Threads

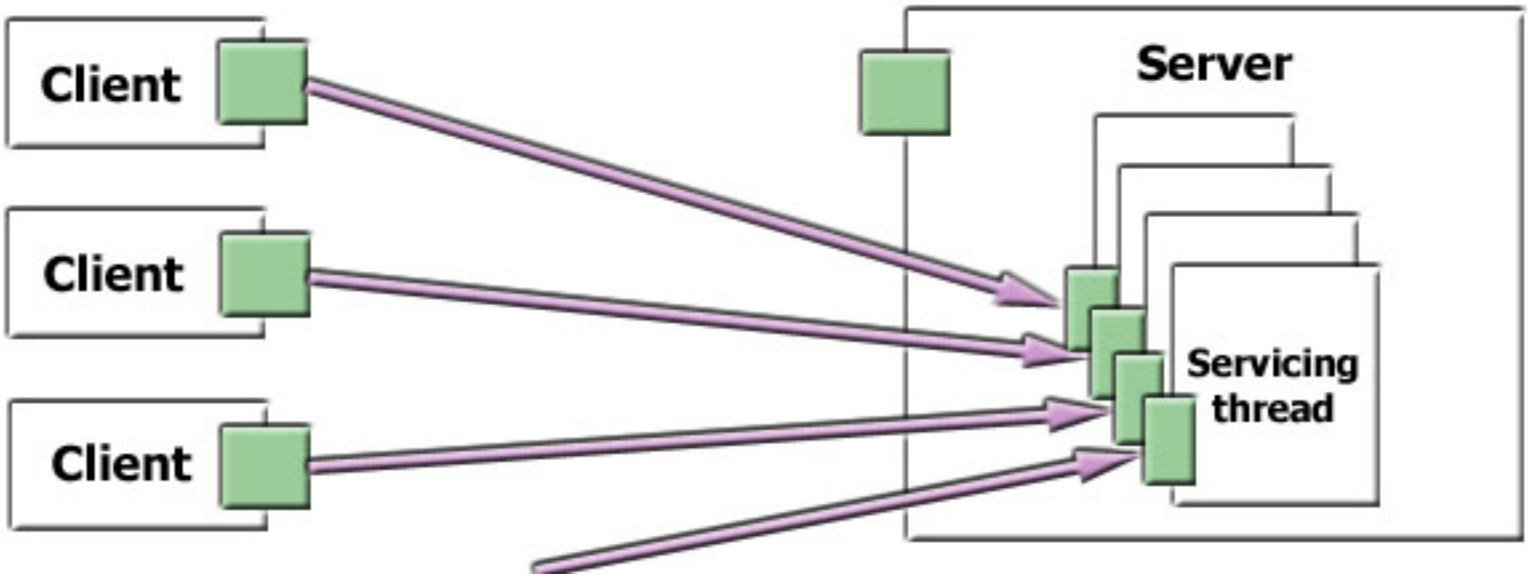
Creating and Managing Multiple Threads

CS3524 Distributed Systems

Lecture 04

Threads

- A thread is a single sequential flow of control within a program
- Example: one server – many clients:
  - How can the server serve all clients?



Processes and Threads

- A process is a self-contained execution environment
  - Has a private set of basic run-time resources, in particular, each process has its own memory space
  - Sometimes regarded as being synonymous with programs or applications started on a computer (although an application may run distributed over many processes)
  - Inter Process Communication (IPC) resources are provided by operating systems to allow exchange of data, such as pipes and sockets
  - E.g.: the Java VM runs as a single process
- **Threads** exist within a process
  - many threads may execute concurrently within such a process (e.g. the Java VM)
  - All Threads share the resources owned by the process, e.g. memory space – this allows for efficient exchange of information between threads but requires extra means to solve problems of concurrent access to these shared resources
  - Multi-threaded execution is an essential feature of Java

Uses of Threads

- To increase responsiveness of a system, actions take place concurrently:
  - ***Network programming*** – messages arrive from the network and must be dealt with while the process is doing other things
  - ***Real-time graphics*** – animation must continue to run while other processing happens
  - ***User Interface Programming*** – applications become far more responsive, e.g. GUI immediately reacts to mouse click

Java Threads

- In Java, programming with multiple threads of control is relatively easy
  - Concentrate on a single thread of execution for a specific task
  - execution environment (Java VM) and operating system handle the scheduling and task switching
- Two objects are always involved in running java threads
  - A Java language object: **`java.lang.Thread`**
  - A "target" object that contains a method for the thread to execute

Creating Java Threads

- Creating Threads – two possibilities:
  - Thread object extends class

    extend **`java.lang.Thread`**

  - Thread object implements interface:

    implement **`java.lang.Runnable`**

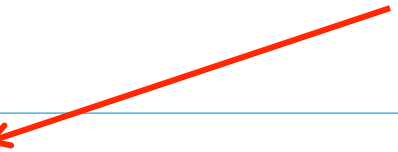
    - instantiate class **`Thread`** itself with a **`Runnable`** object as its target

Creating and Executing Java Threads

- Executing Threads
  - The `start()` method:
    - A thread becomes active, when its **`start()`** method is called
- Implementing thread functionality
  - The `run()` method:
    - The "target" Java class, which implements thread code, executes within a thread and has to implement a method "**`run()`**"
    - this is the code executed by the thread, the thread's **`start()`** method will call **`run()`**

Creating Threads

- The “target” class can be a subclass of **`Thread`**



```
import java.lang.Thread;

class ExampleOne extends Thread
{
  public void run ( )
  {
        //do something
  }
}

public class ExecuteSomeThread
{
  public static void main( String args[])
  {
    new ExampleOne().start();
  }
}
```

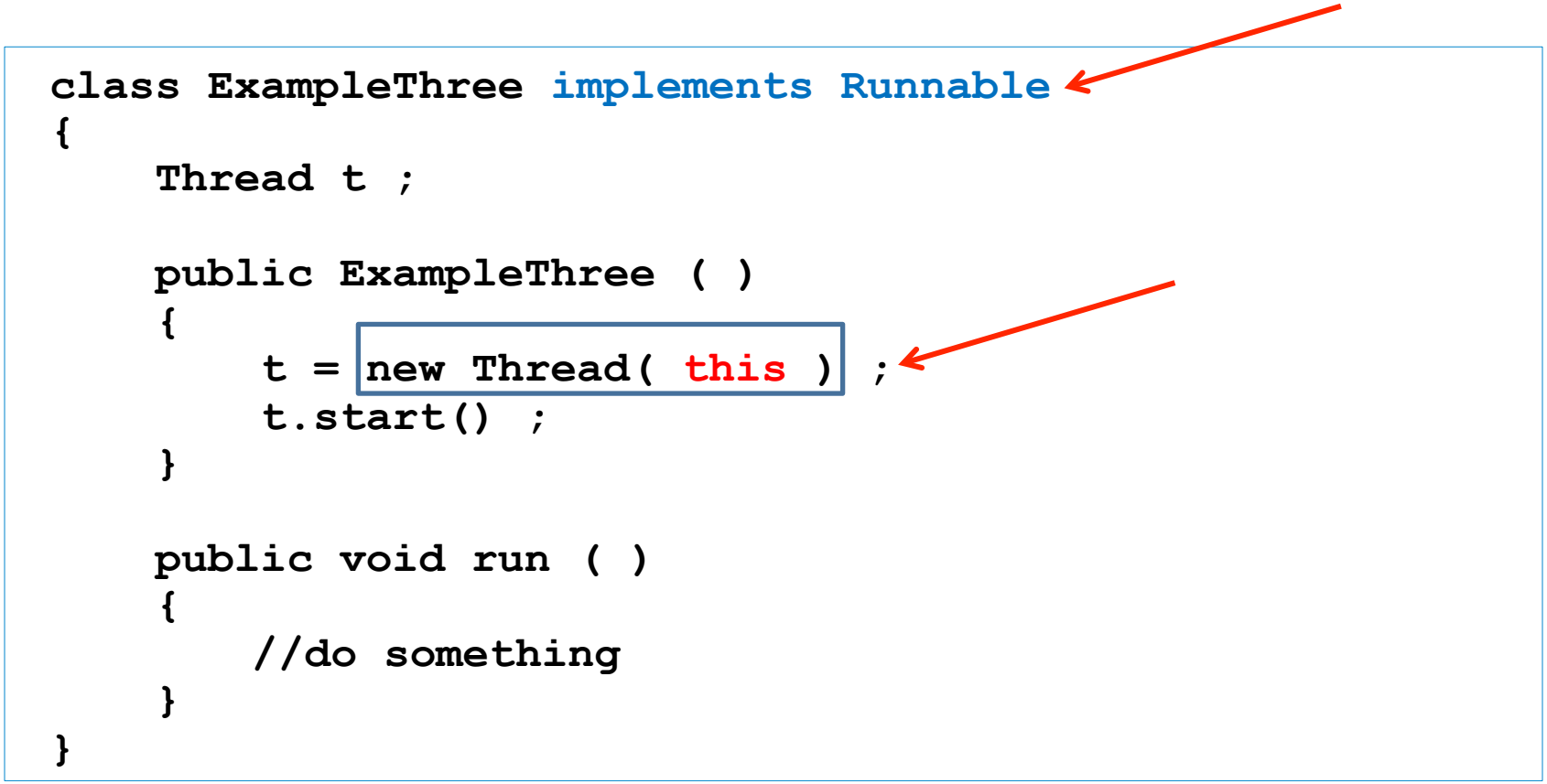
Creating Threads

- The "target" class can implement **Runnable**

```
import java.lang.Thread;
import java.lang.Runnable;

class ExampleTwo implements Runnable
{
  public void run ( )
  {
        //do nothing
  }
}

public class ExecuteSomeThread
{
  public static void main( String args[])
  {
    new Thread( new ExampleTwo() ).start();
  }
}
```

Creating Threads

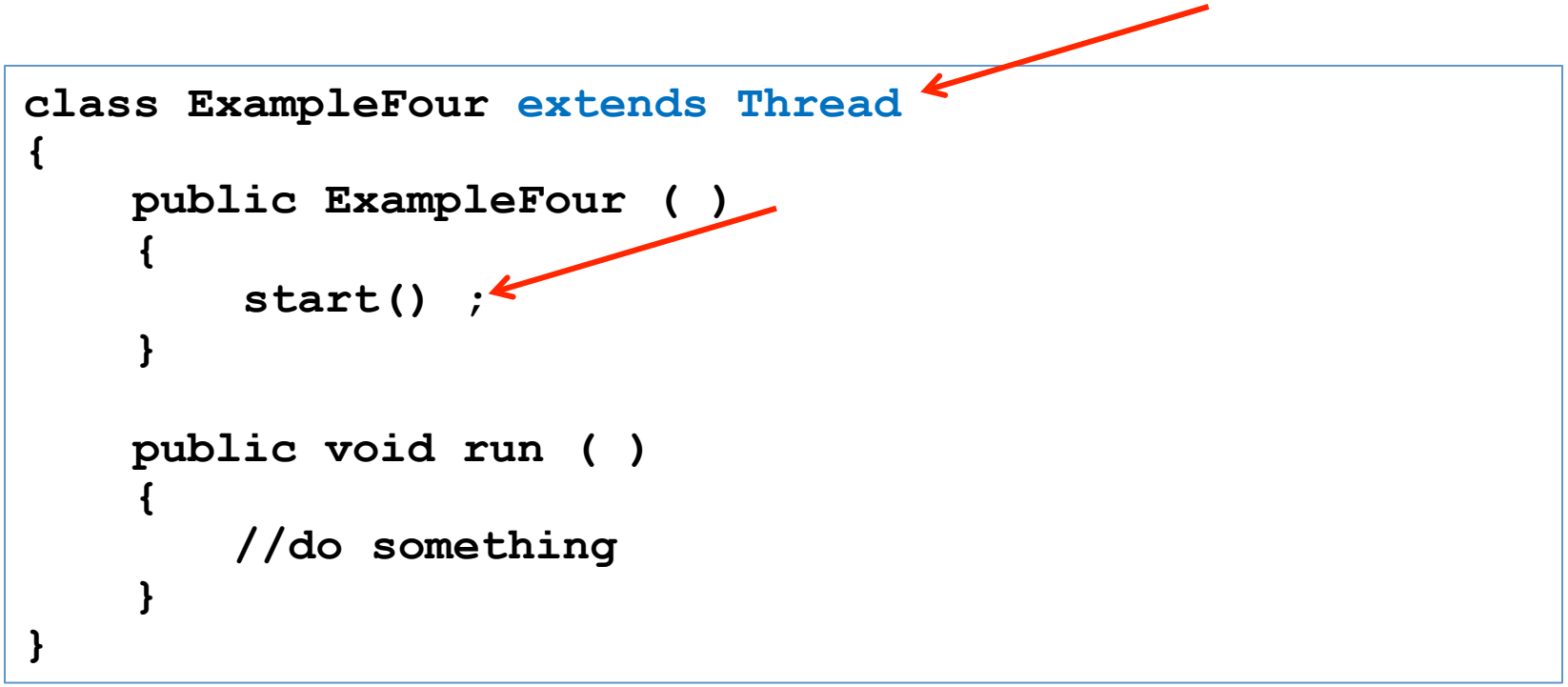
- A target class can implement the creation of its own thread:



```
class ExampleThree implements Runnable
{
    Thread t ;

    public ExampleThree ( )
    {
        t = new Thread( this ) ;
        t.start() ;
    }

    public void run ( )
    {
        //do something
    }
}
```

Creating Threads

- A target class can be an implementation of a thread:



```
class ExampleFour extends Thread
{
    public ExampleFour ( )
    {
        start() ;
    }

    public void run ( )
    {
        //do something
    }
}
```

Example: Extending class **`Thread`**

- The Thread object represents the thread in the Java VM – we use this object to control the thread.
- Once a thread is running, it can call the methods of any object that it has access to in the same program
- First example: extending the **`Thread`** class:

```
public class MyThread extends Thread
```

- Objects of the new subclass can then be instantiated and be used to manage a new thread

Example: Extending class **`Thread`**

```
import java.io.* ;
import java.lang.Thread.* ;

// Design pattern used: extend the Thread class and
// override the run() method

public class MyThread extends Thread
{
   // Record the name of the thread and initialise it
   // in the constructor
   private String _name ;

   public MyThread(String name)
   {
      _name = name ;
   }
   . . .
   . . .
```

Example: Extending class **Thread**

```
public void run()
{
    int count = 5
    int sleeptime = 100 ;

    while ( count > 0 )
    {
        count -- ;
        try
        {
            Thread.sleep ( sleeptime ) ;
            System.out.println ( _name + " Thread" ) ;
        }
        catch ( InterruptedException ie )
        {
            System.out.println ( "Thread problem" ) ;
        }
    }
}
```

Example: Extending class **`Thread`**

```
    // Within the main method, we just create a few
    // named MyThread objects and start them
    // causing the run() method to be called.

    public static void main ( String[] args )
    {
        // Create three objects to run threads.

        MyThread t1 = new MyThread ( "Fred" );
        MyThread t2 = new MyThread ( "Ted" );
        MyThread t3 = new MyThread ( "Ed" );

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Example: Implement interface **`Runnable`**

- Second Example:
  - Class Thread uses a target class that implements the **`Runnable`** interface
    - An object of this class can then be passed to a Thread class constructor
    - This will create a new thread which will start by calling the **`run()`** method of the target class

```
public class MyThreadRunnable implements Runnable
```

- The target's **`run()`** method is the place where a thread starts its flow of control. The **`run()`** method usually calls more specific methods

Example: Implement interface **Runnable**

```
public class MyThreadRunnable implements Runnable
{
   private String _name ;

   public MyThreadRunnable (String name)
   {
      _name = name ;
   }
   public void run () { // same as before. }

   public static void main ( String[] args )
   {
      Thread t1 = new Thread ( new MyThreadRunnable( "Fred" ) ) ;
      Thread t2 = new Thread ( new MyThreadRunnable( "Ted"  ) ) ;
      Thread t3 = new Thread ( new MyThreadRunnable( "Ed"   ) ) ;

      t1.start();
      t2.start();
      t3.start();
   }
}
```

Thread Groups

- Thread groups (the ThreadGroup class in the java.lang package) provide a mechanism for collecting multiple threads into a single object and manipulating all the threads at once
- Creating a new thread without specifying the group in the constructor places it in the same group that created it (default is the main group)

```
ThreadGroup myThreadGroup = new ThreadGroup("My Group");
Thread myThread = new Thread(myThreadGroup, "A thread");

ThreadGroup theGroup = myThread.getThreadGroup();
```

Java Thread Life-Cycle

- The Java thread objects provide a number of methods to control the life-cycle of threads:
  - **`run()`** – implements the code to be executed by the thread
  - **`start()`** – starts a thread
  - **`sleep()`** – a thread will wait for the specified amount of time
  - **`join()`** – can be used to synchronize execution between threads, one thread will wait for another thread to terminate before continuing execution
  - **`yield()`** – causes the thread to give up the processor and allow other threads to execute

Pausing Execution

- We use sleep() to pause the execution:

```
public class SleepMessages
{
    public static void main(String args[]) throw InterruptedException
    {
        String importantInfo[] = { "First message",
                                   "Second message",
                                   "Third message",
                                   "Fourth message" };

       for (int i = 0; i < importantInfo.length; i++)
       {
           //Pause for 4 seconds
           Thread.sleep(4000);
           //Print a message
           System.out.println(importantInfo[i]);
       }
     }
}
```

Java Thread Life-Cycle

- A thread continues to execute until
  - It returns from the target's run() method
  - It is interrupted by an uncaught exception
- A process (the Java VM) will not terminate until all its threads have terminated

Manipulate the Execution of Threads

- Use a static variable to communicate certain information to threads
  - A static variable, specified in a class, is shared between all objects that are instantiated from this class
  - Therefore, all threads will also share a single static variable
  - We can check conditions over shared variables – they are called "guard conditions" as they decide whether a thread continues to execute

Example: Stop a Thread

Implementation

```
import java.io.*;
import java.lang.Thread.*;
import java.util.Date;

public class StopMe implements Runnable
{
   static boolean _keepgoing = true;
   public void run()
   {
      while (_keepgoing)
      {
         System.out.println( new Date() );
         try
         {
            Thread.currentThread().sleep(1000);
         }
         catch (InterruptedException ie) { }
      }
   }

   public static void main(String[] args)
   { . . . }
}
```

Use a static boolean variable to controll the execution of the thread

Example: Stop a Thread

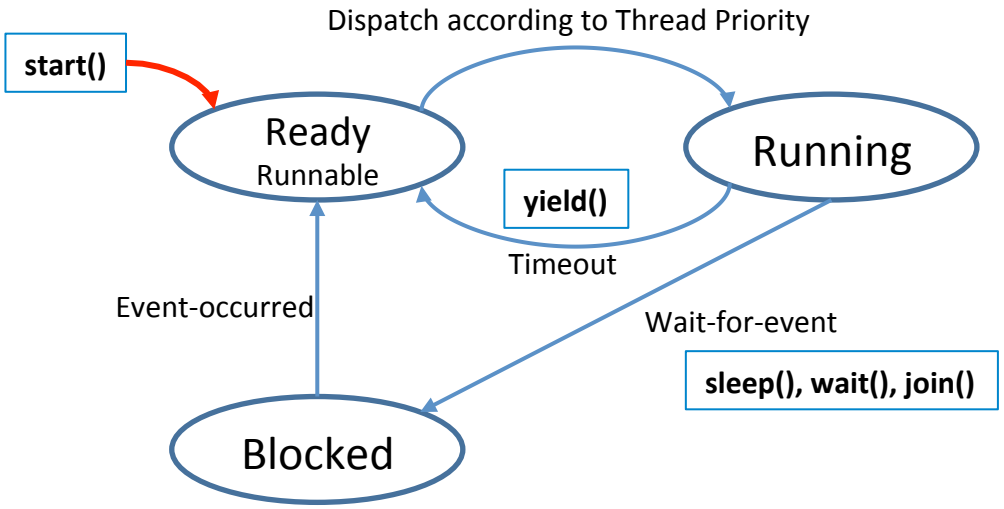
Starting and Stopping

```
import java.io.*;
import java.lang.Thread.*;
import java.util.Date;

public class StopMe implements Runnable
{
   static boolean _keepgoing = true;

   public void run()
   { . . . }

   public static void main(String[] args)
   {
      Thread t = new Thread( new StopMe() );
      t.start();
      try
      {
         Thread.currentThread().sleep(10000);
      }
      catch (InterruptedException ie) { . . .}
      _keepgoing = false;
   }
}
```

In the main method, create the thread that is writing the date every second, sleep for 10 seconds and then communicates to the "StopMe" thread to stop via the static boolean variable "_keepgoing"

Thread Scheduling and Priorities

Thread Scheduling



- Calling start() will make thread "**runnable**", thread is put into Ready queue
- Threads are dispatched according to priority
- Thread can give up processor with yield()
- Thread can put itself into Blocked state
  - If thread calls sleep(), wait(), join() – it will be transferred to the Blocked queue

Thread Scheduling

- Scheduling of Java threads depends on the operating system
  - Modern OS: pre-emptive scheduling, time slices
  - If OS allows thread scheduling, Java threads are mapped onto OS threads
    - Linux, Windows, OSX manage threads
- The Java run-time environment uses a fixed-priority scheme to decide which thread to execute next
  - The thread with the highest priority runs first
  - The thread is run for a specific time (one "time slice"), after that it enters the **runnable** state – OS scheduler decides whether to let the thread continue or schedule a different thread

Thread Priority

- A thread is scheduled according to its priority with respect to other **runnable** threads
- The priority of a thread
  - Priority is an integer – the higher the integer, the higher the priority
  - **`java.lang.Thread`** contains three integer constants for determining the range of thread priorities:
    - **`Thread.MIN_PRIORITY = 1`**
    - **`Thread.NORM_PRIORITY = 5`**
    - **`Thread.MAX_PRIORITY = 10`**
  - The default priority of a thread is **`NORM_PRIORITY`**
  - When a thread is created, it takes the priority of the thread that created it
  - The priority of a thread can be checked with the method **`Thread.getPriority()`**
  - The priority of a thread can be changed with the method **`Thread.setPriority()`**

Scheduling Threads in Java

- A thread chosen by the scheduler will run until
  - time slice has expired
  - It yields by calling the method **`yield()`**
  - It waits by calling **`wait()`** or **`sleep()`**
  - Its **`run()`** method terminates
- Threads scheduled according to priority
  - If a higher-priority thread becomes runnable (and "preempts" the current thread), it will be selected from the Ready queue as the next thread

Using `yield()`

- Use priority for efficiency purposes, not for algorithm correctness
- Well-behaved threads **`yield()`** to threads of the same priority (even if they don't, e.g., **`sleep()`**)

```
public void MyThread extends Thread
{
    static boolean _keepGoing = true ;

    public void run()
    {
        while ( _keepgoing )
        {
            // do something useful
            yield() ;
        }
    }
}
```

Thread Concurrency

Thread Synchronisation

- Multiple threads run within the context of one process
  - Remember: they are called lightweight processes
  - They share a common *memory space*, but
  - They have their own *local variables* ("working memory")
- **Danger of race conditions**: they have to compete for resources within their execution context, e.g. access to global variables
- Therefore: *synchronisation* of thread activity becomes necessary

Thread Synchronisation

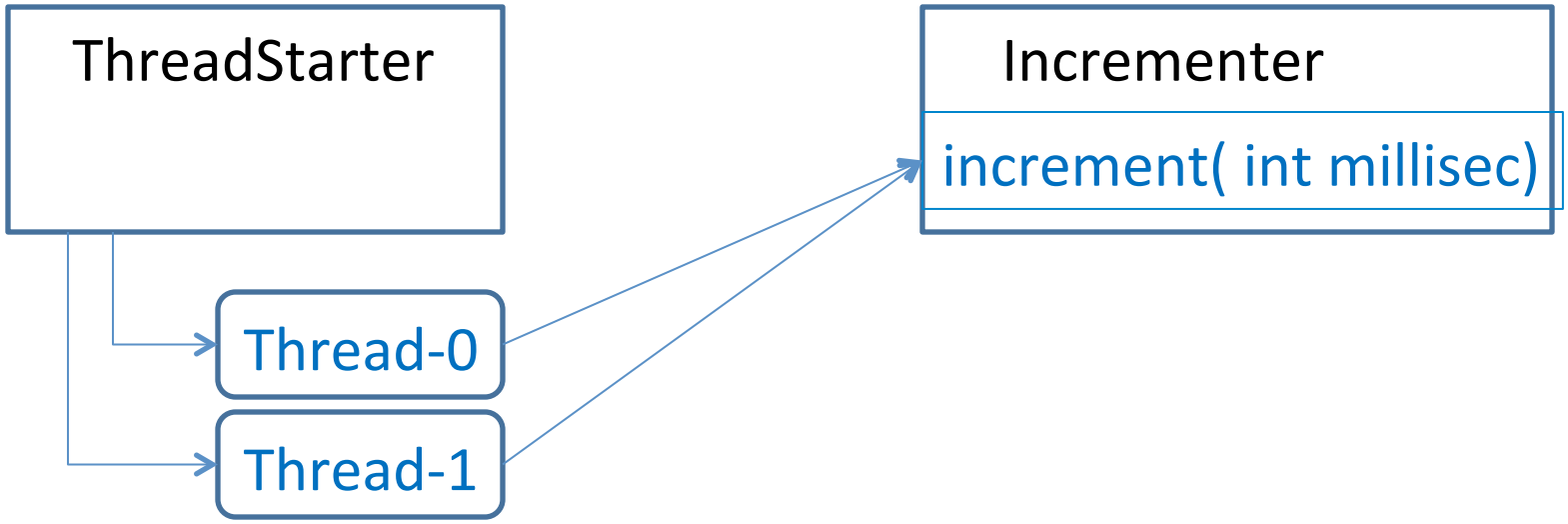
- Depending on operating system and hardware
  - Threads may run ***concurrently*** with the help of a scheduler
  - Threads may run truly ***parallel*** on multiple processors
- We have one class specification, but multiple threads executing code of this class concurrently
  - Threads may execute concurrently and independently without interfering with other threads
  - But: Threads (in most cases) interfere with each other – they access shared resources concurrently
- Goal: **Mutual Exclusion** between threads – only one thread at a time executes critical sections

Thread Synchronisation

- Race Conditions
  - Scheduler may switch between threads in an arbitrary fashion
  - We may create lost update / overwrite of data
- Critical Section
  - Critical sections of code have to be save-guarded in a special way – only one thread at a time should be able to execute such a section **in its completeness**

Example Scenario

- Two classes
  - ThreadStarter: creates two threads
  - Incrementer: provides a method increment() that is called by those threads



Thread Concurrency

```
public class Incrementer
{
   private int i = 0 ;
   private int j = 0 ;
   public void increment ( int milliseconds )
   {
      i ++ ;
      pause ( milliseconds ) ;
      j ++ ;
      System.out.println ( "Method increment(): " +
                                   ((i == j) ? "Same" : "Different") ) ;
   }

   private void pause( int p ) {
      try { Thread.currentThread().sleep( p ) ;
      } catch ( Exception e ) {}
   }
}
```

- What happens to the respective values of i and j ?

Thread Concurrency

```
public class ThreadStarter extends Thread
{
   static private Incrementer incrementer = new Incrementer() ;

   public ThreadStarter ( ) {}

   public void run () {
      incrementer.increment( 1000 ) ;
   }

   static public void main ( String args[] ) {
      ThreadStarter t1 = new ThreadStarter() ;
      ThreadStarter t2 = new ThreadStarter() ;
      t1.start() ;
      t2.start() ;
   }
}
```

- The incrementer is known to both threads

Thread Concurrency

- One particular run:
  - We cannot predict, which thread is scheduled first
  - Both interfere with each other in the incrementation of the variables i and j

```
After  incrementing i, Current thread: Thread-1, i = 1
After  incrementing i, Current thread: Thread-0, i = 2
After  incrementing j, Current thread: Thread-1, j = 1
Method increment(), printed from thread Thread-1: Different
After  incrementing j, Current thread: Thread-0, j = 2
Method increment(), printed from thread Thread-0: Same
```