Threading in Java

Alexandre Bergel
abergel@dcc.uchile.cl
31/05/2010

Roadmap

What are threads?

Example

Multiple execution

Scheduling

Threading Introduction

Threads: expressing logical parallelism in a program

- thread = logical sequence of control
- independent threads = 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:

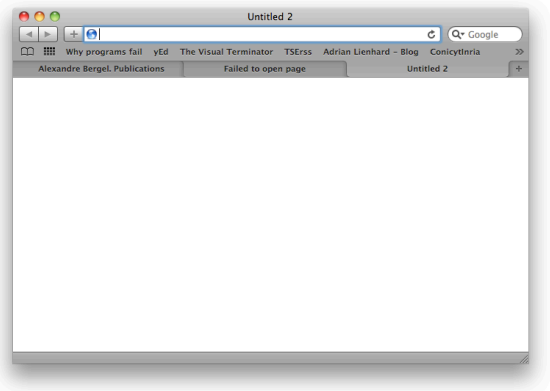
- create -- increase parallelism
- synchronize -- coordinate
- destroy -- decrease parallelism

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

Example: Handing web requests

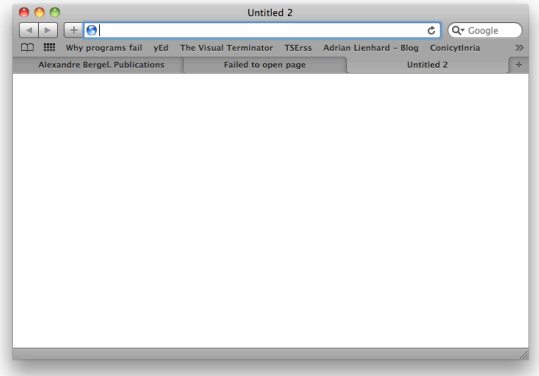


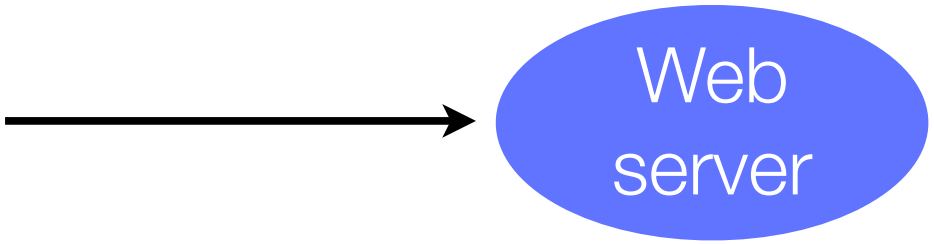
Web server

Thread 1

Example: Handing web requests

http://localhost:8000/index.html

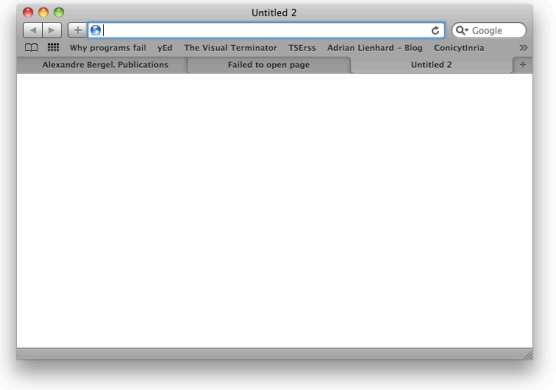




Thread 1

Example: Handing web requests

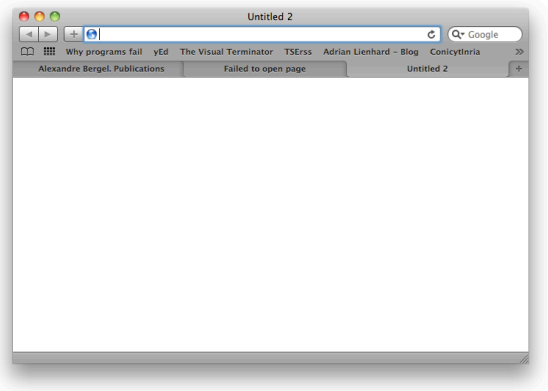
http://localhost:8000/index.html



Thread 1

Thread n

Example: Handing web requests



Web server

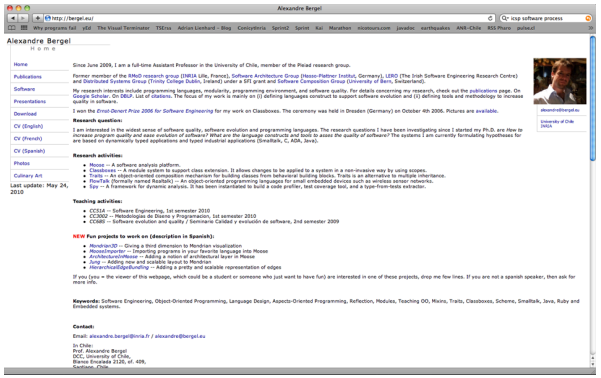
Thread 1

http://localhost:8000/index.html

Request

Thread n

Example: Handing web requests



Web server

Thread 1

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

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

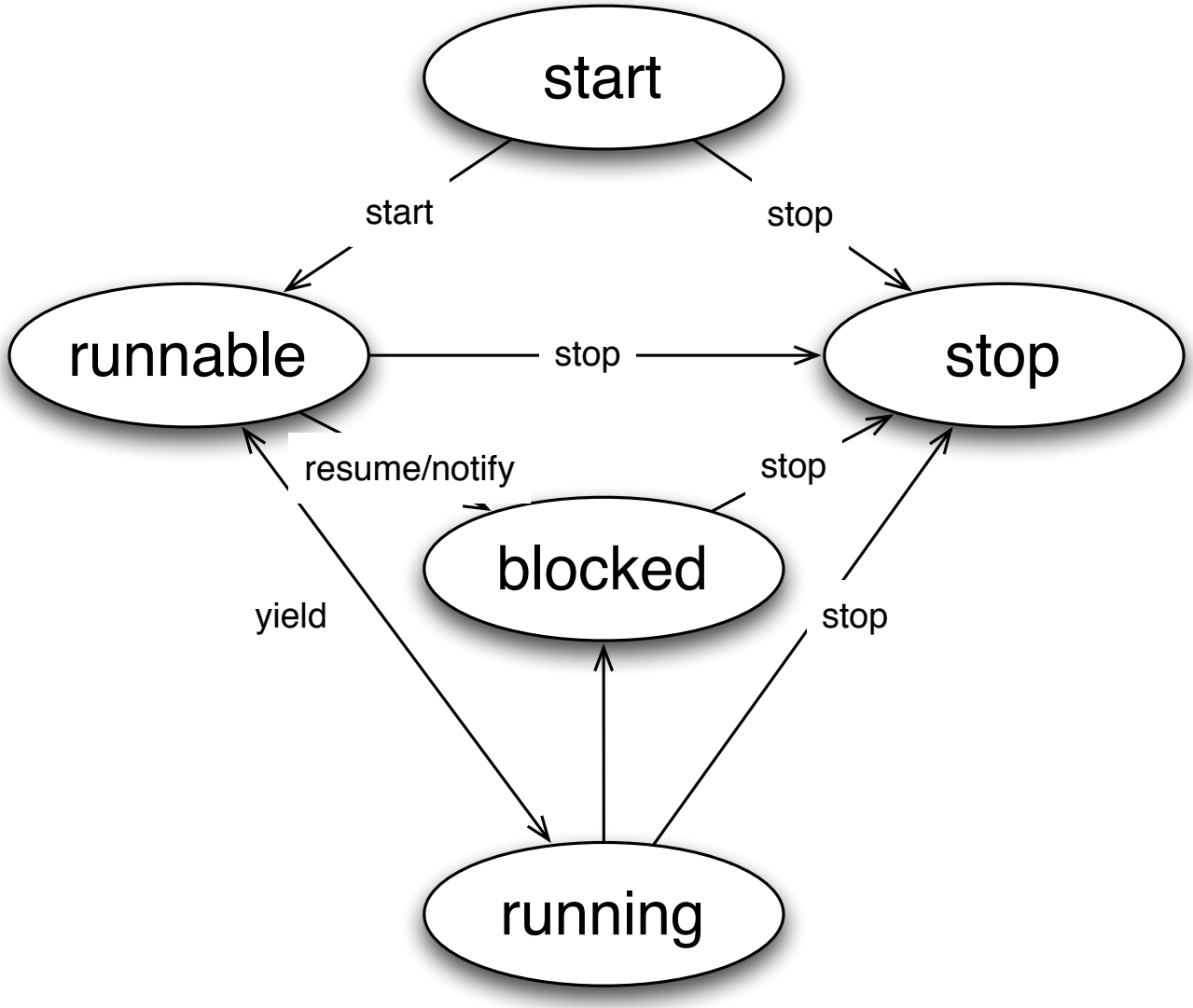
priority

Threads can be run at different priority levels

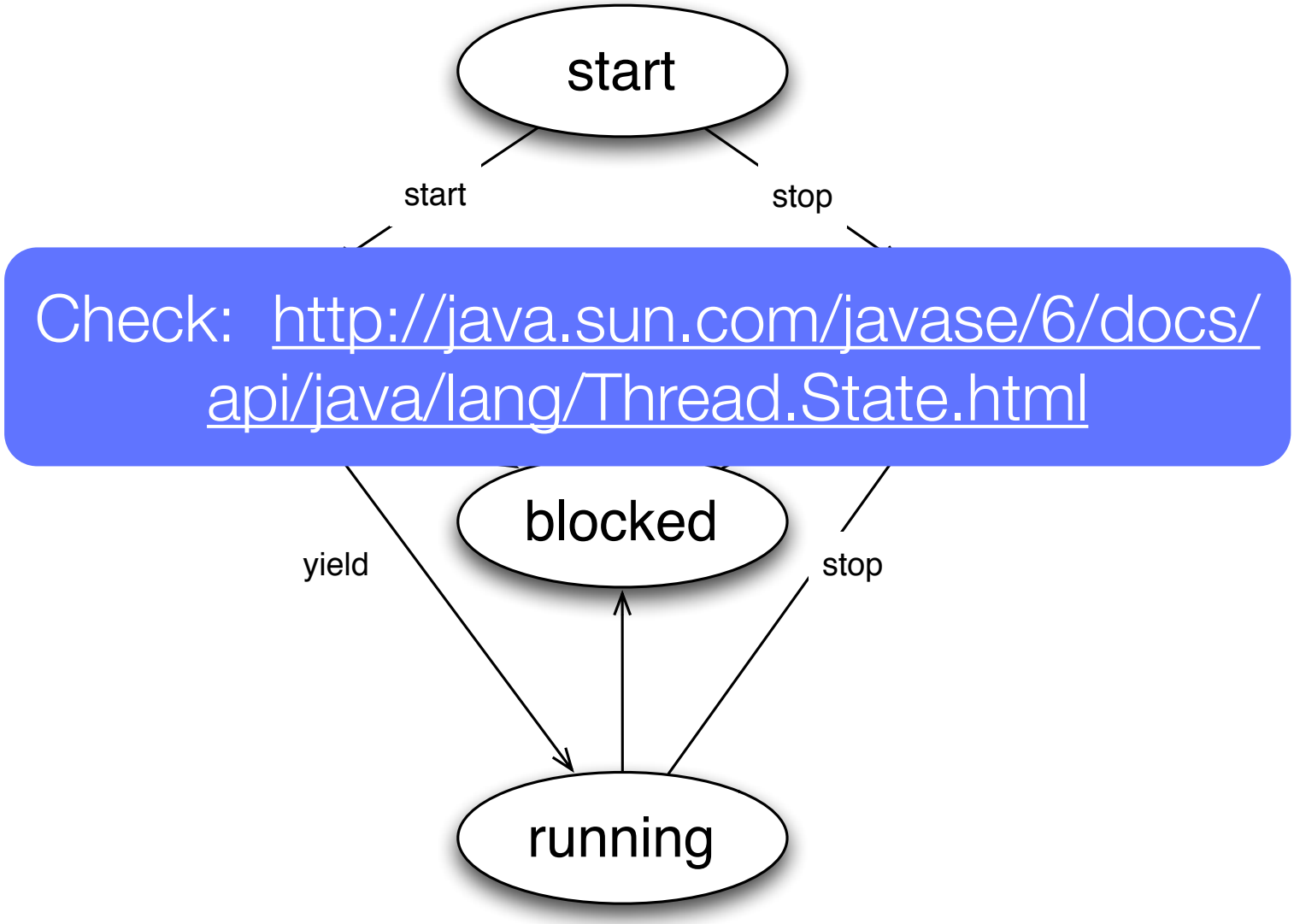
control

this refers to control methods provided by the Thread class

Thread life cycle



Thread life cycle



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 { ... }
```

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

Creating a Thread

Steps

Creating an object of type Runnable

Bind it to a new Thread object

Start it

Start

creates the thread stack for the thread

then invokes the run() method of the Runnable object in the new thread

Example

```
ConcurrentReader readerThread = new ConcurrentReader();

Thread t = new Thread (readerThread);

t.start(); // start the thread and call run()
```

java.lang.Thread class has a constructor that takes an object of type Runnable

You may also subclass Thread

What do you think about that?

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

Cause it to yield to another thread

stop

syspend its execution

resume its execution, etc, ...

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

A bit more refined example

```
public class Counter implements Runnable {
    private int value = 0;
     ...
    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();
    }
}
```

A bit more refined example

```
public class Counter implements Runnable {
    private int value = 0;
    ...
    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!

A bit more refined example

```
public class Counter implements Runnable {
    private int value = 0;
    ...
    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;
    ...
    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;
    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();
    }}
```

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

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

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?