
# Object Oriented Programming with Java

## PART 1: JAVA PROGRAMMING BASICS

Threads, By Aphrodice Rwagaju

Introduction to Concurrency in Java

# Program Execution

- A thread in Java is **the direction or path that is taken while a program is being executed**.
- Generally, all the programs have at least one thread, known as the main thread, that is provided by the JVM at the starting of the program's execution

Java bytecode

```
aload_1
ifnonnull 12
new java.lang.NullPointerException [193]
dup
invokespecial java.lang.NullPointerException() [377]
athrow
iconst_0
istore_2
iload_2
aload_0
getfield java.lang.String.count : int [346]
if_icmpge 93
aload_0
getfield java.lang.String.value : char[] [349]
aload_0
```

# Parallel Execution

- It's easy to see how with multiple CPUs (or CPU cores) we can have parallel execution of multiple processes

CORE: 1
PROCESS: 1

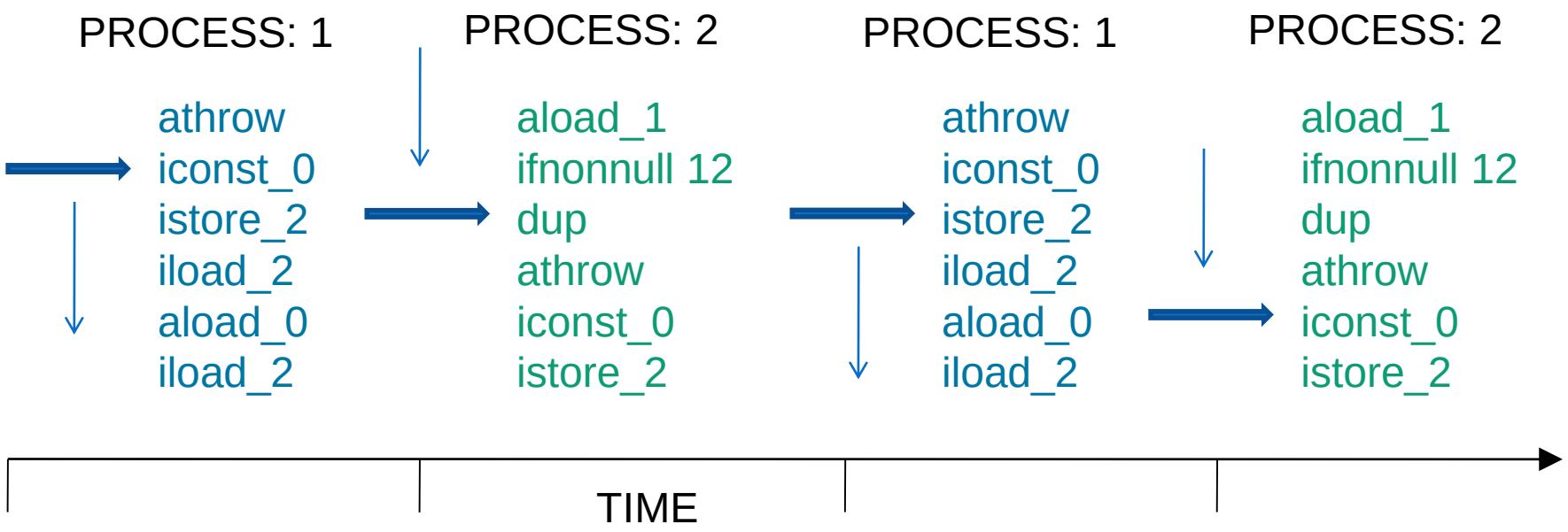
athrow
iconst_0
istore_2
iload_2
aload_0
iload_2

CORE: 2
PROCESS: 2

aload_1
ifnonnull 12
dup
athrow
iconst_0
istore_2

# Sliced Execution

- We can emulate parallel execution even with one CPU core using *time slicing*

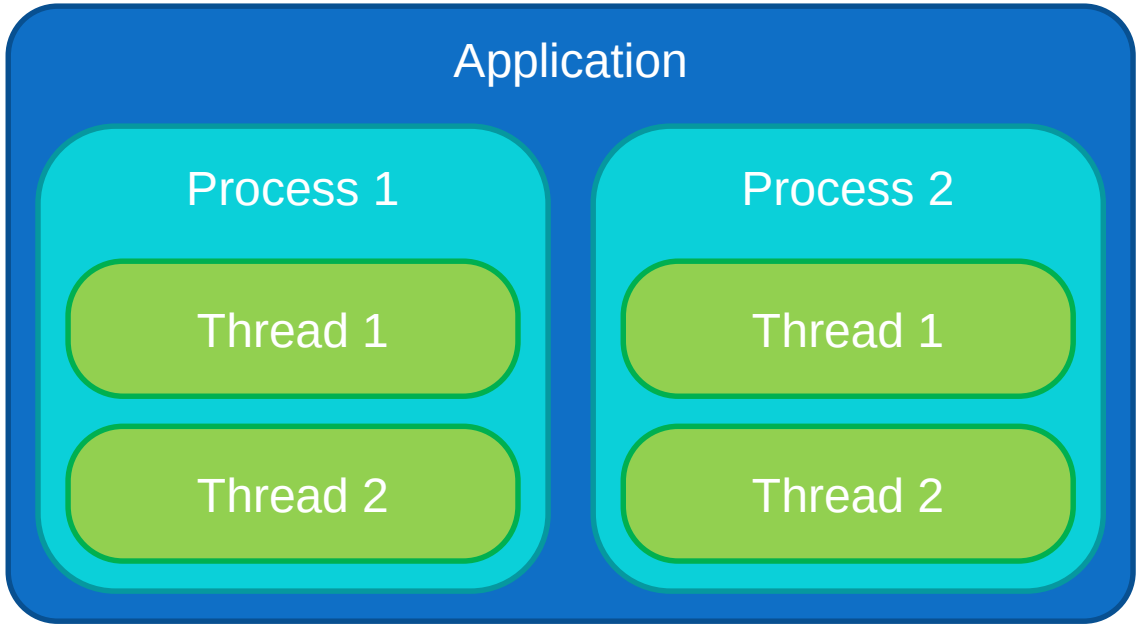



# Processes vs Threads

- An *application* can be made up of one or more processes
- A *process* is a self-contained execution environment, with its own run-time resources such as memory space
- Every process contains at least one *thread*, which is essentially a lightweight process. It shares its parent processes memory and resources

# Anatomy of an Application

- It's common for a process to have more than one thread, so that multiple tasks can be completed simultaneously




# Sharing Resources

- Different processes can only communicate with each other using *Inter Process Communication* (IPC) – a mechanism provided by the operating system
- Different threads however, share the same memory space
  - Communication between threads is simple because they can all access the variables in your program
  - But we can also run into problems if we don't control how they access shared variables

# Java Classes

- Java provides two ways of creating new threads within our applications
  - Extend the `Thread` class and override its `run` method. This is the simplest approach.
  - Implement the `Runnable` interface and its run method, then create a new `Thread` object using our `Runnable` class. This means our class can be a subclass of something other than `Thread`

# Extending Thread

- Override `run` and call `start...`

```
class Worker extends Thread {
  public void run() {
    // TODO
  }
}

public class Main {
  public static void main(String[] args) {
    Worker worker = new Worker();
    worker.start();
  }
}
```

# Implementing Runnable

- Override `run` and pass to a new `Thread`...

```
class Worker implements Runnable {
  public void run() {
    // TODO
  }
}

public class Main {
  public static void main(String[] args) {
    Thread thread = new Thread(new Worker());
    thread.start();
  }
}
```

# Interrupting

- An *interrupt* is a message sent to a thread to tell it to do something else
- Usually a programmer will have a thread respond to an interrupt by terminating
- Every thread has
  - An `interrupt()` method which will cause it to be interrupted
  - An `interrupted()` method which checks to see if it has been interrupted

# Interrupting

- If a thread is processing a task which will take a long time, it should check `interrupted` from time to time to see if it should terminate
- It is good practice therefore to break down large processing tasks into smaller units

```
for (int i < 0; i < data.length; i++) {
  // Do some more processing
  // of the task

  if (interrupted())
    break;
}
```

# Sleeping

- Every thread has a `sleep` method which will cause it to pause for the specified number of milliseconds
- But the thread might be interrupted while it is paused – in which case it will throw an `InterruptedException`
- We can call this method statically as well and it will pause the calling thread

```
try {
  Thread.sleep(1000);
} catch (InterruptedException ex) {
}
```

# Instruction Interleaving

- Because threads are executed using time slicing, their instructions are interleaved
- This can be a problem if they are accessing the same data

# Thread Interference

- Supposing 2 threads access the following method called `foo` simultaneously...

```
int a = 3;

void foo() {
   b = a;
   a = 10;
}
```

- What are the final values of `a` and `b`?

# Thread Interference

- The instructions are interleaved at the *bytecode* level
- One simple Java statement may require several bytecode instructions, e.g.

```
class Test {
  int c = 0;

  public inc() {
    c++;
  }
}
```



```
getfield Test.c     loads c
iconst_1            loads 1
iadd                add 1 to c
putfield Test.c     stores c
```

# Thread Interference

- Thus even 2 threads simultaneously executing a simple statement like `c++` can interfere with each other
- Assuming the value of c is initially 0...

```
getfield Test.c
iconst_1
iadd
putfield Test.c
```

```
getfield Test.c
iconst_1
iadd
putfield Test.c
```

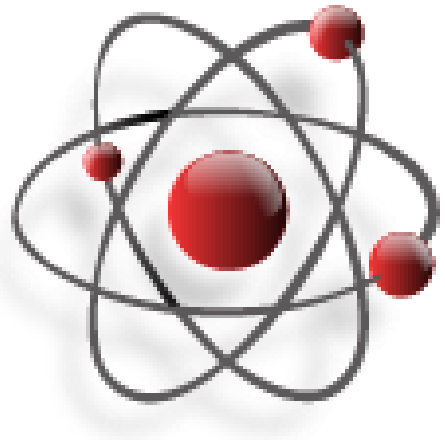
c is now 2

```
getfield Test.c
iconst_1
iadd
putfield Test.c
```

c is now 1

# Atomic Instructions

- An atomic action is one that can't be broken down into more instructions
- It happens all at once, i.e. it can't be stopped in the middle – so it can't be interleaved
- Incrementing a variable is NOT atomic because it requires several instructions which can be interleaved



# Atomic Instructions

- Things which are atomic:
  - Reads and writes are atomic for reference variables and for most primitive variables (all types except `long` and `double`)
  - Reads and writes are atomic for *all* variables declared `volatile` (*including* `long` and `double` variables)

# Memory Inconsistency

- Supposing two threads A and B share the variable c, initially zero
  - Thread A increments c
  - Thread B outputs the value of c
- B will output 0 or 1 – i.e. A and B have inconsistent view of the same data
- We need to define a *happens-before* relationship between the two actions, i.e.
  - **A** incrementing **c** should happen before **B** outputting the value of **c**

# start()

- When we call start on a new thread, we are defining a *happens-before* relationship

```
class Worker
extends Thread {
  public void run() {
    Main.c = 5;
  }
}
```

```
class Main {
 public static int c = 0;
 public static void main(String[] args) {
  System.out.print(c);
  Thread t = new Thread(new Worker());
  t.start();
 }
}
```

- Everything prior to `t.start()` *happens-before* the instructions in `Worker.run()`

# join()

- Join allows us to pause the current thread, until another one completes
- Its another way of defining a happens-before relationship...

```
class Worker
extends Thread {
  public void run() {
    Main.c = 5;
  }
}
```

```
class Main {
 public static int c = 0;
 public static void main(String[] args) {
  System.out.print(c);
  Thread t = new Thread(new Worker());
  t.start();
  System.out.print(c);
  t.join();
  System.out.print(c);
 }
}
```

Main thread will pause here until worker completes

c could be 0 or 5

c is 5

# Synchronized Methods

- Defining a method as being *synchronized* means only one thread at a time can execute it – or any other synchronized methods

```
int a = 3;

synchronized void foo() {
  b = a;
  a = 10;
}
```

Now when one thread enters foo, another thread will wait until the first thread leaves foo

The first thread's call to foo **happens before** the second thread's call

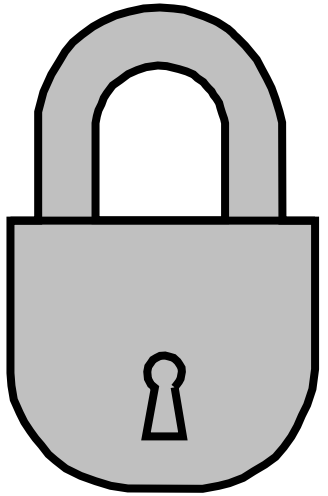
# Synchronized Methods

- There is a *happens-before* relationship between all successive calls to any of the synchronized methods
- Constructors cannot be synchronized, so we must be careful if we expose a reference to an object before its fully constructed...

```
class Counter {
  public ArrayList<Counter> all = new ArrayList<Counter>();
  public Counter() {
    all.add(this);
  }
}
```

# Locks

- When a thread calls a synchronized method, it obtains a lock for that object. When it leaves the synchronized method it releases that lock
- Other threads cannot execute any of the synchronized methods until the lock is released. They wait in a queue until the lock is free
- Static methods use a lock on the class object – a different lock



# Synchronized Statements

- We can create synchronized statement blocks as well as methods
- An object must be specified – its then used as the lock
- If we use `this` keyword, then the lock will be shared with any synchronized methods, e.g.

Synchronized code

Not synchronized

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

# Deadlock

- Deadlock occurs when two threads both block to wait for each other
- For example, if there are two instances of `SyncInt` called `s1` and `s2`, and one thread calls `s1.add(s2)` as another thread calls `s2.add(s1)`

```
class SyncInt {
  private long val = 0;

  public synchronized void add(SyncInt i) {
      val += i.getValue();
  }
  public synchronized long getValue() {
      return val;
  }
}
```

# Summary

- The JVM makes multiple threads appear to run simultaneously by *time slicing*
- Instructions from different threads end up *interleaved*
- An *atomic* instruction is one that can't be broken down into smaller instructions
- The `synchronized` keyword uses an object or class's lock to restrict access to a method or block to one thread at a time

# Guarded Blocks

- Sometimes we need a thread to wait for a condition to become true
- We could implement this with a simple while loop...

```
boolean ready = false;
```

```
while (!ready) { }
```

- Assuming that `ready` can be modified by another thread
- ...but that's a waste of processing time

# Guarded Blocks

- Much better to use to have the thread sleep so its not wasting CPU cycles and have another thread wake it up when its ready
- We can do this with `wait` and `notify` which are both methods of `Object` and so are available to every object

```
try {
   wait();
} catch (InterruptedException e) {
}
```

```
notifiy();
```

# wait

- When a thread enters a `synchronized` method or block it *acquires* the lock – when it leaves it *releases* the lock
- The `wait` method causes the thread to release the lock and go to sleep
- This means another thread can now enter the synchronized method
- `wait` must be called from synchronized code to ensure that the calling thread actually has the lock

# notify and notifyAll

- A thread that calls `wait` will sleep until another thread calls `notify` or `notifyAll`
- We say that such a thread is *blocking* on that object
- `notify` wakes up one thread – `notifyAll` wakes up all threads that are blocking on that object

# Semaphores

- The synchronized keyword restricts access to one thread
- If we need to restrict access to a specific number of threads we can use a *counting semaphore*
- For example – a database may only allow a certain number of connections – a semaphore could be used to ensure that only that number of threads can access it at one time

# Semaphores

- A Semaphore can be easily implemented using `wait` and `notify`

```
class Semaphore {
  private int count;
  public Semaphore(int n) {
    this.count = n;
  }
  public synchronized void acquire() {
    while (count == 0) {
      try { wait(); }
      catch (InterruptedException e) {
        // Keep trying
      }
    }
    count--;
  }
  public synchronized void release() {
    count++;
    notify(); // Wake a thread that's
        // blocking on this semaphore
  }
}
```

# Semaphores

- `wait` should be called within a while loop. It shouldn't assume that because a thread has called `notify` that it should now continue
- It should always check that is ok for it to proceed, and if not, call `wait` again, e.g.

```
public synchronized void acquire() {
   while (count == 0) {
     try { wait(); }
     catch (InterruptedException e) {
       // Keep trying
     }
   }
   count--;
 }
```

# Semaphores

- The code that we want to limit thread access to should occur between calls to `acquire` and `release`, e.g.

```
Semaphore pass = new Semaphore(10);

public void connectToDB() {
  try {
   pass.acquire();
  } catch (InterruptedException ex) {
  }

  // TODO Database code - Only 10 threads at a time can be here

  pass.release();
}
```

# Java Classes

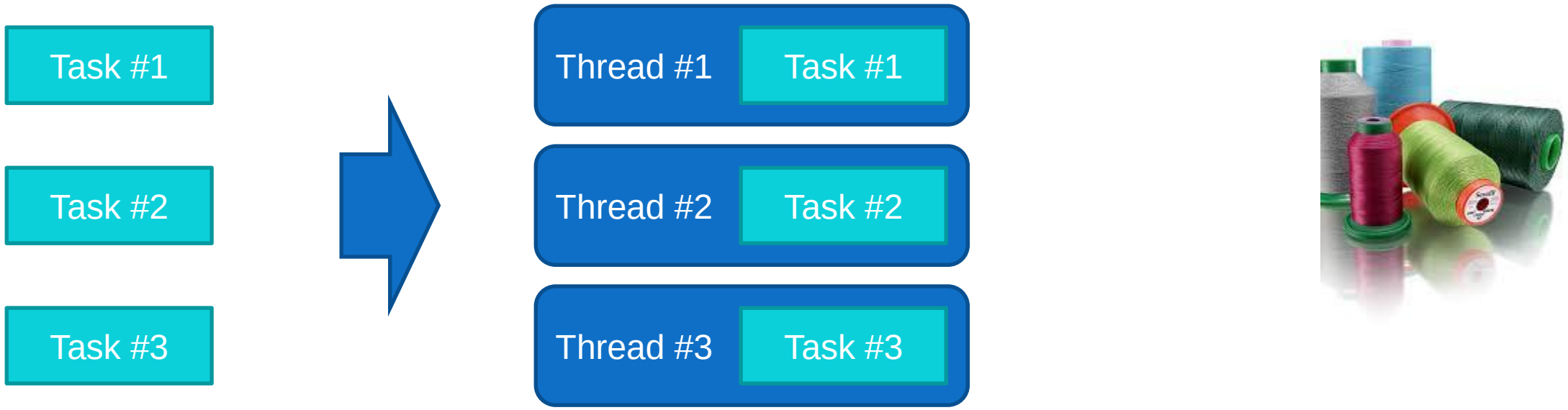
- The `java.util.concurrent` package contains a `Semaphore` class as well as other high level synchronization objects
  - `java.util.concurrent.atomic` contains types such as `AtomicInteger` which emulate a primitive type whose operations are all atomic
  - `java.util.concurrent.locks` contains different types of locks which can be used for synchronization

# Thread Pools

- Semaphores are ideal for limiting the number of threads that can access a resource simultaneously, however...
- Sometimes we need to limit the number of threads created in the first place, e.g.
  - A web server which creates a new thread to handle each HTTP request. Too many requests at one time will crash the system if it can't create that many threads

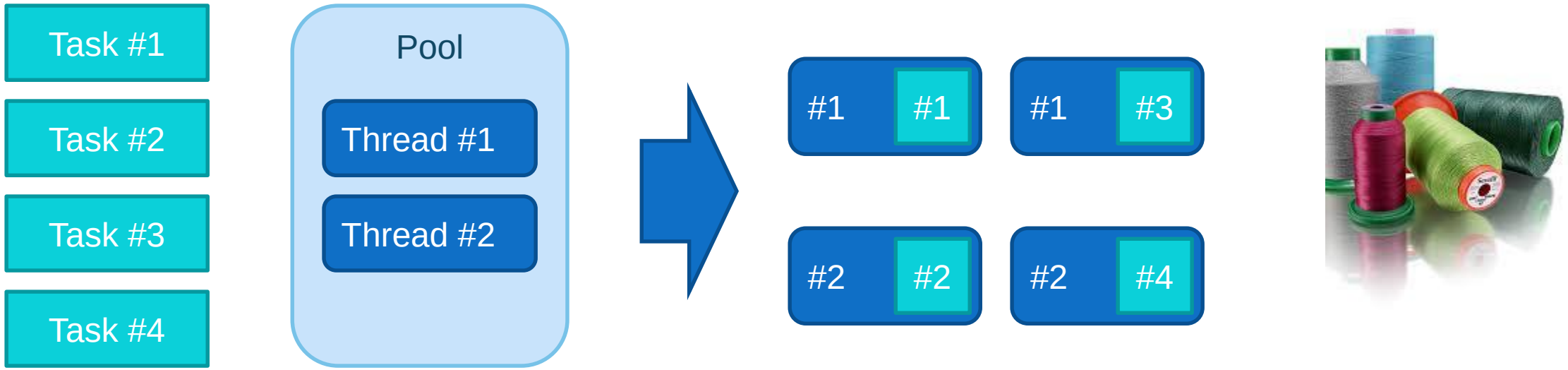
# Thread Pools

- Until now, we've used one thread for each task we want to complete, i.e. each task becomes a thread



# Thread Pools

- With a thread pool, we a create a set of threads, and have them work on a set of tasks until they are all complete



# Thread Pools

- A thread pool is created by using one of the factory methods of the `Executors` class
- Typically we want a thread pool with a fixed number of threads, e.g.

```
Executor pool = Executors.newFixedThreadPool(10);
```

- ... will create a pool of 10 threads
- Any `Runnable` object can now be run by passing it to the pool's `execute` method

# Thread Pool Example

```
class Task implements Runnable {
  public void run() {
    // Do something!
  }
}
```

```
Executor pool = Executors.newFixedThreadPool(10);

for (int i = 0; i < 100; i++)
  pool.execute(new Task());
```

# References

https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

# EoF