SE 552: Lecture 9

Overview

Asynchronous method calls
Worker pools
Timers
Event loops
Polling IO
Asynchronous method calls

What is an asynchronous method call (a.k.a. one-way message)?

```java
public class MultiThreadedServer {
    final int portNum = 2000;
    final ServerSocket server = new ServerSocket (portNum);
    public void start () throws IOException {
        while (true) {
            final Socket socket = server.accept ();
            final Runnable task = new Handler (socket);
            task.run (); // asynchronous method call
        }
    }
}

class Handler implements Runnable {
    final Socket socket;
    Handler (final Socket socket) { this.socket = socket; }
    public void run () { ...handle the connection... } 
}
```

What are other examples of asynchronous method calls?
Asynchronous method calls

One way to implement the multi-threaded server:

```java
public class MultiThreadedServer {
    final int portNum = 2000;
    final ServerSocket server = new ServerSocket(portNum);
    final Executor executor = Executor.singleton;
    public void start() throws IOException {
        while (true) {
            final Socket socket = server.accept();
            final Runnable task = new Handler(socket);
            executor.execute(task); // asynchronous method call
        }
    }
}
```

```java
public interface Executor {
    public void execute(Runnable task);
    public static Executor singleton = new ExecutorImpl();
}
```

How can we implement the ExecutorImpl?
Worker pools

One possible implementation uses a *worker pool*:

class WorkerPool implements Executor, Runnable {
    final TaskQueue queue = TaskQueue.factory.build ();
    public void addWorkers (final int numWorkers) {
        for (int i=0; i<numWorkers; i++) {
            new Thread (this).start ();
        }
    }
    public void execute (final Runnable task) { queue.put (task); }
    public void run () { ... }
}

What should the TaskQueue interface be?

What should the code for run be?
Worker pools

Problems with worker pools:

- Worker threads are recycled, so any code which depends on thread names may not work.
- Tasks may end up blocked on the task queue: this may cause deadlock! (How?)
- When should new worker threads be added? When should worker threads die?
- Should we have a limit on the number of worker threads? What do we do when we reach the limit? (Possibilities: just let the queue grow, drop new requests, drop old requests, apply back pressure...)
Timers

We could extend the Executor interface to allow for delayed execution:

```java
public interface Executor {
    public void execute (Runnable task);
    public void executeAt (Runnable task, long time);
    public static Executor singleton = new ExecutorImpl ();
}
```

Why would we want this? How would we implement it?
Event loops

One extreme is a worker pool with only one thread.

This is the basis of event loops:

```java
public interface EventHandler {
    public void handle (Event e);
    public EventHandler singleton = new EventHandlerImpl ();
}
public interface Event {
    public EventListener[] listeners ();
}
public interface EventListener {
    public void notify (Event e);
}
```

How can we implement EventHandlerImpl?

Note that all events are handled by one thread! What advantages / disadvantages does this have?
Polling IO

Some servers have odd characteristics:

- Large numbers of simultaneous open connections.
- The traffic of an individual connection is small.
- The total traffic of all connections is large.

There may be lots more open connections than threads!
Polling IO

One possible code sketch:

```java
public class PollingServer implements Runnable {
    final int portNum = 2000;
    final int commandSize = 50; // each command is 50 chars
    final ServerSocket server = new ServerSocket(portNum);
    final Thread thread = new Thread(this);
    final SocketPool pool = SocketPool.factory.build();

    public void start() throws IOException {
        thread.start();
        while (true) { SocketPool.add(server.accept()); }
    }

    public void run() {
        while (true) {
            final Socket socket = pool.get(); // get a socket with 50 chars waiting
            process the command waiting on the socket
        }
    }
}
```

We need to complete the implementation by providing SocketPool.
Summary

Asynchronous method calls require thread objects to execute the method call.

These threads can either be created dynamically, or allocated in a thread pool.

Next week: more Chapter 4.