COS40003 Concurrent Programming

Lecture 4b: Thread

CRICOS 00111D
TOID 3059

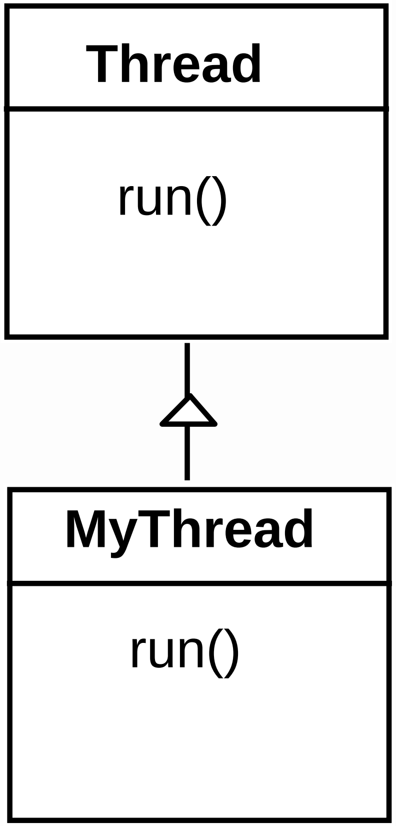
Outline

- Why we need threads?
- What is a thread?
- How threads are working?
- Thread in Java

Java APIs

- Javadocs
  - https://docs.oracle.com/javase/tutorial/index.html
- Java Concurrency
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Threads in Java



The Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

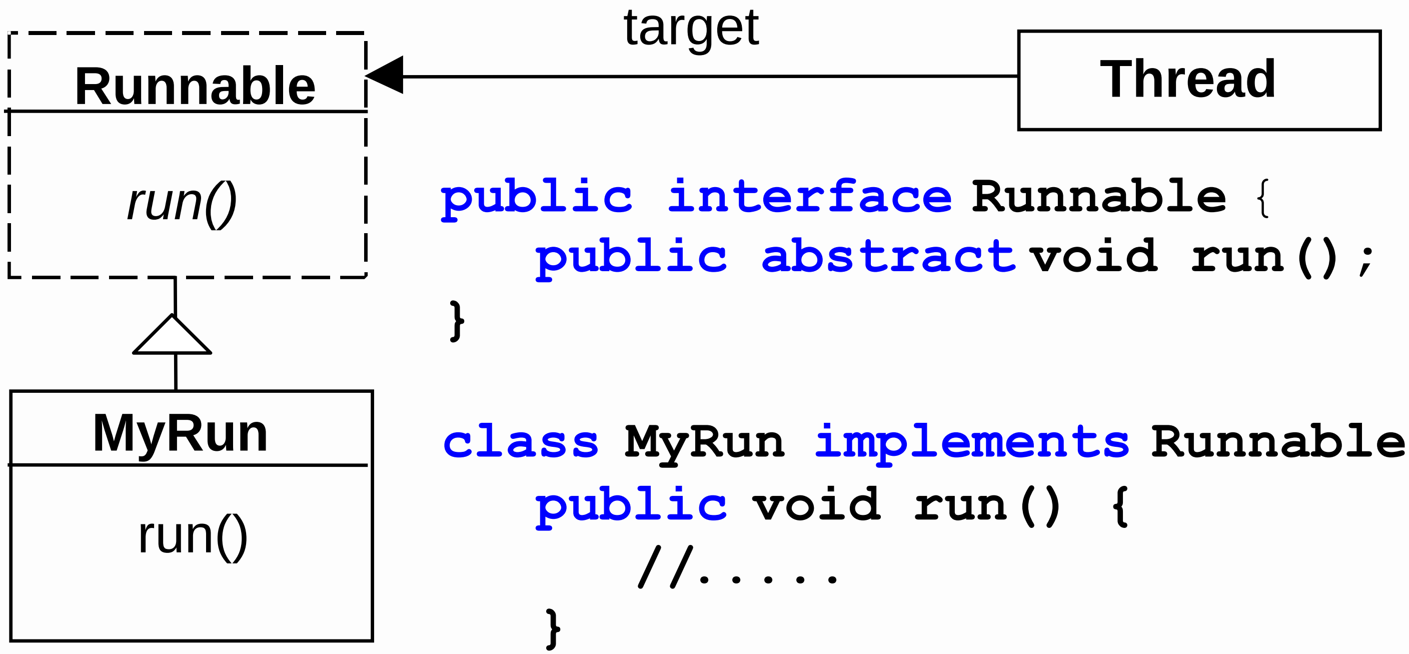
```
class MyThread extends Thread {
     public void run() {
          //......
     }
}
```

Creating and starting a thread object:

```
     Thread a = new MyThread();
     a.start();
```

Threads in Java

Since Java does not permit multiple inheritance, we often implement the **run()** method in a class not derived from Thread but from the interface Runnable. This is also more flexible and maintainable.



```
public interface Runnable {
   public abstract void run();
}

class MyRun implements Runnable{
   public void run() {
      //.....
   }
}
```

Creating and starting a thread object:

```
     Thread b = new Thread(new MyRun());
     b.start();
```

Threads in Java – Example 1

The operation we want to be threaded:

```
public class PrintNumbers {
    static public void printNumbers() {
        for(int i=0; i<10; i++) {
            System.out.println(
                      “Thread " +Thread.currentThread().getId() + “prints " + i);
        }
    }
}
```

Threads in Java – Example 1

Option 1 – extending class Thread:

```
public class Thread1 extends Thread {

  @Override
  public void run() {
    System.out.println("Thread Id: " +
  Thread.currentThread().getId());
    // do our thing
    PrintNumbers.printNumbers();
  }
}
```

Threads in Java – Example 1

Option 1 – extending class Thread (cont’):

```
public static void main(String[] args) {
   System.out.println("Main ThreadId: " +
          Thread.currentThread().getId());
   for(int i=0; i<3; i++) {
     new Thread1().start(); // don't call run!
                }
   PrintNumbers.printNumbers();
}
```

Threads in Java – Example 1

Option 2 – implementing Runnable:

```
public class Thread2 implements Runnable {

   @Override
   public void run() {
      System.out.println("Thread Id: " +
      Thread.currentThread().getId());
      // do our thing
      PrintNumbers.printNumbers();
   }
}
```

Threads in Java – Example 1

Option 2 – implementing Runnable (cont’):

```
public static void main(String[] args) {
   System.out.println("Main ThreadId: " +
          Thread.currentThread().getId());
   for(int i=0; i<3; i++) {
     new Thread(new Thread2()).start();
     // again, don't call run!
   }
   PrintNumbers.printNumbers();
}
```

Creating Threads

- extending the Thread class
  - must implement the *run()* method
  - thread ends when *run()* method finishes
  - call *.start()* to get the thread ready to run

Advantage of Using Runnable

- remember - can only extend one class in Java

- implementing runnable allows class to extend something else

Some useful functions

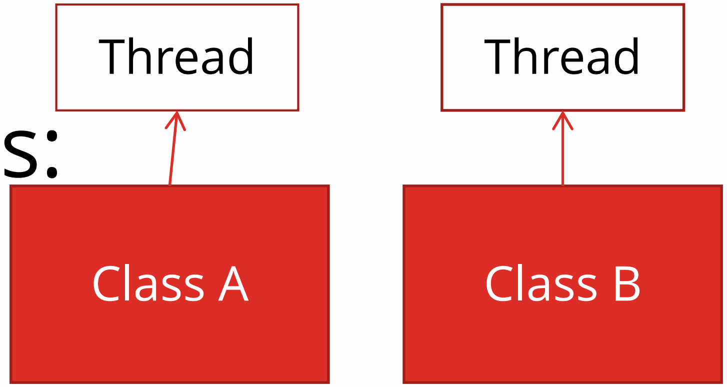
- _.*start()*: begins a thread running
- *wait()* and *notify()*: for synchronization
  - more on this later
- _.*stop()*: kills a specific thread (deprecated)
- _.*suspend()* and *resume()*: deprecated
- _.*join()*: wait for specific thread to finish
- _.*setPriority()*: 0 to 10 (MIN_PRIORITY to MAX_PRIORITY); 5 is default (NORM_PRIORITY)

Example 2 – concurrent independent tasks

- Sometimes we just want to run independent actions concurrently and (possibly) benefit from multi-cores

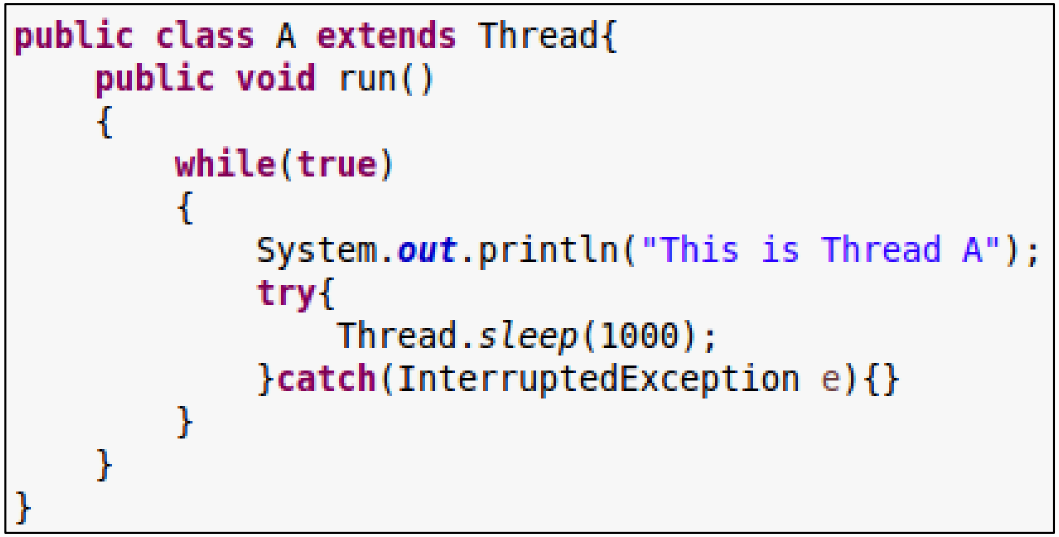
Example 2 – concurrent independent tasks

- Two concurrent tasks:

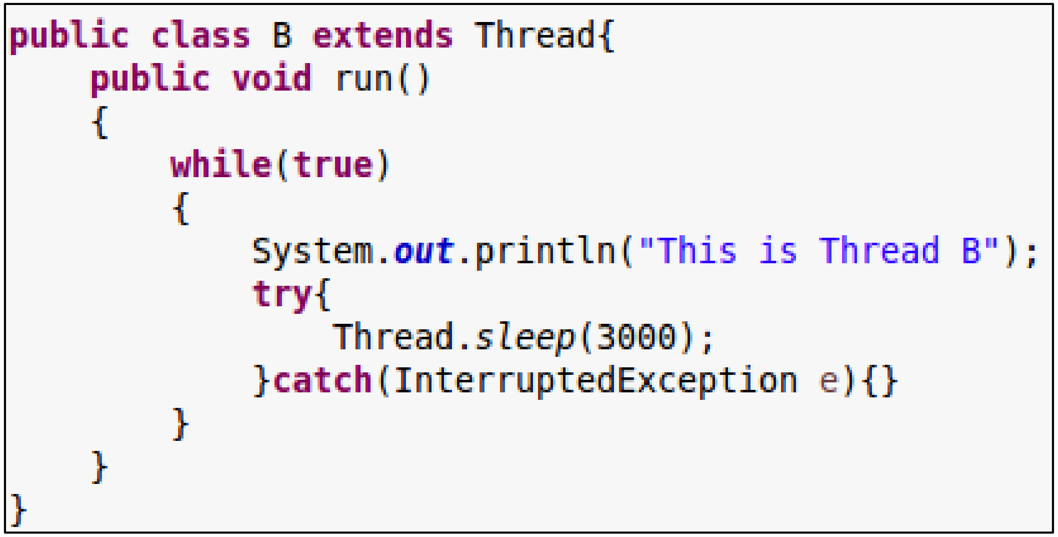


```
public void myFunction() {
  Thread task1 = new A();
  Thread task2 = new B();

  task1.start();
  task2.start();

  task1.join();
  task2.join();
}
```

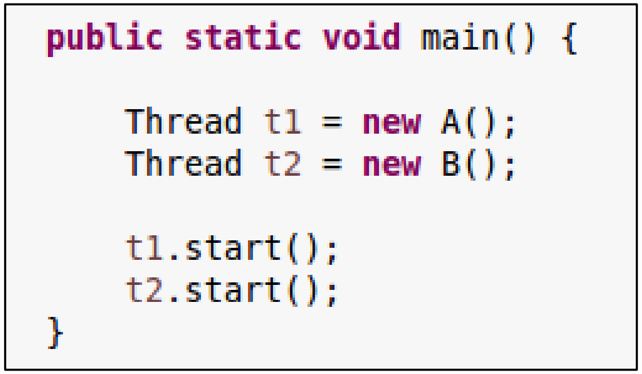
Example 2 – concurrent independent tasks



```
public class A extends Thread{
    public void run()
    {
        while(true)
        {
            System.out.println("This is Thread A");
            try{
                Thread.sleep(1000);
            }catch(InterruptedException e){}
        }
    }
}
```



```
public class B extends Thread{
    public void run()
    {
        while(true)
        {
            System.out.println("This is Thread B");
            try{
                Thread.sleep(3000);
            }catch(InterruptedException e){}
        }
    }
}
```



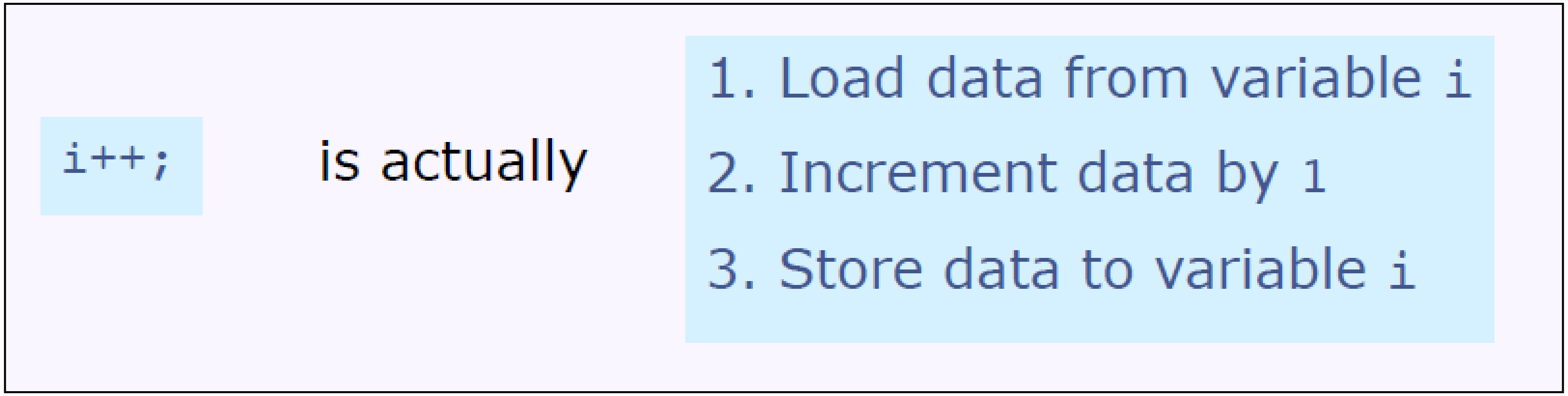
```
public static void main() {

    Thread t1 = new A();
    Thread t2 = new B();

    t1.start();
    t2.start();
}
```

Thread safety in Java libraries

- Many of the Java library classes are *not* thread safe!
- Classes in **java.lang** are thread safe
- Classes in **java.util** might **not** be safe! Check the documentation before use.
- Examples:
  - `ArrayList` and other collections from `java.util` are not thread safe; two threads changing the same list at once may break it.
  - Java GUIs are not thread safe; if two threads are modifying a GUI simultaneously, they may put the GUI into an invalid state.
- Counterexamples:
  - The `Random` class chooses numbers in a thread-safe way.
  - Some input/output (like `System.out`) is thread safe.

Example 3 – Shared variables in concurrency

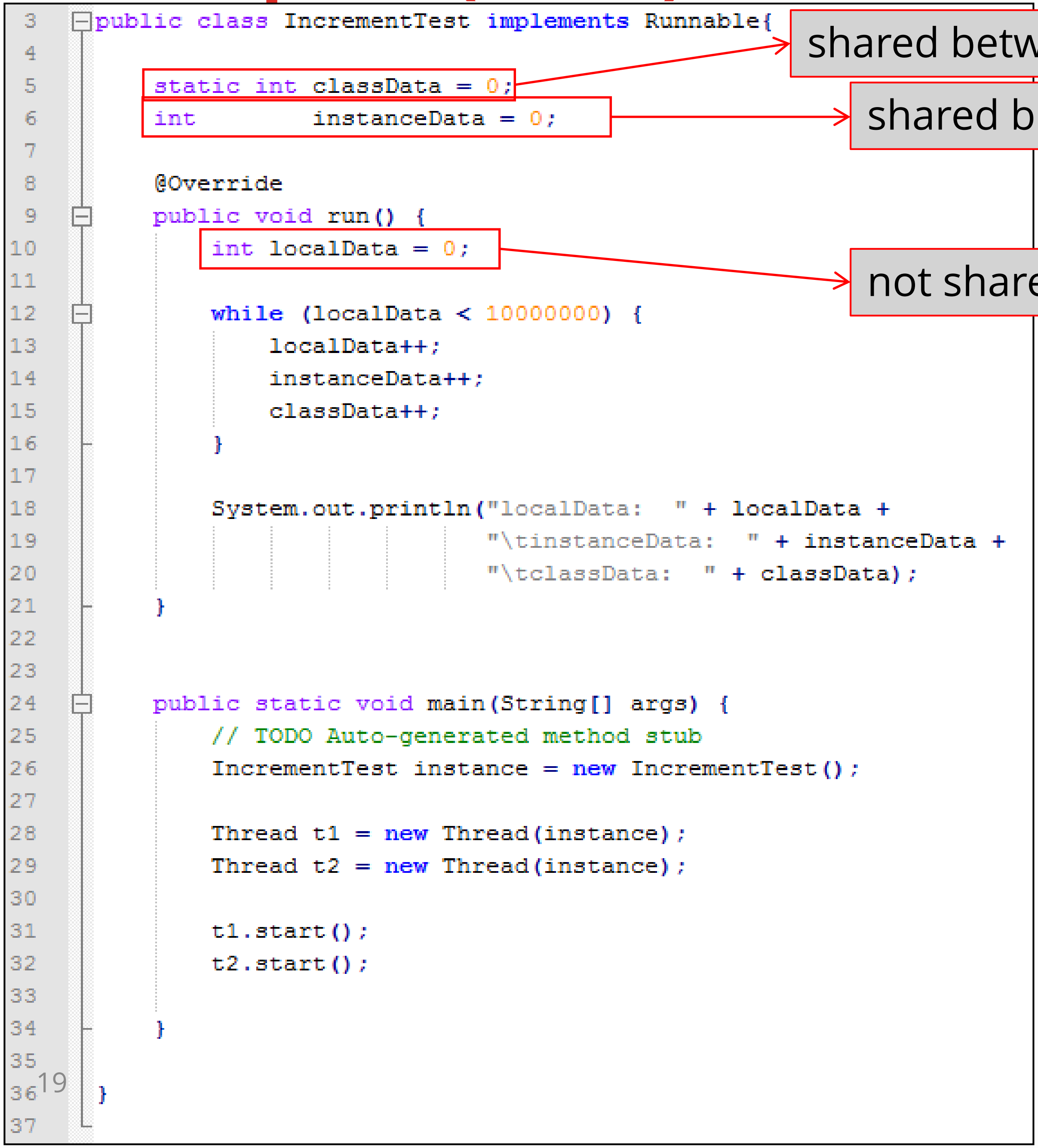
- Atomicity
  - An action is atomic if it is indivisible
  - In Java, integer increment is not atomic



`i++;` is actually

1. Load data from variable `i`
2. Increment data by `1`
3. Store data to variable `i`

Example 3 (cont.)



```
public class IncrementTest implements Runnable{

    static int classData = 0;
    int        instanceData = 0;

    @Override
    public void run() {
        int localData = 0;

        while (localData < 10000000) {
            localData++;
            instanceData++;
            classData++;
        }

        System.out.println("localData:  " + localData +
                           "\tinstanceData:  " + instanceData +
                           "\tclassData:  " + classData);
    }


    public static void main(String[] args) {
        // TODO Auto-generated method stub
        IncrementTest instance = new IncrementTest();

        Thread t1 = new Thread(instance);
        Thread t2 = new Thread(instance);

        t1.start();
        t2.start();

    }

}
```

shared between all instances of this class

shared between functions of an object

not shared with anyone

Task

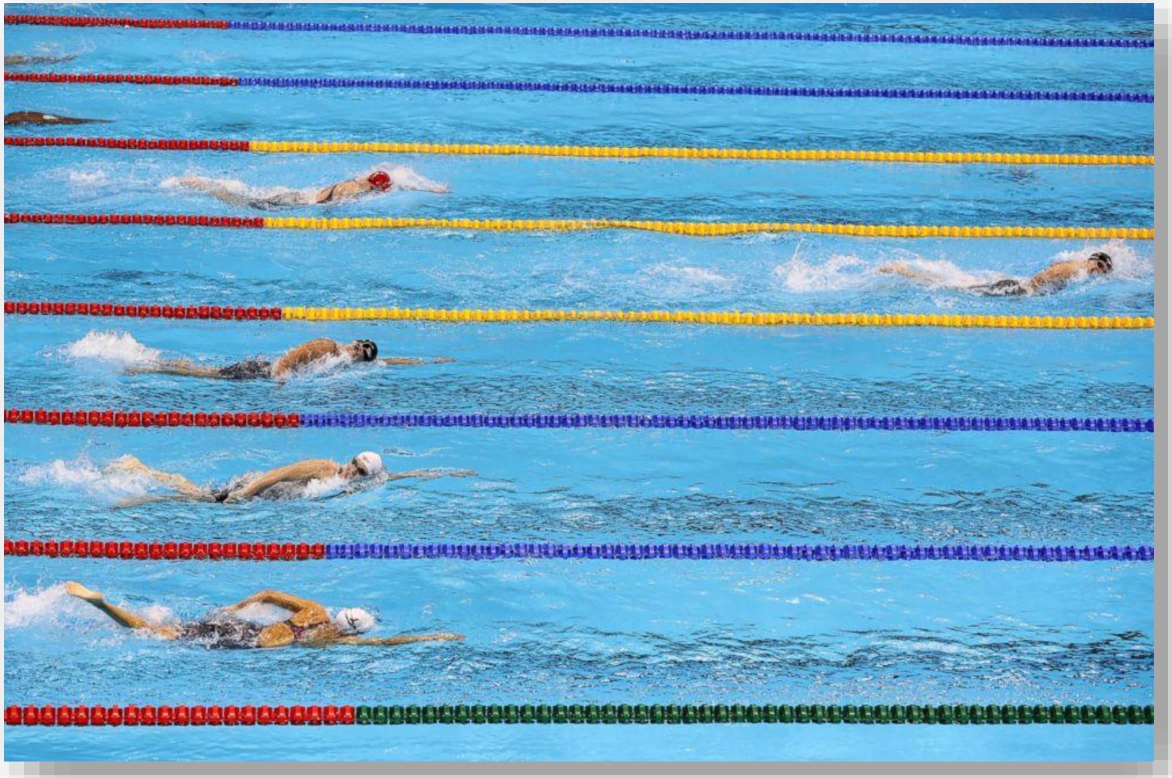
- What is the output of this program with a single thread? Why?
- How about with two threads? Why?

Results

- localData = 10,000,000
- instanceData != 20,000,000
- classData != 20,000,000
- Why?

- Try: create thread t2 with another instance and see the result:

```
IncrementTest instance2 = new IncrementTest();
Thread t2 = new Thread(instance2);
```

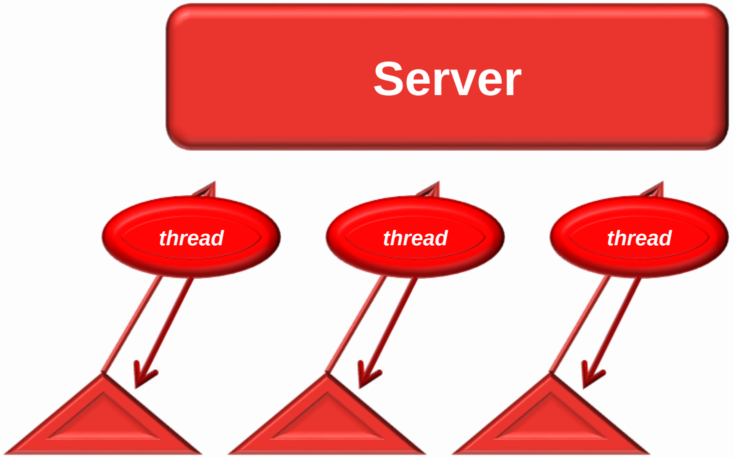


Thread Pools

- A program that creates a **huge number of short-lived** threads can be **inefficient**.
  - Threads are managed by OS. There is a **cost** for creating them.
  - Cost/overhead = memory + time
- The cost can be reduced by Thread Pools
  - A Thread Pool creates a number of threads and keeps them alive. (It asks for memory all at once not on demand).

Client-Server Example

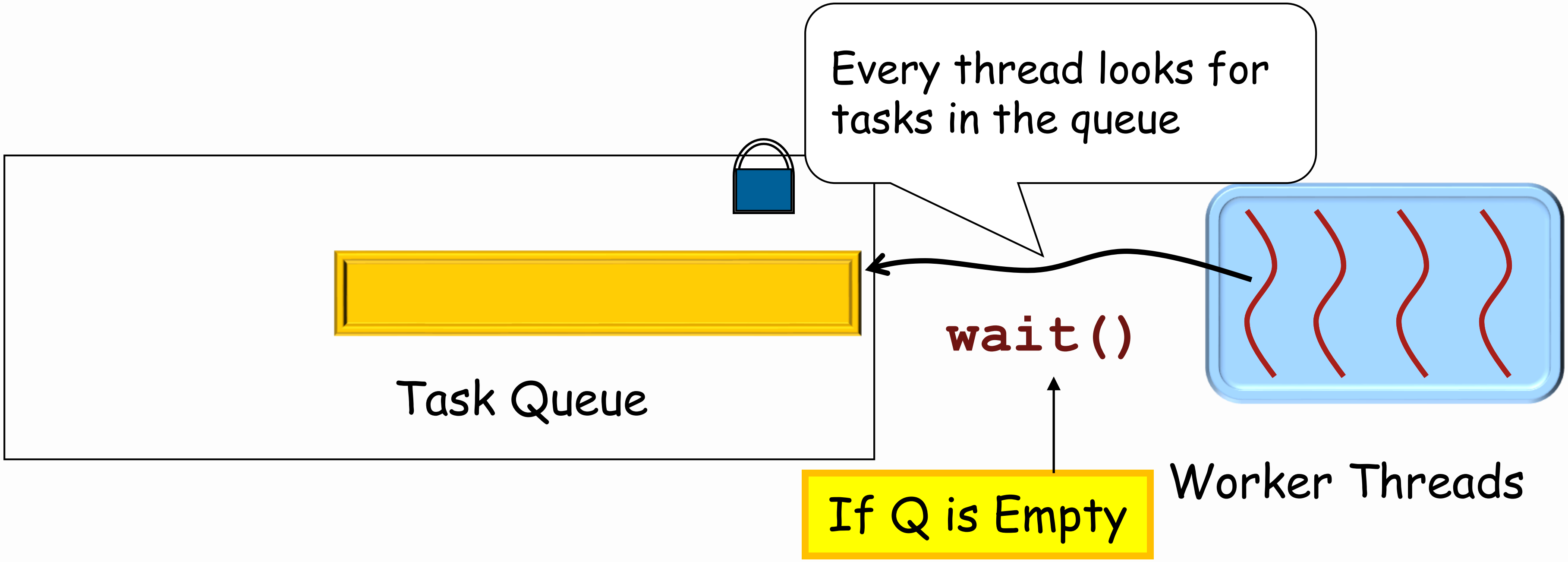
Why it is not good to create a thread upon each request?



Thread Pool, sever Example

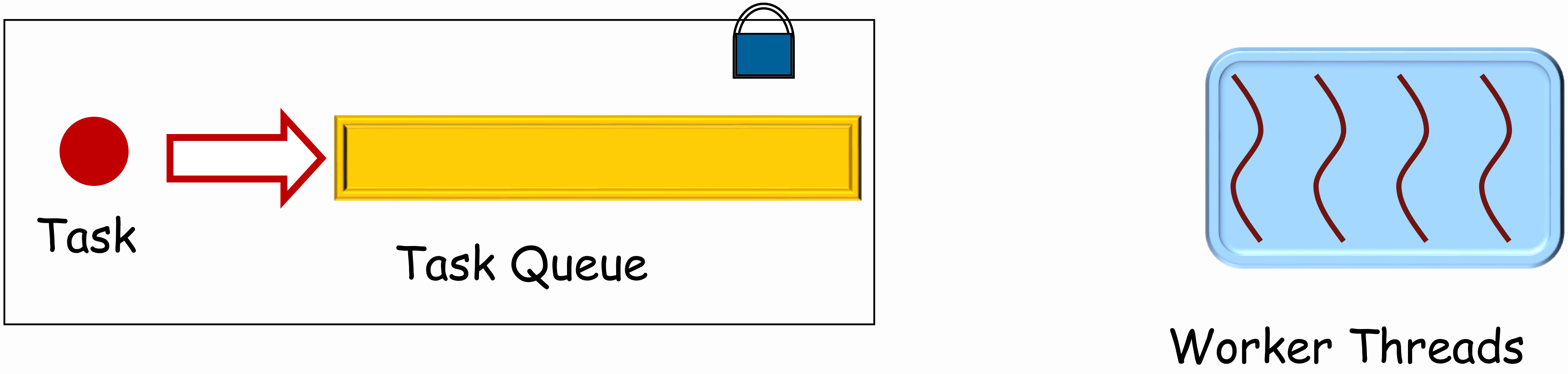
- Thread pools are especially important in client-server applications
  - The processing of each individual task is short-lived and the number of requests is large
  - Servers should not spend more time and consume more system resources creating and destroying threads than processing actual user requests
- **When too many requests arrive, thread pools enable the server to force clients to wait until threads are available**

How Thread Pools work



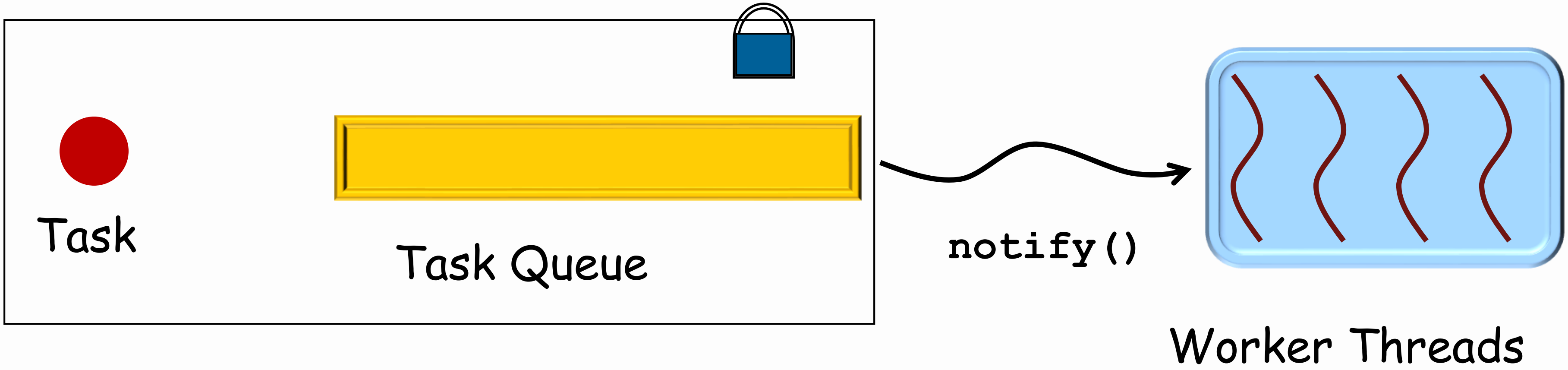
All the worker threads wait for tasks

How Thread Pools work



The number of worker threads is fixed. When a task is inserted to the queue, notify is called

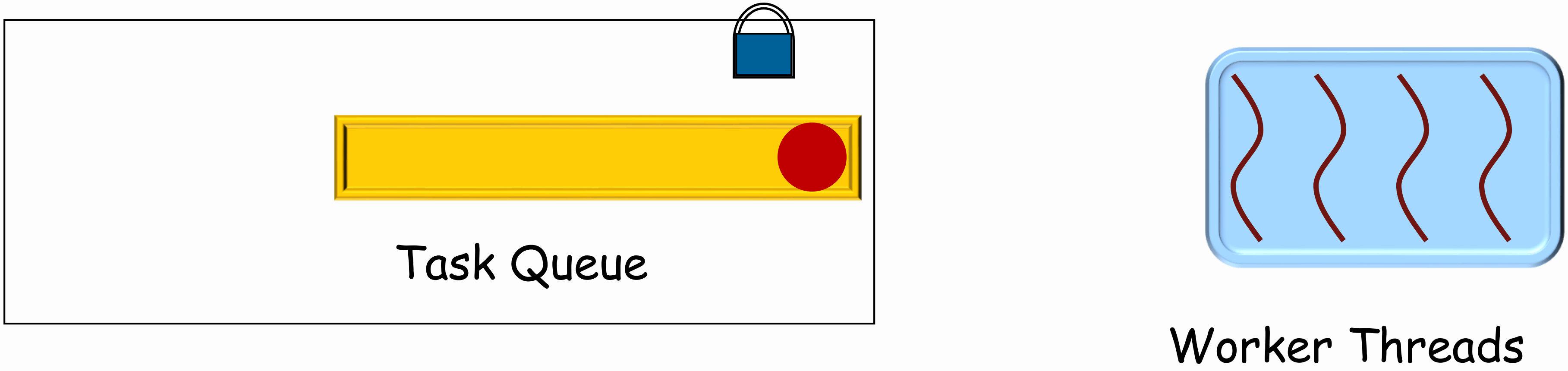
How Thread Pools work



The number of worker threads is fixed. When a task is inserted to the queue, notify is called

How Thread Pools work

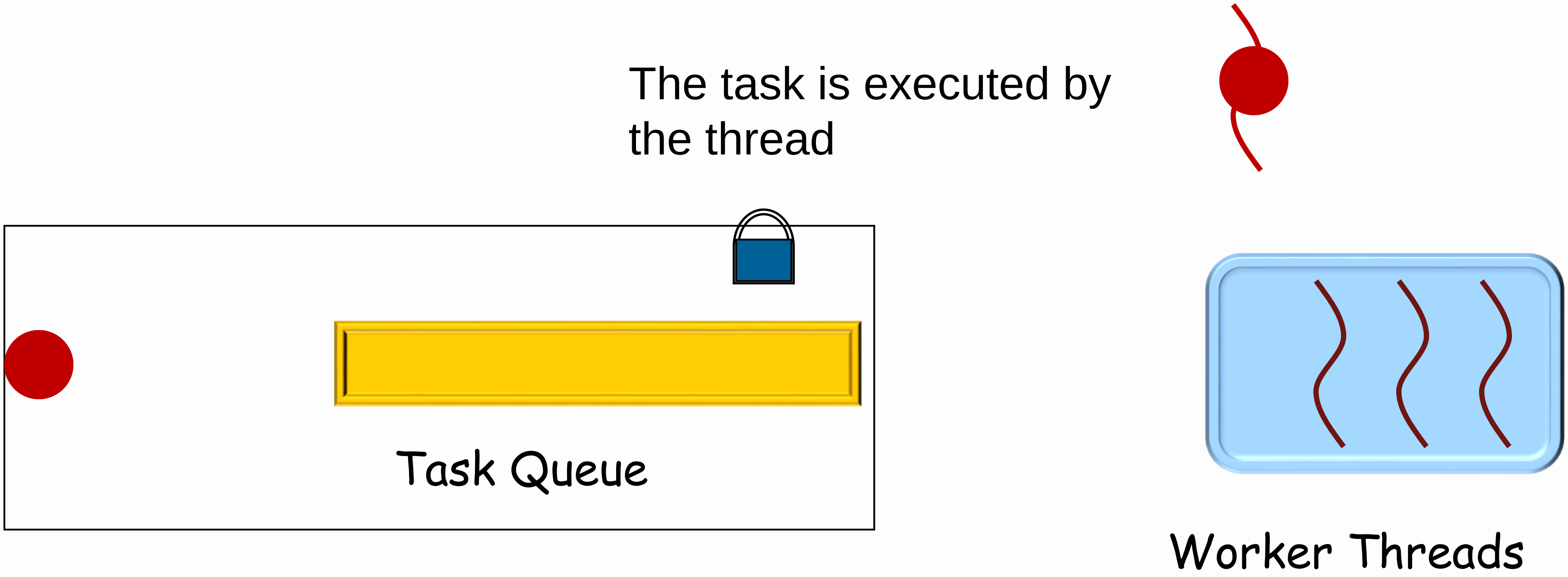
The task is executed by the thread



Task Queue

Worker Threads

How Thread Pools work



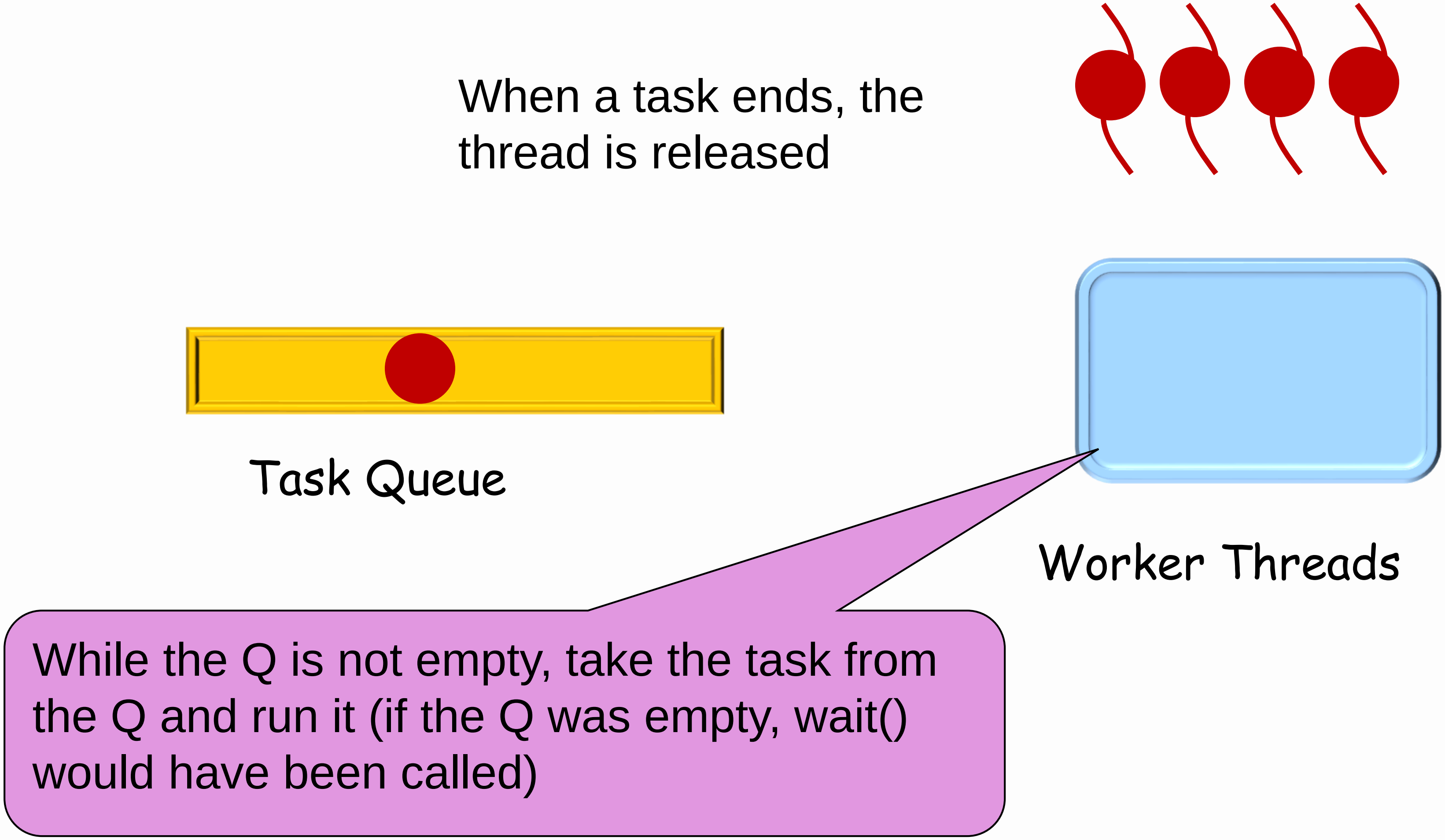
The task is executed by the thread

Task Queue

Worker Threads

The remaining tasks are
executed by the other threads

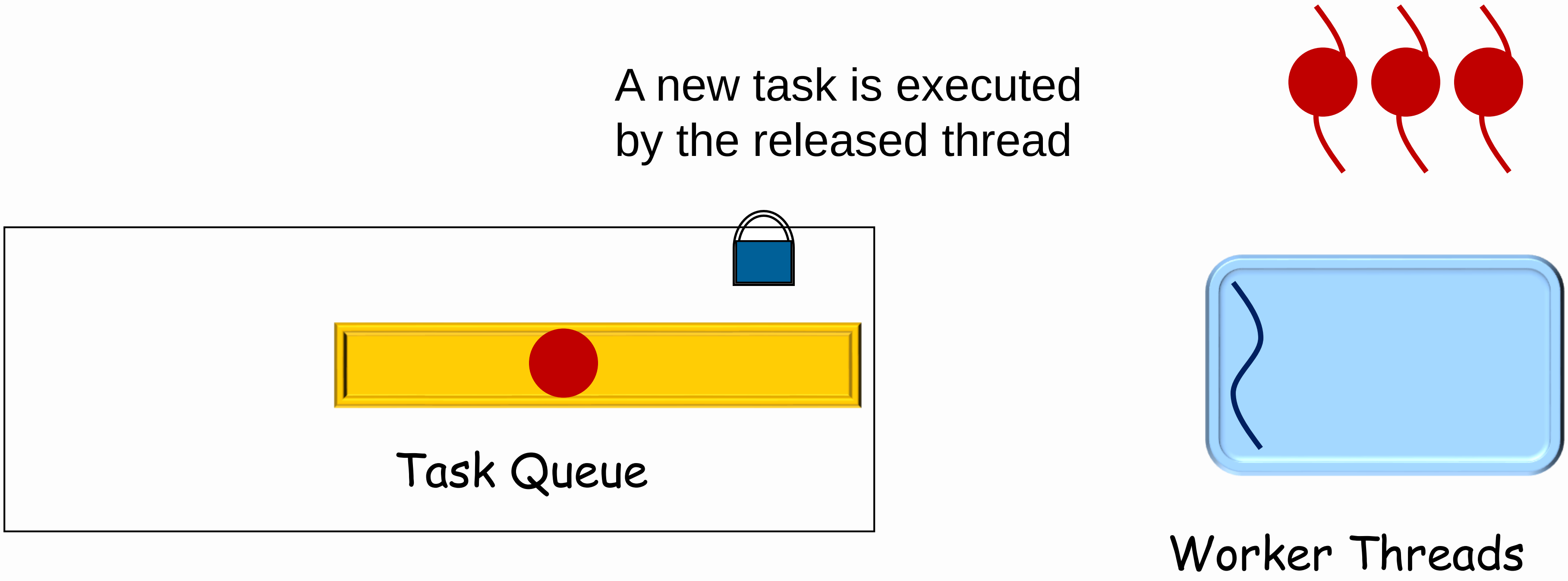
How Thread Pools work



When a task ends, the thread is released

While the Q is not empty, take the task from the Q and run it (if the Q was empty, wait() would have been called)

How Thread Pools work



A new task is executed
by the released thread

Task Queue

Worker Threads

Thread Pools in Java, example 1

(Task class implements Runnable)

```
Task task1 = new Task();
Task task2 = new Task();
Task task3 = new Task();

System.out.println("Starting threads");

ExecutorService executor = Executors.newFixedThreadPool(3);

executor.execute(task1);
executor.execute(task2);
executor.execute(task3);
executor.shutdown(); // shutdown worker threads
executor.awaitTermination(1, TimeUnit.NANOSECONDS);
```

What's happing

- The code in method main executes in the main thread
- Creates three Task objects (no extra threads are in this stage)
- Create an executor that uses a fixed thread pool by invoking **newFixedThreadPool()** method in Executors (returns a ExecutorService object)
- **execute()** creates a new Thread inside the *ExecutorService* and returns immediately from each invocation
- *ExecutorService* method **shutdown()** Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
- **awaitTermination()** waits for submitted tasks to finish
- The program will not terminate until its last thread completes execution

Java Thread Pool APIs

- Thread Pool Tutorial
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html

- java.util.concurrent.ThreadPoolExecutor
  - https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html
  - execute(), shutdown(), awaitTermination() are within

Pool Size

- What is better: to have a large pool or a small pool?
- Each thread consumes resources
  - memory, management overhead, etc.
  - A large pool can cause *starvation*
- Incoming tasks wait for a free thread
  - A small pool can cause *starvation*
- Therefore, you have to *tune the thread pool size* according to the number and characterizations of expected tasks
- There should also be a limit on the size of the task queue (why?)

Handling too Many Requests

- **What is the problem with the server being overwhelmed with requests?**
- What can a server do to avoid a request overload?
  - **Do not add to the queue** all the requests: ignore or send an error response
  - **Use several pool sizes** according to stress characteristics (but do not change the size too often…)

Acknowledgement

- Chapter 26
  - Operating Systems: Three Easy Pieces

- 3.ppt
  - Intro to Operating System at Portland State University
  - by Jonathan Walpole

Questions?