Threads in JDK 5

© 2004 Bruce Eckel
MindView, Inc. www.MindView.net
Training & Project Reviews
Bruce@EckelObjects.com

I’ll put these slides on the site:
http://mindview.net/Etc/ebig
I’ll try to be both

- Informative
- Entertaining

Q & A is during the talk
Java JDK 5 “new” Threads

- Sun has changed threading with every release, because it’s always been broken
- JDK 5 may (probably) work (finally)
- Big rewrite of the concurrency chapter (again) for TIJ4
  – Based on TIC++ V2
The pro(di)gression of Threads

- **JDK 1.2:** `suspend()`, `resume()` and `stop()` are deprecated. `destroy()` never implemented, also deprecated.
- **June 2001:** Josh Bloch points out in *Effective Java* that thread groups have never worked correctly.
  - Also: priorities not particularly useful.
When writing TIJ3 using a dual-processor machine, I discovered very significant behavioral differences when my programs ran on a single-processor machine.
JDK 5

- It turns out there has always been a problem with the `volatile` keyword
- Explains (?) my problems in TIJ3
- JDK 5 is supposed to fix this
- They’ve also added a big new library which basically encapsulates pthreads
“Threat or Menace?”

- From Eric Raymond’s chapter on concurrency in *Art of Unix Programming*
- “Separate processes (separate address spaces) are always best”
- Stroustrup: trying to make multiprocessing perfect made it too slow. Threading arose to compensate for this
- A necessary evil
  - Procedural, not OO
  - Breaks object encapsulation
Demo class

public class LiftOff implements Runnable {
    private int countDown = 10; // Default
    private static int threadCount = 0;
    private int id = threadCount++;
    public LiftOff() {
    }
    public LiftOff(int countDown) {
        this.countDown = countDown;
    }
    public String toString() {
        return "#" + id + ": " +
                (countDown > 0 ? countDown : "Liftoff!");
    }
    public void run() {
        while(countDown-- > 0)
            System.out.println(this);
    }
}
// The most basic use of the Thread class.

public class BasicThreads {
    public static void main(String[] args) {
        Thread t = new Thread(new LiftOff());
        t.start();
        System.out.println("Waiting for LiftOff");
    }
}
Executors

- Simplifies the manipulation of threads
- Can manage threads in pools
  - `CachedThreadPool` (dynamic number)
  - `FixedThreadPool` (fixed number)
- Can be a single “worker” thread with built-in queue: `SingleThreadExecutor`
import java.util.concurrent.*;

public class CachedThreadPool {
    public static void main(String[] args) {
        ExecutorService e = Executors.newCachedThreadPool();
        for (int i = 0; i < 5; i++)
            e.execute(new LiftOff());
        e.shutdown(); // Executes previously submitted tasks
    }
}
import java.util.concurrent.*;

public class FixedThreadPool {
    public static void main(String[] args) {
        // Constructor arg is number of threads:
        ExecutorService e = Executors.newFixedThreadPool(5);
        for(int i = 0; i < 5; i++)
            e.execute(new LiftOff());
        e.shutdown(); // Executes previously submitted tasks
    }
}
import java.util.concurrent.*;

public class SingleThreadExecutor {
    public static void main(String[] args) {
        ExecutorService e = Executors.newSingleThreadExecutor();
        for (int i = 0; i < 5; i++)
            e.execute(new LiftOff());
        e.shutdown(); // Executes previously submitted tasks
    }
}
Callables

- Runnable task doesn’t return a value, can’t throw an exception
- Callable<V> task can return a value and throw an exception
- Since it’s a task, when will the return value be ready?
- Future<V>
  - get() blocks until completion
  - isDone() tells you if it’s ready
import java.util.concurrent.*;
import java.util.*;

class TaskWithResult implements Callable<String> {
    private int id;
    public TaskWithResult(int id) {
        this.id = id;
    }
    public String call() {
        return "result of TaskWithResult " + id;
    }
}
public class CallableDemo {
    static final int SZ = 10;
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        ArrayList<Future<String>> results = new ArrayList<Future<String>>();
        for(int i = 0; i < SZ; i++)
            results.add(Executors.execute(exec, new TaskWithResult(i)));
        for(Future<String> fs : results)
            try {
                // get() blocks until completion:
                System.out.println(fs.get());
            } catch(InterruptedException e) {
                throw new RuntimeException(e);
            } catch(ExecutionException e) {
                throw new RuntimeException(e);
            }
        exec.shutdown();
    }
}
New mutexes

- `java.util.concurrent.locks.Lock`
- `java.util.concurrent.locks.Condition`
  - `await()` instead of `wait()`
    - You can still call `wait()` incorrectly, though!
  - `signal()` instead of `notify()`
// Basic thread cooperation.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

class Car {
    private Lock lock = new ReentrantLock();
    private Condition condition =
        lock.newCondition();
    private boolean waxOn = false;
    public void waxed() {
        lock.lock();
        try {
            waxOn = true; // Ready to buff
            condition.signal();
        } finally {
            lock.unlock();
        }
    }
}
public void buffed() {
    lock.lock();
    try {
        waxOn = false; // Ready for another coat of wax
        condition.signal();
    } finally {
        lock.unlock();
    }
}

public void waitForWaxing() {
    lock.lock();
    try {
        while(waxOn == false)
            try {
                condition.await();
            } catch(InterruptedException e) { /* Exit */ }
    } finally {
        lock.unlock();
    }
}
public void waitForBuffing() {
    lock.lock();
    try {
        while (waxOn == true) {
            try {
                condition.await();
            } catch (InterruptedException e) { /* Exit */ }
        }
        catch (InterruptedException e) { /* Exit */ }
    } finally {
        lock.unlock();
    }
}
class WaxOn implements Runnable {
    private Car car;
    public WaxOn(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                System.out.println("Wax On!");
                Thread.sleep(200);
                car.waxed();
                car.waitForBuffing();
            }
            catch(InterruptedException e) { /* Exit */ } }
        System.out.println("Ending Wax On process");
    }
}
class WaxOff implements Runnable {
    private Car car;
    public WaxOff(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                car.waitForWaxing();
                System.out.println("Wax Off!");
                Thread.sleep(200);
                car.buffed();
            }
        } catch(InterruptedException e) { /* Exit */ } }
        System.out.println("Ending Wax Off process");
    }
}
```java
public class WaxOMatic {
    public static void main(String[] args) {
        System.out.println("Press <Enter> to quit");
        Car car = new Car();
        ExecutorService e = Executors.newCachedThreadPool();
        e.execute(new WaxOff(car));
        e.execute(new WaxOn(car));
        try {
            System.in.read();
        } catch(java.io.IOException ioe) {
            throw new RuntimeException(ioe);
        }
        e.shutdownNow();
    }
}
```
Thread queues

- Synchronize to ensure that no two threads add objects at the same time
- Suspend a consumer thread if that thread tries to get an object from the queue and the queue is empty, and resume when more elements become available
- Queues can solve a remarkable number of problems
import java.util.concurrent.*;

class LiftOffRunner implements Runnable {
    private BlockingQueue<LiftOff> rockets;
    public LiftOffRunner(BlockingQueue<LiftOff> queue) {
        rockets = queue;
    }
    public void add(LiftOff lo) {
        try {
            rockets.put(lo);
        } catch(InterruptedException e) {
            System.out.println("Interrupted during put() ");
        }
    }
}
public void run() {
    try {
        while (!Thread.interrupted()) {
            LiftOff rocket = rockets.take();
            rocket.run(); // Use this thread
        }
    } catch (InterruptedException e) {
        System.out.println("Waking from take()");
    }
    System.out.println("Exiting LiftOffRunner");
}
public class TestBlockingQueues {
    static void getkey() {
        try {
            System.in.read();
        } catch (java.io.IOException e) {
            throw new RuntimeException(e);
        }
    }
    static void getkey(String message) {
        System.out.println(message);
        getkey();
    }
}
static void test(String msg, BlockingQueue<LiftOff> queue) {
    System.out.println(msg);
    LiftOffRunner runner = new LiftOffRunner(queue);
    Thread t = new Thread(runner);
    t.start();
    for(int i = 0; i < 5; i++)
        runner.add(new LiftOff(5));
    getkey("Press <Return> (" + msg + ")");
    t.interrupt();
    System.out.println("Finished " + msg + " test");
}
public static void main(String[] args) {
    test("LinkedBlockingQueue", // Unlimited size
        new LinkedBlockingQueue<LiftOff>();
    test("ArrayBlockingQueue", // Fixed size
        new ArrayBlockingQueue<LiftOff>(3));
    test("SynchronousQueue", // Size of 1
        new SynchronousQueue<LiftOff>();
}
// A toaster that uses queues.
import java.util.*;
import java.util.concurrent.*;
import static com.bruceeckel.Tools.*;
class Toast {
    public enum Status { DRY, BUTTERED, JAMMED }
    private Status status = Status.DRY;
    private int id;
    public Toast(int idn) { id = idn; }
    public void butter() { status = Status.BUTTERED; }
    public void jam() { status = Status.JAMMED; }
    public Status getStatus() { return status; }
    public int getId() { return id; }
    public String toString() {
        return "Toast " + id + ": " + status;
    }
}
class ToastQueue extends LinkedBlockingQueue<Toast> {}
class Toaster implements Runnable {
    private ToastQueue toastQueue;
    private int count = 0;
    private Random rand = new Random();
    public Toaster(ToastQueue tq) { toastQueue = tq; }
    public void run() {
        try {
            while (!Thread.interrupted()) {
                Thread.sleep(100 + rand.nextInt(500));
                // Make toast
                Toast t = new Toast(count++);
                print(t);
                // Insert into queue
                toastQueue.put(t);
            }
        } catch (InterruptedException e) {
            print("Toaster interrupted");
        }
        print("Toaster off");
    }
}
```java
class Butterer implements Runnable {
    private ToastQueue dryQueue, butteredQueue;
    public Butterer(ToastQueue dry, ToastQueue buttered) {
        dryQueue = dry;
        butteredQueue = buttered;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Blocks until next piece of toast is available:
                Toast t = dryQueue.take();
                t.butter();
                print(t);
                butteredQueue.put(t);
            }
        } catch(InterruptedException e) {
            print("Butterer interrupted");
        }
        print("Butterer off");
    }
}
```
// Apply jam to buttered toast:
class Jammer implements Runnable {
    private ToastQueue butteredQueue, finishedQueue;
    public Jammer(ToastQueue buttered, ToastQueue finished) {
        butteredQueue = buttered;
        finishedQueue = finished;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Blocks until next piece of toast is available:
                Toast t = butteredQueue.take();
                t.jam();
                print(t);
                finishedQueue.put(t);
            }
        } catch(InterruptedException e) {
            print("Jammer interrupted");
        }
        print("Jammer off");
    }
}
// Consume the toast:

class Eater implements Runnable {
    private ToastQueue finishedQueue;
    private int counter = 0;
    public Eater(ToastQueue finished) {
        finishedQueue = finished;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Blocks until next piece of toast is available:
                Toast t = finishedQueue.take();
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
// Verify that the toast is coming in order,
// and that all pieces are getting jammed:
if (t.getId() != counter++ ||
    t.getStatus() != Toast.Status.JAMMED) {
    print(">>> Error: " + t);
    System.exit(1);
} else {
    print("Chomp! " + t);
}
} catch (InterruptedException e) {
    print("Eater interrupted");
}
print("Eater off");
public class ToastOMatic {
    public static void main(String[] args) throws Exception {
        ToastQueue dryQueue = new ToastQueue(),
        butteredQueue = new ToastQueue(),
        finishedQueue = new ToastQueue();
        ExecutorService e = Executors.newCachedThreadPool();
        e.execute(new Toaster(dryQueue));
        e.execute(new Butterer(dryQueue, butteredQueue));
        e.execute(new Jammer(butteredQueue, finishedQueue));
        e.execute(new Eater(finishedQueue));
        Thread.sleep(5000);
        e.shutdownNow();
    }
} ///:~
A dizzying # of other things

- Atomicity
- Counting Semaphore
- CyclicBarrier
  - Allows a set of threads to all wait for each other to reach a common barrier point
- Exchanger
  - Provides a synchronization point at which two threads can exchange objects
- **ScheduledExecutor**
  - Schedule commands to run after a given delay, or to execute periodically. Preferable to Timer

- **CountDownLatch**
  - Synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes

- **ConcurrentHashMap, ConcurrentLinkedQueue**
  - Thread safe and reasonable
- CopyOnWriteArrayList, CopyOnWriteArraySet
  - All mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array
Fixing suspend/resume

- Deprecated in JDK 1.3
- `java.util.concurrent.locks.LockSupport`
  - `park()`
  - `unpark()`
Passive Objects

- Threads contend at the entry point to methods
Active Objects

- Serialized on a message basis
// Can only pass constants, immutables, or other active
// objects as arguments to asynch methods

```java
import java.util.*;
import java.util.concurrent.*;

public class ActiveObjectDemo1 {
    private ExecutorService ex = Executors.newSingleThreadExecutor();
    private Random rand = new Random();
    private void pause() {
        try {
            Thread.sleep(100 + rand.nextInt(1500));
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```
public Future<Integer> calculateInt(final int x, final int y) {
    return Executors.execute(ex, new Callable<Integer>() {
        public Integer call() {
            System.out.println("starting " + x + " + " + y);
            pause();
            return x + y;
        }
    });
}
public Future<Float> calculateFloat(final float x, final float y) {
    return Executors.execute(ex, new Callable<Float>() {
        public Float call() {
            System.out.println("starting " + x + " + " + y);
            pause();
            return x + y;
        }
    });
}
public void shutdown() { ex.shutdown(); }
public static void main(String[] args) {
    ActiveObjectDemo1 d1 = new ActiveObjectDemo1();
    ArrayList<Future<?>> results =
        new ArrayList<Future<?>>();
    for(int i = 0; i < 5; i++)
        results.add(d1.calculateInt(i, i));
    for(float f = 0.0f; f < 1.0f; f += 0.2f)
        results.add(d1.calculateFloat(f, f));
    System.out.println("All asynch calls made");
while(results.size() > 0) {
    // Prevent ConcurrentModificationException:
    ArrayList<Future<?>> resultsCopy =
        new ArrayList<Future<?>>(results);
    for (Future<?> f : resultsCopy)
        if (f.isDone()) {
            try {
                System.out.println(f.get());
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            } catch (ExecutionException e) {
                throw new RuntimeException(e);
            }
            results.remove(f);
        }
    d1.shutdown();
}
Latest book

- Intermediate & Advanced topics
  - Templates
  - STL
  - Design Patterns
  - Multithreading
- Downloadable from web site
Working On

- Thinking in Java, 4th Edition
- Rewritten for JDK 5
- In print ~February
Questions

Email: Bruce@EckelObjects.com