Computer Science School
המסלול האקדמי
המכללה למינהל

Advanced Software Development 2 – concurrent design patterns

Dr. Eliahu Khalastchi

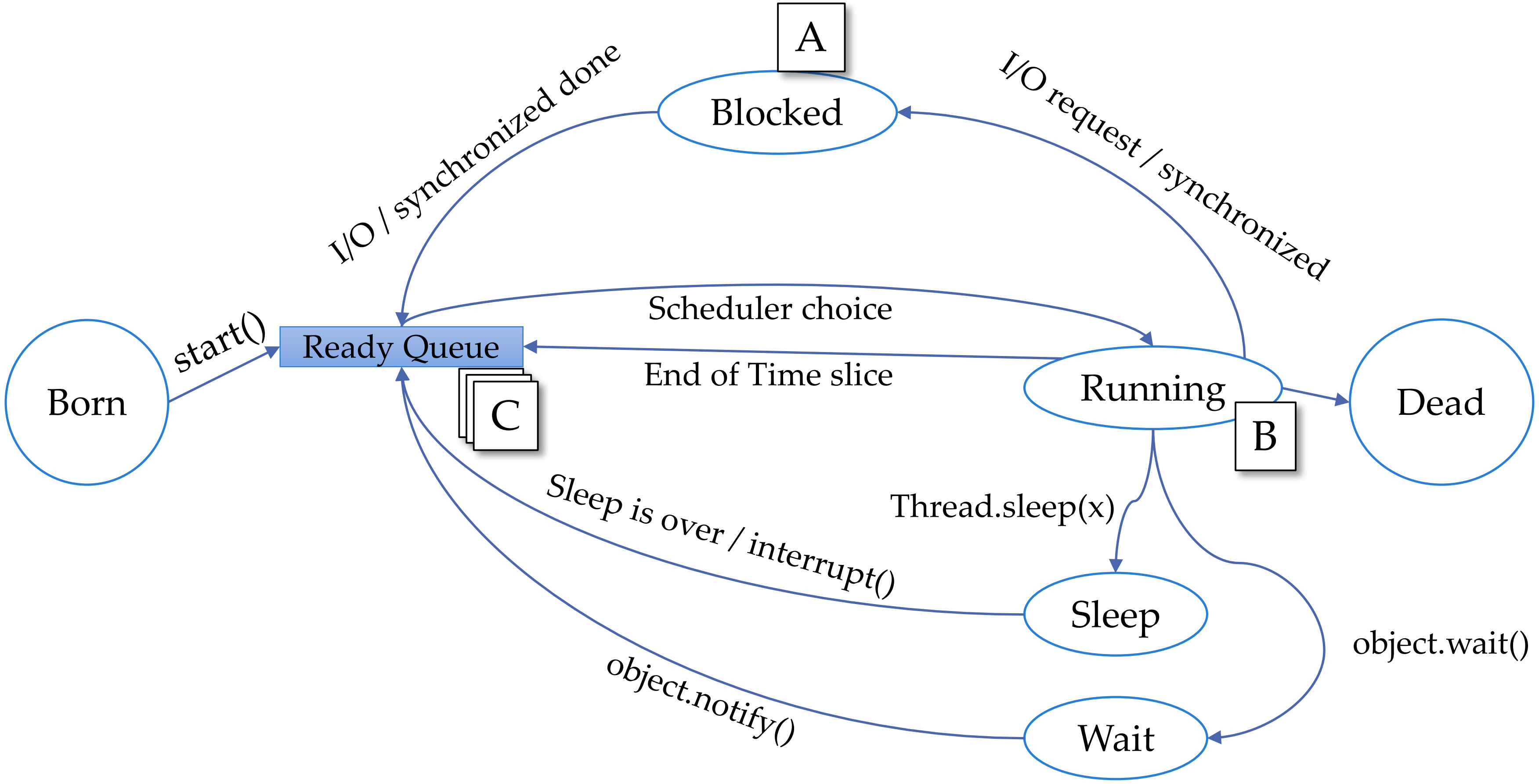
2017

המסלול האקדמי
המכללה למינהל

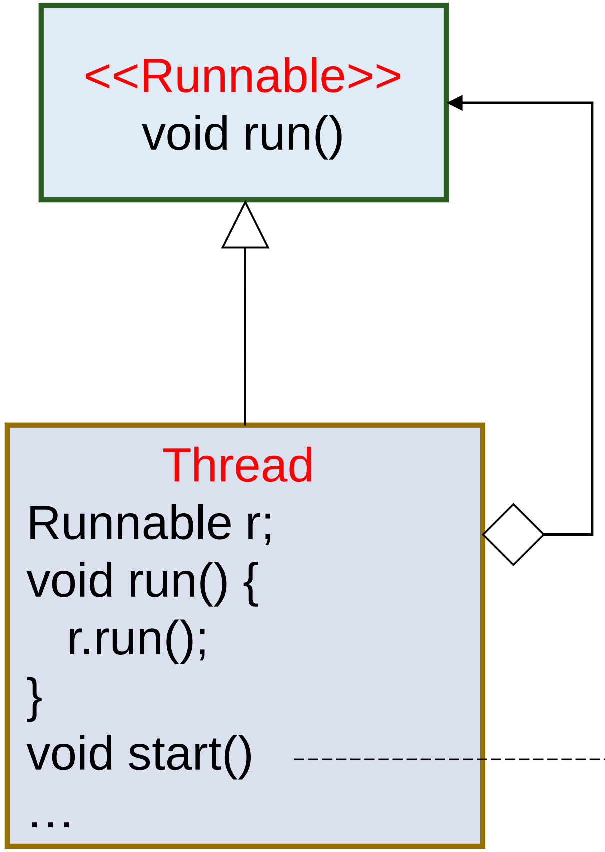
Reminder about Threads

In Java

The Thread Life Cycle

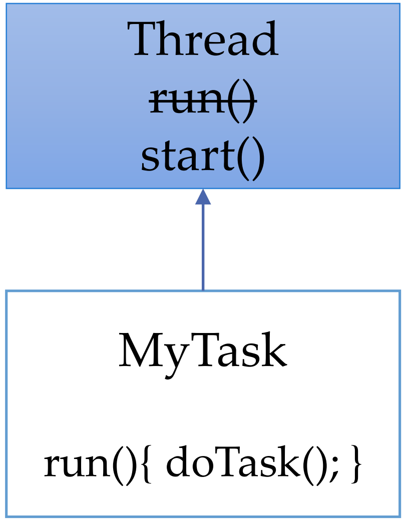


Thread & Runnable



Tells the JVM to execute run() in a thread
i.e., run() enters the *Ready Queue*

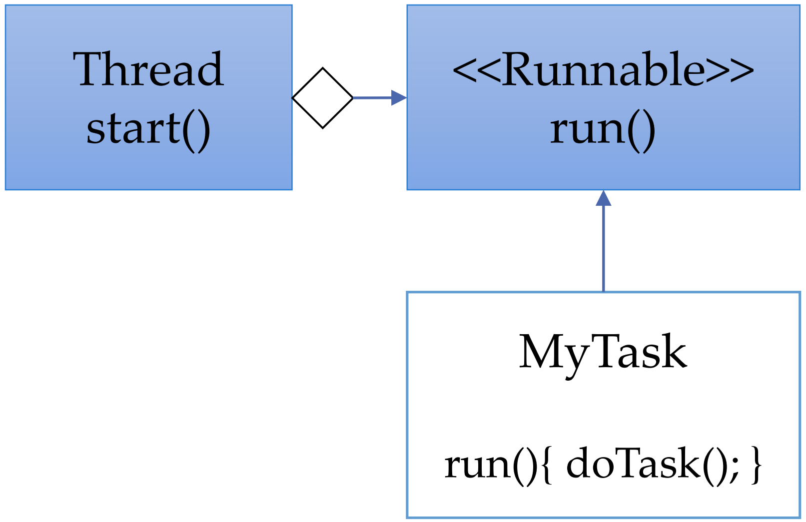
Option 1: extending Thread



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

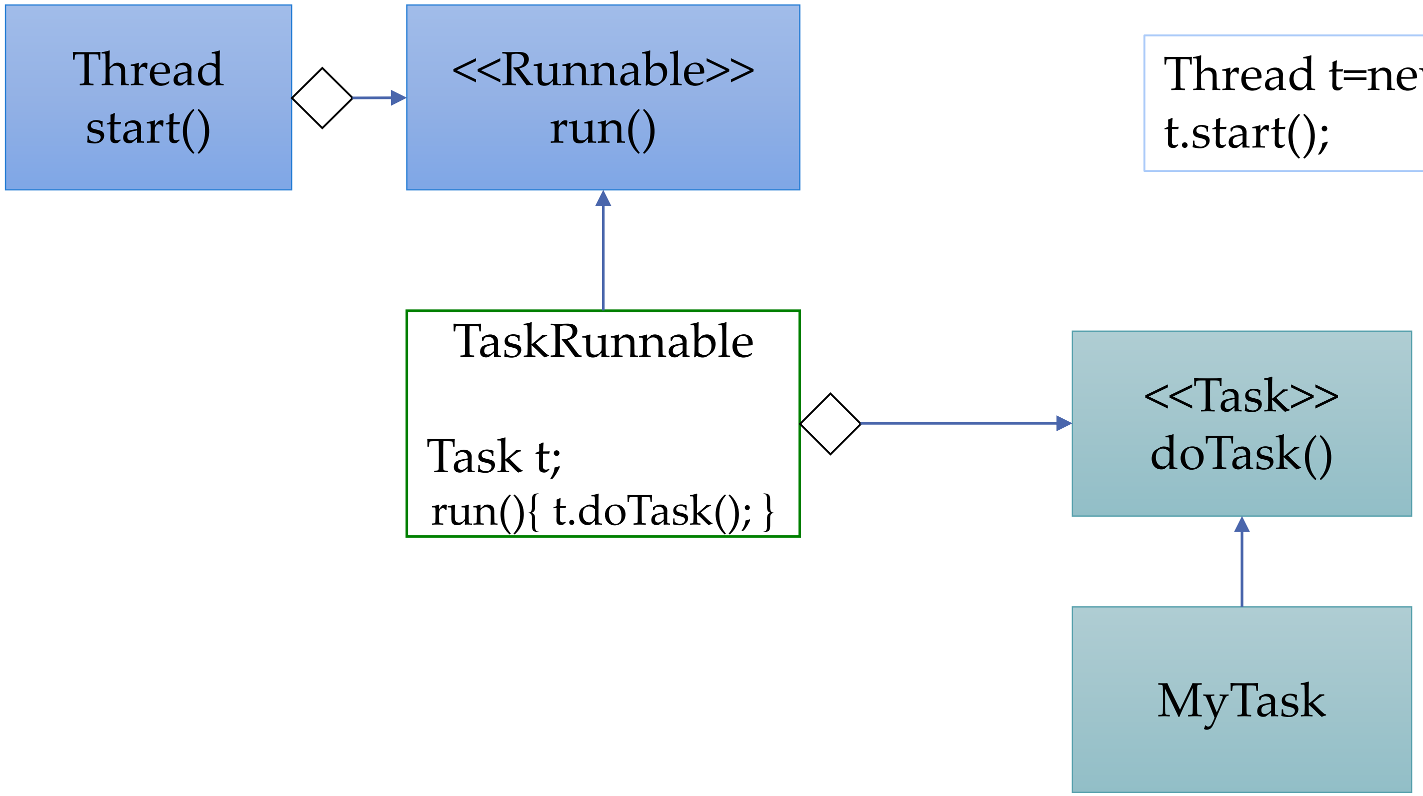
Option 2: implementing Runnable



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

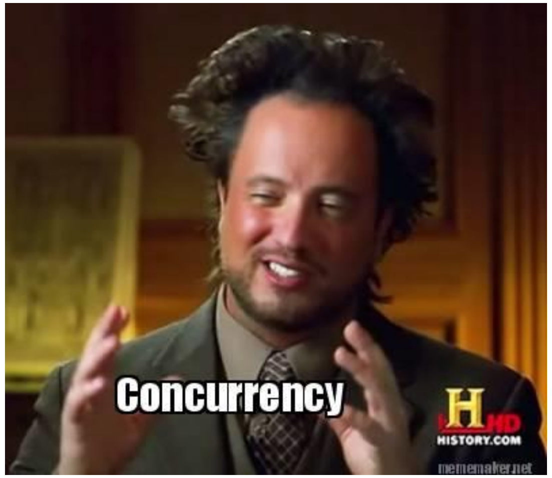
This is a typical strategy pattern, but what if we don't want to (or can't) change MyTask?

Option 3: using object adapters!




```
Thread t=new Thread(new TaskRunnable(new MyTask()));
t.start();
```

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

```java
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

AMI – asynchronous method invocation

```
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();

  }
```

```
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

Not Thread-Safe

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

Example - Singleton

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

Expensive

Example - Singleton

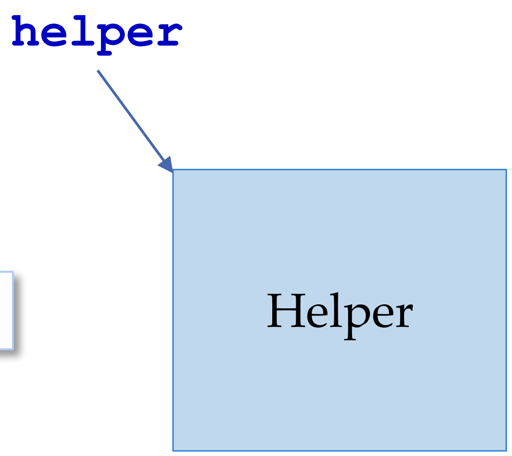
Not Expensive

But its not completely thread-safe ☹

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

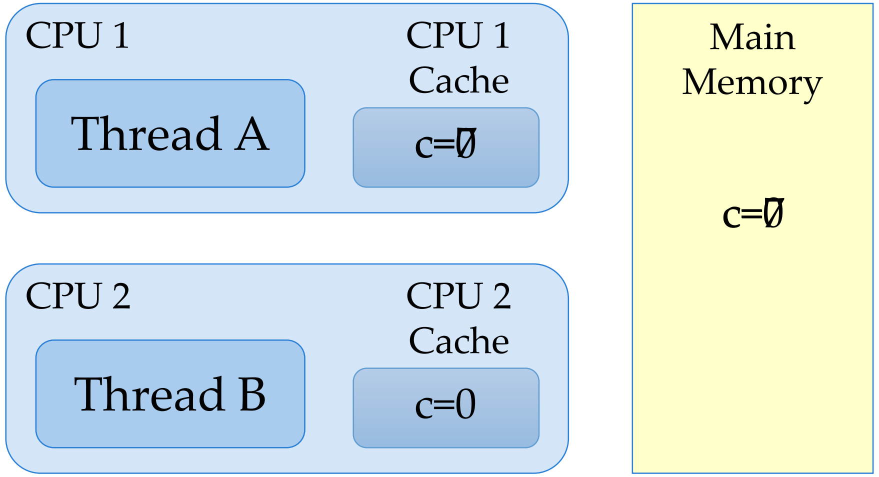
Example - Singleton

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



Volatile

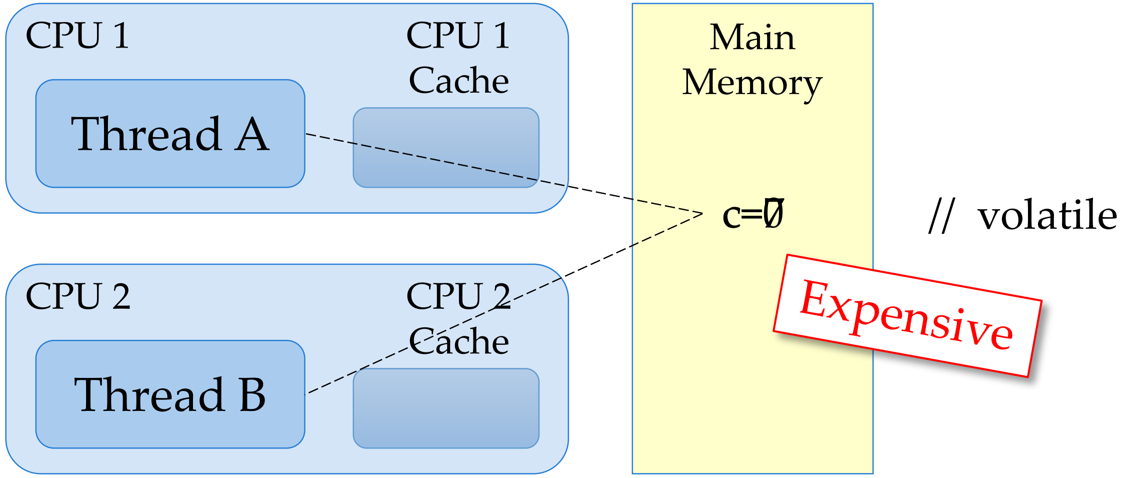
- Every **read & write** to a **volatile** variable will be on the **main memory**
  - **Not** the CPU cache…



// none volatile

Volatile

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



Volatile: ***Happens-Before*** Guarantee

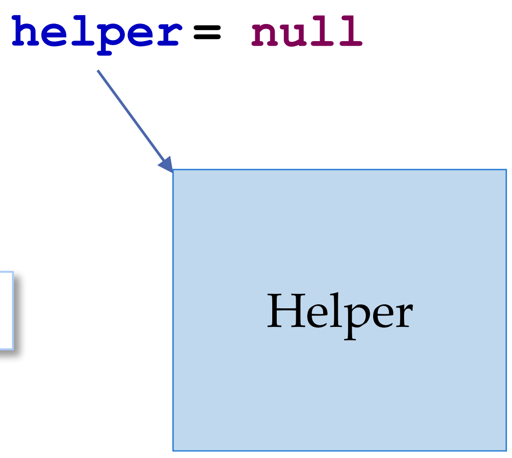
- Every **read & write** to a **volatile** variable will be on the **main memory**
  - **Not** the CPU cache…
- When a thread reads or writes to a volatile variable
  - all other **dependent** variables are flushed to main memory as well
- Reading and writing instructions **cannot be reordered** by the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

Example - Singleton

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

Expensive

Expensive



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;
    }
}
```

"Eager" instead of "Lazy"

Not Expensive

Example - Singleton

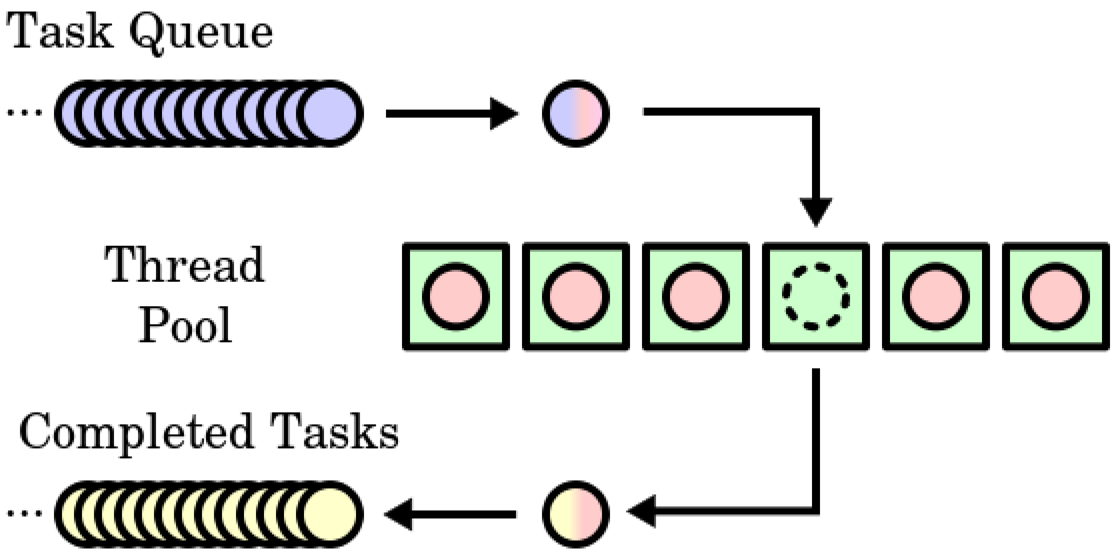
```
class Foo{
    private static class HelperHolder {
        public static final Helper helper = new Helper();
    }

    public static Helper getHelper() {
        return HelperHolder.helper;
    }
}
```

Not Expensive

inner classes are not loaded until they are referenced

Thread Pool



Executor Implementations Example

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

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

```java
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…
```

```
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 ());
}
```

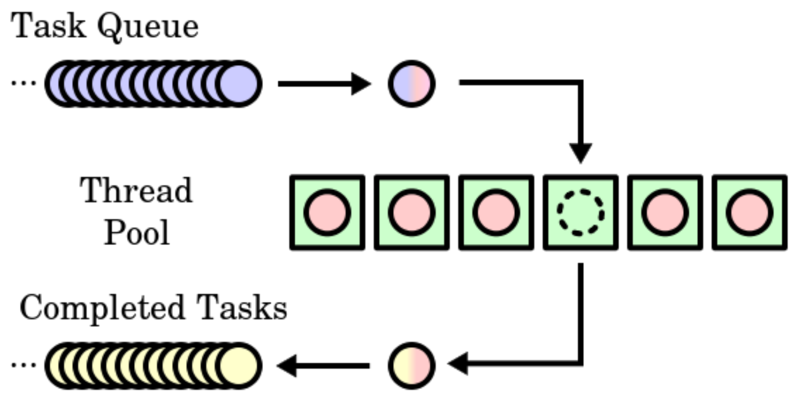
```
task1 started
task2 started
task1 finished
task2 finished
task3 started
task3 finished
```

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

- Runnabler's run() method
  - Cannot return a value
  - Cannot throw an exception
- A Callable Interface can

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

- ExecutorService (a type of thread pool) 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?

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 <V>

- 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 <V> it is set in the instance of Future
- Only then, we may get <V>

Future <V>
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();
 }
}
```