CS342: Software Design

September 12, 2017
Agenda

GitHub introduction

Timer

Thread

- Why threading?
- Runnable Interface, thread lifecycle
- sleep and interrupt
- Synchronize threads: synchronized methods
- Code example: wait/notify
- Code example: round robin, time slicing, priority
- sleep vs. wait vs. yield
- Thread pooling and concurrency utilities
GitHub - web-based Git or version control repository
## Track change history

### Commits on Nov 19, 2015

<table>
<thead>
<tr>
<th>CommitMessage</th>
<th>Author</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>starting to upgrade jasmine version and fix broken tests</td>
<td>derickbailey</td>
<td>Nov 19, 2015</td>
</tr>
</tbody>
</table>

### Commits on Mar 9, 2015

<table>
<thead>
<tr>
<th>CommitMessage</th>
<th>Author</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>ensuring steps run one at a time, sequentially</td>
<td>derickbailey</td>
<td>Mar 9, 2015</td>
</tr>
</tbody>
</table>

### Commits on Jan 1, 2015

<table>
<thead>
<tr>
<th>CommitMessage</th>
<th>Author</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>renamed project to migroose, to avoid naming conflict</td>
<td>derickbailey</td>
<td>Jan 1, 2015</td>
</tr>
</tbody>
</table>

### Commits on Dec 31, 2014

<table>
<thead>
<tr>
<th>CommitMessage</th>
<th>Author</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>rebuild git repo, due to git failure</td>
<td>derickbailey</td>
<td>Dec 31, 2014</td>
</tr>
</tbody>
</table>
Who wrote this line of code?

```javascript
watch: {
  specs: {
    files: ["migroose/**/*.*", "specs/**/*.*"],
    tasks: ["specs"]
  }
}
```
But it’s much more than that...

Branch, merge, flow control

Knowledge sharing

Issue tracking

Analytics - contributors, pulse, activities...

Project management: ZenHub extension, filters

Your name card as a developer
Timer class

Scheduler to set up something to run at designated time

Each object is a background thread watching clock

When scheduled time comes, it triggers the task

```java
import java.util.*;
public class Y2K {
    public static void main(String[] args) {
        Timer timer = new Timer();
        TimerTask task = new TimerTask() {
            public void run() {
                System.out.println("Y2K!" abandonment);
            }
        };
        Calendar cal = new GregorianCalendar(2000, Calendar.JANUARY, 1);
        timer.schedule(task, cal.getTime());
    }
}
```
Schedule Task at certain frequency

```java
import java.util.*;
import java.util.Date.*;
import java.text.SimpleDateFormat;
public class Clock {
    public static void main(String[] args) {
        Timer timer = new Timer();
        TimerTask task = new TimerTask() {
            public void run() {
                SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
                Date date = new Date();
                System.out.println(dateFormat.format(date));
            }
        };
        timer.scheduleAtFixedRate(task, 0, 1000);
    }
}
```
Why Thread?

Event driven programming means things happen simultaneously

Better usage of computing resource

Commonly used in GUI - event dispatching thread (EDT) - AWT

Not so much explicitly in backend, however:

- By nature, backend (server) supports multiple clients at the same time
- "Thread-safe"
- Enterprise email dispatching example
Thread could be messy, so be careful

Multithreaded programming

Theory

Actual

Thread vs. Process
Runnable interface: create “threading” object

Thread is a flow of control within a program

Thread vs. process

All execution in Java is associated with a Thread object. Main thread

Thread begins by run() method of implementation of Runnable

```java
public interface Runnable {
    abstract public void run();
}
```
Create a thread by implementing Runnable

Thread myThread = new Thread(targetObject);
myThread.start();

```java
class HelloComponent4 extends JComponent
    implements MouseMotionListener, ActionListener, Runnable
{
    public HelloComponent4( String message ) {
        Thread t = new Thread( this );
        t.start();
    }
    
    public void run() {
        try {
            while(true) {
                blinkState = !blinkState; // Toggle blinkState.
                repaint(); // Show the change.
                Thread.sleep(300);
            }
        } catch (InterruptedException ie) { } 
    }
}
```
How does this work?

class Animation implements Runnable

```java
myThread = new Thread()
myThread.start();
run()
```

Thread myThread

create

start()
Create a thread by inherit Thread class

This is so convenient, right?

```java
class DrawPicture extends Thread {
    boolean keepGoing = true;
    public void run() {
        while (keepGoing) {
            // do something
        }
    }
}
```
Why do we favor implementing Runnable interface?

By subclassing Thread, the new object is an instance of Thread

- It getting heavy and dirty
- Exposing all public properties and function

I want my class to have ability to thread, not special case of a thread

Implement vs. Inherit

Example: I want to be able to program with Java.
Thread methods and coordination

Deprecated methods

- resume, stop, suspend
- Unexpected behavior, deadlock

sleep() method: make thread to sit idle for specified time

interrupt(): wakes up a thread

join(): block the caller until thread completes.

yield(), wait(), notify(), notifyAll()
Sleep and interrupt

```java
public static void main(String[] args) {
    MyClass si = new MyClass();
    Thread t = new Thread(si);
    t.start();
    try {
        Thread.sleep(2000);
    } catch (InterruptedException x) {
    }

    System.out.println("in main() - interrupting other thread");
    t.interrupt();
    System.out.println("in main() - leaving");
}
```
Synchronized methods

“Serializing” Access to Methods - execution can’t enter two such methods at the same time

```java
synchronized private void changeColor() {
    if (++colorIndex == someColors.length)
        colorIndex = 0;
    setForeground( currentColor( ) );
    repaint( );
}

private void changeColor( ) {
    synchronized (this) {
        if (++colorIndex == someColors.length)
            colorIndex = 0;
        setForeground( currentColor( ) );
        repaint( );
    }
}
Cost of synchronized methods is high..

yield(): hey scheduler, i’m a nice guy...

wait(): wait until another thread invokes the notify() method or the notifyAll()

notify(), notifyAll(): wakes up one or multiple waiter methods

Producer and consumer example
Thread State - threadObj.getState()

NEW
RUNNABLE
BLOCKED
WAITING
TERMINATED
Thread State - threadObj.getState()

NEW
RUNNABLE
BLOCKED
WAITING
TERMINATED
Thread scheduling - round-robin

Scheduler picks up next thread when

- Sleeps, by calling Thread.sleep( ) or wait( )
- Waits for a lock, in order to run a synchronized method
- Blocks on I/O, for example, in a read( ) or accept( ) call
- Explicitly yields control, by calling yield( )
- Terminates, by completing its target method or with a stop( ) call (deprecated)
Thread scheduling - time slicing

Thread processing is chopped up so that each thread runs for a short period
How do I know?

```java
public class Thready {
    public static void main(String args[]) {
        new ShowThread("Foo").start();
        new ShowThread("Bar").start();
    }

    static class ShowThread extends Thread {
        String message;
        ShowThread(String message) {
            this.message = message;
        }
        public void run() {
            while (true)
                System.out.println(message);
        }
    }
}
```

However, you should assume round-robin!!!
Set thread priority

```java
public class ThreadPriority {
    public static void main(String args[]) {
        new ShowThread("Foo").start();
        foo.setPriority(Thread.MIN_PRIORITY);
        new ShowThread("Bar").start();
        bar.setPriority(Thread.MAX_PRIORITY);
    }
}
```

Yielding

```java
public void run() {
    while (true) {
        System.out.println(message);
        yield();
    }
}
```