CS 580 Client-Server Programming
Spring Semester, 2010
Doc 9 Threads, GUIs & Servers
25 Feb, 2010

References

Creating a GUI With JFC/Swing, http://java.sun.com/docs/books/tutorial/uiswing/

Concurrency in Swing,http://java.sun.com/docs/books/tutorial/uiswing/concurrency/index.html

Cancellable Activities, Doug Lea, October 1998, http://gee.cs.oswego.edu/dl/cpj/cancel.html

Concurrent Programming in Java: Design Principles and Patterns, Doug Lea, Addison-Wesley, 1997

Java on-line documentation http://java.sun.com/javase/6/docs/api/

Java Network Programming 2nd Ed., Harold, O'Reilly, Chapter 5

Java Performance and Scalability Vol. 1, Dov Bulka, 2000

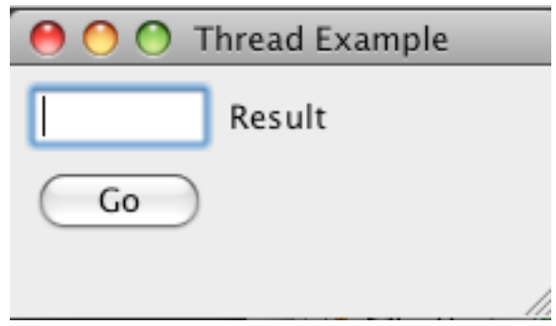
Threads & GUIs

Swing & Events

GUI programs are event driven

Actions on GUI components generate events

Events are placed in a event queue & processed



The Rule

"All Swing components and related classes, unless otherwise documented, must be accessed on the event dispatching thread."

From Java Swing documentation

How

javax.swing.SwingUtilities.invokeLater(Runnable run)

javax.swing.SwingUtilities.invokeAndWait (Runnable run)

Swing Hello World

```
import javax.swing.*;

public class HelloWorldSwing {
    private static void createAndShowGUI() {
        JFrame frame = new JFrame("HelloWorldSwing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel label = new JLabel("Hello World");
        frame.getContentPane().add(label);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            createAndShowGUI();
         }
        });
    }
}
```


Source from Creating a GUI With JFC/Swing, http://java.sun.com/docs/books/tutorial/uiswing/examples/start/HelloWorldSwingProject/src/start//HelloWorldSwing.java

Swing Thread Example

SDChat client reading a message from server

Example 1 - Blocking GUI actions

Example 2- One shot background thread

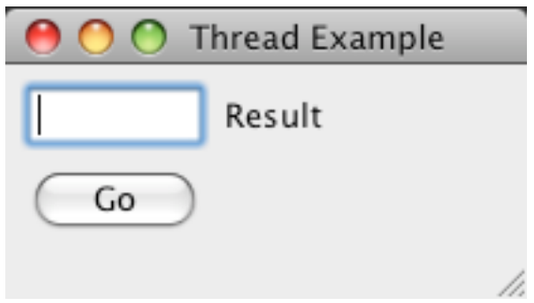
Example 3 - Repeating background thread

GUI Thread Example - Blocking GUI actions

```
public class SwingGUI extends javax.swing.JFrame{
    private javax.swing.JLabel fieldLabel;
    private javax.swing.JButton runButton;
    private javax.swing.JTextField resultField;
    private SDChatClient client = new SDChatClient();

    public SwingGUI() {
        initComponents();
    }

public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new SwingGUI().setVisible(true);
        }
    });
}
```



Creating the Window Components

```
private void initComponents() {
    resultField = new javax.swing.JTextField();
    fieldLabel = new javax.swing.JLabel();
    runButton = new javax.swing.JButton();

    setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
    setTitle("Thread Example");

    fieldLabel.setText("Result");
    runButton.setText("Go");
    runButton.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            goButtonActionPerformed(evt);
        }
    });

    lots code to lay out the window
    pack();
}
```

The Action

```
private void goButtonActionPerformed(java.awt.event.ActionEvent evt) {
   boolean result = client.nicknameAvailable("foo");
   resultField.setText(String.valueOf(result));
}
```

Main Points

client.nicknameAvailable("foo") is done on the event dispatching thread

While the client.nicknameAvailable("foo") the GUI freezes

SwingWorker<T, V> - Background Thread

abstract class to perform lengthy GUI-interacting tasks in a dedicated thread

Important Methods

T doInBackground()
    Run in separate thread
done()
    Run in event dispatching thread
    Called when doInBackground is done
process(List<v> chunks)
    Run in event dispatching thread
    Receives data from publish asynchronously
publish(V... chunks)
    Sends data to process(List)
get()
get(long timeout, TimeUnit unit)
    Returns result doInBackground
    Waits if needed

Only in Java 1.6

NicknameWorker - Background Thread

```
import javax.swing.SwingWorker;
import javax.swing.JTextField;

public class NicknameWorker extends SwingWorker<String, Object> {
        private SDChatClient client;
        private JTextField output;

        public NicknameWorker(SDChatClient aClient,JTextField aField) {
                client = aClient;
                output = aField;
}
```

NicknameWorker

```
public String doInBackground() {
            boolean result = client.nicknameAvailable("foo");
       return String.valueOf(result);
    }

public void done() {
            try {
                  String answer = get();
                  output.setText(answer);
               } catch (InterruptedException e) {/*do something here */}
               catch (java.util.concurrent.ExecutionException e) {/*do something here */ }
         }
}
```

Using NicknameWorker

```
private void goButtonActionPerformed(java.awt.event.ActionEvent evt) {
   NicknameWorker worker = new NicknameWorker(client, resultField);
   worker.execute();
}
```

Outline

worker.**execute**(); Starts the threads

doInBackground() Is run in separate thread, execute starts this

done() called when doInBackground ends,
Runs on event dispatching thread

get() Returns the value returned by doInBackground()

One Life

SwingWorker's objects can only be run once

SwingWorker Updates - Repeating thread

Use one SwingWorker to use read multiple messages from a conversation

MessageWorker

```
public class MessageWorker extends SwingWorker<Void, String> {
        private SDChatClient client;
        private JTextField output;
        public MessageWorker(SDChatClient aClient,JTextField aField) {
                client = aClient;
                output = aField;
        }

        public Void doInBackground() {
                while (! isCancelled()) {
                        String message = client.readMessage();
                        publish(message);
                }
                return null;
        }

        protected void process(List<String> messages) {
                for (String message : messages) {
                        output.setText(message);
                }
        }
}
```

Starting the Worker

```
private void goButtonActionPerformed(java.awt.event.ActionEvent evt) {
    MessageWorker worker = new MessageWorker(client, resultField);
    worker.execute();
  }
```

Outline

worker.**execute**(); Starts the threads

doInBackground() Is run in separate thread, execute starts this

publish(message) Sends message to process
Can send multiple items in one call

process(List<String> messages)
Runs on event dispatching thread

isCancelled() Returns true if the thread has been canceled

worker.cancel(true); Sends the cancel message to the worker

Server Types

Types of Servers

Connectionless(UDP) verse Connection-Oriented (TCP)

Iterative verses Concurrent

Stateless verse stateful

Iterative verses Concurrent Server

Iterative

Single process

Handles requests one at a time

Good for low volume & requests that are answered quickly

Iterative verses Concurrent Server

Concurrent

Handle multiple requests concurrently

Normally uses thread/processes

Needed for high volume & complex requests

Harder to implement than iterative

Must deal with currency

Sample Concurrent Server

```
require 'socket'
class DateServer
  def initialize(port)
    @port = port
  end

  def run()
    server = TCPServer.new( @port)
    puts("start " + @port.to_s)
    while (session = server.accept)
      Thread.new(session) do |connection|
        process_request_on(connection)
        connection.close
      end
    end
  end
```

```
  def process_request_on(socket)
    request = canonical_form( socket.gets("\n") )
    now = Time.now
    answer = case request
      when 'time'
        now.strftime("%X")
      when 'date'
        now.strftime("%x")
      else
        "Invalid request"
    end
    socket.send(answer + "\n",0)
  end

  def canonical_form(string)
    string.lstrip.rstrip.downcase
  end
end
```

Can you spot the problem?

Single Thread Concurrent Server

One can implement a concurrent server using one thread/ process

```
while (true) {
        check if any new connects (non-block accept)
        if new connection accept
        process a little on each current request
}
```

Stateless verses Stateful Servers

State information

Information maintained by server about ongoing interactions with clients

Consumes server resources

How long does one maintain the state?

Modes of Operation

Stateful servers sometimes have different modes of operation

Each mode has a set of legal commands

In Login mode only the commands password & username are acceptable

After successful login client-server connection in transaction mode

In transaction mode command X, Y Z are legal

These modes are also called server states or just states

Some threading issues

- Passing Data
- Polling
- Callbacks
- Thread Pools
- Resuing Threads

Passing Data – Multiple Thread Access

Situation

An object is passed between threads

Issue

```
anObject = anotherThreadObject.getFoo();    // line A
System.out.println( anObject);               // line B
```

If multiple threads have access to anObject

The state of anObject can change after line A ends and before line B starts!

This can cause debugging nightmares

Passing Data – Possible Solutions

Pass copies

Returning data

```
pubic foo getFoo() {
    return foo.clone();
}
```

Parameters

```
anObject.doSomeMunging( bar.clone());
```

Passing Data – Possible Solutions

Immutable Objects

Pass objects that cannot change

Java's base types (Integer) and Strings are immutable

Passing Data – Possible Solutions

Thread one does not maintain reference to data after passing it on

Passing Data - How to Pass

Shared Queues are commonly used

Background Operations

Situation

Perform operation in the background

At same time perform operations in the foreground

Need to get the result when operation is done

Issue

Don’t make the code sequential

Avoid polling

```
public class Poll  {
    public static void main( String args[] ) {

    TimeConsumingOperation background =
        new TimeConsumingOperation();
    background.start();

    while ( !background.isDone() ) {
        performSomethingElse;
    }
    Object neededInfo = background.getResult();
    }
}
```

Futures

A future starts a computation in a thread

When you need the result ask the future

You will block if the result is not ready

Sample Java Future

```
class FutureWrapper {
    TimeConsumingOperation myOperation;

    public FutureWrapper() {
        myOperation =
            new TimeConsumingOperation();
        myOperation.start();
    }

    public Object value() {
        try {
            myOperation.join();
            return myOperation.getResult();
        } catch (InterruptedException trouble ) {
            DoWhatIsCorrectForYourApplication;
        }
    }
}
```

```
public class FutureExample  {
    public static void main( String args[] ) {

    FutureWrapper myWorker =
        new FutureWrapper();

    DoSomeStuff;
    DoMoreStuff;

    x = myWorker.value();
    }
}
```

java.util.concurrent.FutureTask<V> (JDK 1.5)

```
import java.util.concurrent.*;

public class SimpleRunable<V> implements Callable<V> {
        V fakeResult;

        public SimpleRunable(V x) {
                fakeResult = x;
        }

        public V call() throws Exception {
                Thread.sleep((long)1000);
                return fakeResult;
         }
 }
        public void testFuture() throws InterruptedException, ExecutionException
{
                Callable<String> faked = new SimpleRunable<String>("go");
                FutureTask<String> example = new FutureTask(faked);
                example.run();
                assertTrue( "go" == example.get());
        }
```

Callbacks

Have the background thread call a method when it is done

```
class MasterThread {
    public void normalCallback( Object result ) {
        processResult;
    }

    public void someMethod() {
        compute;
        TimeConsumingOperation backGround =
            new TimeConsumingOperation( this );

        backGround.start();
        moreComputation;
    }
}
```

```
class TimeConsumingOperation extends Thread {
    MasterThread master;

    public TimeConsumingOperation(
                        MasterThread aMaster ) {
        master = aMaster;
    }

    public void run() {
            DownLoadSomeData;
            PerformSomeComplexStuff;
            master.normalCallback( resultOfMyWork );
    }
}
```

Thread Pools - Some Background

Iterative Server

```
while (true)
        {
        Socket client = serverSocket.accept();
        Sequential code to handle request
        }
```

When usable

TP = Time to process a request

A = arrival time between two consecutive requests

Then we need TP << A

Thread Pools - Some Background

Basic Concurrent Server

```
while (true)
        {
        Socket client = serverSocket.accept();
        Create a new thread to handle request
        }
```

When usable

Let TC = time to create a thread

Let A = arrival time between two consecutive requests

We need TC << A

Often this is good enough

Problem with Threads

Thread consume resources

- Memory
- CPU cycles

A program has a limit of

- Threads it can productively support
- Sockets it can have open

We need to insure we don’t create too many threads

Some Timing Results

VisualWorks Smalltalk				
	Iterations or Number Created (n)			
	100	1,000	10,000	100,000
Empty Loop	0	0	0	1
Integer add	0	0	0	3
Collection create	0	0	4	34
Thread create	1	5	45	380
Thread create & run	1	5	152	1268
Thread create & run	0	3	82	1048

Ruby				
	Iterations or Number Created (n)			
	100	1,000	10,000	100,000
Empty Loop	0	1	10	83
Integer add	0	3	44	248
Collection create	0	4	53	356
Thread create & run	38	289	2153	21526
Thread create & run	37	268	2116	22298

Java				
	Iterations or Number Created (n)			
	100	1,000	10,000	100,000
Empty Loop	0	0	1	7
Integer add	0	0	1	7
Vector create	0	5	10	42
Thread create	2	62	185	1771
Thread create & run	40	402	2626	24909
Thread create & run	51	351	2507	24767

Macintosh PowerBook with 1.25GHz PowerPC processo

OS 10.3.3 (OS 10.4.3 for Ruby)

VW 7.2nc

Java 1.4.2_03 with HotSpot Client

Ruby 1.8

VisualWorks Details

Loop	n timesRepeat:[ ]
Integer add	n timesRepeat:[x := 3 +4]
Collection create	n timesRepeat:[x := OrderedCollection new]
Thread create	n timesRepeat:[[x := 3 +4] newProcess]
Thread create & run	n timesRepeat:[[x := 3 +4] fork]

Code used

```
Transcript clear.
#(100 1000 10000 100000) do:
    [:n |
    | x |
ObjectMemory garbageCollect.
time := Time millisecondsToRun:
   [n timesRepeat: [[x := 3 + 4] fork]].
Transcript
   print: n;
   tab;
   print: time;
   tab;
   print: x;
   cr;
   flush]
```

Java Code Timed

Loop	for (int k = 0; k < n; k++){
	}

Integer add	for (int k = 0; k < n; k++){
	x = 3 + 4
	}

Collection create	for (int k = 0; k < n; k++){
	x = new Vector();
	}

Thread create	for (int k = 0; k < n; k++){
	x = new SampleThread();
	}

Thread create & run	for (int k = 0; k < n; k++){
	x = new SampleThread();
	x.start();
	}

Java Program Used

```
import java.util.*;
import sdsu.util.Timer;

public class TimeTests {
  public static void main (String args[]) {
    Timer clock = new Timer();
    SampleThread x = new SampleThread();

    for (int n = 100; n < 200000; n = n * 10) {
      System.gc();
      clock.start();
      for (int k = 0; k < n; k++){
        x = new SampleThread();
        x.start();
      }
      long time = clock.stop();
      x.x();
      System.out.println("" +  n + "\t" + time);
      clock.reset();
    }
  }
}
```

```
class SampleThread extends Thread {
      int x;
      public void run() {
            x = 3 + 4;
      }

      public int x() {
            return x;
      }
}
```

Ruby Code

```
[100, 1000, 10000, 100000].each do |
size|
    start = Time.now
    size.times do
        x = Thread.new do
            x = 3 + 4
        end
    end
    endTime = Time.now

    puts ((endTime - start)*1000).round
end
```

Empty Loop

```
size.times do
end
```

Collection Creation

```
start = Time.now
x = 4
size.times do
   x = Array.new
end
endTime = Time.now
x[0] = 4
```

Warning about Micro-benchmarks

Micro-benchmarks are

Hard to do well

Misleading

Better to measure the performance of your system

Concurrent Server With Thread Pool

```
Create N worker threads
while (true)
        {
        Socket client = serverSocket.accept();
        Use an existing worker thread to handle
request
        }
```

When usable

TP = Time to process a request

A = arrival time between two consecutive requests

N = Thread Pool size

Then we need TP << A * N

Concurrent Server - Thread Pool

```
Create N worker threads
while (true)
        {
        Socket client = serverSocket.accept();
        if worker thread is idle
                Use an existing worker thread to handle
request
        else
                create new worker thread to handle the
request
        }
```

When usable

Number of requests we can handle in a unit of time

$$TP / N + 1/TC$$

where N is not constant

TP = Time to process a request
TC = time to create thread

What to do with the new Worker Threads?

Client requests are not constant over time

Requests can come in bursts

Threads consume resources

Don’t want a large pool of threads sitting idle

Common strategy

Have a minimum number of threads in a pool

When needed add threads to the pool up to some maximum

When traffic slows down remove idle threads

Threads & Memory Cache

Threads require a fair amount of memory (why?)

Virtual memory divides memory into pages

A page may be in

- Memory
- Memory Cache
- Disk Cache
- Disk

Access to a page is faster if it is in memory

Last thread that completed is likely to be in memory or cache

Reusing last thread that complete can improve performance

Which Should I use?

Which method to use?

Which values (number of threads, etc) to use?

Depends on your

- Application
- Implementation
- Hardware
- Performance requirements

How to reuse a Thread?

Classic idea

Server places client requests in a queue

Worker repeats forever

- Read request from queue
- Process request

Queue

- Block on read if queue is empty
- Signals waiting threads when data is added

Java Example - SharedQueue

```
import java.util.ArrayList;
public class SharedQueue  {
  ArrayList elements = new ArrayList();

  public synchronized void append( Object item ) {
    elements.add( item);
    notify();
  }

  public synchronized Object get( )  {
    try {
      while ( elements.isEmpty() )
        wait();
    }
    catch (InterruptedException threadIsDone )  {
      return null;
    }
    return elements.remove( 0);
}

public int size(){
  return elements.size();
  }
}
```

DateHandler

```
import java.net.*;
import java.io.*;
import java.util.Date;

public class DateHandler extends Thread  {
  SharedQueue workQueue;

  public DateHandler(SharedQueue workSource ) {
    workQueue = workSource;
  }

public void run() {
  while (!isInterrupted() )
    try {
      Socket client = (Socket) workQueue.get();
      processRequest(client);
    }
    catch (Exception error ){
      /* log error*/
      }
}
```

```
void processRequest(Socket client) throws IOException {
    try {
      client.setSoTimeout( 10 * 1000 );
      processRequest(
        client.getInputStream(),
        client.getOutputStream());
    } finally {
      client.close();
    }
  }

  void processRequest(InputStream in, OutputStream out)
      throws IOException{
    BufferedReader parsedInput =
        new BufferedReader(new InputStreamReader(in));
    PrintWriter parsedOutput = new PrintWriter(out,true);
    String inputLine = parsedInput.readLine();
    if (inputLine.startsWith("date"))  {
      Date now = new Date();
      parsedOutput.println(now.toString());
    }
  }
}
```

DateServer

```
import java.util.*;
import java.net.*;
import java.io.*;

public class DateServer {
   SharedQueue workQueue;
   ServerSocket listeningSocket;
   ArrayList workers = new ArrayList();

   public static void main( String[] args ) {
      System.out.println( "Starting");
      new DateServer( 33333).run();
   }

public void run() {
      Socket client = null;
      while (true) {
         try {
            client = listeningSocket.accept();
            workQueue.append( client);
         } catch (IOException acceptError){
            // need to log error and make sure client is closed
            }
         }
      }
   }
```

```
public DateServer( int port ) {
      try {
         listeningSocket = new ServerSocket(port);
         workQueue = new SharedQueue();
         for (int k = 0; k < 5; k++) {
            Thread worker = new DateHandler( workQueue);
            worker.start();
            workers.add( worker);
         }
      } catch (IOException socketCreateError) {
         //log and exit here
      }
   }
```