

Threads

(2008.12.23)

Frank Ducrest

I. Introduction to Java Threads

Threads

- **streams of code execution**
- **allows a running program to switch between simultaneously executing sets of code**
- **low overhead compared to operating system multiprocessing**
- **example: can allow program to switch back and forth between a slow but lengthy process to a more rapid and intense process**

Reasons to use Threads

- **avoid having an interactive program go dead on a user while a slow task completes or waits for a resource**
example: saving the state of a document while the user continues to work on that document
- **some programs make more sense if they run sub tasks as separate and simultaneous operations**
example: a server that must serve an unknown number of clients
- **some programs are amenable to parallel processing**
example: a program that must compute two separate results that are to be combined later

Obtaining a Thread using *Runnable*

- implement interface *Runnable* (*Thread* is implemented by *Runnable*) and override *run()*

```
class MyThread implements Runnable {  
    public void run() { ... code ... }  
    ... more code ...  
}
```

```
MyThread mt = new MyThread();  
Thread t = new Thread(mt);  
t.start();
```

<http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html>

<http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Runnable.html>

Obtaining a Thread using *Thread*

- extend `java.lang.Thread` and override `run()`

```
class MyThread extends Thread {  
    public void run() { ... code ... }  
    ... more code ...  
}
```

```
MyThread mt = new MyThread();  
mt.start();
```

<http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html>

<http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Runnable.html>

Some Thread Methods

```
void start() // start the thread running

void run() // executed on start

static void sleep(long millis)
// thread sleeps for millis

void destroy() // destroys thread

static Thread currentThread()
// returns current thread

void setName(String)
// sets name of thread

void getName() // returns name of thread
```

Some Thread Methods (continued)

```
void setPriority(int)
                // sets priority of thread

int getPriority()
                // returns thread priority

void join()      // waits for thread to die

void join(long millis)
                // waits up to millis for
                // thread to die

static int activeCount()
                // returns number of active
                // threads in group
```

Some Thread Methods (continued)

```
void interrupt()    // interrupt thread

static boolean interrupted()
                    // true if thread
                    // interrupted

boolean isAlive()
                    // true if thread alive

boolean isDaemon()
                    // true if thread is daemon

void yield()        // thread yields to waiting
                    // thread

void notifyAll()    // wake up all waiting
                    // threads
```

Thread Life-Cycle

- a thread dies when *run* halts, such as through a return or exception
- a thread of a higher priority can preempt a thread of a lower priority
- threads of equal priority are not guaranteed equal time (better to have a CPU intensive thread yield)
- **warning: threads created in anonymous classes are NOT garbage collected until *run* ends, can hang program**

Thread Groups

- a group of threads created with class *ThreadGroup*
- operations can be performed on the group, the group can be passed
- creating a thread group:

```
private ThreadGroup myGroup;
```

- adding a thread to a group:

```
myGroup = Thread.currentThread().getThreadGroup();
```

Demon Thread

- thread that is marked as a demon

`aThread.setDaemon(true);`

- program running only demon threads terminates when the demon threads have nothing more to do
- example: **printer queue**

Thread Programming

- **synchronization refers to coordinating threads**
- **thread programming types**
 - **unrelated threads**
 - **related but unsynchronized threads**
 - **mutually exclusive threads**
 - **communicating mutually exclusive threads**

II. Unrelated Threads

Unrelated Thread Example

```
// file UncleBob.java
public class UncleBob extends Thread {
    public void run() {
        for (int x=0; x<5; x++) {
            System.out.println("... and Bob's your uncle.");
            yield();
        }
    }
}
```

```
// file NotUncleBob.java
public class NotUncleBob extends Thread {
    public void run() {
        for (int x=0; x<5; x++) {
            System.out.println("... and he is not Bob.");
            yield();
        }
    }
}
```

Unrelated Thread Example

```
// file SayUncle.java
public class SayUncle {
    public static void main(String[] args) {
        UncleBob ub = new UncleBob();
        NotUncleBob nub = new NotUncleBob();
        ub.start();
        nub.start();
    }
}
```

Output:

```
... and Bob's your uncle.
... and he is not Bob.
... and Bob's your uncle.
... and he is not Bob.
... and Bob's your uncle.
... and he is not Bob.
... and Bob's your uncle.
... and he is not Bob.
... and Bob's your uncle.
... and he is not Bob.
```

III. UnSynchronized Related Threads

Related but UnSynchronized Thread Example

```
// file IHaveThePower.java
public class IHaveThePower extends Thread {
    private long num;
    private static long pow;

    public IHaveThePower(long num, long pow) {
        this.num = num;
        this.pow = pow;
    }

    public void run() {
        long numToPower = 1;
        for (long x=1; x<=pow; x++)
            numToPower *= num;
        System.out.println(num + " to power " + pow
            + " is " + numToPower);
        yield();
    }
}
```

Related but UnSynchronized Thread Example

```
// file Powers.java
import javax.swing.*;

public class Powers {

    public static void main(String[] args) {
        String input;
        long hi, lo, pow;

        input = JOptionPane.showInputDialog(
            "Enter High Integer");
        if (input == null) input = "0";
        hi = Long.parseLong(input);

        input = JOptionPane.showInputDialog(
            "Enter Low Integer");
        if (input == null) input = "0";
        lo = Long.parseLong(input);
```

Related but UnSynchronized Thread Example

```
// file Powers.java continued

    input = JOptionPane.showInputDialog(
        "Enter Positive Integer Power");
    if (input == null) input = "0";
    pow = Long.parseLong(input);

    for (long x=lo; x<=hi; x++) {
        new IHaveThePower(x, pow).start();
    }
}

} // end of Powers.java
```

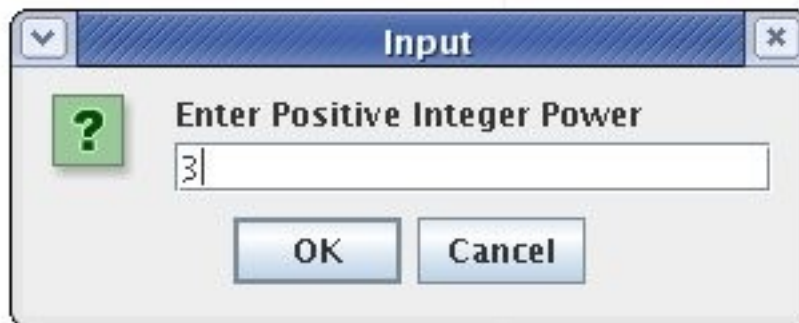
Related but UnSynchronized Thread Example



Input dialog box titled "Input" with a question mark icon. The text "Enter High Integer" is displayed above a text input field containing the value "100". Below the input field are "OK" and "Cancel" buttons.



Input dialog box titled "Input" with a question mark icon. The text "Enter Low Integer" is displayed above a text input field containing the value "90". Below the input field are "OK" and "Cancel" buttons.



Input dialog box titled "Input" with a question mark icon. The text "Enter Positive Integer Power" is displayed above a text input field containing the value "3". Below the input field are "OK" and "Cancel" buttons.

Output:

90 to power 3 is 729000

97 to power 3 is 912673

98 to power 3 is 941192

95 to power 3 is 857375

96 to power 3 is 884736

94 to power 3 is 830584

99 to power 3 is 970299

93 to power 3 is 804357

100 to power 3 is 1000000

91 to power 3 is 753571

92 to power 3 is 778688

IV. Thread Priorities

- thread of higher priorities executes ahead of a thread of lower priority
- threads of equal priorities depend on time-slicing, which is not guaranteed; can lead to CPU starvation for a thread
- highest priority is **10**, constant **MAX_PRIORITY**
- lowest priority is **1**, constant **MIN_PRIORITY**
- default priority is **5**, constant **NORM_PRIORITY**

V. Alternating Threads with Priorities

Alternating Threads with Priorities

```
// file MyThread.Java
public class MyThread implements Runnable {

    Thread thread;

    public MyThread() {
        thread = new Thread(this, "My Thread");
        thread.start();
        thread.setPriority(6);
    }

    public void run() {
        try {
            for (int a=0; a<7; a++) {
                System.out.println(
                    Thread.currentThread().getName() + a);
                Thread.sleep((long)(Math.random()*2000));
            }
        } catch (InterruptedException e) {}
    }
}
```

Alternating Threads (continued)

```
// file YourThread.Java
public class YourThread implements Runnable {

    Thread thread;

    public YourThread() {
        thread = new Thread(this, "Your Thread");
        thread.start();
        thread.setPriority(4);
    }

    public void run() {
        try {
            for (int a=0; a<10; a++) {
                System.out.println(
                    Thread.currentThread().getName() + a);
                Thread.sleep((long)(Math.random()*1000));
            }
        } catch (InterruptedException e) {}
    }
}
```

Alternating Threads (continued)

```
// file Demo.java
public class Demo {

    public Demo() {
        MyThread mt = new MyThread();
        YourThread yt = new YourThread();
    }

    public static void main(String[] args) {
        Demo demo = new Demo();
    }
}
```

Alternating Threads (continued)

Run 1

My Thread0
Your Thread0
Your Thread1
Your Thread2
Your Thread3
Your Thread4
My Thread1
Your Thread5
My Thread2
Your Thread6
My Thread3
Your Thread7
My Thread4
Your Thread8
Your Thread9
My Thread5
My Thread6

Run 2

My Thread0
Your Thread0
Your Thread1
Your Thread2
Your Thread3
My Thread1
Your Thread4
Your Thread5
Your Thread6
My Thread2
My Thread3
Your Thread7
Your Thread8
My Thread4
Your Thread9
My Thread5
My Thread6

Run 3

My Thread0
Your Thread0
Your Thread1
Your Thread2
My Thread1
Your Thread3
My Thread2
Your Thread4
Your Thread5
Your Thread6
My Thread3
Your Thread7
My Thread4
Your Thread8
Your Thread9
My Thread5
My Thread6

VI. A CPU Intensive Thread that Yields

A CPU Intensive Thread that Yields

```
// file MyThread.Java
public class MyThread implements Runnable {

    Thread thread;

    public MyThread() {
        thread = new Thread(this, "My Thread");
        thread.start();
        thread.setPriority(5);
    }

    public void run() {
        try {
            for (int a=0; a<7; a++) {
                System.out.println(
                    Thread.currentThread().getName() + a);
                Thread.sleep((long)(Math.random()*5));
            }
        } catch (InterruptedException e) {}
    }
}
```

A CPU Intensive Thread that Yields (continued)

```
// file YourThread.Java
public class YourThread implements Runnable {

    Thread thread;

    public YourThread() {
        thread = new Thread(this, "Your Thread");
        thread.start();
        thread.setPriority(5);
    }

    public void run() {
        for (int a=0; a<10000; a++) {
            if (a % 1000 == 0)
                System.out.println(
                    Thread.currentThread().getName() + a);
            thread.yield();
        }
    }
}
```

A CPU Intensive Thread that Yields (continued)

```
// file Demo.Java
public class Demo {

    public Demo() {
        MyThread mt = new MyThread();
        YourThread yt = new YourThread();
    }

    public static void main(String[] args) {
        Demo demo = new Demo();
    }

}
```

A CPU Intensive Thread that Yields (continued)

Run 1

My Thread0

Your Thread0

My Thread1

Your Thread1000

Your Thread2000

My Thread2

Your Thread3000

Your Thread4000

Your Thread5000

Your Thread6000

Your Thread7000

My Thread3

My Thread4

Your Thread8000

Your Thread9000

My Thread5

My Thread6

Run 2

My Thread0

Your Thread0

My Thread1

Your Thread1000

Your Thread2000

Your Thread3000

My Thread2

Your Thread4000

Your Thread5000

Your Thread6000

Your Thread7000

My Thread3

Your Thread8000

Your Thread9000

My Thread4

My Thread5

My Thread6

Run 3

My Thread0

Your Thread0

My Thread1

Your Thread1000

Your Thread2000

My Thread2

Your Thread3000

Your Thread4000

Your Thread5000

Your Thread6000

My Thread3

My Thread4

My Thread5

Your Thread7000

Your Thread8000

Your Thread9000

My Thread6

VII. Synchronized Threads

Synchronization

- **used to insure mutual exclusion between threads**
- **mutual exclusion may be necessary when two or more threads access the same data**

Synchronization (continued)

Example of why mutual exclusion is necessary:

Starting Bank Balance: \$200

- 1) husband and wife log on to separate ATMS, account balance is copied to ATMs
- 2) husband withdraws \$100 and wife withdraws \$50
- 3) ATM one updates the the bank balance to \$100
- 4) ATM two updates the bank balance to \$150

Final bank balance: \$150

Synchronization (continued)

- each Object has its own semaphore
- when the semaphore is set (captured, accessed) by the call of a thread, no other thread may use this code
- any Object or method or class may be modified to be mutually exclusive with *synchronized* modifier

Synchronization Example

- with communicating mutually exclusive threads
- from:

<http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html>

- Controlling the Greedy Consumer:

A value is placed in a mutually accessible object by the producer. The consumer is allowed to access each newly stored value only once.

Synchronization Example (continued)

- synchronized *class CubbyHole*
 - 1) use synchronize modifier on *get* and *put* methods
 - 2) add a boolean variable to indicate if the data member is ready to be read or modified
 - 3) use *wait()* from Object class to wait until another thread invokes the *notify()* method or the *notifyAll()* method for this object
 - 4) use *notifyAll()* from Object class to wake up all threads that are waiting on this object

Synchronization Example (continued)

// Synchronized Class *CubbyHole*

```
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        notifyAll();
        return contents;
    }
}
```

Synchronization Example (continued)

// Synchronized Class *CubbyHole* (continued)

```
public synchronized void put(int value) {
    while (available == true) {
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    contents = value;
    available = true;
    notifyAll();
}
} // end of class Cubbyhole
```

Synchronization Example (continued)

// Class Producer

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number
                               + " put: " + i);

            try {
                sleep((int)(Math.random() * 100));
                // option -> yield();
            } catch (InterruptedException e) { }
        }
    }
}
```

Synchronization Example (continued)

// Class Consumer

```
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" +
                this.number + " got: " + value);
        }
    }
}
```

Synchronization Example (continued)

// Class *ProducerConsumerTest*

```
public class ProducerConsumerTest {  
    public static void main(String[] args) {  
        CubbyHole c = new CubbyHole();  
        Producer p1 = new Producer(c, 1);  
        Consumer c1 = new Consumer(c, 1);  
  
        p1.start();  
        c1.start();  
    }  
}
```

Synchronization Example (continued)

Output:

```
Producer #1 put: 0
Consumer #1 got: 0
Producer #1 put: 1
Consumer #1 got: 1
Producer #1 put: 2
Consumer #1 got: 2
Producer #1 put: 3
Consumer #1 got: 3
Producer #1 put: 4
Consumer #1 got: 4
Producer #1 put: 5
Consumer #1 got: 5
Producer #1 put: 6
Consumer #1 got: 6
Producer #1 put: 7
Consumer #1 got: 7
Producer #1 put: 8
Consumer #1 got: 8
Producer #1 put: 9
Consumer #1 got: 9
```

VIII. Piped Input and Output

Piped I/O

- a pipe is a buffer that can be written to with *PipedInputStream* and read from with *PipedOutputStream*
- buffer is limited – when full, writer cannot write until room is made by the reader

Piped I/O Example

// Class *Producer*

```
import java.io.*;

public class Producer extends Thread {
    protected PipedOutputStream pos = new PipedOutputStream();
    private    DataOutputStream dos = new DataOutputStream(pos);

    public void run() {
        for (int x=0;;x++)
            Produce(x);
    }

    void Produce(int n) {
        System.out.println("Produced: " + n);
        try {
            dos.writeInt(n);
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

Piped I/O Example (continued)

// Class Consumer

```
import java.io.*;
import java.util.*;

public class Consumer extends Thread {
    private PipedInputStream pip;
    private DataInputStream dis;

    public Consumer(Producer p) {
        try {
            pip = new PipedInputStream(p.pos);
            dis = new DataInputStream(pip);
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

Piped I/O Example (continued)

// Class Consumer (continued)

```
int get() {
    int i = 0;

    try {
        i = dis.readInt();
    } catch (IOException e) {
        System.out.println(e);
    }

    return i;
}
```

Piped I/O Example (continued)

// Class Consumer (continued)

```
public void run() {
    Random r = new Random();
    int n;

    for (;;) {
        n = get();
        System.out.println("Consumed: " + n);

        // just to do something else
        try {
            sleep(r.nextInt() % 1000);
        } catch (Exception e) {}
    }
} // end class Consumer
```

Piped I/O Example (continued)

// Class *PipeTest*

```
public class PipeTest {
    public static void main(String[] args) {
        Producer p = new Producer();
        Consumer c = new Consumer(p);

        c.start();
        p.start();
    }
}
```

Piped I/O Example (continued)

Example Output:

```
...  
Produced: 254  
Produced: 255  
Produced: 256  
Consumed: 0  
Consumed: 1  
Produced: 257  
Produced: 258  
Consumed: 2  
Produced: 259  
Consumed: 3  
Consumed: 4  
Consumed: 5  
Consumed: 6  
Consumed: 7  
Consumed: 8  
Consumed: 9  
Produced: 260  
Produced: 261  
...
```