Java Threads

Notice: The slides for this lecture have been largely based on those accompanying the textbook Operating Systems Concepts with Java, by Silberschatz, Galvin, and Gagne (2003). Many, if not all, the illustrations contained in this presentation come from this source.
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many
Many-to-One Model

Many user-level threads mapped to single kernel thread.
One-to-One Model

Each user-level thread maps to kernel thread.
Many-to-Many Model

Several user level threads are mapped to several kernel threads. Allows the operating system to create a sufficient number of kernel threads.
Two-Level Model

Similar to M:M, except that it allows a user thread to be bound to kernel thread.
Thread Cancellation

• Terminating a thread before it has finished.

• Two general approaches:
  – **Asynchronous cancellation** terminates the target thread immediately.
  – **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.
Thread Specific Data

• Allows each thread to have its own copy of data.

• Useful when you do not have control over the thread creation process (i.e., when using a thread pool).
Java Threads

- Java threads are managed by the JVM.

- Java threads may be created by:
  - Extending Thread class.
  - Implementing the Runnable interface.
Extending the Thread Class

class Worker1 extends Thread
{
   public void run()
   {
      System.out.println("I Am a Worker Thread");
   }
}

public class First
{
   public static void main(String args[])
   {
      Worker1 runner = new Worker1();
      runner.start();

      System.out.println("I Am The Main Thread");
   }
}
The Runnable Interface

```java
public interface Runnable
{
    public abstract void run();
}
```
Implementing the Runnable Interface

class Worker2 implements Runnable {
    public void run() {
        System.out.println("I Am a Worker Thread");
    }
}

public class Second {
    public static void main(String args[]) {
        Runnable runner = new Worker2();
        Thread thrd = new Thread(runner);
        thrd.start();

        System.out.println("I Am The Main Thread");
    }
}
Java Thread States

new

new

start()

I/O

I/O is available

runnable

sleep()

I/O

blocked

exits run() method
Joining Threads

class JoinableWorker implements Runnable {
    public void run() {
        System.out.println("Worker working");
    }
}

public class JoinExample {
    public static void main(String[] args) {
        Thread task = new Thread(new JoinableWorker());
        task.start();

        try { task.join(); } catch (InterruptedException ie) { }

        System.out.println("Worker done");
    }
}
Thread Cancellation

Thread thrd = new Thread (new InterruptibleThread());
Thrd.start();

... 

// now interrupt it
Thrd.interrupt();

One could also use the stop() method in the thread class, but that is deprecated (that is, still exists, but is being phased out). Note that while stop() is asynchronous cancellation, interrupt() is deferred cancellation.
Thread Cancellation

```java
public class InterruptibleThread implements Runnable {
    public void run() {
        while (true) {
            /* *
            * do some work for awhile
            */
            if (Thread.currentThread().isInterrupted()) {
                System.out.println("I'm interrupted!");
                break;
            }
            // clean up and terminate
        }
    }
}
```

With deferred cancellation, the thread must periodically check if it's been cancelled.
Thread-Specific Data

All one needs to do in order to create data that is specific to a thread is to subclass the `Thread` class declaring its own private data.

This approach doesn’t work when the developer has no control over the thread creation process.
Thread Specific Data

class Service
{
    private static ThreadLocal errorCode = new ThreadLocal();

    public static void transaction() {
        try {
            // some operation where an error may occur
            catch (Exception e) {
                errorCode.set(e);
            }
        }
    }

    // get the error code for this transaction
    public static Object getErrorCode() {
        return errorCode.get();
    }
}
Thread Specific Data

class Worker implements Runnable
{
    private static Service provider;

    public void run()
    {
        provider.transaction();
        System.out.println(provider.getErrorCode());
    }
}
Producer-Consumer Problem

```java
public class Factory
{
    public Factory() {
        // first create the message buffer
        Channel mailbox = new MessageQueue();
        // now create the producer and consumer threads
        Thread producerThread = new Thread(new Producer(mailbox));
        Thread consumerThread = new Thread(new Consumer(mailbox));
        producerThread.start();
        consumerThread.start();
    }

    public static void main(String[] args) {
        Factory server = new Factory();
    }
}
```
class Producer implements Runnable
{
    private Channel mbox;

    public Producer(Channel mbox) {
        this.mbox = mbox;
    }

    public void run() {
        Date message;

        while (true) {
            SleepUtilities.nap();
            message = new Date();
            System.out.println("Producer produced "+ message);

            // produce an item & enter it into the buffer
            mbox.send(message);
        }
    }
}
Consumer Thread

class Consumer implements Runnable
{
    private Channel mbox;

    public Consumer(Channel mbox) {
        this.mbox = mbox;
    }

    public void run() {
        Date message;

        while (true) {
            SleepUtilties.nap();
            // consume an item from the buffer
            System.out.println("Consumer wants to consume.");

            message = (Date) mbox.receive();
            if (message != null)
                System.out.println("Consumer consumed " + message);
        }
    }
}