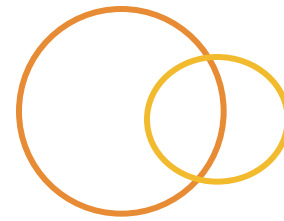
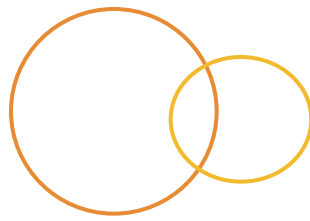




Advanced Concurrent Programming Part II



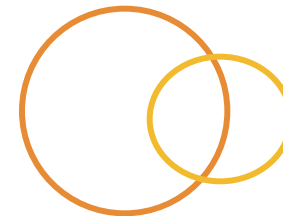
Objectives





When we are done, you should be able to:

- 🕒 Understand how to use the Fork-Join framework
- 🕒 Explain the importance of the `java.util.concurrent.Future` class

Parallelism



Multiple CPUs deserve Multiple Threads

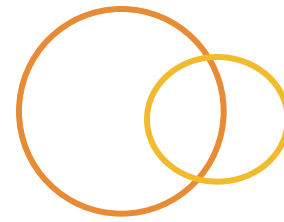


Introduction to Parallelism



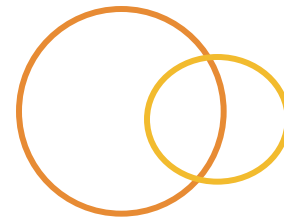
- Modern computers have multiple CPUs
- Java uses one thread per task
 - Large task means unused CPUs
- We want to take large task and break it into smaller ones
 - Theoretically, each thread gets own CPU
- Easy example: One client query requiring multiple database queries
- Complex example: Doing calculation on gigabytes of data

Do it Yourself



- ◉ Works with easy things
 - ◉ Obvious logic breaks in tasks
- ◉ No way for us to optimize for:
 - ◉ Different speeds of CPUs
 - ◉ Different 'real' difficulty across data chunks
 - ◉ Noticing CPU is not in use so give it more work
 - ◉ Other non-Java tasks using CPUs

Fork-Join Framework



- Introduced as part of Java SE 7
- Fork-Join has algorithm to better utilize CPUs
 - We break down task into subsets
 - Subsets assigned to a thread's processing queue
 - When queue is empty, thread can steal work from other threads

Fork-Join Framework [cont.]



Classes involved:

- ◉ `java.util.concurrent.ForkJoinPool`
 - ◉ Specialized `ExecutorService`
 - ◉ Implements core work-stealing algorithm
 - ◉ Provides means for managing and monitoring operations
- ◉ `java.util.concurrent.ForkJoinTask`
 - ◉ Abstract parent to `RecursiveAction` and `RecursiveTask`
 - ◉ Uses `fork()` to break task into group of asynchronous executions
 - ◉ Uses `join()` to bring them back together when task is completed

Fork-Join Framework [cont.]



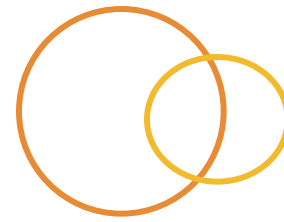
Classes involved [cont.]:

- ◉ `java.util.concurrent.RecursiveAction`
 - ◉ No return result
 - ◉ If something returned, it is `null`
- ◉ `java.util.concurrent.RecursiveTask`
 - ◉ Returns something

java.util.concurrent.Future<V>

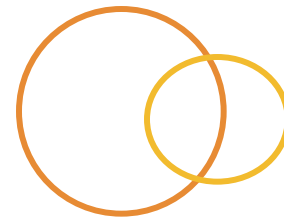
- Interface Representing the result of an asynchronous computation
- Is implemented by ForkJoinTask, RecursiveAction and RecursiveTask
- There are others, but they aren't germane to the fork-join discussion

Best Practices



- ⦿ Avoid synchronized methods or blocks in calculations
- ⦿ Use synchronizer classes defined to work with fork-join if synchronization is required within task
- ⦿ Avoid blocking I/O
- ⦿ Do not throw checked exceptions
- ⦿ These practices can be broken, but if broken often, can lead to performance degradation

Writing the Task



```
package examples.concurrent.advanced;

import java.util.concurrent.RecursiveTask;

public class MinValueTask extends RecursiveTask<Integer>{

    public Integer compute(){
        if(end - start < threshold){
            int foundMin = Integer.MAX_VALUE;
            for(int i = start; i<=end; i++){
                int value = dataSet[i];
                if(value < foundMin ){
                    foundMin = value;
                }
            }
            return foundMin;
        }
        else {
            int middle = (end-start) /2+start;
            MinValueTask t1 = new MinValueTask(dataSet, start, middle, threshold);
            t1.fork();
            MinValueTask t2 = new MinValueTask(dataSet, middle+1, end, threshold);
            return Math.min(t2.compute(), t1.join());
        }
    }
}
```

Writing the Task [cont.]



```
private final int threshold;  
private final int[] dataSet;  
private int start;  
private int end;
```

```
public MinValueTask(int dataSet[], int start, int end, int threshold){  
    this.dataSet = dataSet;  
    this.start=start;  
    this.threshold = threshold;  
    this.end = end;  
}
```

```
}
```

Defining the ForkJoinPool



```
import java.util.Random;
import java.util.concurrent.ForkJoinPool;

public class Test {
    public static void main(String[] args){
        long start = System.nanoTime();
        int dataSet[] = Test.calculateDataSet();
        ForkJoinPool pool = new ForkJoinPool();
        MinValueTask task = new MinValueTask(dataSet, 0,
            dataSet.length-1, dataSet.length/16);
        Integer minimumValue = pool.invoke(task);
        long end = System.nanoTime();
        System.out.println("The minimum value found was "+ minimumValue);
        System.out.println("It took "+ (end-start) + " nanoseconds to perform");
    }

    private static int[] calculateDataSet(){
        int size = 365*24*60;
        int[] data = new int[size];
        Random generator = new Random();
        for(int x=0; x<data.length; x++){
            data[x] = generator.nextInt();
        }
        return data;
    }
}
```

Copyright DevelopIntelligence 2012

Lab: Verifying Fork-Join Usefulness



- ◉ **Purpose:** To see if there is a difference in finding something in a large data set when using Fork-Joins
- ◉ **How:** Write a non-threaded 'application' that makes a large data set of random numbers and then finds the maximum value in the data set. Similarly to how it was done in the example, see how long it takes to run.

Modify the application to do the same thing, but now using the Fork-Join framework. Is there a difference in performance? You may want to run both versions several times to get an average of how long they take.