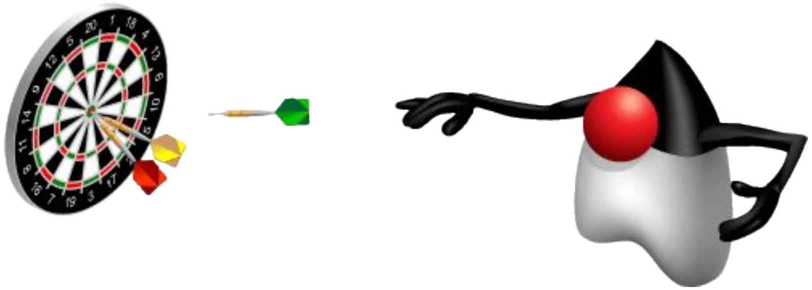
Lesson 12
Concurrency

Objectives

After completing this lesson, you should be able to:

- Use atomic variables
- Use a `ReentrantReadWriteLock`
- Use the `java.util.concurrent` collections
- Describe the synchronizer classes
- Use an `ExecutorService` to concurrently execute tasks
- Apply the Fork-Join framework



The `java.util.concurrent` Package

Java 5 introduced the `java.util.concurrent` package, which contains classes that are useful in concurrent programming. Features include:

- Concurrent collections
- Synchronization and locking alternatives
- Thread pools
  - Fixed and dynamic thread count pools available
  - Parallel divide and conquer (Fork-Join) new in Java 7

The `java.util.concurrent.atomic` Package

The `java.util.concurrent.atomic` package contains classes that support lock-free thread-safe programming on single variables

```
AtomicInteger ai = new AtomicInteger(5);
if(ai.compareAndSet(5, 42)) {
    System.out.println("Replaced 5 with 42");
}
```

An atomic operation ensures that the current value is 5 and then sets it to 42.

The `java.util.concurrent.locks` Package

The `java.util.concurrent.locks` package is a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors.

```
public class ShoppingCart {
    private final ReentrantReadWriteLock rwl =
        new ReentrantReadWriteLock();

    public void addItem(Object o) {
        rwl.writeLock().lock();
        // modify shopping cart
        rwl.writeLock().unlock();
    }
```

A single writer, multi-reader lock

Write Lock

java.util.concurrent.locks

```
    public String getSummary() {
        String s = "";
        rwl.readLock().lock();
        // read cart, modify s
        rwl.readLock().unlock();
        return s;
    }
    public double getTotal() {
        // another read-only method
    }
}
```

Read Lock

All read-only methods can concurrently execute.

Thread-Safe Collections

The `java.util` collections are not thread-safe. To use collections in a thread-safe fashion:

- Use synchronized code blocks for all access to a collection if writes are performed
- Create a synchronized wrapper using library methods, such as `java.util.Collections.synchronizedList(List<T>)`
- Use the `java.util.concurrent` collections

Note: Just because a `Collection` is made thread-safe, this does not make its elements thread-safe.

Quiz

A `CopyOnWriteArrayList` ensures the thread-safety of any object added to the `List`.

a. True

b. False

Synchronizers

The `java.util.concurrent` package provides five classes that aid common special-purpose synchronization idioms.

Class	Description
Semaphore	Semaphore is a classic concurrency tool.
CountDownLatch	A very simple yet very common utility for blocking until a given number of signals, events, or conditions hold
CyclicBarrier	A resettable multiway synchronization point useful in some styles of parallel programming
Phaser	Provides a more flexible form of barrier that may be used to control phased computation among multiple threads
Exchanger	Allows two threads to exchange objects at a rendezvous point, and is useful in several pipeline designs

java.util.concurrent.CyclicBarrier

The `CyclicBarrier` is an example of the synchronizer category of classes provided by `java.util.concurrent`.

```
final CyclicBarrier barrier = new CyclicBarrier(2);

new Thread() {
    public void run() {
        try {
            System.out.println("before await - thread 1");
            barrier.await();
            System.out.println("after await - thread 1");
        } catch (BrokenBarrierException|InterruptedException ex) {

        }
    }
}.start();
```

Two threads must await before they can unblock.

May not be reached

High-Level Threading Alternatives

Traditional `Thread` related APIs can be difficult to use properly. Alternatives include:

- `java.util.concurrent.ExecutorService`, a higher level mechanism used to execute tasks
  - It may create and reuse `Thread` objects for you.
  - It allows you to submit work and check on the results in the future.
- The Fork-Join framework, a specialized work-stealing `ExecutorService` new in Java 7

java.util.concurrent.ExecutorService

- An `ExecutorService` is used to execute tasks.
  - It eliminates the need to manually create and manage threads.
  - Tasks **might** be executed in parallel depending on the `ExecutorService` implementation.
  - Tasks can be:
    - `java.lang.Runnable`
    - `java.util.concurrent.Callable`
  - Implementing instances can be obtained with `Executors`.

```
ExecutorService es = Executors.newCachedThreadPool();
```

java.util.concurrent.Callable

The `Callable` interface:

- Defines a task submitted to an `ExecutorService`
- Is similar in nature to `Runnable`, but can:
  - Return a result using generics
  - Throw a checked exception

```
package java.util.concurrent;
public interface Callable<V> {
    V call() throws Exception;
}
```

java.util.concurrent.Future

The `Future` interface is used to obtain the results from a `Callable`'s `V call()` method.

`ExecutorService` controls when the work is done.

```
Future<V> future = es.submit(callable);
//submit many callables
try {
    V result = future.get();
} catch (ExecutionException|InterruptedException ex) {

}
```

Gets the result of the `Callable`'s `call` method (blocks if needed).

If the `Callable` threw an `Exception`

Shutting Down an `ExecutorService`

Shutting down an `ExecutorService` is important because its threads are nondaemon threads and will keep your JVM from shutting down.

Stop accepting new `Callable`s.

If you want to wait for the `Callable`s to finish

```
es.shutdown();

try {
    es.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
    System.out.println("Stopped waiting early");
}
```

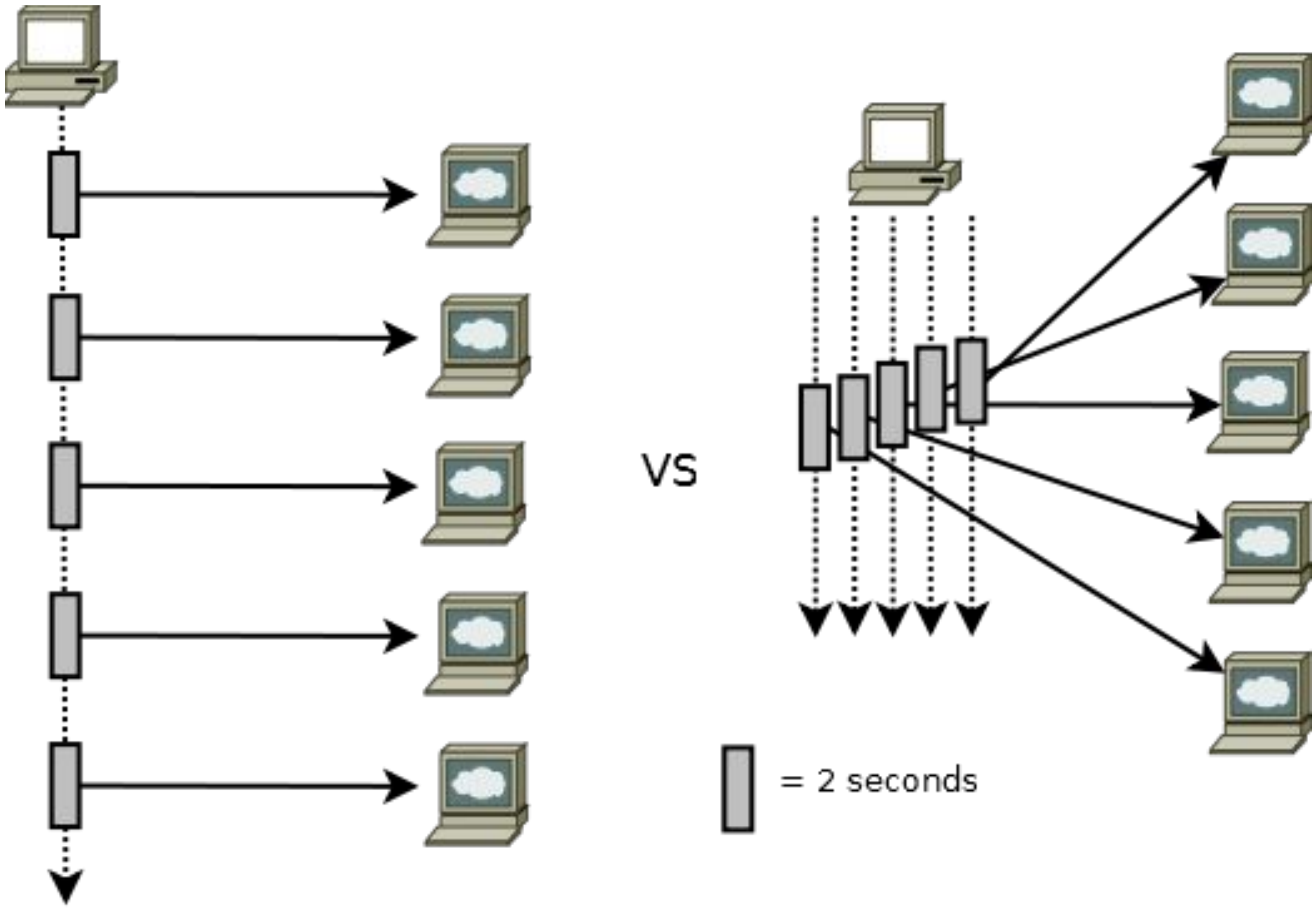
Quiz

An `ExecutorService` will always attempt to use all of the available CPUs in a system.

a. True
b. False

Concurrent I/O

Sequential blocking calls execute over a longer duration of time than concurrent blocking calls.



A Single-Threaded Network Client

```
public class SingleThreadClientMain {
    public static void main(String[] args) {
        String host = "localhost";
        for (int port = 10000; port < 10010; port++) {
            RequestResponse lookup =
             new RequestResponse(host, port);
            try (Socket sock = new Socket(lookup.host, lookup.port);
                 Scanner scanner = new Scanner(sock.getInputStream());){
                lookup.response = scanner.next();
                System.out.println(lookup.host + ":" + lookup.port + " " +
                    lookup.response);
            } catch (NoSuchElementException|IOException ex) {
                System.out.println("Error talking to " + host + ":" +
                    port);
            }
        }
    }
}
```

A Multithreaded Network Client (Part 1)

```
public class MultiThreadedClientMain {
    public static void main(String[] args) {
        //ThreadPool used to execute Callables
        ExecutorService es = Executors.newCachedThreadPool();
        //A Map used to connect the request data with the result
        Map<RequestResponse,Future<RequestResponse>> callables =
            new HashMap<>();

        String host = "localhost";
        //loop to create and submit a bunch of Callable instances
        for (int port = 10000; port < 10010; port++) {
            RequestResponse lookup = new RequestResponse(host, port);
            NetworkClientCallable callable =
                new NetworkClientCallable(lookup);
            Future<RequestResponse> future = es.submit(callable);
            callables.put(lookup, future);
        }
```

A Multithreaded Network Client (Part 2)

```
//Stop accepting new Callables
es.shutdown();

try {
    //Block until all Callables have a chance to finish
    es.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
    System.out.println("Stopped waiting early");
}
```

A Multithreaded Network Client (Part 3)

```
        for(RequestResponse lookup : callables.keySet()) {
            Future<RequestResponse> future = callables.get(lookup);
            try {
                lookup = future.get();
                System.out.println(lookup.host + ":" + lookup.port + " " +
                    lookup.response);
            } catch (ExecutionException|InterruptedException ex) {
                //This is why the callables Map exists
                //future.get() fails if the task failed
                System.out.println("Error talking to " + lookup.host +
                    ":" + lookup.port);
            }
        }
    }
}
```

A Multithreaded Network Client (Part 4)

```
public class RequestResponse {
    public String host; //request
    public int port; //request
    public String response; //response

    public RequestResponse(String host, int port) {
        this.host = host;
        this.port = port;
    }

    // equals and hashCode

}
```

A Multithreaded Network Client (Part 5)

```
public class NetworkClientCallable implements Callable<RequestResponse> {
    private RequestResponse lookup;

    public NetworkClientCallable(RequestResponse lookup) {
        this.lookup = lookup;
    }

    @Override
    public RequestResponse call() throws IOException {
        try (Socket sock = new Socket(lookup.host, lookup.port);
             Scanner scanner = new Scanner(sock.getInputStream());) {
            lookup.response = scanner.next();
            return lookup;
        }
    }
}
```

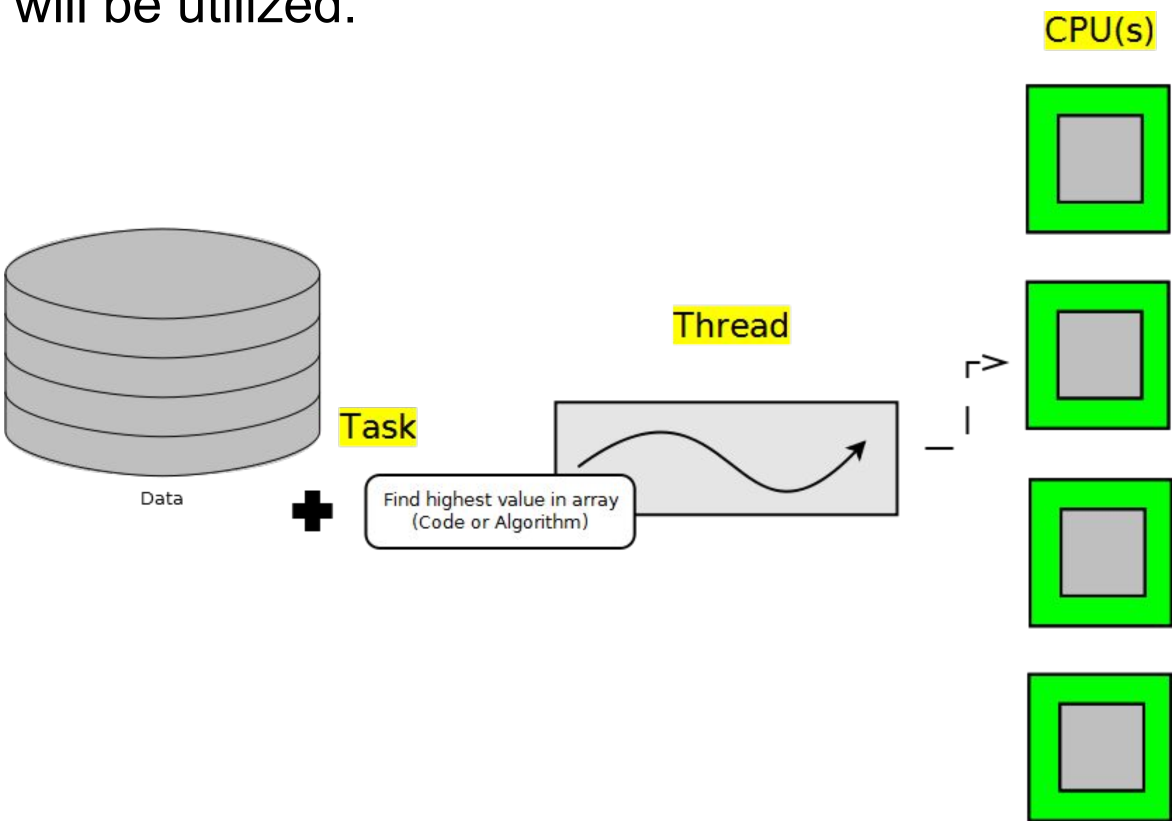
Parallelism

Modern systems contain multiple CPUs. Taking advantage of the processing power in a system requires you to execute tasks in parallel on multiple CPUs.

- Divide and conquer: A task should be divided into subtasks. You should attempt to identify those subtasks that can be executed in parallel.
- Some problems can be difficult to execute as parallel tasks.
- Some problems are easier. Servers that support multiple clients can use a separate task to handle each client.
- Be aware of your hardware. Scheduling too many parallel tasks can negatively impact performance.

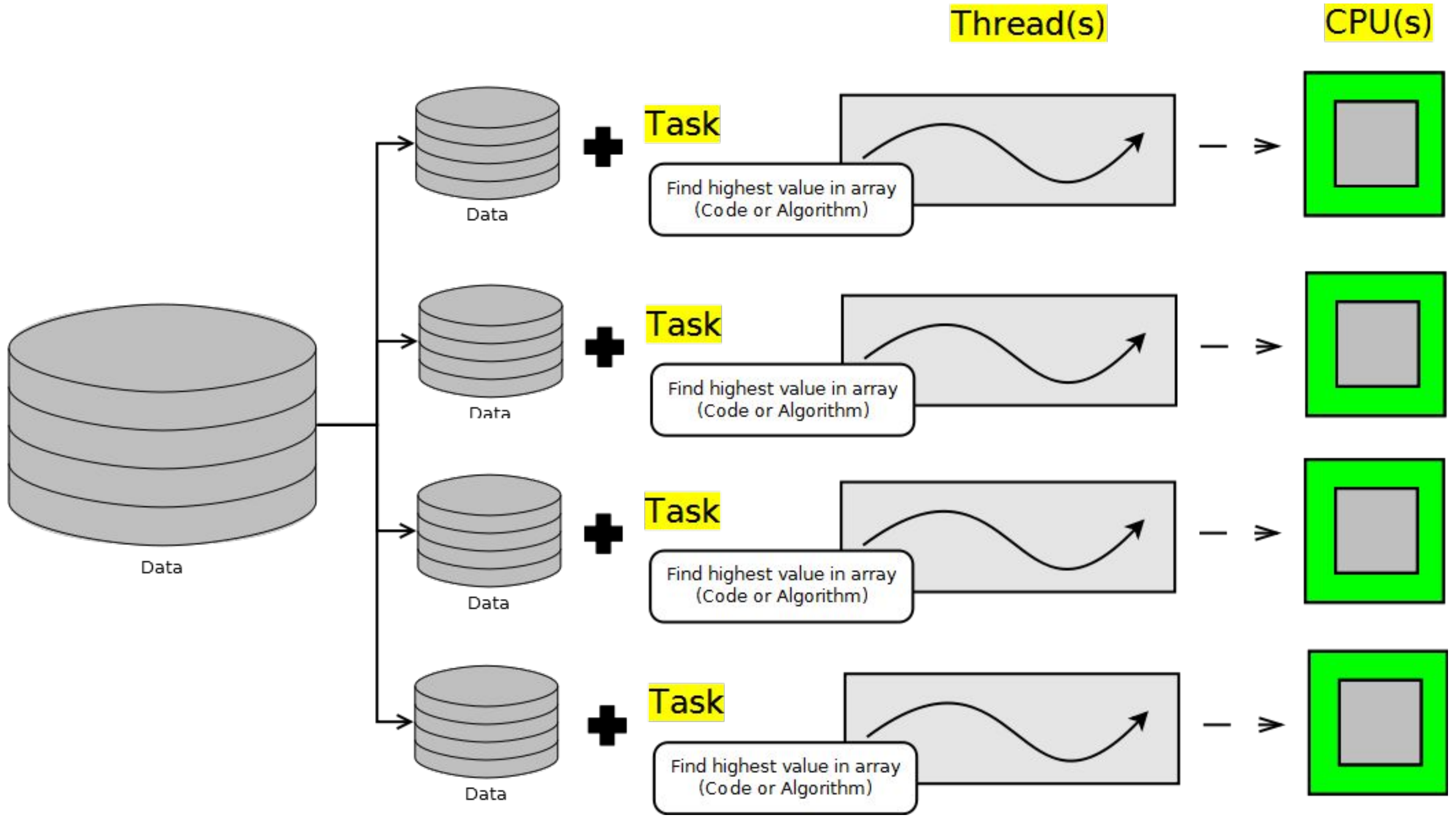
Without Parallelism

Modern systems contain multiple CPUs. If you do not leverage threads in some way, only a portion of your system's processing power will be utilized.



Naive Parallelism

A simple parallel solution breaks the data to be processed into multiple sets. One data set for each CPU and one thread to process each data set.

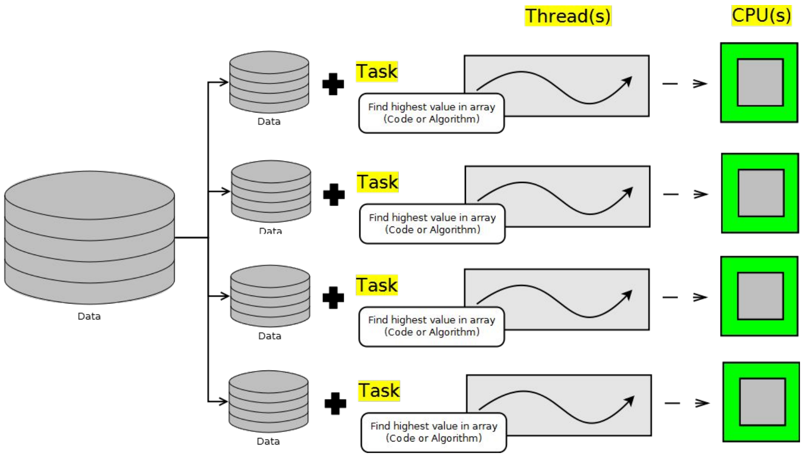


The Need for the Fork-Join Framework

Splitting datasets into equal sized subsets for each thread to process has a couple of problems. Ideally all CPUs should be fully utilized until the task is finished but:

- CPUs may run a different speeds
- Non-Java tasks require CPU time and may reduce the time available for a Java thread to spend executing on a CPU

- The data being analyzed may require varying amounts of time to process

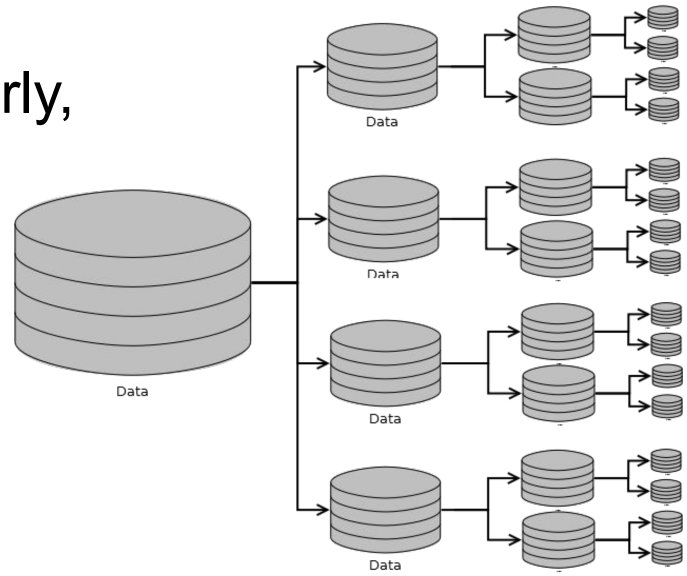


Work-Stealing

- To keep multiple threads busy:
  - Divide the data to be processed into a large number of subsets
  - Assign the data subsets to a thread's processing queue

- Each thread will have many subsets queued

If a thread finishes all its subsets early, it can "steal" subsets from another thread.



A Single-Threaded Example

```
int[] data = new int[1024 * 1024 * 256]; //1G

for (int i = 0; i < data.length; i++) {
    data[i] = ThreadLocalRandom.current().nextInt();
}

int max = Integer.MIN_VALUE;
for (int value : data) {
    if (value > max) {
        max = value;
    }
}
System.out.println("Max value found:" + max);
```

A very large dataset

Fill up the array with values.

Sequentially search the array for the largest value.

java.util.concurrent.ForkJoinTask<V>

A `ForkJoinTask` object represents a task to be executed.

- A task contains the code and data to be processed. Similar to a `Runnable` or `Callable`.
- A huge number of tasks are created and processed by a small number of threads in a Fork-Join pool.
  - A `ForkJoinTask` typically creates more `ForkJoinTask` instances until the data to processed has been subdivided adequately.
- Developers typically use the following subclasses:
  - `RecursiveAction`: When a task does not need to return a result
  - `RecursiveTask`: When a task does need to return a result

`RecursiveTask` Example

```
public class FindMaxTask extends RecursiveTask<Integer> {
    private final int threshold;
    private final int[] myArray;
    private int start;
    private int end;

    public FindMaxTask(int[] myArray, int start, int end,
  int threshold) {
        // copy parameters to fields
    }
    protected Integer compute() {
        // shown later
    }
}
```

Result type of the task

The data to process

Where the work is done.
Notice the generic return type.

compute Structure

```
protected Integer compute() {
    if DATA_SMALL_ENOUGH {
        PROCESS_DATA
        return RESULT;
    } else {
        SPLIT_DATA_INTO_LEFT_AND_RIGHT_PARTS
        TASK t1 = new TASK(LEFT_DATA);
        t1.fork();
        TASK t2 = new TASK(RIGHT_DATA);
        return COMBINE(t2.compute(), t1.join());
    }
}
```

Asynchronously execute

Process in current thread

Block until done

`compute` Example (Below Threshold)

```
protected Integer compute() {
    if (end - start < threshold) {
        int max = Integer.MIN_VALUE;
        for (int i = start; i <= end; i++) {
            int n = myArray[i];
            if (n > max) {
                max = n;
            }
        }
        return max;
    } else {
        // split data and create tasks
    }
}
```

You decide the threshold.

The range within the array

`compute` Example (Above Threshold)

```
protected Integer compute() {
    if (end - start < threshold) {
        // find max
    } else {
        int midway = (end - start) / 2 + start;
        FindMaxTask a1 =
    new FindMaxTask(myArray, start, midway, threshold);
        a1.fork();
        FindMaxTask a2 =
    new FindMaxTask(myArray, midway + 1, end, threshold);
        return Math.max(a2.compute(), a1.join());
    }
}
```

Task for left half of data

Task for right half of data

`ForkJoinPool` Example

A `ForkJoinPool` is used to execute a `ForkJoinTask`. It creates a thread for each CPU in the system by default.

```
ForkJoinPool pool = new ForkJoinPool();
FindMaxTask task =
 new FindMaxTask(data, 0, data.length-1, data.length/16);
Integer result = pool.invoke(task);
```

The task's `compute` method is automatically called .

Fork-Join Framework Recommendations

Avoid I/O or blocking operations.

- Only one thread per CPU is created by default. Blocking operations would keep you from utilizing all CPU resources.

Know your hardware.

- A Fork-Join solution will perform slower on a one-CPU system than a standard sequential solution.
- Some CPUs increase in speed when only using a single core, potentially offsetting any performance gain provided by Fork-Join.

Know your problem.

- Many problems have additional overhead if executed in parallel (parallel sorting, for example).

Quiz

Applying the Fork-Join framework will always result in a performance benefit.

a. True
b. False

Summary

In this lesson, you should have learned how to:

- Use atomic variables
- Use a `ReentrantReadWriteLock`
- Use the `java.util.concurrent` collections
- Describe the synchronizer classes
- Use an `ExecutorService` to concurrently execute tasks
- Apply the Fork-Join framework