Threads in Java

Alexandre Bergel
abergel@dcc.uchile.cl
05/05/2011

Roadmap

1. What are threads?
2. Example
3. Multiple execution
4. Scheduling
5. Synchronization
6. Closing words

Roadmap

1. **What are threads?**
2. Example
3. Multiple execution
4. Scheduling
5. Synchronization
6. Closing words

More than one thing

Most of the programs we have seen use *a single thread* for their execution

This may cause problem when multiple events or actions need to occur *at the same time*

- Drawing a graphical tictactoe while handling user's clicks
- Serving a HTTP request while waiting for new request

The solution to these problems is the *seamless execution* of two or more sections of a program, at the same time

Threading Introduction

Threads: expressing logical parallelism in a program

- thread = logical sequence of control, a program's path of execution
- independent logical sequences of control
- generally share one memory

Threads give the illusion to do some work in parallel

What are threads?

Threads are a control mechanism offered by both a library and the programming language

Used to express concurrency and parallelism in a program

The following operations are necessary:

- create: increase parallelism
- synchronize: coordinate
- destroy: decrease parallelism

Multiple execution

Multithreading means *multiple execution lines* for a single program *at the same time*

However, it is *not the same as starting a program twice*

In this case, the operating system is treating the programs as two separate and distinct process

The idea of *sharing data* is very beneficial

but brings up some areas of concern

What are threads in Java?

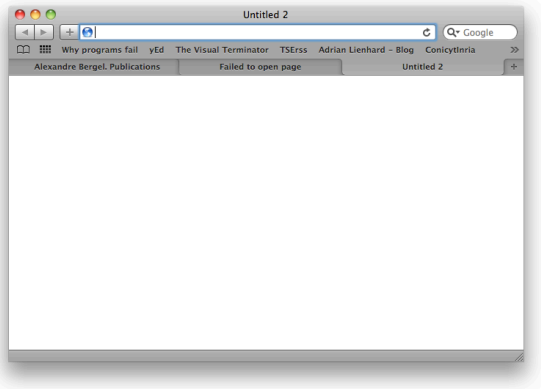
Threads are exposed as a special kind of object

- Operations are methods on thread objects
- Each thread object is a unit of parallelism
- A thread can be executed independently therefore

Roadmap

1. What are threads?
2. **Example**
3. Multiple execution
4. Scheduling
5. Synchronization
6. Closing words

Example: Handing web requests

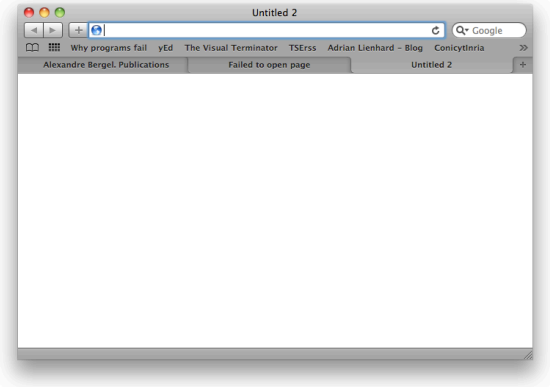


Web server

Thread 1

Example: Handing web requests

Example: Handing web requests



http://localhost:8000/index.html

Web server

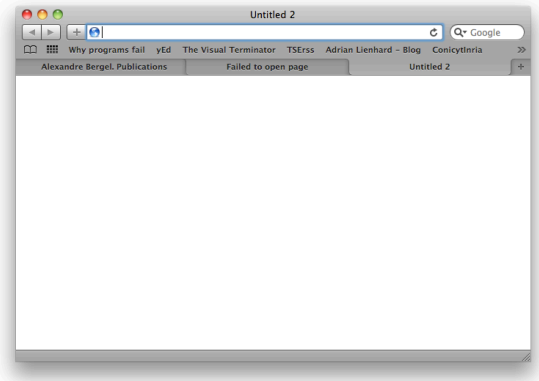
Thread 1

creates

Request

Thread 2

Example: Handing web requests



Web server

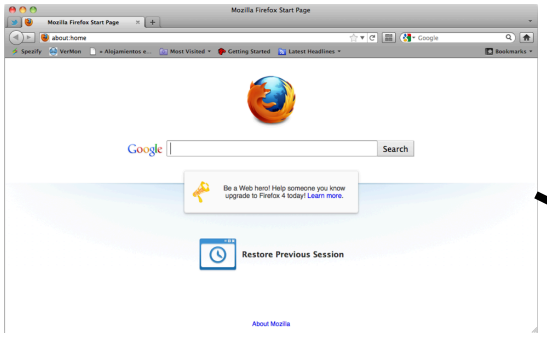
Thread 1

http://localhost:8000/index.html

Request

Thread 2

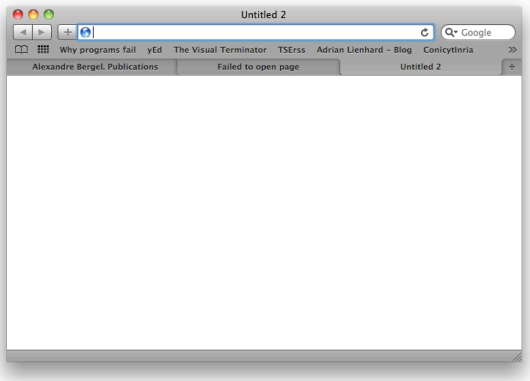
Example: Handing web requests



http://localhost:8000/index.html

Web server

Thread 1



http://localhost:8000/index.html

Request

Thread 2

The SimpleWebServer class

```
public class SimpleWebServer extends Thread {

    public SimpleWebServer(File rootDir, int port) throws IOException {

        // ...

        start();
    }
     /* Starting point of the application */
    public static void main(String[] args) {
        try {
            new SimpleWebServer(new File("~/"), 8000);
        }
        catch (IOException e) {
            System.out.println(e);
        }
    }
    ...
}
```

The SimpleWebServer class

```
public class SimpleWebServer extends Thread {

    private boolean _running = true;

    public void shutdown() { // Used in the tests

        this._running = false;
    }
    ...
}
```

The SimpleWebServer class

```
public class SimpleWebServer extends Thread {

    private boolean _running = true;

    public void run() {

        while (_running) {
            try {
                Socket socket = _serverSocket.accept();
                RequestThread requestThread =
                    new RequestThread(socket, _rootDir);
                requestThread.start();
            }
            catch (IOException e) {
                System.exit(1);
            }
        }
    }
   ...
}
```

The RequestThread class

```
public class RequestThread extends Thread {

    public void run() {

    ...
    }
}
```

Roadmap

1. What are threads?
2. Example
3. **Multiple execution**
4. Scheduling
5. Synchronization
6. Closing words

Two ways to create a thread

Java's creators have graciously designed two ways of creating threads

- Implementing the interface Runnable
- Extending the class Thread

Subclassing Thread

```
public class RequestThread extends Thread
{
        public void run()
        {
        ....
        }
}
```

The Thread class

defined as a class in the core Java language

implements an interface called Runnable

define a single abstract method called run()

```
public interface Runnable {
   public void run();
}
public class Thread implements Runnable { ... }
```

java.lang.Thread

There are a number of methods defined on the Thread class

To query the thread to find its *priority*

To put it to *sleep* (note that sleep() is a static method)

Cause it to *yield* to another thread

stop

suspend its execution

resume its execution, etc, ...

Implementing Runnable

```
public class RequestThread implements Runnable
{
        Thread t;
        public RequestThread () {
              t = new Thread(this);

        }
        public void run()
        {
        ....
        }
}
```

Using the Runnable Interface

A class must implement the Runnable interface

- provide an implementation of the run method
- initiates the computation in the thread

```
public class Counter implements Runnable {
    public void run () {
        /* code here executed concurrently with callers */ }
    ...
}
```

Creating a Thread

Steps

- Create an object of type Runnable & bind it to a new Thread object
- Or create an instance of a subclass of Thread
- Start it

The Thread.start() method

- creates the thread stack for the thread
- then invokes the run() method of the Runnable object in the new thread

Threads operations

construction

usually done by passing a runnable object to the thread on construction

starting

Invoking a thread's start() method cases the run() method of the runnable object to run

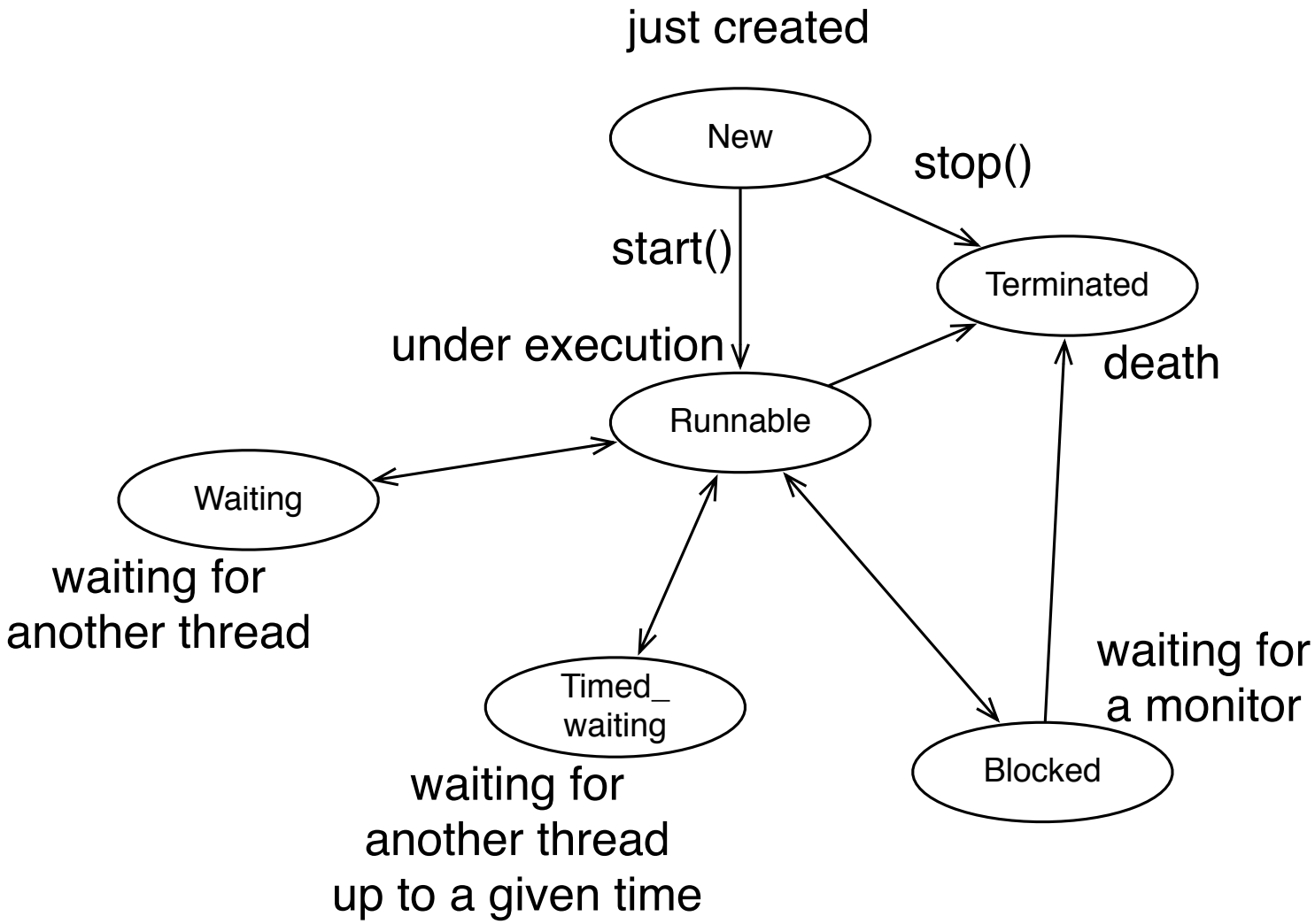
priority

Threads can be run at different priority levels

control

this refers to control methods provided by the Thread class

Thread life cycle



http://java.sun.com/javase/6/docs/api/java/lang/Thread.State.html

Issues with threads

Sharing and Synchronization

Threads may share access to objects (object state, open files, and other resources) associated with a single process

Scheduling

if # of threads != # of processes, scheduling of threads is an issue

Operations in different threads may occur in variety of orders

Roadmap

1. What are threads?
2. Example
3. Multiple execution
4. **Scheduling**
5. Synchronization
6. Closing words

Example: a simple counter

```
public class SmallExample implements Runnable {
    private String info;
    public SmallExample(String info) { this.info = info; }

    public void run () {
        for(int i = 1; i < 10; i++) {
            System.out.println(info + " " + i);
        }
    }

    public static void main(String[] argv) {
        new Thread(new SmallExample("thread1")).start();
        new Thread(new SmallExample("thread2")).start();
        new Thread(new SmallExample("thread3")).start();
    }
}
```

Example

```
thread1 1
thread1 2
thread1 3
thread1 4
thread1 5
thread1 6
thread1 7
thread1 8
thread1 9
thread2 1
thread2 2
thread2 3
thread2 4
thread2 5
thread2 6
thread3 1
thread3 2
thread3 3
thread3 4
thread3 5
thread3 6
thread3 7
thread3 8
thread3 9
thread2 7
thread2 8
thread2 9
```

Buh?
No parallelism?
What happened?

Example

```
thread1 1
thread1 2
thread1 3
thread1 4
thread1 5
thread1 6
thread1 7
thread1 8
thread1 9
thread2 1
thread2 2
thread2 3
thread2 4
thread2 5
thread2 6
thread3 1
thread3 2
thread3 3
thread3 4
thread3 5
thread3 6
thread3 7
thread3 8
thread3 9
thread2 7
thread2 8
thread2 9
```

Buh?
No parallelism?
What happened?

Each thread did not wait for the others

Letting other thead execute

```
public class SmallExample implements Runnable {
    private String info;
    public SmallExample(String info) { this.info = info; }

    public void run () {
        for(int i = 1; i < 10; i++) {
            System.out.println(info + " " + i);
            Thread.yield();
        }
    }

    public static void main(String[] argv) {
        new Thread(new SmallExample("thread1")).start();
        new Thread(new SmallExample("thread2")).start();
        new Thread(new SmallExample("thread3")).start();
    }
}
```

Letting other thead execute

```
public class SmallExample implements Runnable {
    private String info;
    public SmallExample(String info) { this.info = in
    public void run () {
        for(int i = 1; i < 10; i++) {
            System.out.println(info + " " + i);
            Thread.yield();
        }
    }
    public static void main(String[] argv) {
        new Thread(new SmallExample("thread1")).start();
        new Thread(new SmallExample("thread2")).start();
        new Thread(new SmallExample("thread3")).start();
    }
}
```

Causes the currently executing thread object to temporarily pause and allow other threads to execute.

Slow down!

```
public class SmallExample implements Runnable {
    private String info;
    public SmallExample(String info) { this.info = info; }

    public void run () {
        for(int i = 1; i < 10; i++) {
            System.out.println(info + " " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    ...
```

Slow down!

```
public class SmallExample implements Runn
    private String info;
    public SmallExample(String info) { this.info

    public void run () {
        for(int i = 1; i < 10; i++) {
            System.out.println(info + " " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException
                e.printStackTrace();
            }
        }
    }
    ...
```

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds

Thread interruption

Threads may interrupt themselves

Thread.interrupt()

When this happens, an exception `InterruptedException` is raised

Making a thread live

```
public class Counter implements Runnable {
    private int value = 0;
    private String info;
    public void run () {
        while (true) {
            System.out.println(info + " " + value);
            value ++;
            waitSecond(1);
        }
    }
    ...
    public static void main(String[] argv) {
        Counter counter = new Counter("Counter");
        Thread thread = new Thread(counter);
        thread.start();
        counter.waitSecond(3);
        thread.stop();
    }
}
```

Making a thread live

```
public class Counter implements Runnable {
    private int value = 0;
    private String info;
    public void run () {
        while (true) {
            System.out.println(info + " " + value);
            value ++;
            waitSecond(1);
        }
    }
    ...
    public static void main(String[] argv) {
        Counter counter = new Counter("Counter");
        Thread thread = new Thread(counter);
        thread.start();
        counter.waitSecond(3);
        thread.stop();
    }
}
```

Deprecated method!

Making a thread live

```
public class Counter implements Runnable {
    private int value = 0;
    private String info;
    public void run () {
        while (true) {
            System.out.println(info + " " + value);
            value ++;
            waitSecond(1);
        }
    }
    ...
    public static void main(String[] argv) {
        Counter counter = new Counter("Counter");
        Thread thread = new Thread(counter);
        thread.start();
        counter.waitSecond(3);
        thread.stop();
    }
}
```

"This method is inherently unsafe."

How to make a counter stop then?

```
public class Counter implements Runnable {
    private int value = 0;
    private String info;
    public void run () {
        while (true) {
            System.out.println(info + " " + value);
            value ++;
            waitSecond(1);
        }
    }
...
    public static void main(String[] argv) {
        Counter counter = new Counter("Counter");
        Thread thread = new Thread(counter);
        thread.start();
        counter.waitSecond(3);
        thread.stop();
    }
}
```

"This method is inherently unsafe."

How to make a counter stop then?

```
public class Counter implements Runnable {
    private boolean shouldRun;
    private String info;
    public Counter(String info) { this.info = info; shouldRun = true; }
     public void run () {
          while (shouldRun) {
               System.out.println(info + " " + value);
               value ++;
               waitSecond(1);
          }
    }
    ...
    private void stopRunning() { shouldRun = false; }
    public static void main(String[] argv) {
          Counter counter = new Counter("Counter");
          Thread thread = new Thread(counter);
          thread.start();
          counter.waitSecond(3);
          counter.stopRunning();
    }}
```

Thread Scheduler

Java has a *scheduler* that *monitors* all running threads

The Scheduler *decides* which threads should be running and which are in line to be executed

Two important characteristics

- Thread daemon
- Thread priority

Thread Daemon

According to Webster's, a daemon (variant of demon) is an attendant power or spirit

In Java, any thread can be a Daemon thread

```
Thread.setDaemon(true)
```

The difference between *threads* and *daemon threads* is that the JVM will only *shut down* a program when all *user threads have terminated*

Daemon threads are terminated by the JVM when there are no longer any user threads running, including the *main thread of execution*

Thread priority

Each thread *runs* a given *priority*

The runtime chooses the runnable thread with the *highest priority* for execution

A thread *gets* the *Runnable* state according to their *priority*

When a Java thread is created, it *inherits* its priority from the thread that created it

Preemptive scheduling

In Java, *preemptive scheduling* algorithm is applied

Always the thread of the *highest priority* is chosen

If two threads of the same priority are waiting to be executed by the CPU, then the *round-robin* algorithm is applied

Roadmap

1. What are threads?
2. Example
3. Multiple execution
4. Scheduling
5. **Synchronization**
6. Closing words

Synchronization

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Synchronization

If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:

First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized

Synchronization

If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:

> First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executi threads that inv ock (suspen object.
>
> Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

http://java.sun.com/docs/books/tutorial/essential/concurrency/syncmeth.html

Note that constructors cannot be synchronized

Roadmap

1. What are threads?
2. Example
3. Multiple execution
4. Scheduling
5. Synchronization
6. **Closing words**

Conclusion

Java threads are the basis for expression of parallelism

- convenient, nice encapsulation, cleanly integrated
- can build flexible expression and management

Do not overuse Threads

It may leads to complex and hard-to-debug situations

What you should know

What are threads?

What threads are often necessary?

How to define a thread?

When you need to employ threads?

Understand what are the synchronization problems in threading

What is a scheduler?

Can you answer to these questions?

Why each web request must be handled in a separate thread?

Can you provide an example of synchronization problem?

Why stop() is deprecated?