Concurrent Programming in Java

Yuh-Jzer Joung
Dept. of Information Management
National Taiwan University
June, 2005

References

• The Java Tutorial, Third Edition.
  You can download the book materials from
  http://java.sun.com/docs/books/tutorial/index.html

Java execution environment

source code

Libraries

Compiler

JVM

Computer

byte code

portability

security

Java Environment

Compile-time environment

Run-time environment (Java Platform)

Java Class Libraries

Java Class Libraries

Bytecodes (.class)

Bytecodes (java)

Class Loader

Bytecode Verifier

Java Interpreter

Just-in-time Compiler

Runtime system

Operating system

Hardware

Hello World

public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}

Output:

Sum = 55

Language Basics

public class BasicsDemo {
    public static void main(String[] args) {
        int sum = 0;
        for (int current = 1; current <= 10; current++) {
            sum += current;
        }
        System.out.println("Sum = " + sum);
    }
}
THE LIFE CYCLE OF AN OBJECT

A typical Java program creates many objects, which interact with one another by sending each other messages. Through these object interactions, a Java program can implement a GUI, run an animation, or send and receive information over a network. Once an object has completed the work for which it was created, it is garbage-collected and its resources are recycled for use by other objects.

Here’s a small program, called CreateObjectDemo, that creates three objects: one Point object and two Rectangle objects. You will need all three source files to compile this program.

Processes and Threads

A process is an executing program. It has been allocated memory by the operating system. A thread is an execution or flow of control in the address space of a process; the program counter register points to the next instruction to be executed. A process is a program with one thread. A process can have more than one thread. All the threads in a process have their own program counter and their own stack for local (also called automatic) variables and return addresses of invoked procedures.

Threads

- A thread is a single sequential flow of control within a program. A thread is also called a lightweight process. A thread is similar to a real process in that a thread and a running program are both a single sequential flow of control. However, a thread is considered lightweight because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program’s environment.

Threads (cont.)

- Shared memory multiprocessors are cheaper and more common so each thread can be allocated a CPU;
- It is less expensive and more efficient to create several threads in one process that share data than to create several processes that share data;
- IO on slow devices like networks, terminals, and disks can be done in one thread while another thread does useful computation in parallel;
- Multiple threads can handle the events (e.g., mouse clicks) in multiple windows in the windowing system on a workstation;
- In a LAN cluster of workstations or in a distributed operating system environment, a server running on one machine can spawn a thread to handle an incoming request in parallel with the main thread continuing to accept additional incoming requests.

Threads in Java

In Java, a thread in the run-time interpreter calls the main() method of the class on the java command line. Each object created can have one or more threads, all sharing access to the data fields of the object.

A Simple Thread Example

```
class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {
            }
        }
        System.out.println("DONE! " + getName());
    }
}
```
CREATING THREADS

When a thread is created and ready to run, you invoke its `start` method. The `start` method spawns a new thread of control based on the data in the `Thread` object and then returns. Then the virtual machine invokes the new thread's `run` method, making the thread active.

When a thread's `run` method returns, the thread has exited.

The standard implementation of `Thread.run` does nothing. You either need to extend `Thread` to provide a new run method, or create a `Runnable` object and pass it to the thread's constructor.

CREATING THREADS (cont.)

In Java, there are two ways to create threaded programs:

That is, you can choose one of two ways to provide a customized `run()` method for a Java thread:

- Subclass the `Thread` class defined in the java.lang package and override the `run()` method.
- Provide a class that implements the `Runnable` interface, also defined in the java.lang package. Now, when you instantiate a thread (either directly from the `Thread` class, or from a subclass of `Thread`), give the new thread a handle to an instance of your `Runnable` class. This `Runnable` object provides the `run()` method to the thread.

Extend the Thread class:

```java
class MyThread extends Thread {
    public void run() {
        //...}
}
```

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

Implement the interface Runnable:

```java
class MyThread implements Runnable {
    public void run() {
        //...}
}
```

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

USING Runnable

```java
class RunPingPong implements Runnable {
    String word;
    int delay;
    RunPingPong (String whatToSay, int delayTime) {
        word = whatToday;
        delay = delayTime;
    }
    public void run() {
        try { for (;;) {
            System.out.print(word + " ");
            Thread.sleep(delay);
        } catch (InterruptedException e) {return;}
    }
    public static void main(String[] args) {
        Runnable ping = new RunPingPong("ping", 33);
        Runnable pong = new RunPingPong("Pong",100);
        new Thread(ping).start();
        new Thread(pong).start();
```
An applet is a special kind of Java program

- Runs when embedded in:
  - Java-enabled web browser
  - Specialized tool such as appletviewer
- Applet code can therefore be
- Sent via the network from a Web Server to a Browser, which
  upon rendering the HTML web page, creates the applet
  object and executes the Java code.

The application (Browser) runs the Java code in the applet
object by invoking the “life cycle” methods – pre-defined
for applets running within an application.

---

Applet operation

<table>
<thead>
<tr>
<th>Security restrictions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Byte-code verification</td>
</tr>
<tr>
<td>- Limited system access</td>
</tr>
<tr>
<td>- No access to local file system</td>
</tr>
<tr>
<td>- Limited network operations</td>
</tr>
<tr>
<td>- ...</td>
</tr>
</tbody>
</table>

---

Applet Life Cycle

Applets do not invoke a main() method as Java
applications do.

Instead, applets inherit four special methods associated
with an applet’s life cycle:

- init()
- start()
- stop()
- destroy()

These methods are invoked by the browser and are
not typically called explicitly.

---

Running Applets

Create an applet application
Compile the applet with javac translator
Create an html file hosting the applet

```html
<applet code=“appletname.class” width=w height=h>
</applet>
```

---

“Hello World”, again

```java
import java.applet.Applet;
import java.awt.Graphics;

public class FirstApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello, World!", 20, 50);
    }
}
```

where to draw it
EXAMPLES:
The Clock Applet Source Code

The Producer and Consumer Problem
The Producer generates an integer between 0 and 9 (inclusive), stores it in a "CubbyHole" object, and prints the generated number.
The Consumer consumes all integers from the CubbyHole (the exact same object into which the Producer put the integers in the first place) as quickly as they become available.
To prevent race conditions, the storage of a new integer into the CubbyHole by the Producer must be synchronized with the retrieval of an integer from the CubbyHole by the Consumer.
The Java language and runtime system support thread synchronization through the use of monitors

The Producer and Consumer Problem
Main Program
Producer
Consumer
CubbyHole

Synchronization
Conflicts may arise when more than one thread are inside an object!

class Buffer
{
    private int [100] buf;
    private int index=0;
    public void processing (int item) {
        buf[index] = item;
        index++;
    }
}

MONITORS
In general, a monitor is associated with a specific data item (a condition variable) and functions as a lock on that data. When a thread holds the monitor for some data item, other threads are locked out and cannot inspect or modify the data.
The code segment within a program that must be accessed by concurrent threads in an exclusive manner is called a critical section. In Java, you mark critical sections in your program with the synchronized keyword.

Declare synchronized methods
Class someObject {
    synchronized void f() { /* ... */ }
    synchronized void g(){ /* ... */}
    ...
}
If f() is called for an object, g() cannot be called for the same object until f() is completed
Conceptually, each object has exactly one exclusive lock, and any synchronized method must have that lock to start.
Critical section

Prevent multiple thread access to part of the code inside a method instead of the entire method.

```java
synchronized (any_sync_Object) {
    // This code can be accessed
    // by only one thread at a time
}
```

Java Monitors Are Re-entrant

Java monitors are re-entrant so as to prevent the possibility of a single thread deadlocking itself on a monitor that it already holds.

```java
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()");
    }
}
```

Output:

```
here I am, in b()
here I am, in a()
```

Inter-Thread Communication

Often one thread has to wait for another thread to do something before it can perform its task.

The threads need to communicate between each other when they have completed their task.

The notify() and wait() methods

notify() - Notifies a thread in a wait state that the condition it is waiting for has been achieved.
notifyAll() - Performs similarly to the notify() method except that all the waiting threads are awakened.
wait() - Causes the invoking thread to wait for a condition to occur.

The Object class contains two other versions of 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 nanoseconds have elapsed.

Note: The notify() and wait() methods can be invoked only from within a synchronized method or within a synchronized block or statement.

The Dining Philosophers Problem

The applet code
Abstract classes

You can create an abstract class when you want to manipulate a set of classes through this common interface.

A class containing abstract methods is called an abstract class. If a class contains one or more abstract methods, the class must be qualified as abstract.

If you inherit from an abstract class and you want to make objects of the new type, you must provide method definitions for all the abstract methods in the base class.

Example: abstractAreas.java

```
abstract class Shape {
    double d1,d2;
    Shape(double a, double b) {
        d1=a;
        d2=b;
    }
    abstract double area();
}

class triangle1 extends Shape {
    triangle1(double a,double b) {
        super(a,b);
    }
    double area() {
        System.out.println("area for triangle:");
        return d1*d2/2;
    }
}

abstract class Shape {
    double d1,d2;
    Shape(double a, double b) {
        d1=a;
        d2=b;
    }
    abstract double area();
}

class triangle1 extends Shape {
    triangle1(double a,double b) {
        super(a,b);
    }
    double area() {
        System.out.println("area for triangle:");
        return d1*d2/2;
    }
}
```

Example: abstractAreas.java (cont.)

```
public class abstractAreas {
    public static void main(String[] args){
        rectangle1 R=new rectangle1(8,6);
        triangle1 T=new triangle1(10,12);
        Shape shapref;
        shapref=R;
        System.out.println("Area is"+shapref.area());
        shapref=T;
        System.out.println("Area is"+shapref.area());
    }
}
```

Interface

The interface keyword produces a completely abstract class, one that provides no implementation at all. The interface is more than just an abstract class taken to the extreme, since it allows you to perform a variation on C++’s “multiple inheritance,” by creating a class that can be upcast to more than one base type.

Example: abstractAreas.java (cont.)

```
interface Interface {
    double area();
}
```

public class abstractAreas {
    public static void main(String[] args){
        Interface intf = new Interface() {
            @Override
            public double area() {
                return 5; // Example area calculation
            }
        };
        System.out.println("Area is"+intf.area());
    }
}
```

Interface (cont.)

An interface says: “This is what all classes that implement this particular interface will look like.”
To create an interface, use the interface keyword instead of the class keyword.
To make a class that conforms to a particular interface (or group of interfaces) use the implements keyword.
Multiple Inheritance

In C++, this act of combining multiple class interfaces is called multiple inheritance, and it carries some rather sticky baggage because each class can have an implementation.

In Java, you can perform the same act, but only one of the classes can have an implementation, so the problems seen in C++ do not occur with Java when combining multiple interfaces.

If A has a method x, and B overwrites it, then which does D.x refer to? C.x(A.x) or B.x?

In C++, such ambiguity must be explicitly resolved in the program (by qualifying a name with its class).

Multiple inheritance in Java

[pUBLIC] INTERFACE interfaceName [EXTENDS interface1, interface2,...]

PUBLIC INTERFACE drawing EXTENDS drawable, transformable{ }

PUBLIC INTERFACE CanFight {
    void fight();
}
PUBLIC INTERFACE CanSwim {
    void swim();
}
PUBLIC INTERFACE CanFly {
    void fly();
}
PUBLIC CLASS ActionCharacter {
    PUBLIC void fight();
}
PUBLIC CLASS Hero extends ActionCharacter implements CanFight, CanSwim, CanFly {
    PUBLIC void swim();
    PUBLIC void fly();
}

Example

public class Adventure {
    static void t(CanFight x) { x.fight(); }
    static void u(CanSwim x) { x.swim(); }
    static void v(CanFly x) { x.fly(); }
    static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Treat it as a CanFight
        u(h); // Treat it as a CanSwim
        v(h); // Treat it as a CanFly
        w(h); // Treat it as an ActionCharacter
    }
}