Thread Programming

Comp-303 : Programming Techniques
Lecture 11

Alexandre Denault
Computer Science
McGill University
Winter 2004

Announcements

- We have a big lecture ahead of us today.
- You're getting back the Project Req&Spec Document.
- You're getting back Assignment 1.
- Assignment 2 will be handed out today.
- The midterm is next week.
- Valentine's day is Saturday.

Project Req&Spec

- No changes in any project.
- Stapling the sheets together is a MINIMUM!
- Here are the class projects.

Assignment 1

- Read the specification (largerThan is not greaterThan)
- A BigInt powers 0 gives 1.
- Code reuse was important.
- Immutable means the properties are read only.
- Storing larger numbers was OK.
- Scripts used to correct are available on the course website.
- Questions about grades are to be directed to the T.A. (office hours)
- T.A.'s policy: No paper copy, no grades

Assignment 2

In this assignment, you must implement the BigIntSet data abstraction using the BigInt abstraction you built in your first assignment.

BigIntSet can store a collection of large integer numbers (BigInts). Users can arbitrary add and remove BigInt from the set. Using the Adder interface, users can calculate in various ways the sum of elements in this set. Users can also use the Comparator interface to define an order in the set and obtain a sorted iterator.

Assignment 2

To solve this assignment, you will need a BigInt class. You are free to use either your solution or the T.A.'s solution (found on the course webpage). Please note that the T.A.'s solution is not guaranteed to be free of bugs, thought it has been extensively tested. If you do find a bug, it is up to you to fix it (this is just one of the quirks of working with somebody else's code).

Assignment 2

- Modify the BigInt class so it can implement the Comparable interface.
- Modify the BigInt class so it can implement the Cloneable interface.
- Fill in the bodies of the methods of class BigIntSet. (constructors, add, choose, elements, isIn, remove, sort and sum)
- Add methods ”String toString()” and ”boolean equals(Object o) with specifications.
- Build a generator class for the BigIntSet iterator function (see elements).

Assignment 2

- Build a CollectionHasChangedException class extending the Runtime Exception class. You are free to add any methods you think are appropriate.
- Build the following object using the Adder interface:
  - BigIntAdder: This object will simply add two BigInt and return the result as a BigInt.
    i.e. return o1+o2

Assignment 2

- Build the following two objects using the Comparator interface.
  - BigIntCompare: Compares two BigInt object in the following way:
    - if o1 is larger than o2, return a positive number
    - if o1 is equal to o2, return 0
    - if o1 is smaller than o2, return a negative number
  - BigIntInverseCompare: Compares two BigInt object in the following way:
    - if o1 is larger than o2, return a negative number
    - if o1 is equal to o2, return 0
    - if o1 is smaller than o2, return a positive number
- Create the appropriate test code to test all of these functionalities.

Assignment 2

This assignment must be submitted on WebCT and in paper format (drop off box in McConnell). You must submit all the Java files the T.A. will need to correct your assignment (that means don't forget to resubmit your modified BigInt class).

Make sure you respect the specifications. This means that a method without REQUIRES specifications should accept all inputs (Total implementation), and that a method with REQUIRES specs should preserve those specs.

Last lecture . . .

- Polymorphic abstractions provide a way to abstract from the types of parameters.
- Procedures, Iterators and data abstractions can benefit from polymorphic abstraction.
- A polymorphic abstraction usually requires certain methods to apply to the parameters.
- In the element subtype approach, an interface defines these methods which take a parameter object as receiver (define a supertype).
- In the related subtype approach, an interface defines a new object type which has these methods that take parameter objects as arguments.

Sorting Game

Sort order: As, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K

References and additional material

- This course is *heavily* inspired from the Sun's Thread Tutorial (i.e. a lot of material was taken directly from the tutorial):

  `http://java.sun.com/docs/books/tutorial/essential/threads/index.html`

- A good (but advanced) book on concurrency would be:

  `Multithreaded, Parallel, and Distributed Programming`
  `Gregory R. Andrews`
  `Addison Wesley`

Sequential Programming

- Every programmer knows how to program sequentially.
- These programs have a beginning, an execution sequence and an ending.
- At any given point, there is only one point of execution.

Use of threading

- Threading is often used without us recognizing it :
  - In a Web Browser, you can scroll a webpage while it is being loaded.
  - In a Word Processor, you mistakes are highlighted while you type.
  - In an Email client, new emails are retrieved while you read older messages.
  - In XCode, your program is being compiled as you write it.

What are threads

- A thread is similar to a real process . . .
  - Both have independent flow of control.
  - Both are time-shared (and scheduled) on single CPU machines.
- However, threads lack several features of real processes . . .
  - Threads within a process share resources (memory, file descriptors, etc).
  - Since less resources are allocated to threads, the context switch between them is much faster.
- Sometimes, the names *lightweight process* or *execution context* are used to describe a thread.

Building a thread

- The execution of a thread is defined by a *run* method.
- By implementing this *run* method, you define the behavior of that thread.
- There are two ways of implementing a run method for a thread
  - Subclassing *Thread* and Overriding run
  - Implementing the *Runnable* Interface

Subclassing *Thread* and Overriding run

```
public class SimpleThread extends Thread {

   private String name;

   public SimpleThread(String str) {
      this->name = str;
   }

   public void run() {
      for (int i = 0; i < 8; i++) {
         System.out.println(i + " " + this->name);
         try {
            sleep((long)(Math.random() * 1000));
         } catch (InterruptedException e) {}
      }

      System.out.println("DONE! " + this->name);
   }
}
```

Using our new thread

```
public class TwoThreadsDemo {
   public static void main (String[] args) {

      Thread firstThread = new SimpleThread("Apple");
      Thread secondThread = new SimpleThread("Orange");

      firstThread.start();
      secondThread.start();
    }
  }
```

And the results . . .

```
0 Apple
0 Orange
1 Orange
1 Apple
2 Apple
2 Orange
3 Orange
3 Apple
4 Apple
4 Orange
5 Apple
5 Orange
6 Apple
6 Orange
7 Orange
DONE! Orange
7 Apple
DONE! Apple
```

Implementing the *Runnable* Interface

```
public class SimpleThread extends Objects implements Runnable {

   private String name;

   public SimpleThread(String str) {
      this->name = str;
   }

   public void run() {
      for (int i = 0; i < 8; i++) {
         System.out.println(i + " " + this->name);
         try {
            Thread.sleep((long)(Math.random() * 1000));
         } catch (InterruptedException e) {}
      }

      System.out.println("DONE! " + this->name);
   }
}
```

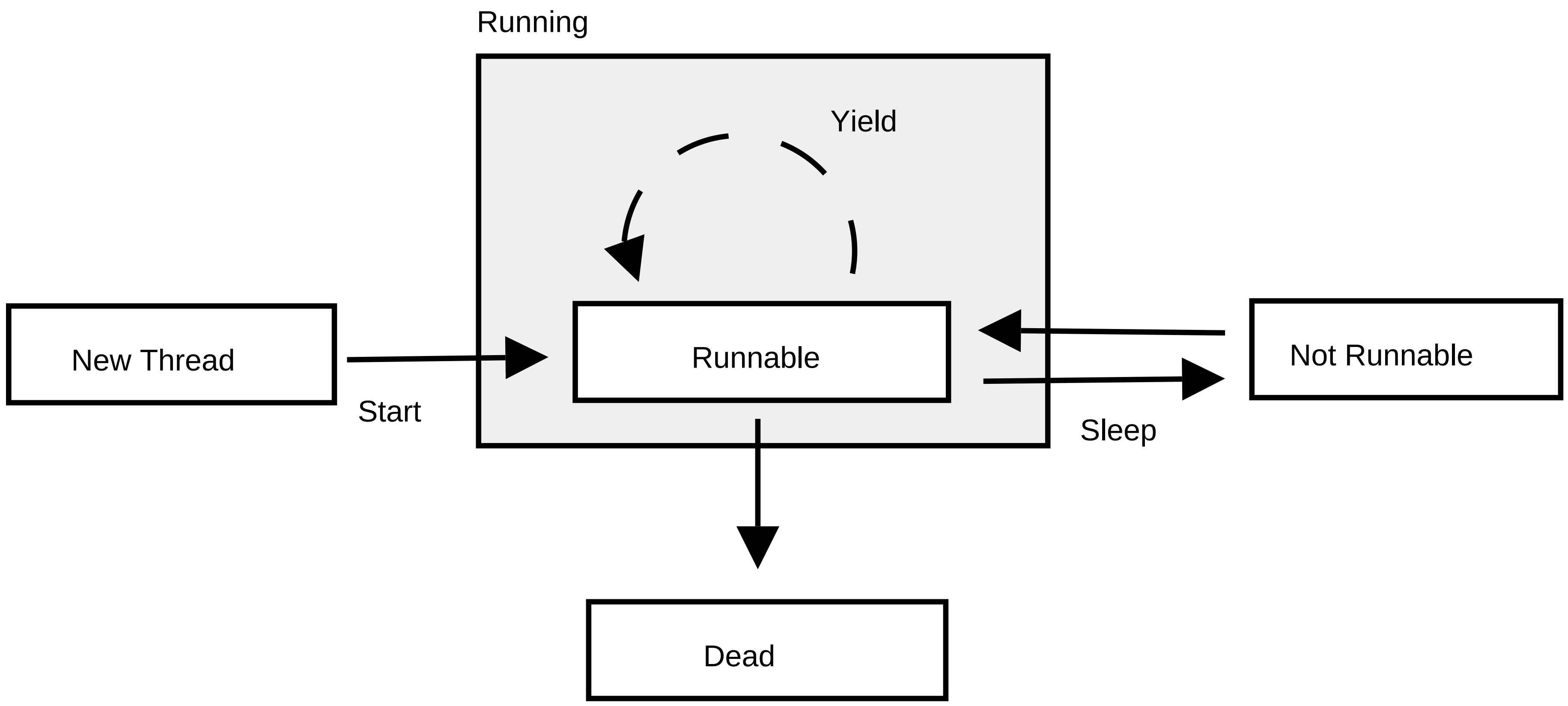
Using the *Runnable* object

```
public class TwoThreadsDemo {
   public static void main (String[] args) {

      Thread firstThread = new Thread(new SimpleThread("Apple"));
      Thread secondThread = new Thread(new SimpleThread("Orange"));

      firstThread.start();
      secondThread.start();
    }
  }
```

Thread VS *Runnable*

- There are good reasons for using either (thread groups, coding style, abstraction, etc).
- Rule of Thumb: If your class must subclass some other class (the most common example being Applet), you should use Runnable.

The life and dead of a thread



The Birth of a Thread

```
Runnable newProject = new Project();
Thread projectThread = new Thread(newProject);
firstThread.start();
```

- When a *Thread* object is created (after calling *new*), it is in the *New Thread* state.
- Once the *start* method is called, the thread is transferred in the *Runnable* state.
- Resource needed to run the thread are only allocated when the *start* method is called.

Making a Thread Not Runnable

- A thread becomes Not Runnable when one of these events occurs:
  - Its sleep method is invoked.
  - The thread calls the wait method to wait for a specific condition to be satisfied.
  - The thread is blocking on I/O.
- For each entrance into the Not Runnable state, there is a specific and distinct escape route that returns the thread to the Runnable state.
  - If a thread has been put to sleep, then the specified number of milliseconds must elapse.
  - If a thread is waiting for a condition, then another object must notify the waiting thread of a change in condition.
  - If a thread is blocked on I/O, then the I/O must complete

Stopping a Thread

- A thread arranges for its own death by having a run method that terminates naturally (like the *main* method).
- We usually control the life and death of a thread through a *while* loop.

```
public void run() {

   while ( booleanFinished == false ) {
      \\ Do thread stuff here
   }

   \\End of the thread
}
```

Understanding Thread Priority

- In theory, threads should run concurrently.
- In practice, this isn't true.
- Most computer have only one CPU, true cannot run two threads concurrently.
- We can create the illusion of concurrency through scheduling.
- By alternating the running thread, the JVM gives the appearance of concurrency.
- Java supports a deterministic scheduling algorithm known as *fixed priority scheduling*.
- This algorithm schedules threads based on their priority relative to other runnable threads.

Thread priority

- A thread inherits its priority from the thread that created it.
- Users can modify a thread's priority at any time after its creation using the *setPriority* method.
- Thread priorities are integers ranging between MIN_PRIORITY and MAX_PRIORITY (constants defined in the Thread class).
- The higher the integer, the higher the priority.
- When multiple threads are ready to be executed, the runtime system usually chooses the runnable thread with the highest priority for execution.
- In the JVM specification, the actual implementation of this scheduling algorithm is loosely defined.
- You should avoid thread priority if you can.

Rule of thumb

At any given time, the highest priority thread is running. However, this is not guaranteed. The thread scheduler may choose to run a lower priority thread to avoid starvation. For this reason, use priority only to affect scheduling policy for efficiency purposes. Do not rely on thread priority for algorithm correctness.

Selfish Threads

```
public class SimpleThread extends Thread {
   public int tick = 1;

   public void run() {
      while (tick < 10000000) tick++;
   }
}
```

- The following thread implements a selfish behavior.
- The while loop in the run method is in a tight loop.
- In other words, the thread will not stop until the loop is finished.
- This will starve the other threads.

How do we fix selfish behavior?

```
public void run() {
   while (tick < 400000) {
      tick++;
      if ((tick % 50000) == 0)
         System.out.println("Thread #" + num + ", tick = " + tick);
    }
}
```

- In this example, the selfish behavior is avoided because of the call to the *println* method.
- Since the *println* method does I/O, it forces the thread to yield its position.
- We can force a thread to yield its position using the *yield()* method.

Synchronizing multiple threads

- Some section of code should never be executed concurrently.
- For example, you might not want two Threads sending information to the Printer at the same time.
- These section of code are called *critical section.*
- The proper manipulation of these sections is much too large for the scope of this class.
- However, Java does provide mechanism to handle these at a rudimentary level.

Locking an Object

- Java allows you to use the synchronized keyword
  - In a method header / signature
  - In a block
- To enter a synchronized section of code (or method), the thread must obtain the lock for that object.
- Once obtained, the thread will not release the lock until it exits the synchronized section.
- Proper use of synchronize blocks/methods can remove race conditions.

  In computer programming and electronics, a race condition is the anomalous behavior due to unexpected critical dependence on the relative timing of events. (Wikipedia)

Examples of Synchronized

```
public synchronized int get() {
        ...
}
```

or

```
public int get() {
        ...
        synchronized {
         ...
        }
        ...
}
```

Reentrant Locks

- The Java runtime system allows a thread to re-acquire a lock that it already holds because Java locks are reentrant.
- If the locks were not reentrant, a call to *a()* would deadlock the system.

```
public class Reentrant {

   public synchronized void a() {
      b();
      System.out.println("here I am, in a()");
   }

   public synchronized void b() {
      System.out.println("here I am, in b()");
   }
}
```

Blocking Behavior

```
public class Box {

   bool available;
   Object contents;

   public synchronized Object get() {    // won't work!
      if (available == true) {
         available = false;
         return contents;
      }
    }

    public synchronized void put(Object value) {    // won't work!
       if (available == false) {
          available = true;
          contents = value;
       }
   }
}
```

Example: wait(), notify(), notifyAll()

```
public synchronized Object get() {
   while (available == false) {
      try {
         // wait for User to put value
           wait();
        } catch (InterruptedException e) { }
     }

     available = false;
     // notify User that value has been retrieved
     notifyAll();

     return contents;
}
```

Example: wait(), notify(), notifyAll()

```
public synchronized void put(Object value) {
   while (available == true) {
      try {
         // wait for Consumer to get value
         wait();
      } catch (InterruptedException e) { }
   }

   contents = value;
   available = true;

   // notify Consumer that value has been set
   notifyAll();
}
```

wait() method

- The *wait()* method allows us to stop a thread until a condition is met.
- When *wait()* is called, the thread relinquished its locks of the object.
- It will try regain that lock when it is notified. It cannot continue without the lock.
- There are two variations to the wait method
  - wait(long timeout) : Waits for notification or until the timeout period has elapsed. timeout is measured in milliseconds.
  - wait(long timeout, int nanos) : Waits for notification or until timeout milliseconds plus nanos nanoseconds have elapsed.

notify(), notifyAll() methods

- The *notify()* method will wake up one waiting thread.
- The *notifyAll()* method will wake up all waiting thread.
- The threads will compete to be the first to requires the lock. The other will go back to waiting.

sleep() method

- The *sleep()* method is quite similar to the *wait()* method.
- Both delay a thread for the requested amount of time.
- However, a sleeping thread cannot be awakened prematurely.

Avoiding Starvation and Deadlock

- Starvation is caused when a thread is not allowed to run (other threads are selfish).
- Deadlock is caused when multiple thread block trying to reserve multiple resources.
- These problems are most often illustrated using the dining philosophers problem.
- The tools Java provided will not protect you.
- It's quite easy to deadlock in Java.

Grouping Threads

- Threads can be grouped using ThreadGroups.
- These are outside the scope of this tutorial.

Summary

- Java provides many tools to implement threading behavior.
- When implementing threads, you have the choice between extending Thread and implementing Interface.
- Methods such as *yield()*, *sleep()*, *wait()*, *notify()* and *notifyAll()* allow you to control the behavior of your threads.
- We have barely scratched the surface: timers, thread groups, priorities, etc.
- There are many more issues you have to deal with when programming concurrent behavior: race condition, atomicity, sharing, etc.
- If you're interested in learning more about concurrency, check out Comp-409.

Tool of the day: Java 2 SE 1.5

- Java 2 SE 1.5 is the newest version of the Java Environment.
- It is still Beta, so you shouldn't be using for your project.
- It has many new features:
  - Metadata: annotation which can be read by the javac compiler or other tools
  - Generic Types: or Templates, if you know C++
  - Autoboxing and Auto-unboxing of Primitive Types
  - Enhanced for loop: for loops that uses iterators
  - Enumerated types: list of constants
  - Formatted Output: printf strikes back
  - Formatted Input: Easier way to read from streams
  - Concurrency Utilities: Easier to shoot yourself in the foot
- We will take a detailed look at 1.5 at the end of the session.