Java - Threads

CS 4354
Summer II 2015

Jill Seaman

Threads

- What is a process?
  - ✦a self-contained running program with its own address space.
  - ✦processes are controlled by the operating system.
- What is a thread?
  - ✦A thread is an execution stream within a process.
- A thread is also called a lightweight process.
  - ✦Has its own execution stack, local variables, and program counter.
  - ✦Very much like a process, but it runs within a process.
- There may be more than one thread in a process.
  - ✦Is called a **multithreaded** process.

Multithreading

- Multithreading:
  - ✦Provides the capability to run tasks in parallel for a process.
  - ✦All threads **share** with each other **resources** allocated to the process.
  - ✦In fact, they compete and may interfere with each other.
- Threads allow the programmer to turn a program into separate, independently running subtasks
- In all cases, thread programming:
  1. Seems mysterious and requires a shift in the way you think about programming
  2. Looks similar to thread support in other languages, so when you understand threads, you understand a common tongue

Threads in Java

- "In general, you'll have some part of your program tied to a particular event or resource, and you don't want that to hold up the rest of your program. So, you create a thread associated with that event or resource and let it run independently of the main program."
- The java.lang.Thread class has all the wiring necessary to create and run threads.
- The run( ) method contains the code that will be executed "simultaneously" with the other threads in a program
- The Java Thread class provides a generic thread that, by default, does nothing.
  - ✦Its run() method is empty, and should be overridden by all subclasses of Thread.

The Runnable Interface

- The java.lang.Runnable Interface
  - ✦This interface should be implemented by any class whose instances are intended to be executed by a thread (but do not want to or cannot subclass Thread).
  - ✦The class must define a method of no arguments called run.
  - ✦A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target.
- Runnable is implemented by the class java.lang.Thread.

Threads in Java

- There are two techniques to implement threads in Java:
  - ✦To subclass Thread and override run().
  - ✦To implement the Runnable interface (by defining run()) and embed class instances in a Thread object.

    This allows a class to have a superclass other than Thread, but still implement a thread.
- Once a Thread instance is created, call the start() method to make it run.
  - ✦ This causes the run() method to be executed in a separate thread.
  - ✦ The code following the call to start() will execute concurrently with the thread's run method.

Subclassing Thread: example

```
public class YinYang extends Thread {
     private String word;             // what to say

     public YinYang(String whatToSay) {
         word = whatToSay;
     }

     public void run() {
         for (int i = 0; i < 10; i++) {
             System.out.print(word + " ");
             yield();                 // to give another thread a chance
         }
     }

     public static void main(String[] args) {
         YinYang yin = new YinYang("Yin");    // to create Yin thread
         YinYang yang = new YinYang("Yang");  // to create Yang thread
         yin.start();                         // to start Yin thread
         yang.start();                        // to start Yang thread
     }
 }
```

output:

```
Yin Yang Yang Yang Yin Yang Yin Yang Yin Yang
Yin Yang Yin Yang Yin Yang Yin Yang Yin Yin
```


Implementing Runnable: example

```
public class YangYin implements Runnable {
    private String word;              // what to say
    public YangYin(String whatToSay) {
        word = whatToSay;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.print(word + " ");
            Thread.yield();           // to give another thread a chance
        }
    }
    public static void main(String[] args) {
        Runnable rYang = new YangYin("Yang");   // to instantiate YangYin
        Runnable rYin = new YangYin("Yin");     // to instantiate again

        Thread yang  = new Thread(rYang);       // to create Yang thread
        Thread yin  = new Thread(rYin);         // to create Yin thread
        yang.start();                           // to start Yang thread
        yin.start();                            // to start Yin thread
    }
}
```

output:

```
Yin Yin Yang Yin Yang Yin Yang Yang Yin Yang Yin
Yang Yang Yin Yang Yin Yang Yin Yang Yin
```

Thread methods

- run()
  - ✦The code that will be run concurrently (in its own thread)
- start()
  - ✦Causes the run method to execute in a separate thread, continues execution (immediately returns control to caller).
- yield()
  - ✦Causes the currently executing thread object to temporarily pause and allow other threads to execute.
- sleep(long milllis)
  - ✦Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds

Thread methods

- join()
  - ✦Causes the calling thread to wait for this thread to complete before proceeding.
- getName()
  - ✦Returns this thread's name (set in the constructor).
- interrupt()
  - ✦Called from outside the thread.
  - ✦interrupts a thread that is paused via sleep(), or join().
  - ✦InterruptedException is generated in the sleep/join
  - ✦Calls to sleep/join must be in a try/catch block

```
public final void join()
                  throws InterruptedException
```


interrupt() example

```
class Sleeper extends Thread {
  private int duration;
  public Sleeper(String name, int sleepTime) {
    super(name);
    duration = sleepTime;
    start();              //starts itself
  }
  public void run() {
    try {
      sleep(duration);
    } catch (InterruptedException e) {
      System.out.println(getName() + " was interrupted.");
      return;
    }
    System.out.println(getName() + " has awakened");
  }
}

class Joiner extends Thread {
  private Sleeper sleeper;
  public Joiner(String name, Sleeper sleeper) {
    super(name);
    this.sleeper = sleeper;
    start();            //starts itself
  }
  public void run() {
    try {
      sleeper.join();
    } catch (InterruptedException e) {
      System.out.println(getName() + " was interrupted. ");
      return;
    }
    System.out.println(getName() + " join completed");
  }
}
```

```
public class Joining {
  public static void main(String[] args) {
    Sleeper
      sleepy = new Sleeper("Sleepy", 1500),
      grumpy = new Sleeper("Grumpy", 1500);
    Joiner
      dopey = new Joiner("Dopey", sleepy),
      doc = new Joiner("Doc", grumpy);
    grumpy.interrupt();
    // doc.interrupt();
  }
}
```

```
Grumpy was interrupted.
Doc join completed
Sleepy has awakened
Dopey join completed
```

```
Doc was interrupted.
Sleepy has awakened
Grumpy has awakened
Dopey join completed
```


Thread synchronization

- We now have the possibility of two or more threads trying to use the same limited resource at once.
  - ✦i.e. two threads trying to access the same bank account at the same time

```
public class Account{
  private float balance = 0;
  public float getBalance(){ return balance; }
  public void incrBalance(float value){ balance = balance + value; }
}
```

```
class Teller extends Thread {
  Account a;
  public Teller(Account a) {
    this.a = a;
  }
  public void run() {
    a.incrBalance(100.00);
 }
}
```

```
public class Demo {
 public static void main(String[] args) {
   Account a = new Account();
   Teller t1 = new Teller(a);
   Teller t2 = new Teller(a);
   t1.start();
   t2.start();
   System.out.println(a.getBalance());
}
```

Thread synchronization

- The output of the previous Demo should be 200.00
  - ✦But depending on timing of t1 and t2, it could be 100.00
- Note that (in assembly/byte code) incrBalance is really 2 steps:

```
public void incrBalance(float value){
  float f = getbalance;     //get
  setbalance (f + value); } //set
}
```

- So either one of these sequences of events is possible:

```
t1 gets a balance of 0
t1 sets the balance to 100
t2 gets a balance of 100
t2 sets the balance to 200
```

```
t1 gets a balance of 0
t2 gets a balance of 0
t1 sets the balance to 100
t2 sets the balance to 100
```


Thread synchronization

- If a certain method should **not** be called from two threads at the same time, you can use the keyword “synchronized”.
  - ✦If a thread is inside one of the synchronized methods, all other threads are blocked from entering any of the synchronized methods of the class until the first thread returns from its call

```
public class Account{
  private float balance = 0;
  public float synchronized getBalance(){ return balance; }
  public void synchronized incrBalance(float value){
    balance = balance + value;
  }
}
```