M9(a)-Concurrency

Jin L.C. Guo

Image source: https://cdn.pixabay.com/photo/2016/10/10/00/48/chefs-1727446_960_720.jpg

Objective

- Understand the concept of a Thread and its usefulness for programming;
- Be able to write basic concurrent programs in Java;
- Understand the causes of basic concurrency errors
- Understand the mechanisms that help prevent the basic concurrency errors.

Make this ravioli dish

- Make the pasta dough
- Let the dough rest
- Make the filling
- Make the ravioli
- Boil the water
- Cook ravioli in water
- Make the butter sauce
- Finish cooking ravioli in sauce
- Make garnishing

Source: https://www.jamieoliver.com/recipes/pasta-recipes/amazing-ravioli/

Units of execution

- Process
  - A self-contained execution environment
  - Has its own memory space
  - Most implementations of Java virtual machine run as a single process

Units of execution

- Thread
  - Lightweight process
  - One process has at least one thread.
  - Threads in one process share process resources
    - Memory
    - File handles
    - Security credentials
  - Thread has their own
    - Program counter
    - Stack
    - Local variables
  - Each java application start with one thread: main thread.

Units of execution

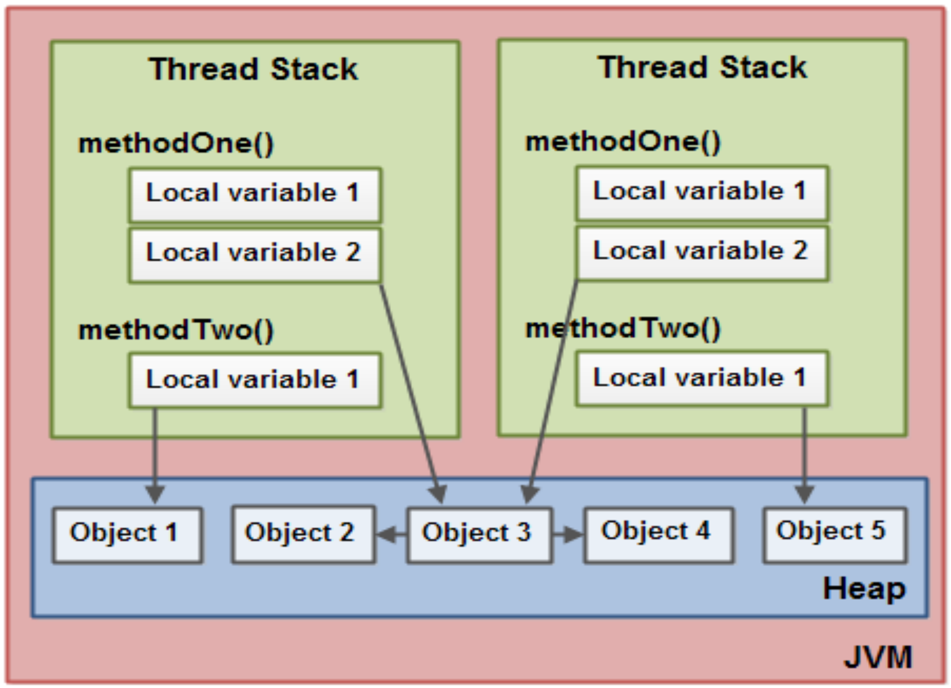
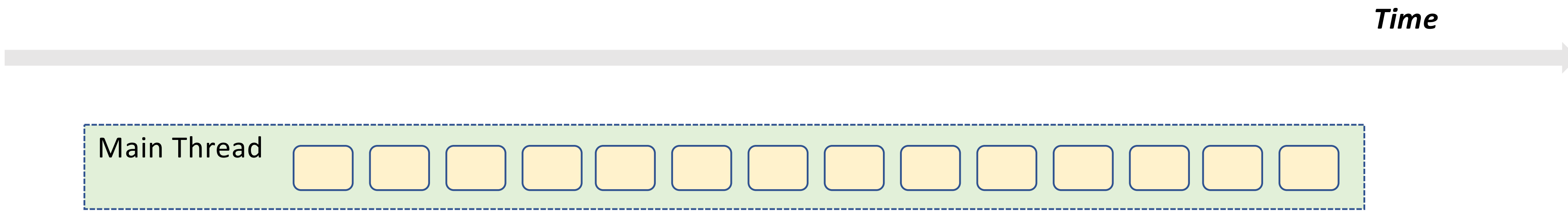


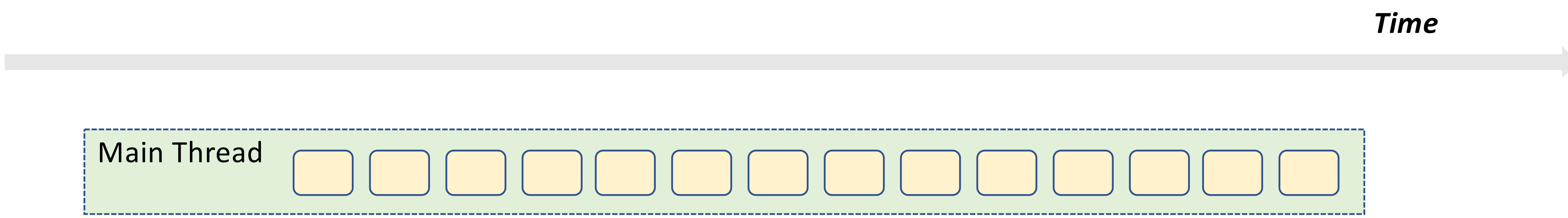
Image source: http://tutorials.jenkov.com/images/java-concurrency/java-memory-model-3.png

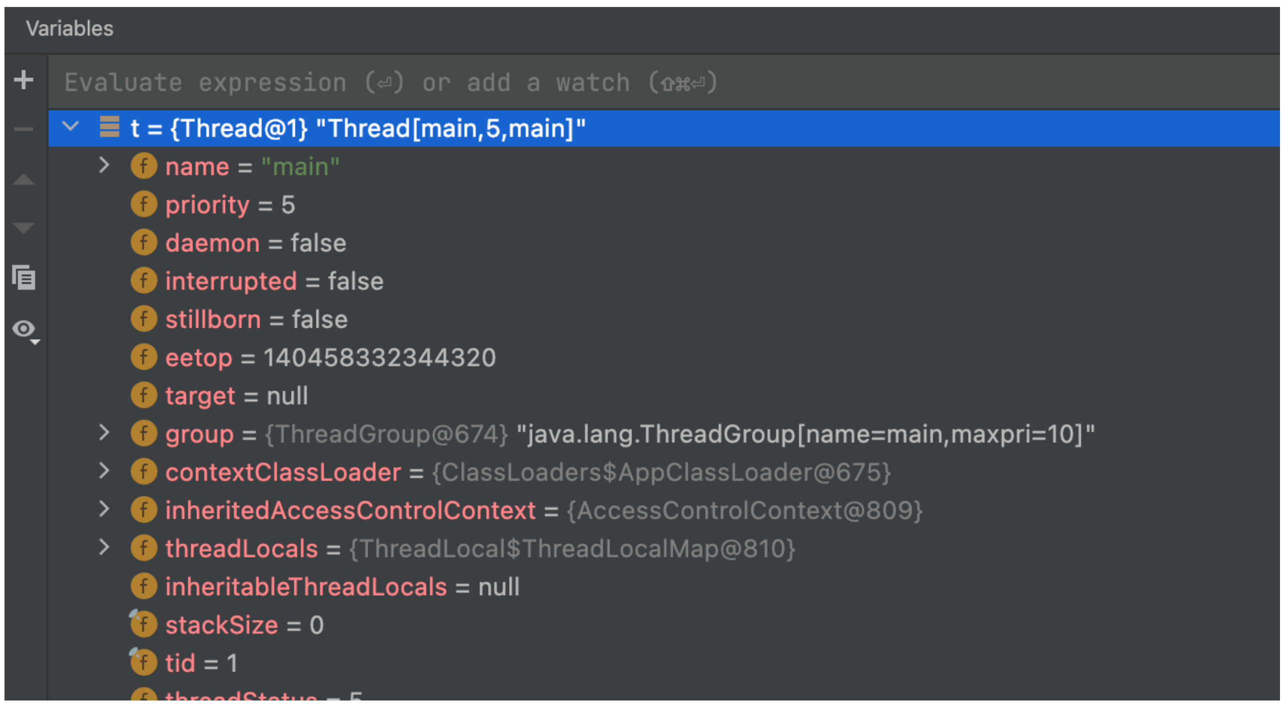
Why concurrency

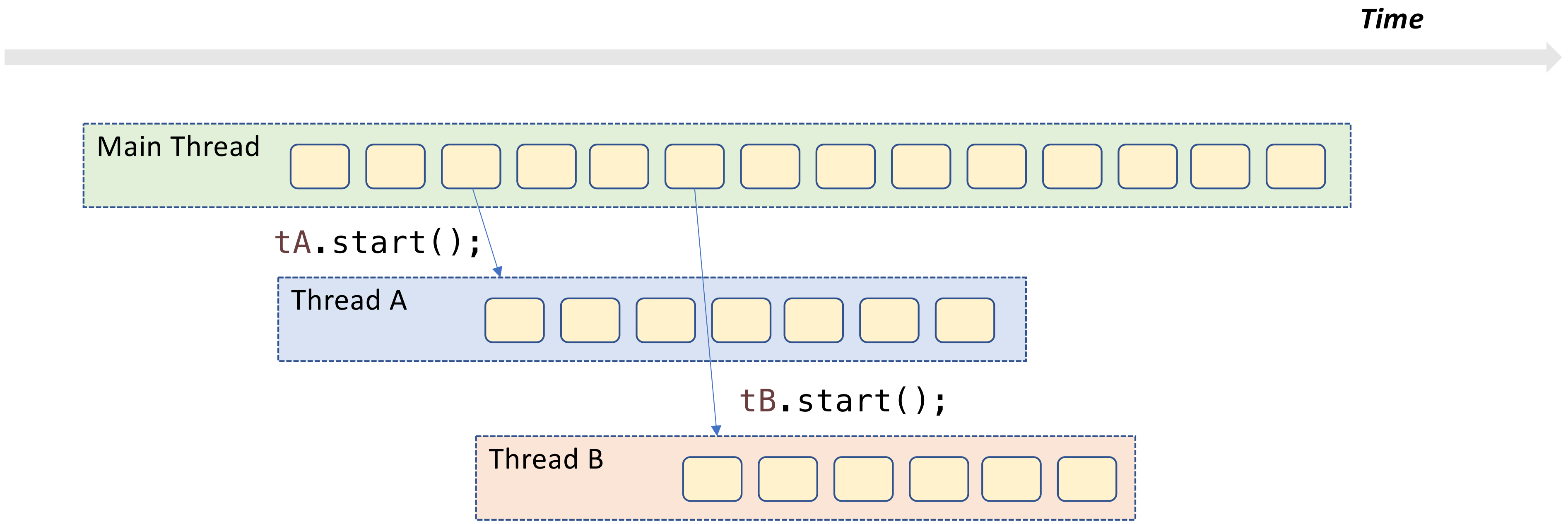
- **Exploiting Multiple Processors**
  - Improve throughput by utilizing available processor resources more effectively.
- **Simplicity of Modeling**
  - Decompose complicated, asynchronous workflow into a number of simpler, synchronous workflows interacting only at specific synchronization points.
- **Improve GUI applications**
  - Enable more responsive user interfaces




```java
public class MainThread
{
    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t.getName());
    }
}
```


Time
Main Thread


Variables
Evaluate expression (⏎) or add a watch (⇧⌘⏎)
t = {Thread@1} "Thread[main,5,main]"
name = "main"
priority = 5
daemon = false
interrupted = false
stillborn = false
eetop = 140458332344320
target = null
group = {ThreadGroup@674} "java.lang.ThreadGroup[name=main,maxpri=10]"
contextClassLoader = {ClassLoaders$AppClassLoader@675}
inheritedAccessControlContext = {AccessControlContext@809}
threadLocals = {ThreadLocal$ThreadLocalMap@810}
inheritableThreadLocals = null
stackSize = 0
tid = 1



Each thread is associated an instance of the class Thread

Create New Thread – Method 1

To declare a class to be a subclass of Thread. This subclass should override the run method of class Thread.

```java
// This thread compute the prime number larger than minPrime
class PrimeThread extends Thread
{
   long minPrime;
   PrimeThread(long minPrime) { this.minPrime = minPrime; }

   @Override
   public void run()
   {
      // actual computation
   }
}
```

Create New Thread – Method 1

Then create a thread and start it running:

```
public class MainThread
{
  public static void main(String[] args)
  {
    PrimeThread p = new PrimeThread(143);
    p.start();
  }
}
```

Create New Thread – Method 2

To declare a class that implements the Runnable interface. That class then implements the run method.

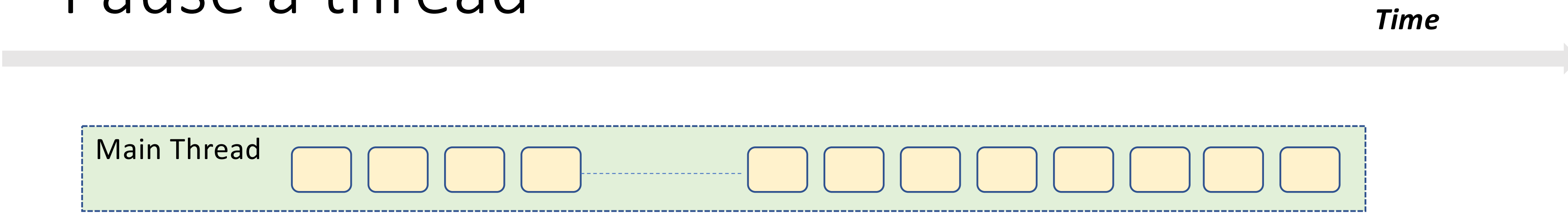
```
class PrimeRun implements Runnable
{
   long minPrime;
   PrimeRun(long minPrime) { this.minPrime = minPrime; }

   @Override
   public void run()
   {
      // compute primes larger than minPrime  . . .
   }
```

Create New Thread – Method 2

An instance of the class can then be allocated, passed as an argument when creating Thread, and started.

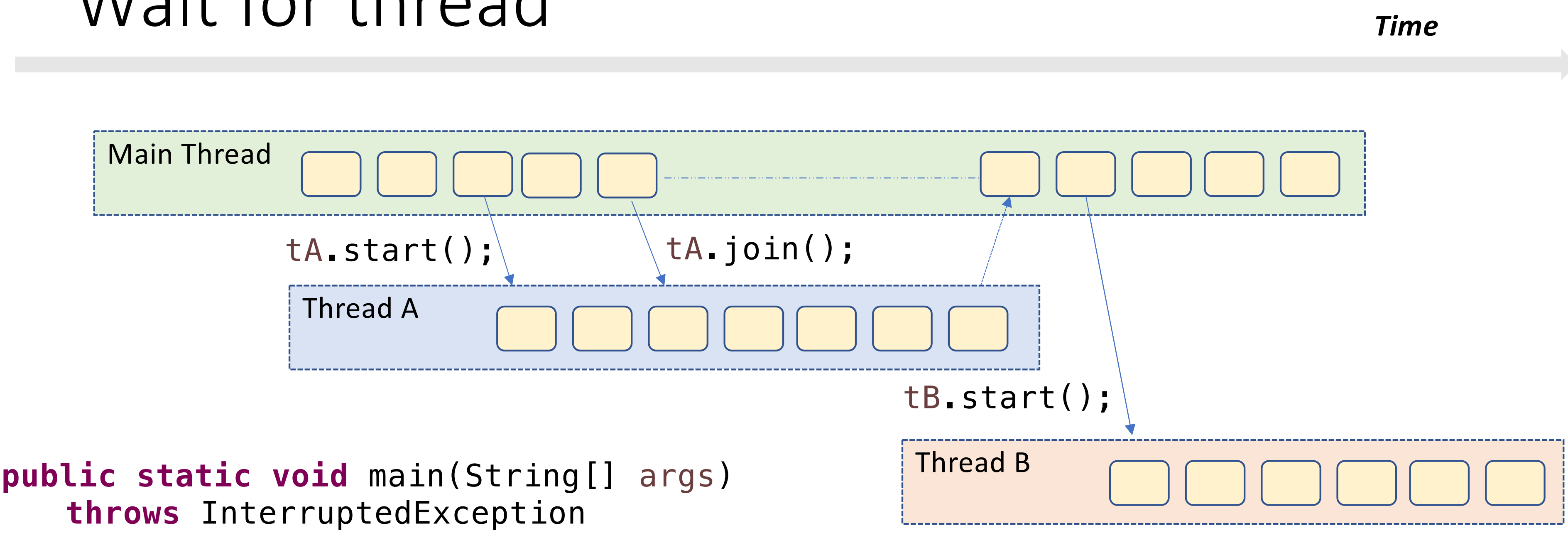
```
public class MainThread
{
   public static void main(String[] args)
   {
      PrimeRun p = new PrimeRun(143);
      new Thread(p).start();
   }
}
```

Pause a thread




```java
public static void main(String[] args)
   throws InterruptedException
{
        for (int i = 0;i < 10;i++)
        {
            Thread.sleep(1000);
            System.out.println(i);
        }
}
```

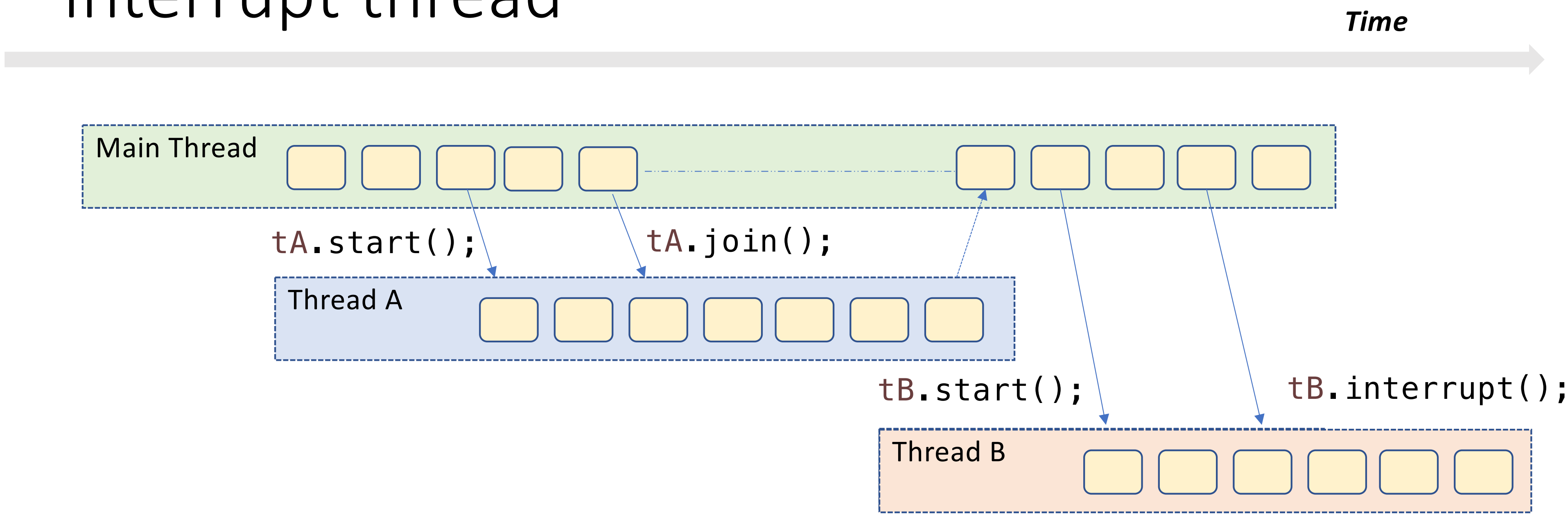
Wait for thread




```java
public static void main(String[] args)
    throws InterruptedException
{
    Thread tA = new Thread(()->System.out.println("Running Thread A"));
    Thread tB = new Thread(()->System.out.println("Running Thread B"));
    tA.start();
    tA.join();
    System.out.println("Main Thread Waiting for Thread A");
    tB.start();
}
```

DistributedComputation Demo

Interrupt thread

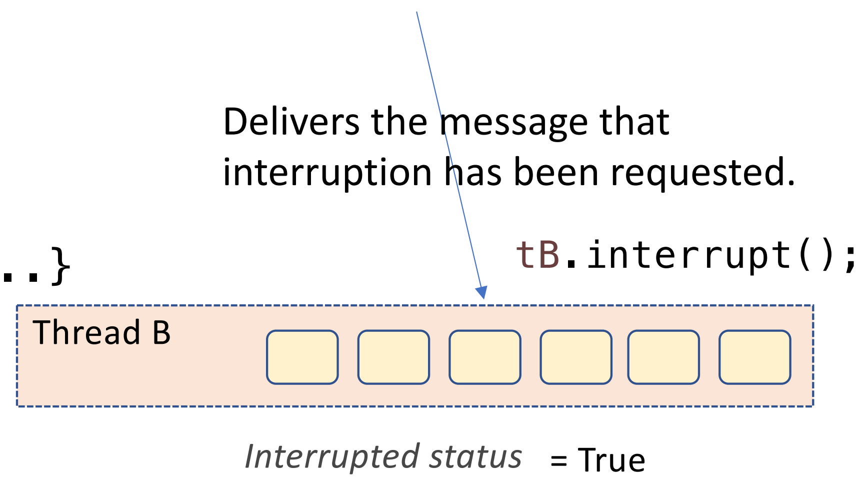


Thread B has to support its own interruption

Interrupt thread

- A cooperative mechanism

```java
public class Thread {
   public void interrupt() {...}
   public boolean isInterrupted() {...}
   public static boolean interrupted() {...}
   ...
}
```



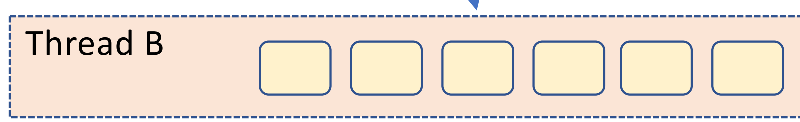
Interrupt thread

- A cooperative mechanism

```java
public void run() {
    for (int i = 0; i < inputLength; i++) {
        // Heavy operation
        if (Thread.currentThread().isInterrupted()) {
            return;
        }
    }
}
```

Delivers the message that interruption has been requested.

```java
tB.interrupt();
```



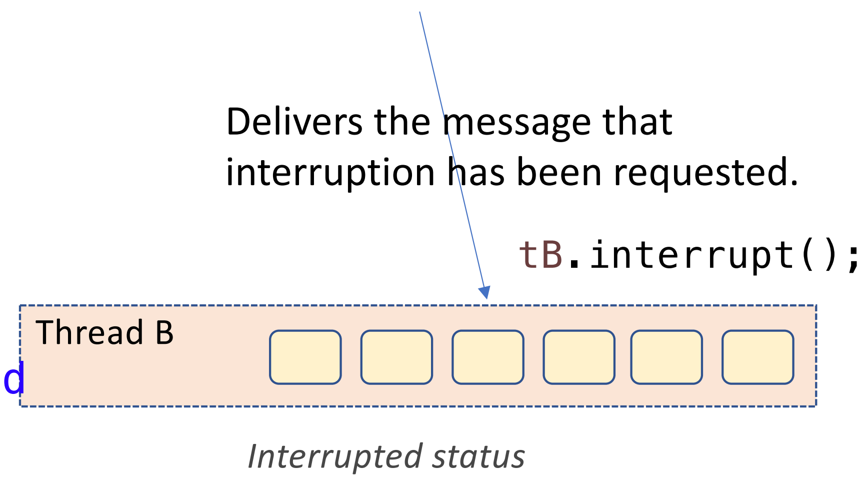
Interrupted status

vs `if (Thread.interrupted())`

Interrupt thread

- A cooperative mechanism

```java
public void run()
{
    try
    {
        Thread.sleep(5000);
        System.out.println("Thread completed
        normally");
    }
    catch(InterruptedException e)
    {
        System.out.println("Thread interrupted");
    }
}
```



InterruptedThread and JoiningHands Demo

Objective

- Understand the concept of a Thread and its usefulness for programming;
- Be able to write basic concurrent programs in Java;
- Understand the causes of basic concurrency errors
- Understand the mechanisms that help prevent the basic concurrency errors.

Risks of threads

- Safety Hazard
  - System behave incorrectly
- Liveness Hazard
  - System fails to make forward progress (deadlock, starvation, livelock)
- Performance Hazard
  - Impair service time, responsiveness, throughput, resource consumption, or scalability of the system.

Thread Safety

- Is about correctness

  Correctness means that a class conforms to its specification.
  A good specification defines invariants constraining an object's state and postconditions describing the effects of its operations.

- Is about managing access to ***state***

  in particular to *shared, mutable state*.

```java
public class Factorizer {

    public void service(Request req, Response resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
       encodeIntoResponse(resp, lastFactors);
    }
}
```

Stateless objects are always thread-safe!

```java
public class Counter {
    private int value;
    /**
    *
    * @return the counter value that’s increased by 1
    */
    public int getNext() {
        return ++value;
    }
}
```

not *atomic*

read-modify-write operation

S1. read the value

S2. add one to it

S3. write out the new value

Thread			
Thread A	value -> 4	4+1 -> 5	value = 5
Thread B	value -> 4	4+1 -> 5	value = 5

```java
class LazyInit {
   private ExpensiveObject instance = null;

   public ExpensiveObject getInstance() {
      if (instance == null)
         instance = new ExpensiveObject();
      return instance;
   }
}
```

Check-then-act operation

not *atomic*

S1. observe instance to be null

S2. create new instance

Race Condition

- A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime;

- Result of compound actions:
  - read-modify-write sequences
  - Check-then-act

Fix: ensuring atomicity

- Using thread safe objects
- Locking

```java
public class Counter {
   private AtomicInteger value = new AtomicInteger(0);
   /**
   *
   * @return A unique value in the sequence
   */
   public int getNext() {
      return value.incrementAndGet();
   }
}
```

What about when there’s more than one state variables?

The java.util.concurrent.atomic package contains *atomic variable* classes for effecting atomic state transitions on numbers and object references.

```java
class MutableName {
  private AtomicReference<String> aName;
  private AtomicInteger aCount;

  public void setName(String pName) {
    aName.set(pName);
    aCount.incrementAndGet();
  }
}
```

Still has the problem of race condition

Locking

Reference to an object

```java
synchronized (lock) {
   // Access or modify shared state guarded by lock
}
```

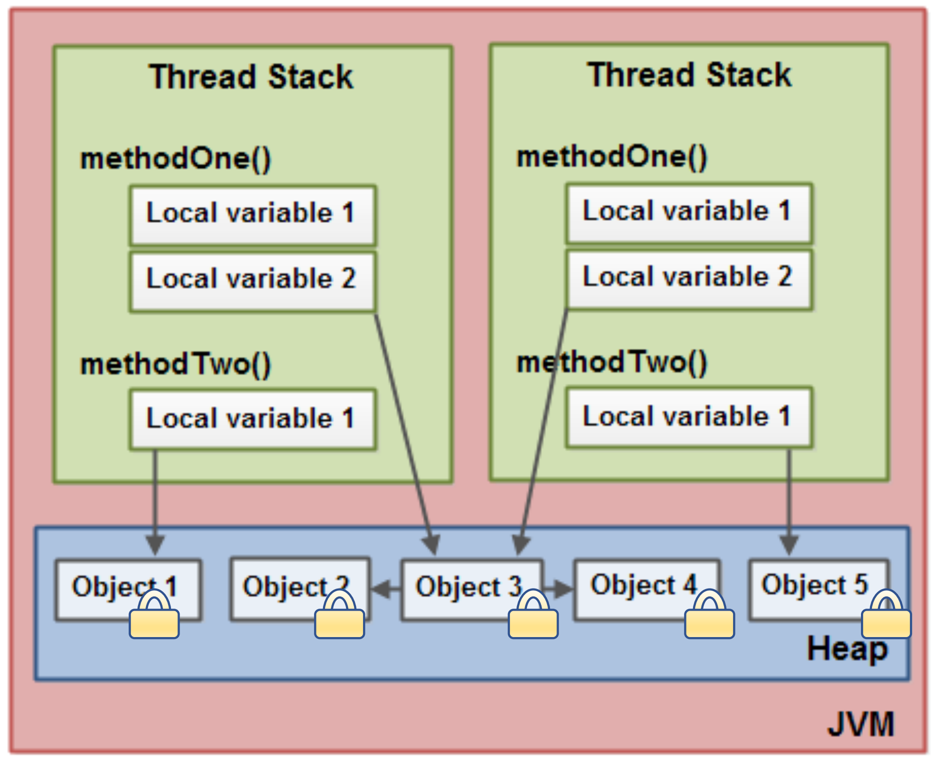
A group of statements in the block appear to execute as a single, indivisible unit.

No thread executing a synchronized block can observe another thread to be in the middle of a synchronized block guarded by the same lock.

```java
class MutableName {
   private String aName;
   private int aCount;

   public synchronized void setName(String pName) {
        aName = pName;
        ++aCount;
   }
}
```

When Thread A is executing a setName, Thread B that also try to invoke setName has to wait until the Thread A is done.

Intrinsic lock->



Memory Visibility

- To ensure that when a thread modifies the state of an object, other threads can actually *see* the changes that were made.

Memory Visibility

```
class MutableName {
   private String aName;
   private int aCount;

   public void setName(String pName) {
      aName = pName;
      ++aCount;
   }

   public int getCount() {
      return aCount;
   }
}
```

Thread A

Thread B

Memory Visibility

```java
class MutableName {
   private String aName;
   private int aCount;

   public synchronized void setName(String pName) {
      aName = pName;
      ++aCount;
   }

   public synchronized int getCount() {
      return aCount;
   }
}
```

Thread A

Thread B

Memory Visibility

Preserve memory visibility but not atomicity

```
class MutableName {
  private volatile String aName;

  public void setName(String pName) {
    aName = pName;
  }

  public String getName() {
    return aName;
  }
}
```

Thread Safety

- A class is *thread-safe* if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.