

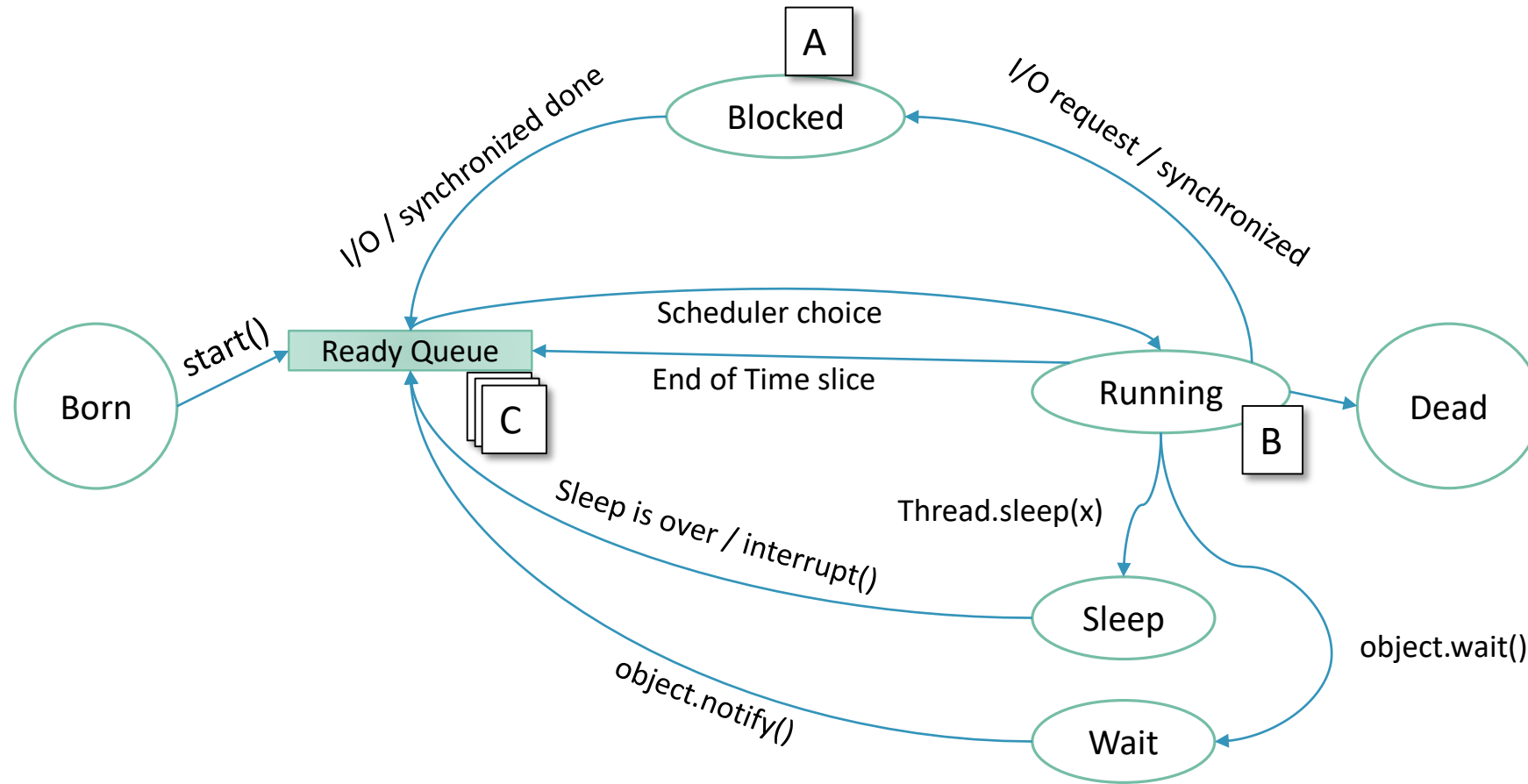


Advanced Programming 2 - Concurrency Patterns

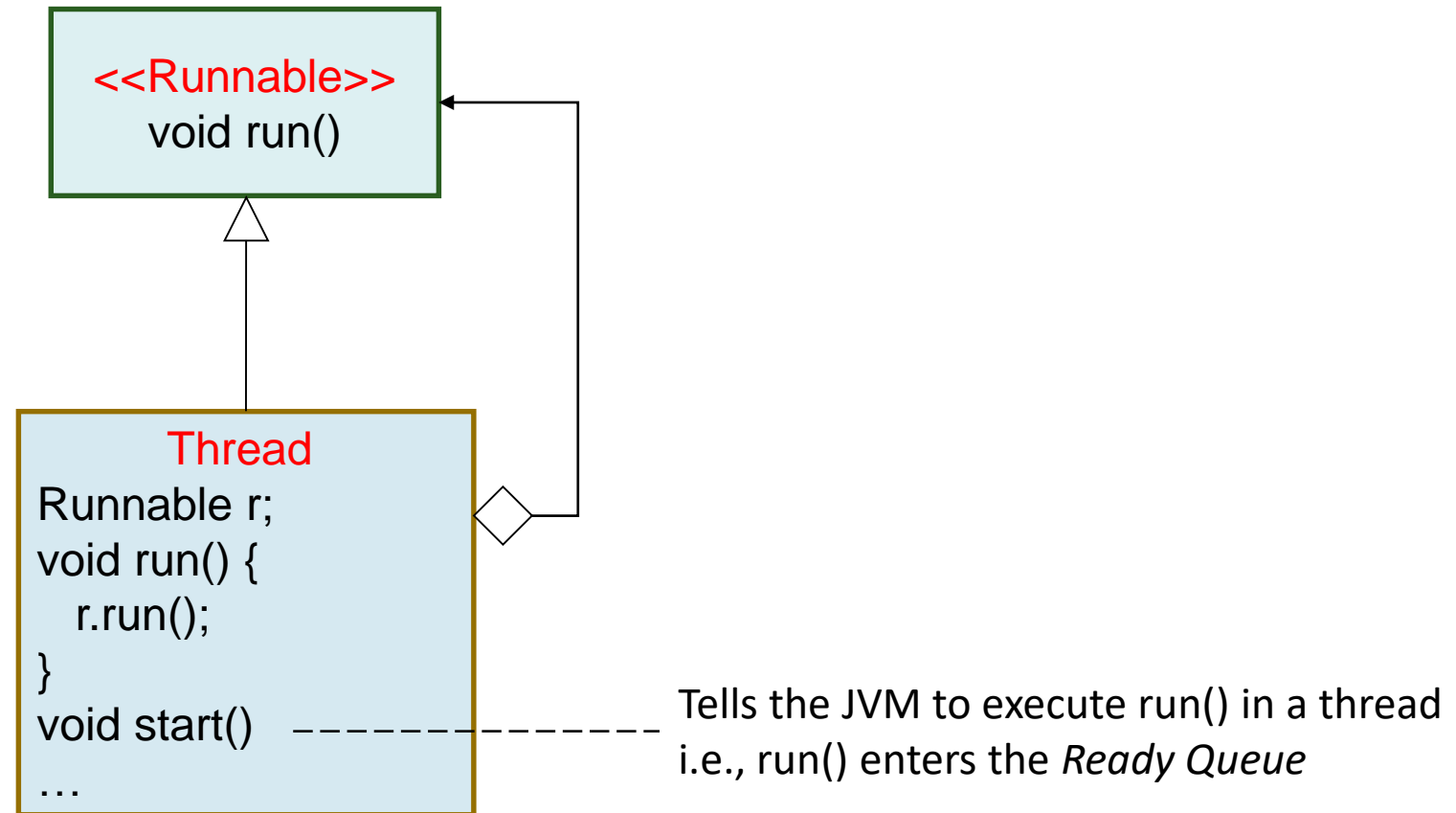
DR. ELIAHU KHALASTCHI

2016

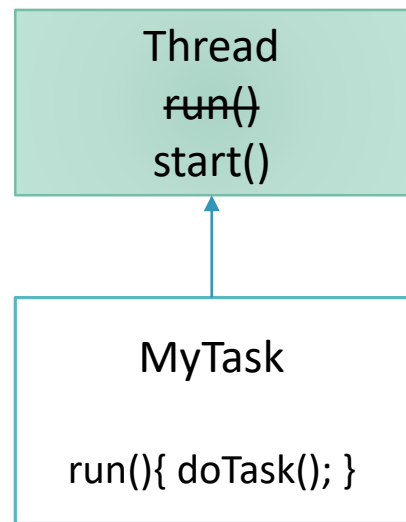
The Thread Life Cycle



Thread & Runnable



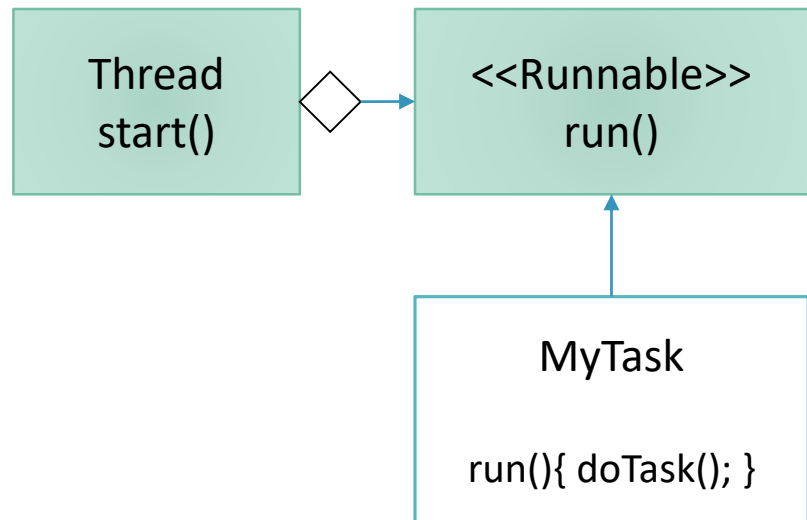
Option 1: extending Thread



1. Extended the Thread class
2. Override the run() method
3. Call start to execute in parallel

But sometimes our class is not a type of Thread or it already extends something else

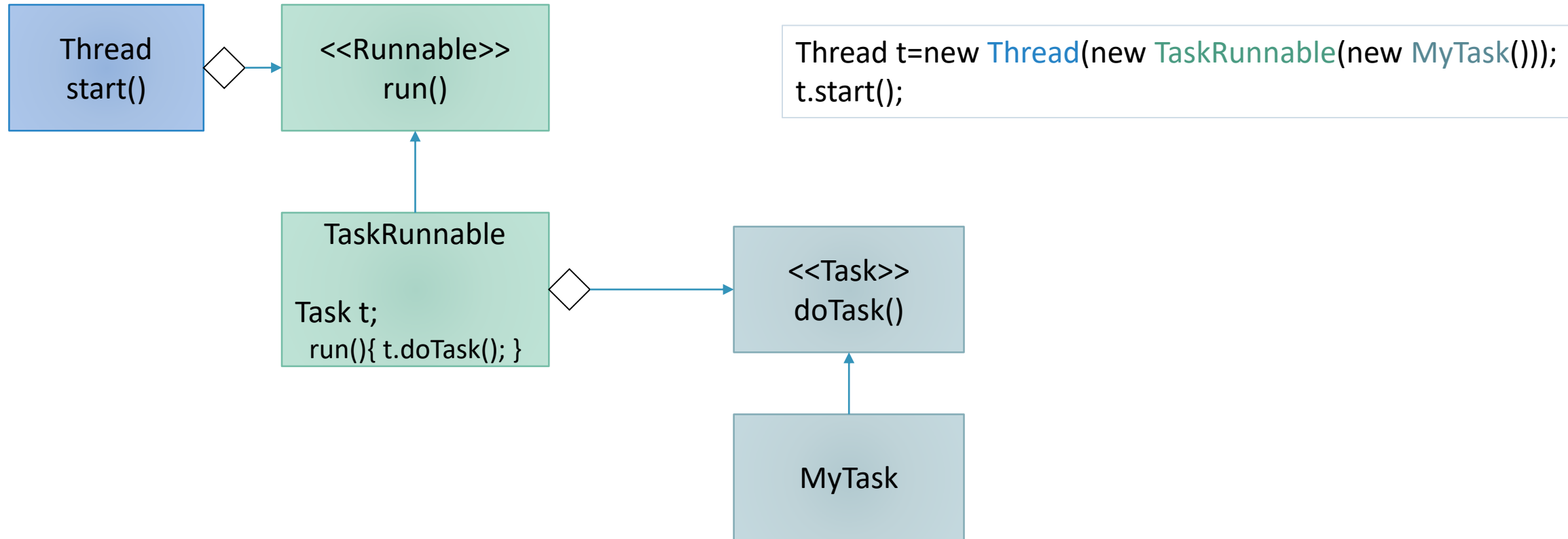
Option 2: implementing Runnable



1. Implement the Runnable interface
2. Create an instance of Thread
3. Inject the Runnable
4. Call start

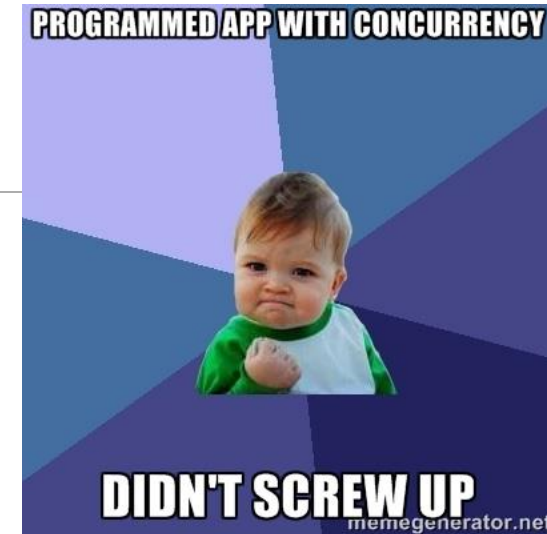
This is a typical strategy pattern, but what if we don't want to (or can't) change MyTask?

Option 3: using object adapters!





Concurrency Design Patterns



Active Object

Active Object

- Decouples method execution from method invocation
- for objects that each reside in their own thread of control
- The goal is to introduce concurrency,
 - by using asynchronous method invocation
 - and a scheduler for handling requests

Example

```
class MyModel implements Model{

    Maze maze;
    Solution solution;

    void generateMaze () {
        maze=MazeGenerator.generateMaze (/**/);
    }

    void solve(Maze m) {
        solution=searcher.search(m);
    }
}
```

Not an active object
Method invocation is coupled to execution

Example

```
class MyActiveModel implements Model {

    Maze maze;
    Solution solution;
    BlockingQueue<Runnable> dispatchQueue
        = new LinkedBlockingQueue<Runnable>();

    public MyActiveModel() {

        new Thread(new Runnable() {
            public void run() {
                while (true) {
                    try {
                        // take() blocks, so no busy waiting
                        dispatchQueue.take().run();
                    } catch (InterruptedException e) {}
                }
            }
        }).start();

    }
}
```

AMI – asynchronous method invocation

```
void generateMaze() throws InterruptedException {
    dispatchQueue.put(new Runnable() {
        public void run() {
            maze = MazeGenerator.generateMaze(/**/);
        }
    });
}

void solve(Maze m) throws InterruptedException {
    dispatchQueue.put(new Runnable() {
        public void run() {
            solution = searcher.search(m);
        }
    });
}
```

Double-checked locking

Double-checked locking

- Goal: to reduce the overhead of acquiring a lock
 - by first testing the locking
 - without actually acquiring the lock
- Only if the locking is required then do the actual locking

Example - Singleton

```
class Foo {  
    private Helper helper;  
    public Helper getHelper() {  
        if (helper == null) {  
            helper = new Helper();  
        }  
        return helper;  
    }  
}
```

Not Thread-Safe

Example - Singleton

```
class Foo {  
    private Helper helper;  
    public synchronized Helper getHelper() {  
        if (helper == null) {  
            helper = new Helper();  
        }  
        return helper;  
    }  
}
```

Expensive

Example - Singleton

Not Expensive

```
class Foo {  
    private Helper helper;  
    public Helper getHelper() {  
        if (helper == null) {  
            synchronized(this) {  
                if (helper == null) {  
                    helper = new Helper();  
                }  
            }  
        }  
        return helper;  
    }  
}
```

But its not completely thread-safe ☹️

Example - Singleton

```
class Foo {  
    private Helper helper;  
    public Helper getHelper() {  
        if (helper == null) {  
            synchronized(this) {  
                if (helper == null) {  
                    helper = new Helper();  
                }  
            }  
        }  
        return helper;  
    }  
}
```

← Thread B

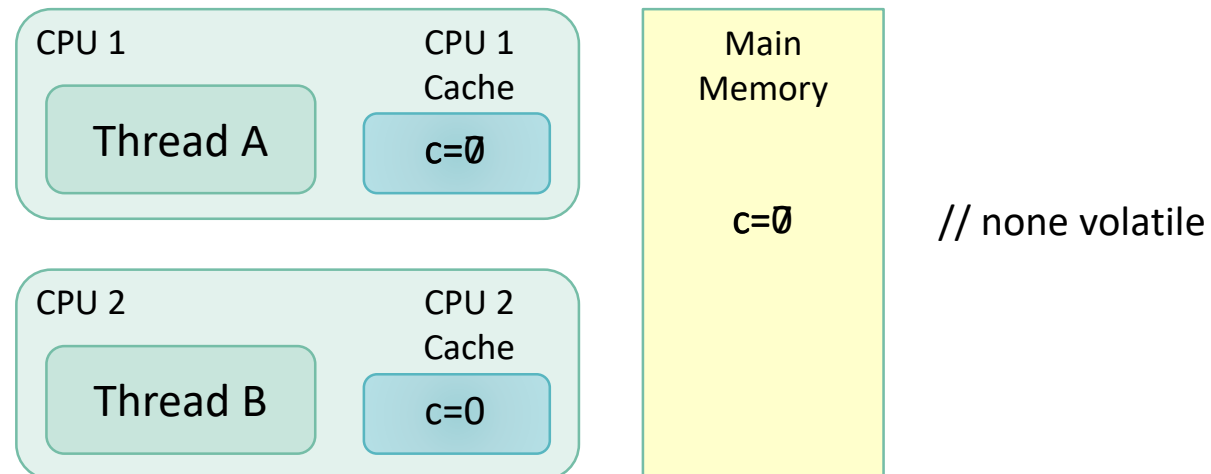
← Thread A

helper

Helper

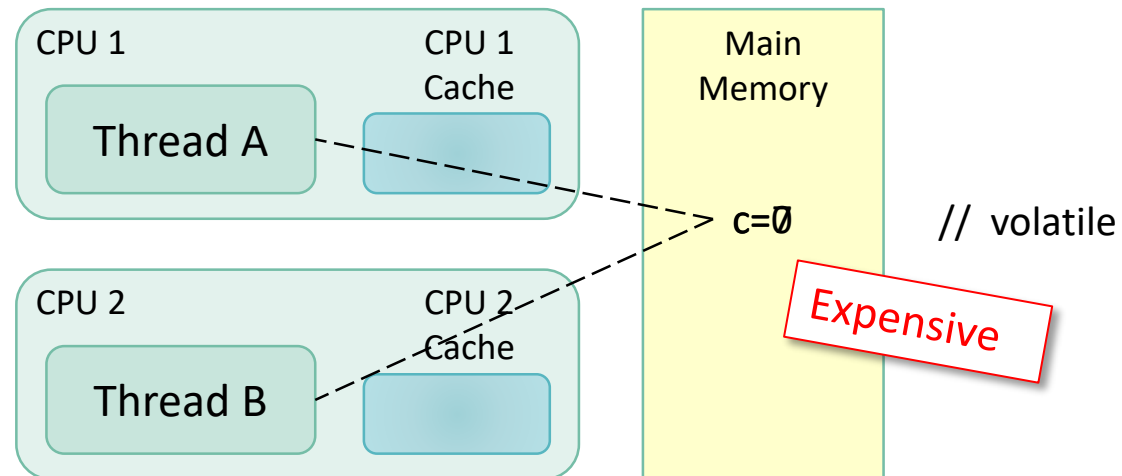
Volatile

- Every **read & write** to a **volatile** variable will be on the **main memory**
 - **Not** the CPU cache...



Volatile

- Every **read & write** to a **volatile** variable will be on the **main memory**
 - **Not** the CPU cache...



Volatile: *Happens-Before* Guarantee

- Every **read & write** to a **volatile** variable will be on the **main memory**
 - **Not** the CPU cache...
- When a thread reads or writes to a volatile variable
 - all other **dependent** variables are flushed to main memory as well
- Reading and writing instructions **cannot be reordered** by the JVM

Example - Singleton

```
class Foo {  
    private volatile Helper helper;  
    public Helper getHelper() {  
        if (helper == null) {  
            synchronized(this) {  
                if (helper == null) {  
                    helper = new Helper();  
                }  
            }  
        }  
        return helper;  
    }  
}
```

Expensive

← Thread B

Expensive

← Thread A

helper = null

Helper

Example - Singleton

```
class Foo{
    private volatile Helper helper;
    public Helper getHelper() {
        Helper result = helper;
        if (result == null) {
            synchronized(this) {
                result = helper;
                if (result == null) {
                    helper = result = new Helper();
                }
            }
        }
        return result;
    }
}
```

Expensive

Not Expensive

As much as 25% performance improvement

Another solution for concurrent Singleton

Example - Singleton

```
class Foo{  
    private static final Helper helper = new Helper();  
    public static Helper getHelper() {  
        return helper;  
    }  
}
```

Not Expensive

"Eager" instead of "Lazy"

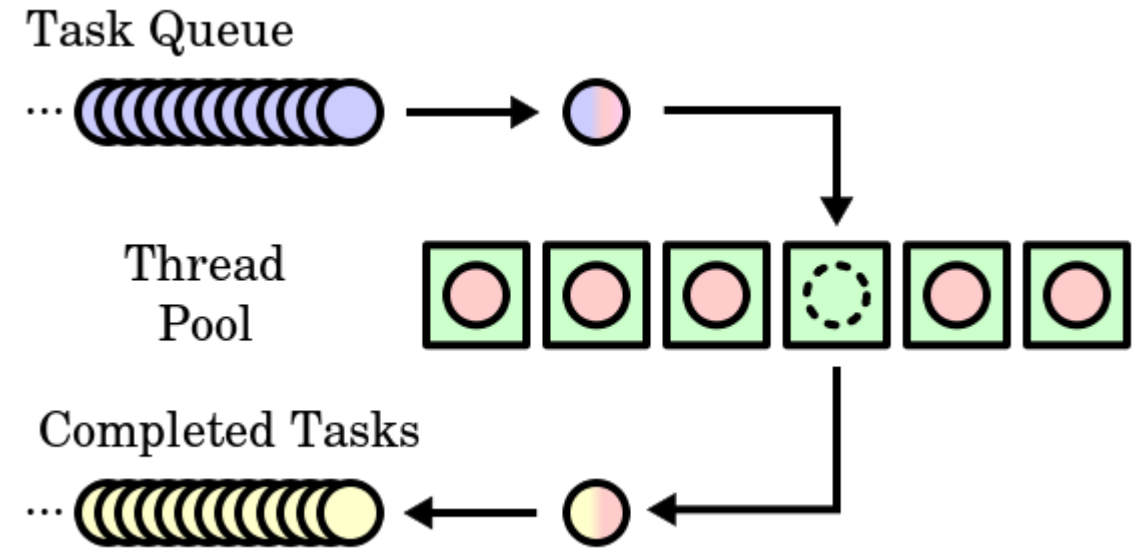
Example - Singleton

```
class Foo{  
    private static class HelperHolder {  
        public static final Helper helper = new Helper();  
    }  
  
    public static Helper getHelper() {  
        return HelperHolder.helper;  
    }  
}
```

Not Expensive

inner classes are not loaded until they are referenced

Thread Pool



Executor Implementations Example

```
interface Executor {  
    void execute(Runnable r);  
}
```

```
class DirectExecutor implements Executor{  
    public void execute(Runnable r) {  
        r.run();  
    }  
}
```

```
class ThreadPerTaskExecutor implements Executor{  
    public void execute(Runnable r) {  
        new Thread(r).start();  
    }  
}
```

And if we wanted to control the number of threads?

Thread Pools Example

```
public class RunnableTask1 implements Runnable{
    public void run(){
        System.out.println("task1 started");
        try { Thread.sleep(10000);}
        catch (InterruptedException e) {}
        System.out.println("task1 finished");
    }
}
// RunnableTask2 & RunnableTask3 are the same...
```

```
task1 started
task2 started
task1 finished
task2 finished
task3 started
task3 finished
```

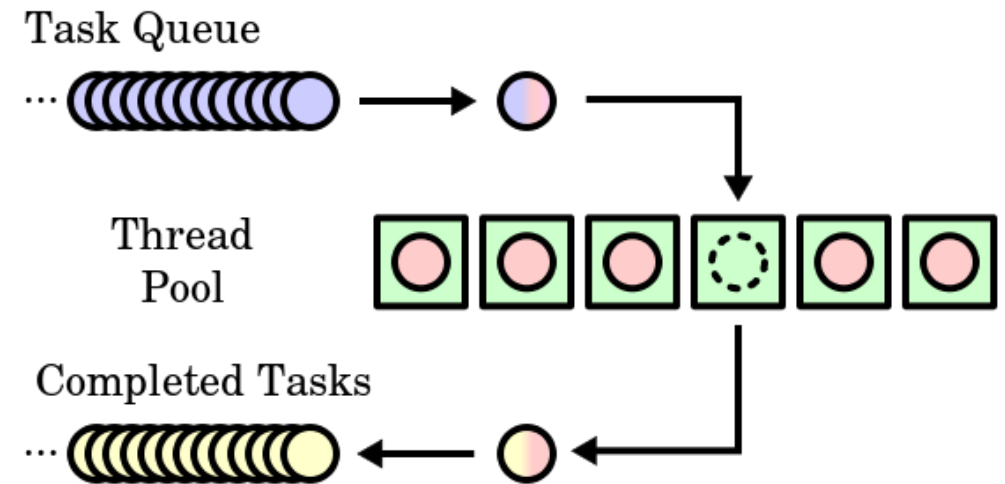
```
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
//...
public static void main(String[] args) {
    ExecutorService executor =
        Executors.newFixedThreadPool(2);
    executor.execute (new RunnableTask1 ());
    executor.execute (new RunnableTask2 ());
    executor.execute (new RunnableTask3 ());
}
```

Thread Pool

- Control the number of threads
- No thread creation / destruction overhead

```
// a thread that can run task after task
class PooledThread extends Thread{
    Runnable task;
    Object lock;
    boolean terminated=false;

    public void assignTask(Runnable r){
        task=r;
        unSuspendMe();
    }
    public void run(){
        while(!terminated){
            task.run();
            suspendMe();
        }
    } // the pooled thread dies
    // ...
}
```



AMI – Asynchronous Method Invocation

- Doesn't block the calling thread while waiting for a reply
 - Instead, the calling thread is notified when the reply arrives
 - Polling for a reply is an undesired option.
-
- One common use of AMI is in the ***active object*** design pattern
 - Alternatives are synchronous method invocation and ***future objects***.

Callable

- Runnable's run() method
 - Cannot return a value
 - Cannot throw an exception
- A Callable Interface can
- ExecutorService can
 - **execute**(Runnable r); // as we have seen
 - **submit**(Callable c);
 - It puts the callable in the thread pool and immediately returns
 - What can be returned by submit?

```
interface Callable<V> {  
    V call() throws Exception;  
}
```

The problem

```
public class MyCallable implements Callable<Worker>{  
  
    Worker call() throws Exception{  
        // after 10 minutes or so...  
        return someworker;  
    }  
}
```

```
ExecutorService executor = Executors. newFixedThreadPool (2);  
_____ = executor.submit (new MyCallable ());
```

1. The submit() method was written years ago... the Worker class was created just now...
2. submit() should return a value now! And not in 10 minutes

The Solution – Future!

- Future is a holder for a value of type $\langle V \rangle$
- The submit method returns immediately an instance of Future
 - *Future<V> submit(Callable<V> callable);*
 - We should define the same V in the Callable and the Future
- When the Callable's call() returns $\langle V \rangle$ it is set in the instance of Future
- Only then, we may get $\langle V \rangle$

Future $\langle V \rangle$
V value;
set(V v); V get();

The Solution – Future!

```
public class MyCallable implements Callable<Worker>{  
  
    Worker call() throws Exception{  
        // after 10 minutes or so...  
        return someworker;  
    }  
}
```

Future <V>
V value;
set(V v); V get();

```
ExecutorService executor = Executors. newFixedThreadPool (2);
```

```
Future<Worker> f = executor.submit (new MyCallable ());
```

```
// ...
```

```
Worker w = f.get(); // waits for the call() to return
```

Guarded suspension pattern 

Guarded Suspension

Guarded Suspension

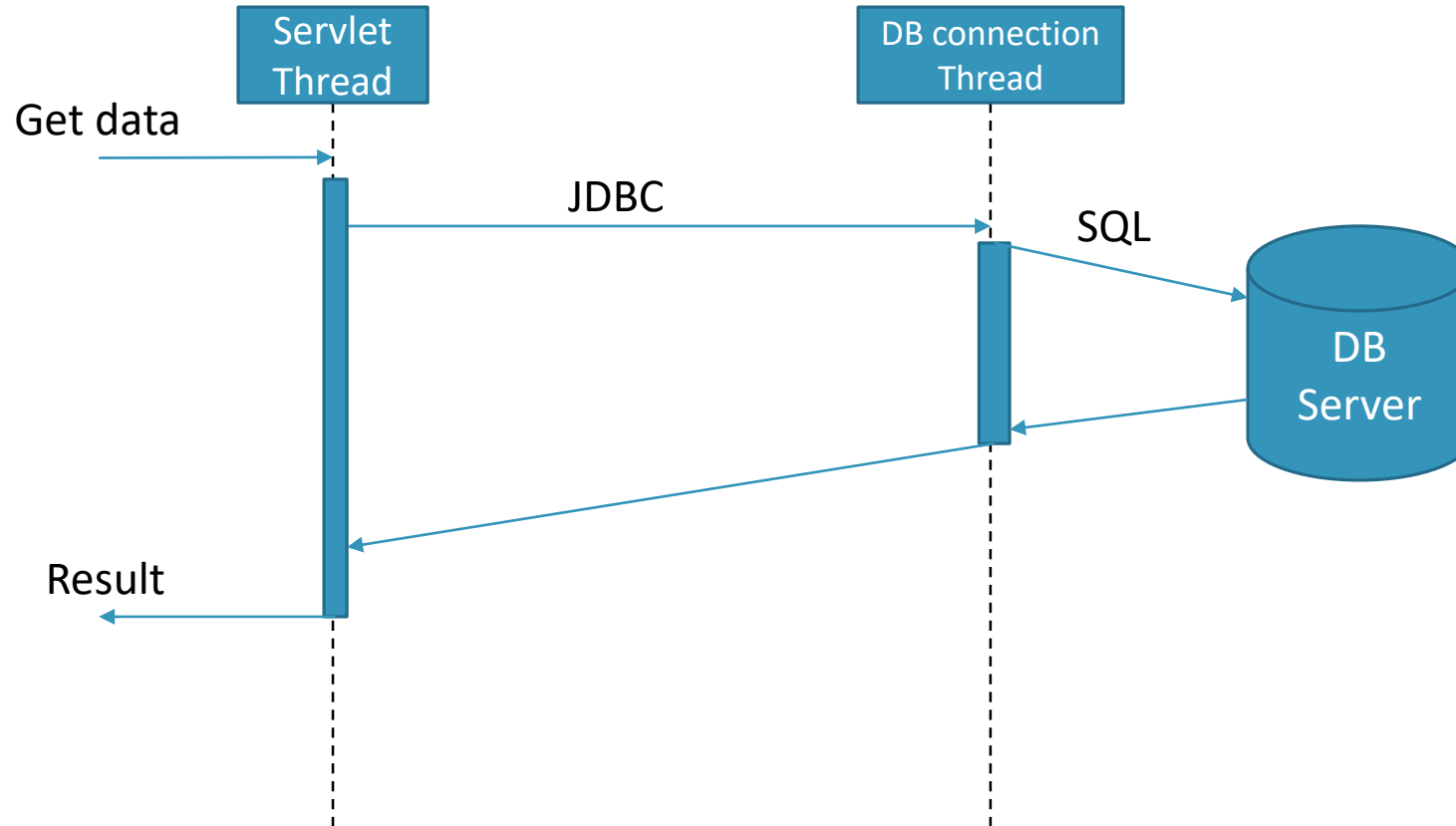
- Manages operations that require both
 - a lock to be acquired
 - and a precondition to be satisfied
- before the operation can be executed

```
public class GameCharacter {  
    boolean victory;  
    int score;  
  
    synchronized void victoryDance() { // guarded method  
        while (!victory) {  
            try { wait(); } catch (InterruptedException e) {}  
        }  
        // Actual task implementation  
        // victory dance!!  
    }  
  
    synchronized void updateScore(int x) {  
        // ...  
        // Inform waiting threads  
        notify();  
    }  
}
```

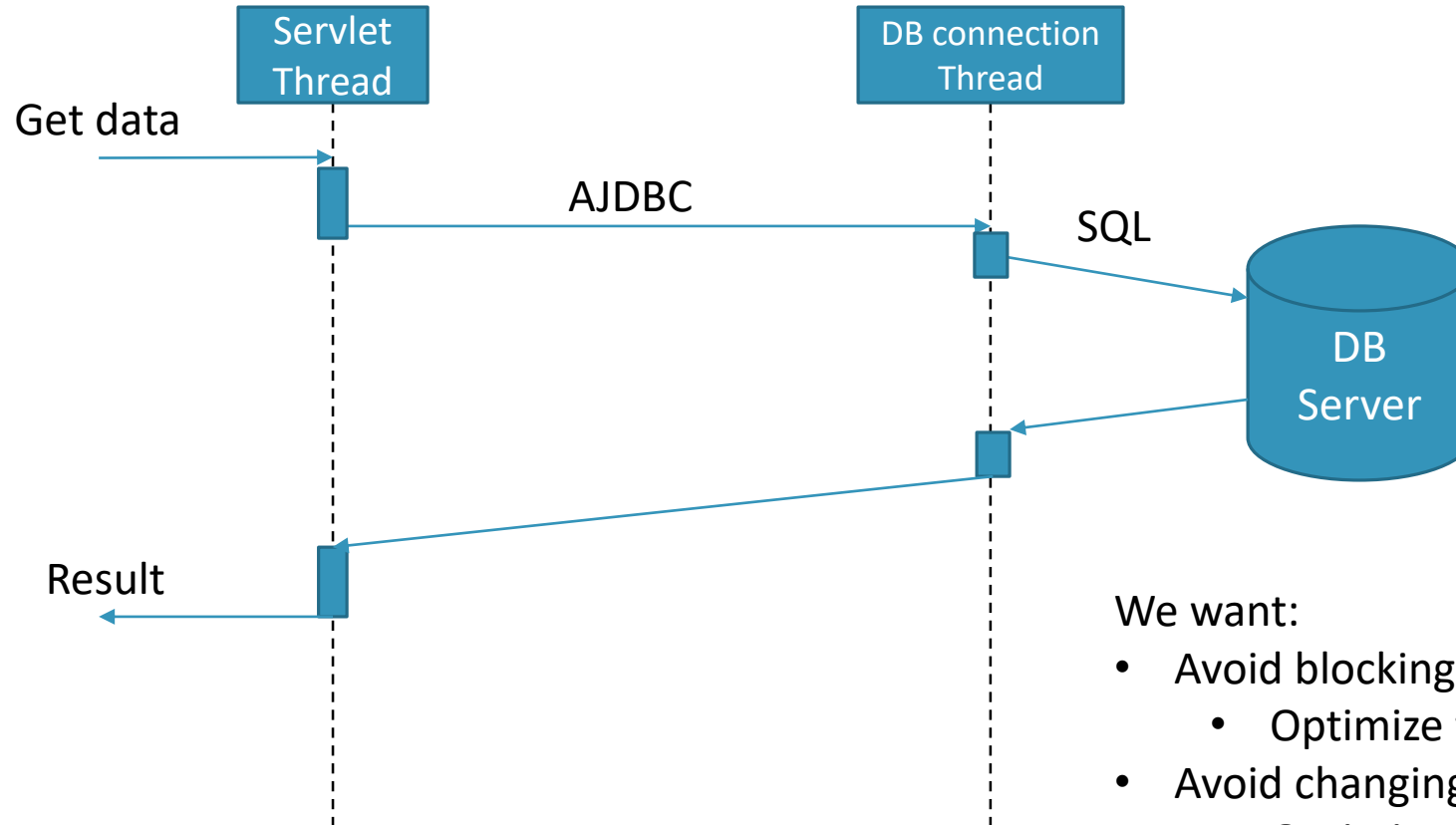
CompletableFuture

JAVA 8

Blocking (yet asynchronous)



Non-Blocking



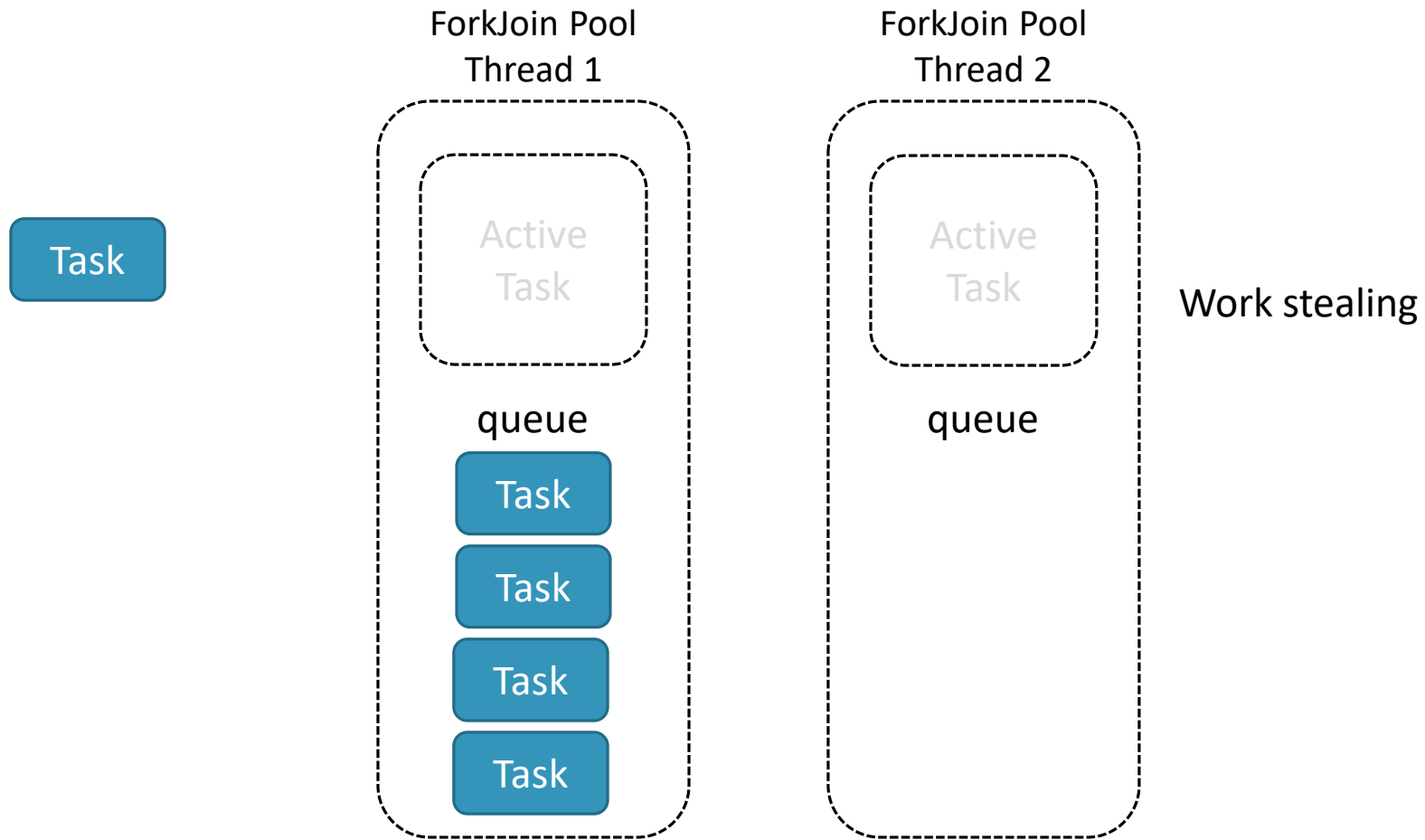
We want:

- Avoid blocking
 - Optimize the use of a multicore
- Avoid changing threads
 - Optimize the use of cache

Fork-Join Pool

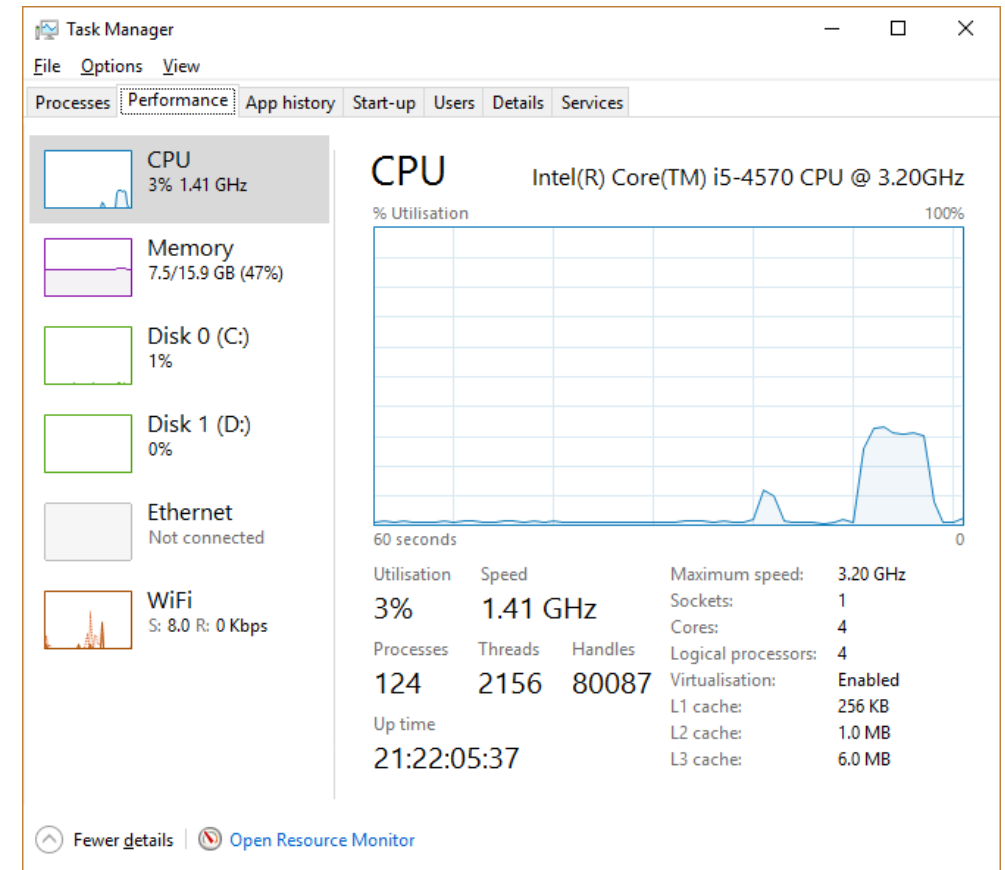
JAVA 7

ForkJoin Pool (JDK 7)



Fibonacci Example

```
public class Fib {  
  
    int num;  
    public Fib(int num) {  
        this.num=num;  
    }  
  
    public int compute(){  
        if(num<=1)  
            return num;  
        Fib fib1= new Fib(num-1);  
        Fib fib2= new Fib(num-2);  
        return fib2.compute()+fib1.compute();  
    }  
    public static void main(String[] args) {  
        System.out.println(new Fib(45).compute());  
    }  
}
```

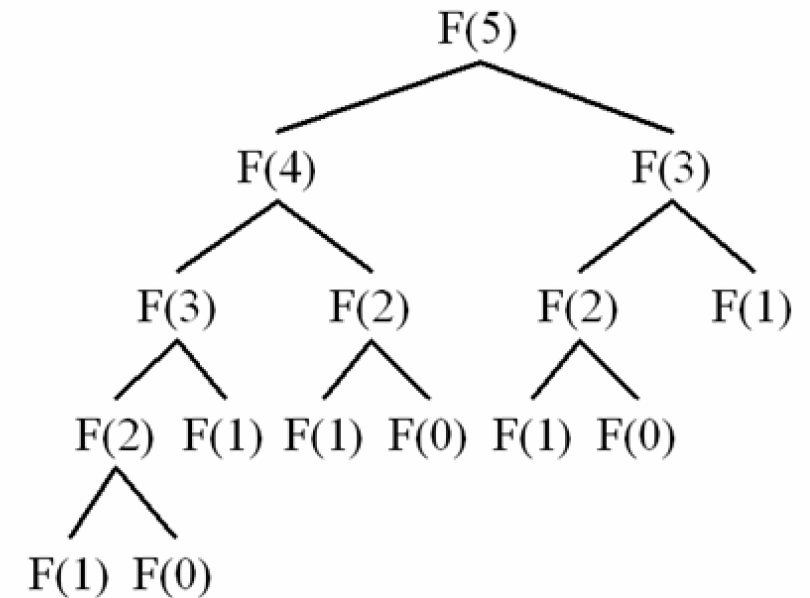


Fibonacci + Dynamic Programming

```
public class Fib_DP { // without concurrency
    // but with dynamic programming
    static HashMap<Integer,Integer> fibs=new HashMap<>();

    int num;
    public Fib_DP(int num) { this.num=num;}

    public int compute(){ // a recursive task
        if(num<=1)
            return num;
        if(fibs.get(num)!=null)
            return fibs.get(num);
        Fib_DP fib1= new Fib_DP(num-1);
        Fib_DP fib2= new Fib_DP(num-2);
        int result=fib2.compute()+fib1.compute();
        fibs.put(num,result);
        return result;
    }
    public static void main(String[] args) {
        System.out.println(new Fib_DP(2048).compute());
    }
}
```



However, we wish to simulate a multithreaded task

Fibonacci + Thread Pool (JDK 6)

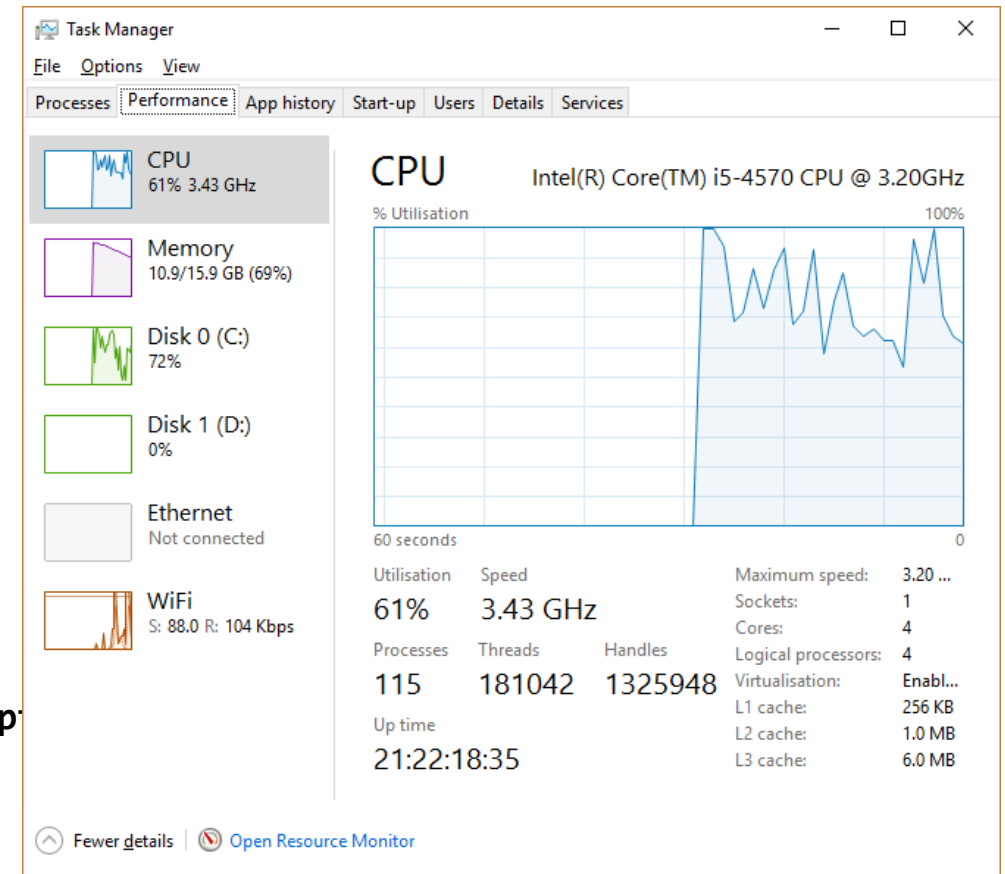
```
public class Fib_TP implements Callable<Integer>{

    static ExecutorService es=Executors.newCachedThreadPool();

    int num;
    public Fib_TP(int num) {this.num=num;}

    @Override
    public Integer call() throws Exception {
        if(num<=1)
            return num;
        Future<Integer> fib1 = es.submit(new Fib_TP(num-1));
        Future<Integer> fib2 = es.submit(new Fib_TP(num-2));
        return fib2.get()+fib1.get();
    }

    public static void main(String[] args) throws InterruptedException{
        Future<Integer> f=es.submit(new Fib_TP(45));
        System.out.println(f.get());
    }
}
```

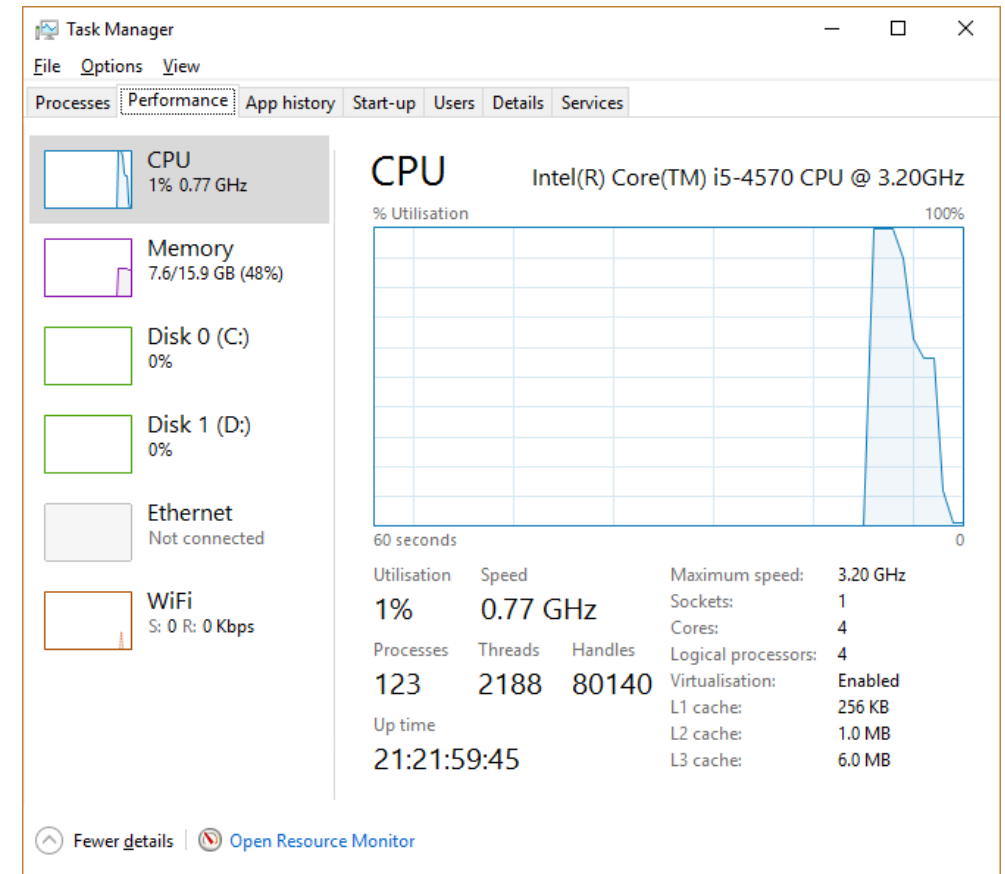


Fibonacci + Fork-Join Pool (JDK 7)

```
public class Fib_FJ extends RecursiveTask<Integer>{
    // with fork-join pool
    int num;
    public Fib_FJ(int num) { this.num=num; }

    @Override
    public Integer compute(){ // a recursive task
        if(num<=1)
            return num;
        Fib_FJ fib1= new Fib_FJ(num-1);
        fib1.fork();
        Fib_FJ fib2= new Fib_FJ(num-2);
        return fib2.compute()+fib1.join();
    }

    public static void main(String[] args) {
        Fib_FJ fib=new Fib_FJ(45);
        ForkJoinPool pool = new ForkJoinPool();
        System.out.println(pool.invoke(fib));
    }
}
```



Since JDK 5 – Callable & Future



```
public String deepThought() {  
    // takes a really really long time...  
    return "42";  
}
```

```
ExecutorService executor=Executors.newCachedThreadPool();  
  
Future<String> f = executor.submit(new Callable<String>() {  
    @Override  
    public String call() throws Exception {  
        return deepThought();  
    }  
});
```

```
//...  
System.out.println(f.get()); // blocks until an answer is given
```

Back to deep thought...



```
public String deepThought() {  
    // takes a really really long time...  
    return "42";  
}
```

```
ExecutorService executor=Executors.newCachedThreadPool();  
  
Future<String> f = executor.submit( ()-> {  
    return deepThought();  
});
```

Still, resources are wasted because of the blocking get() call

```
//...  
System.out.println(f.get()); // blocks until an answer is given
```

Using CompletableFuture

```
public String deepThought() {  
    // takes a really really long time...  
    return "42";  
}
```

```
ExecutorService executor=Executors.newCachedThreadPool();  
  
// an asynchronous call  
CompletableFuture.supplyAsync( ()->{  
    return deepThought();  
}, executor);
```



Using CompletableFuture

```
public String deepThought() {  
    // takes a really really long time...  
    return "42";  
}
```

```
// an asynchronous call  
CompletableFuture.supplyAsync( () -> {  
    return deepThought();  
});
```

Uses the default ForkJoin Pool



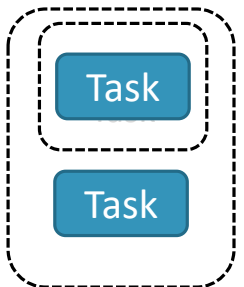
Adding callbacks (instead of blocking)



```
public String deepThought() {  
    // takes a really really long time...  
    return "42";  
}
```

```
CompletableFuture<String> fc = CompletableFuture.supplyAsync( () -> {  
    return deepThought();  
});  
  
fc.thenAccept( (String answer) -> {System.out.println("answer: "+answer);});
```

Reactive pattern: This action will be taken right after deep thought is finished



Adding callbacks



```
public String deepThought() {  
    // takes a really really long time...  
    return "42";  
}
```

```
CompletableFuture<String> fc = CompletableFuture.supplyAsync( () -> {  
    return deepThought();  
});  
  
fc.thenAccept( answer -> System.out.println("answer: "+answer));
```

Reactive pattern: This action will be taken right after deep thought is finished

Fluent Programming: each method returns its object, allowing chained calls

Returns
CompletableFuture<String>

Adding callbacks

```
public String deepThought() {  
    // takes a really really long time...  
    return "42";  
}
```



```
CompletableFuture.supplyAsync( () -> {return deepThought();})  
    .thenApply(answer -> Integer.parseInt(answer))  
    .thenApply(x -> x*2)  
    .thenAccept(answer -> System.out.println("answer: "+answer));
```

Adding callbacks



```
public String deepThought() {  
    // takes a really really long time...  
    return "42";  
}
```

```
CompletableFuture.supplyAsync( () -> {return deepThought();}, executor)  
    .thenApply(answer -> Integer.parseInt(answer))  
    .then
```

- `thenAccept(Consumer<? super Void> action) : CompletableFuture<Void> - CompletableFuture`
- `thenAcceptAsync(Consumer<? super Void> action) : CompletableFuture<Void> - CompletableFuture`
- `thenAcceptAsync(Consumer<? super Void> action, Executor executor) : CompletableFuture<Void> - CompletableFuture`
- `thenAcceptBoth(CompletionStage<? extends U> other, BiConsumer<? super Void,? super U> action) : CompletableFuture<Void> - CompletableFuture`
- `thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super Void,? super U> action) : CompletableFuture<Void> - CompletableFuture`
- `thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super Void,? super U> action, Executor executor) : CompletableFuture<Void> - CompletableFuture`
- `thenApply(Function<? super Void,? extends U> fn) : CompletableFuture<U> - CompletableFuture`
- `thenApplyAsync(Function<? super Void,? extends U> fn) : CompletableFuture<U> - CompletableFuture`
- `thenApplyAsync(Function<? super Void,? extends U> fn, Executor executor) : CompletableFuture<U> - CompletableFuture`
- `thenCombine(CompletionStage<? extends U> other, BiFunction<? super Void,? super U,? extends V> fn) : CompletableFuture<V> - CompletableFuture`
- `thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super Void,? super U,? extends V> fn) : CompletableFuture<V> - CompletableFuture`
- `thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super Void,? super U,? extends V> fn, Executor executor) : CompletableFuture<V> - CompletableFuture`
- `thenCompose(Function<? super Void,? extends CompletionStage<U>> fn) : CompletableFuture<U> - CompletableFuture`
- `thenComposeAsync(Function<? super Void,? extends CompletionStage<U>> fn) : CompletableFuture<U> - CompletableFuture`
- `thenComposeAsync(Function<? super Void,? extends CompletionStage<U>> fn, Executor executor) : CompletableFuture<U> - CompletableFuture`
- `thenRun(Runnable action) : CompletableFuture<Void> - CompletableFuture`
- `thenRunAsync(Runnable action) : CompletableFuture<Void> - CompletableFuture`
- `thenRunAsync(Runnable action, Executor executor) : CompletableFuture<Void> - CompletableFuture`

Press 'Ctrl+Space' to show Template Proposals

Please look at

- New Concurrency Utilities in Java 8
 - https://www.youtube.com/watch?v=Q_0_1mKTlnY
- How to use CompletableFuture
 - https://www.youtube.com/watch?v=HdnHmbFg_hw
- Reactive Programming patterns
 - <https://www.youtube.com/watch?v=tiJEL3oiHIY>
- Disruptor Pattern
 - <https://www.youtube.com/watch?v=DCdGlxBbKU4>
 - <https://disruptor.googlecode.com/files/Disruptor-1.0.pdf>