61FIT3JSD
Fall 2022

Lecture 4

Multi-threaded programming & Applications

Lecture outline

- Multi-tasking & thread
- Multi-threaded
  - Design
  - Implementation
  - Applications

Multi-tasking

- Computers can execute multiple tasks:
  - in parallel or concurrently
- A task is a (decomposed) sub-problem
- Examples of multi-tasking:
  - display progress (in log in, searching, …)
  - perform multiple GUI tasks
  - display and receive messages from the network

Example: multi-threaded drawing

`lect04.gui.MultiprocessingDemo2`

Drawing a piece of the Mandelbrot set by dividing up the computation into multiple smaller tasks

Example: a network chat program

`lect04.net.GUIChat`

Thread

- Represents a single task
- Corresponds to a sub-set of procedures in a program
- Every Java program has the `main` thread
  - created by the JVM
- Other threads are created from `main` and so on

Multi-threaded

- Design guidelines
- Threadable class
- Mutual exclusion
- Volatile variable
- Thread management

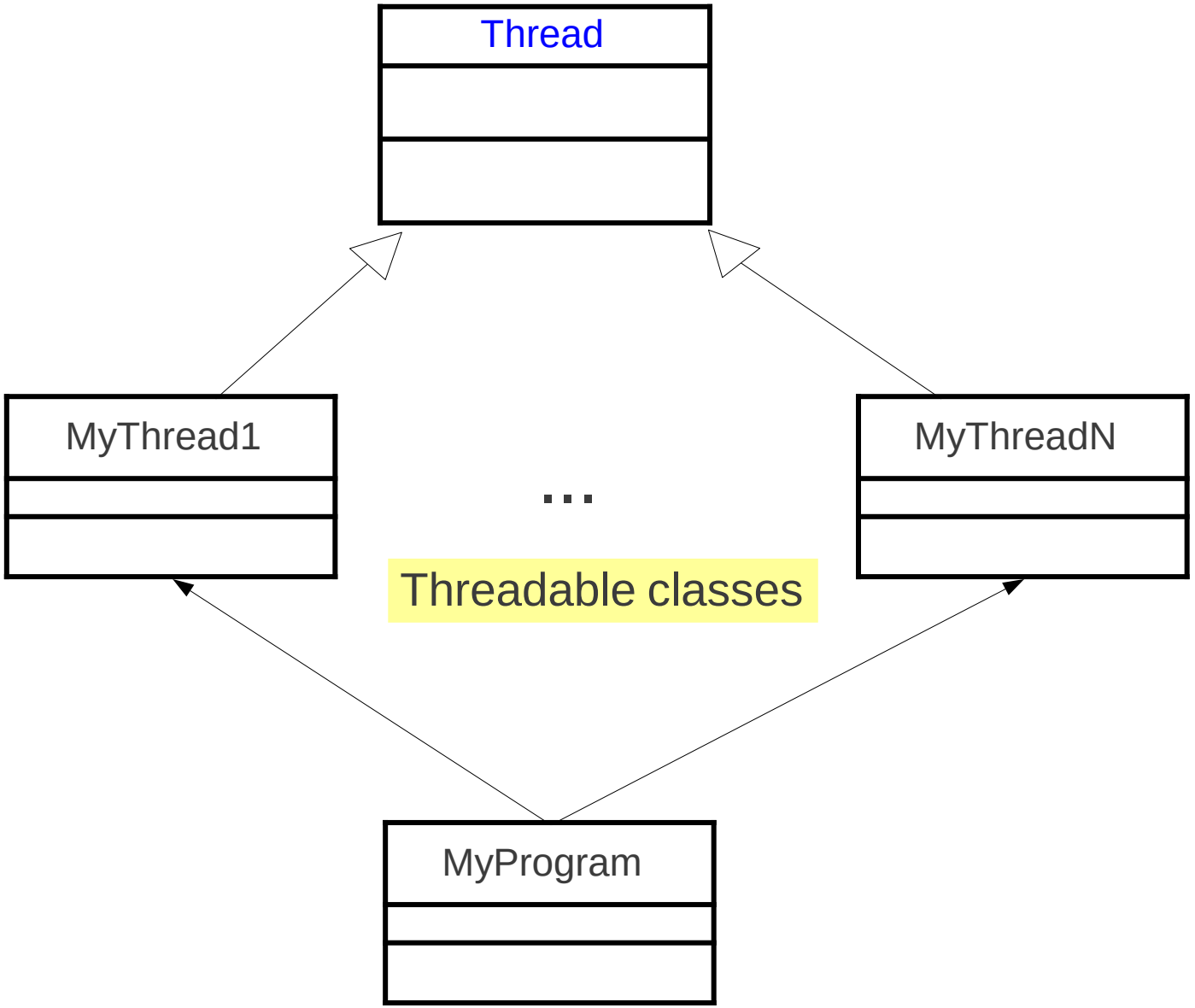
Design guidelines

- Decompose into subproblems
- Each subproblem is reasonably small
  - more subproblems than number of processors
- Subproblems may take unequal times
- Assign each subproblem to a thread
- Protect shared data
  - through mutual exclusion
- Manage threads
  - thread pool and queue

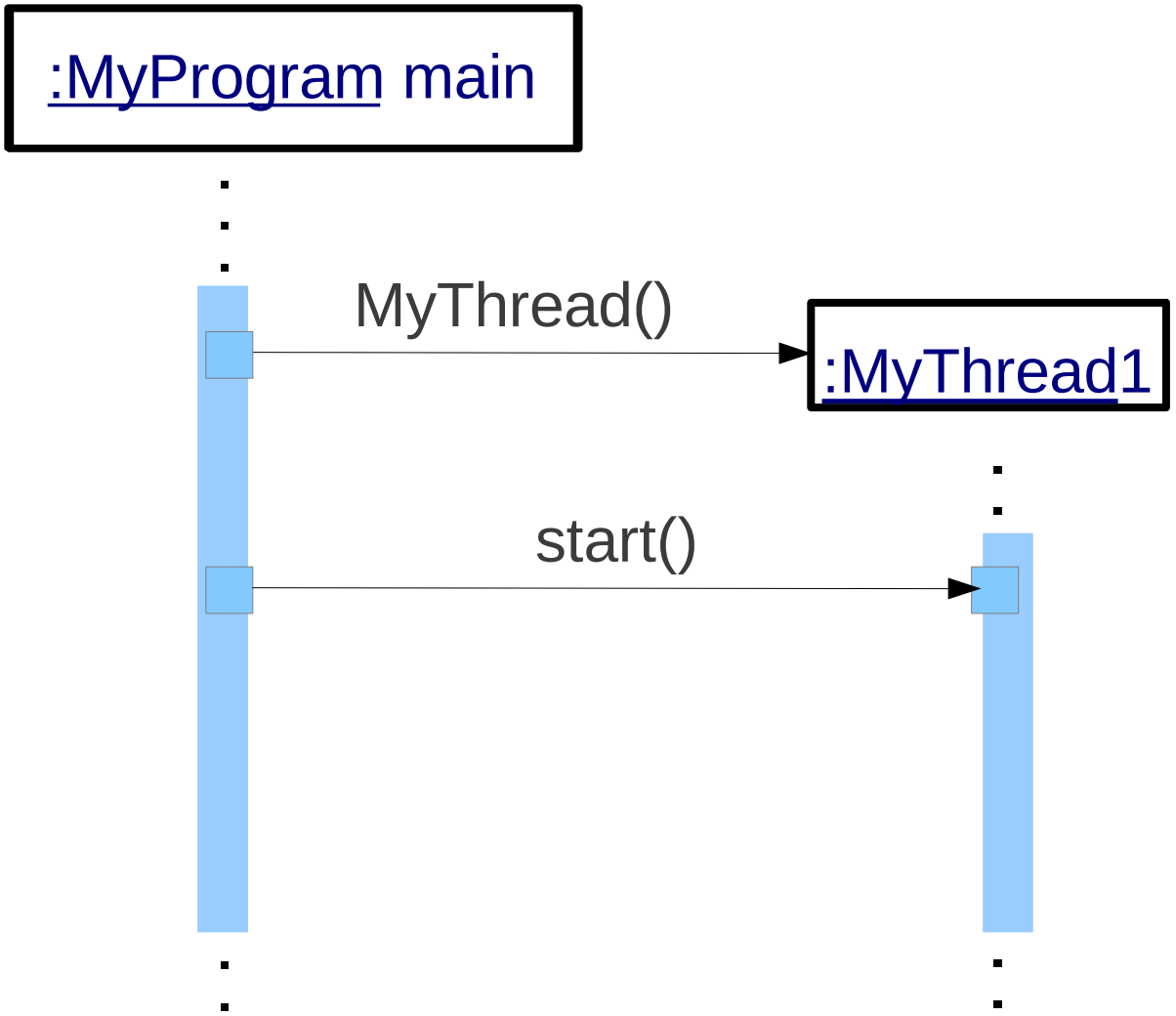
Threadable class

- A thread is an object
- Thread objects are instances of a *threadable class*
- Java supports two threadable class designs:
  - (1) as a sub-type of class `java.lang.Thread`
  - (2) as a sub-type of interface `java.lang.Runnable`

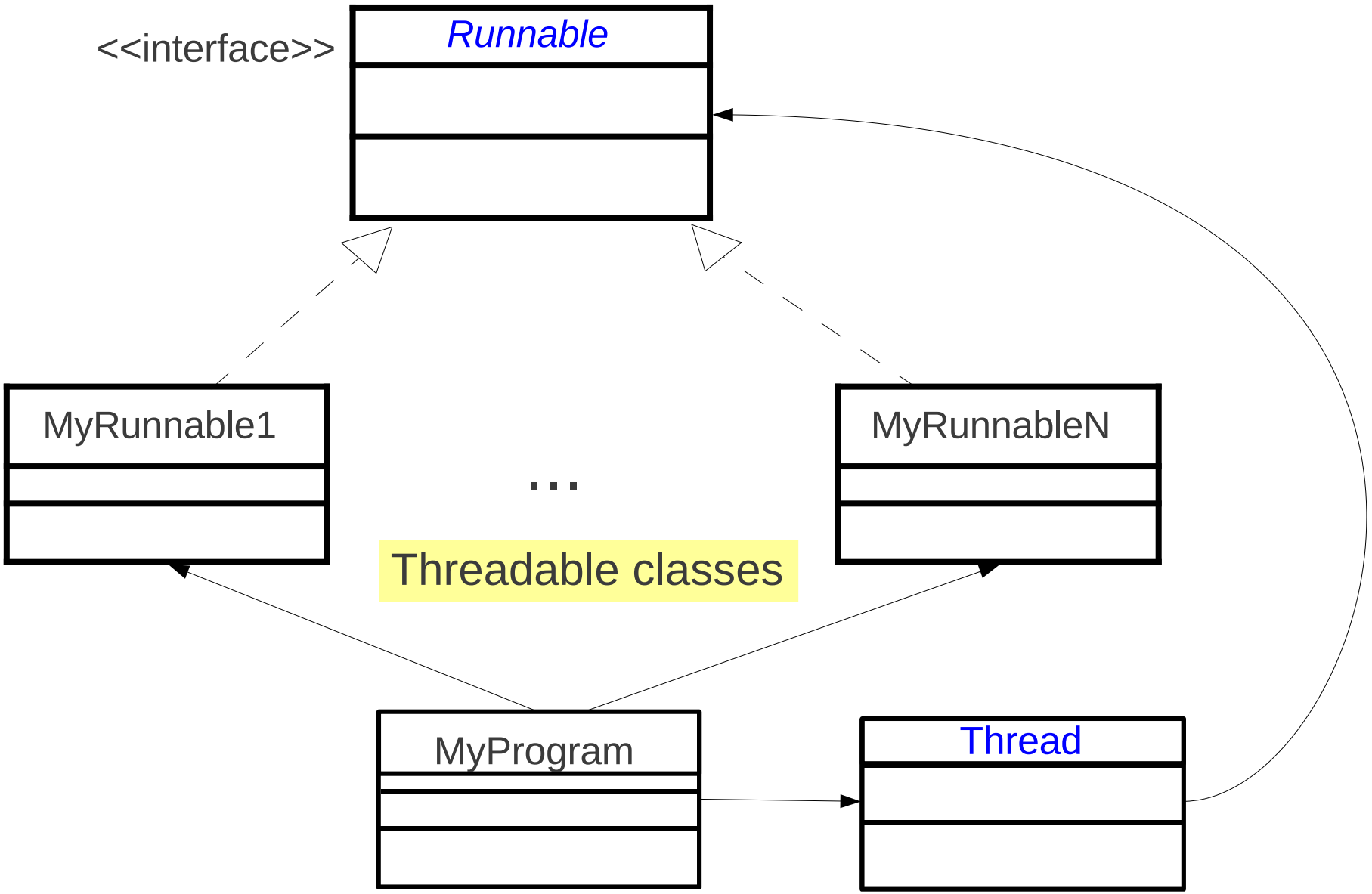
Threadable class (1)



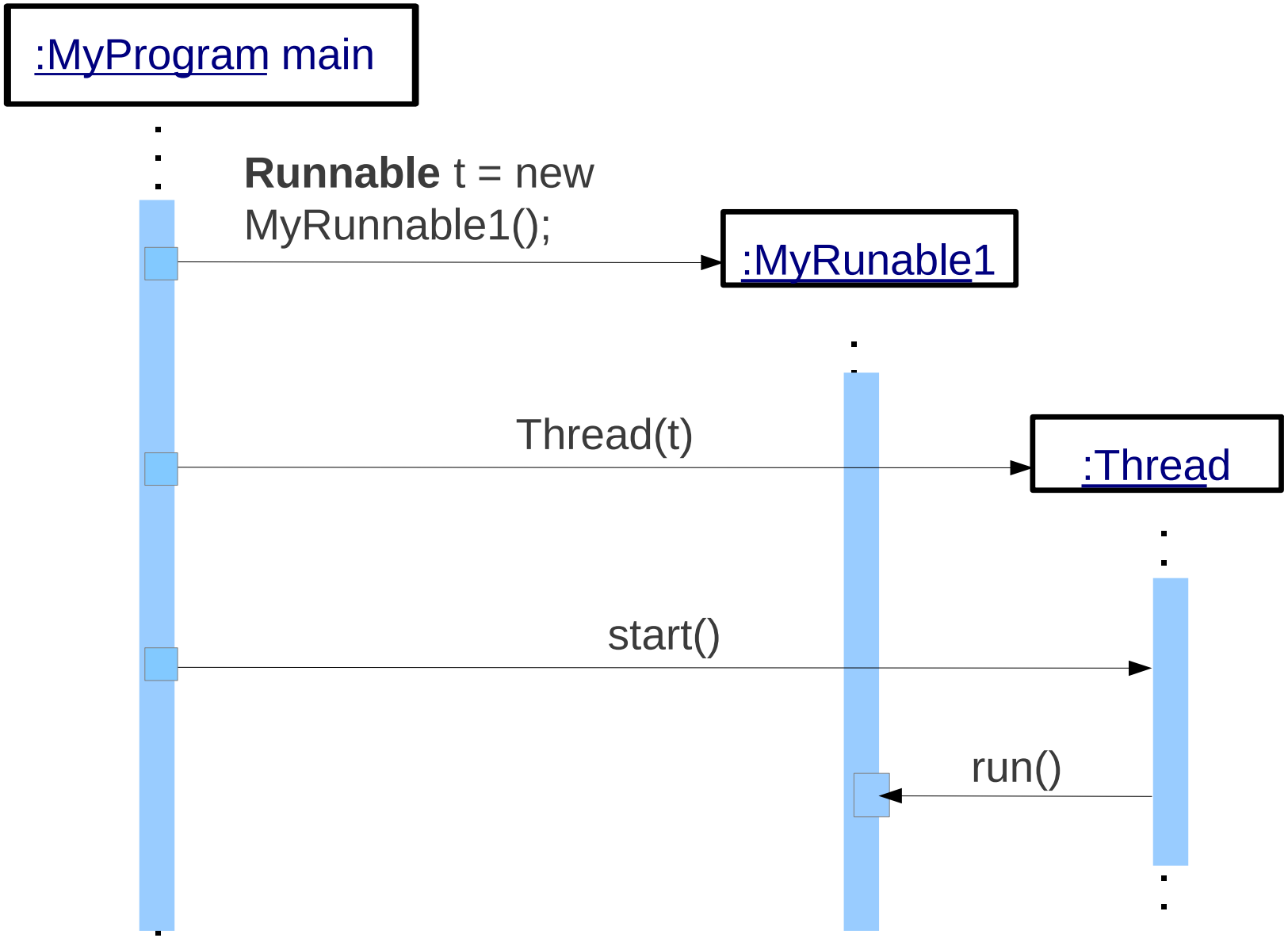
Sequence diagram



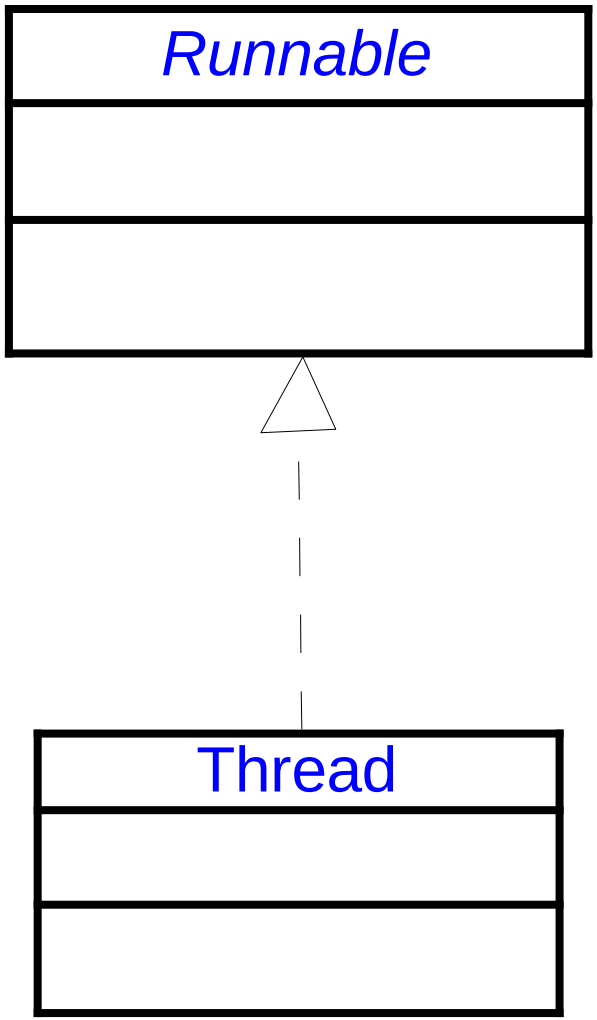
Threadable class (2)



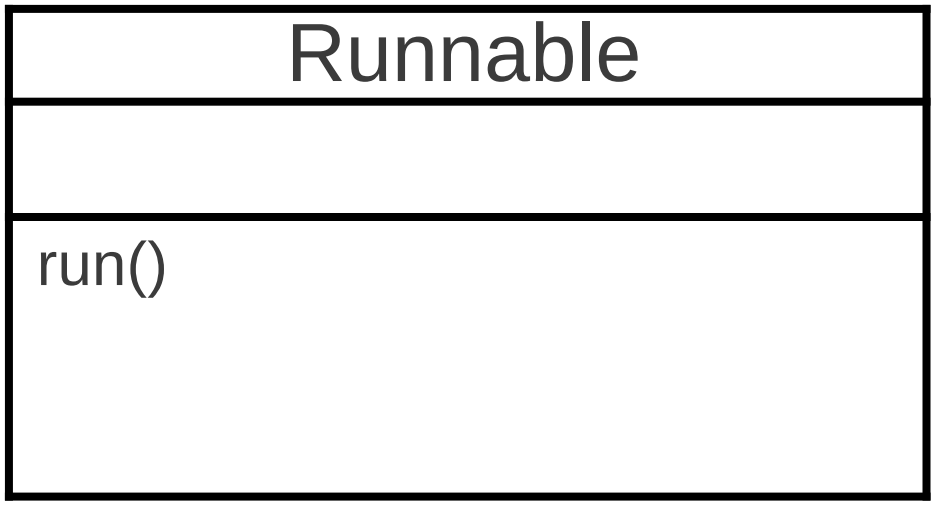
Sequence diagram



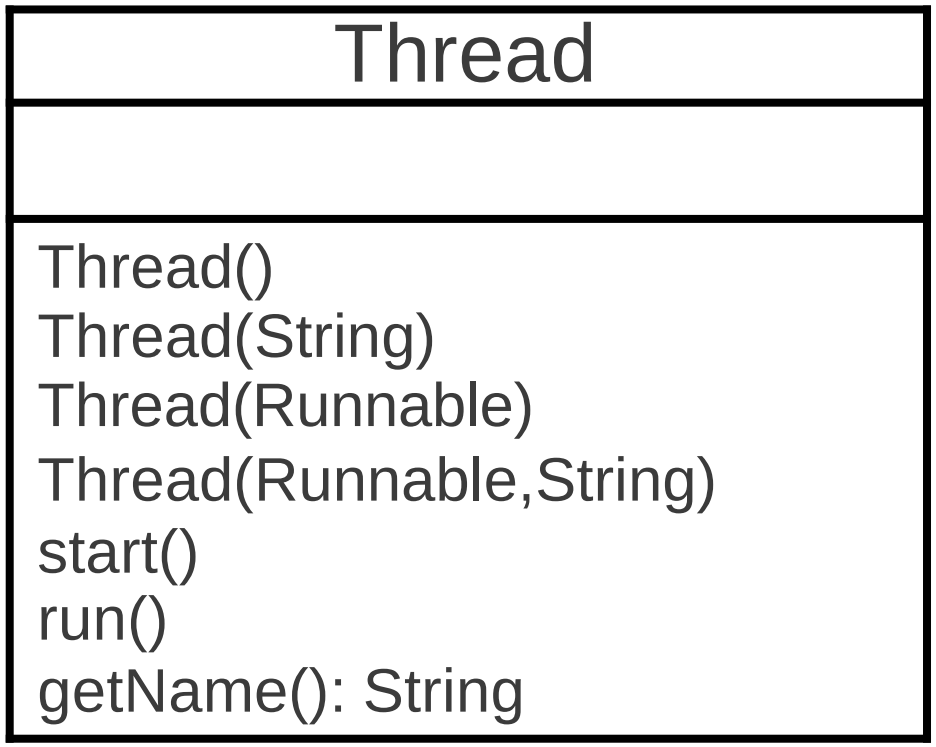
Thread implements Runnable



interface java.lang.Runnable



class java.lang.Thread



Implementation

- Design choice (1):
  - MyThread extends Thread
  - implements MyThread() or MyThread(String)
    - invokes super constructor
  - overrides run()
- Design choice (2):
  - implements Runnable
  - implements method run()

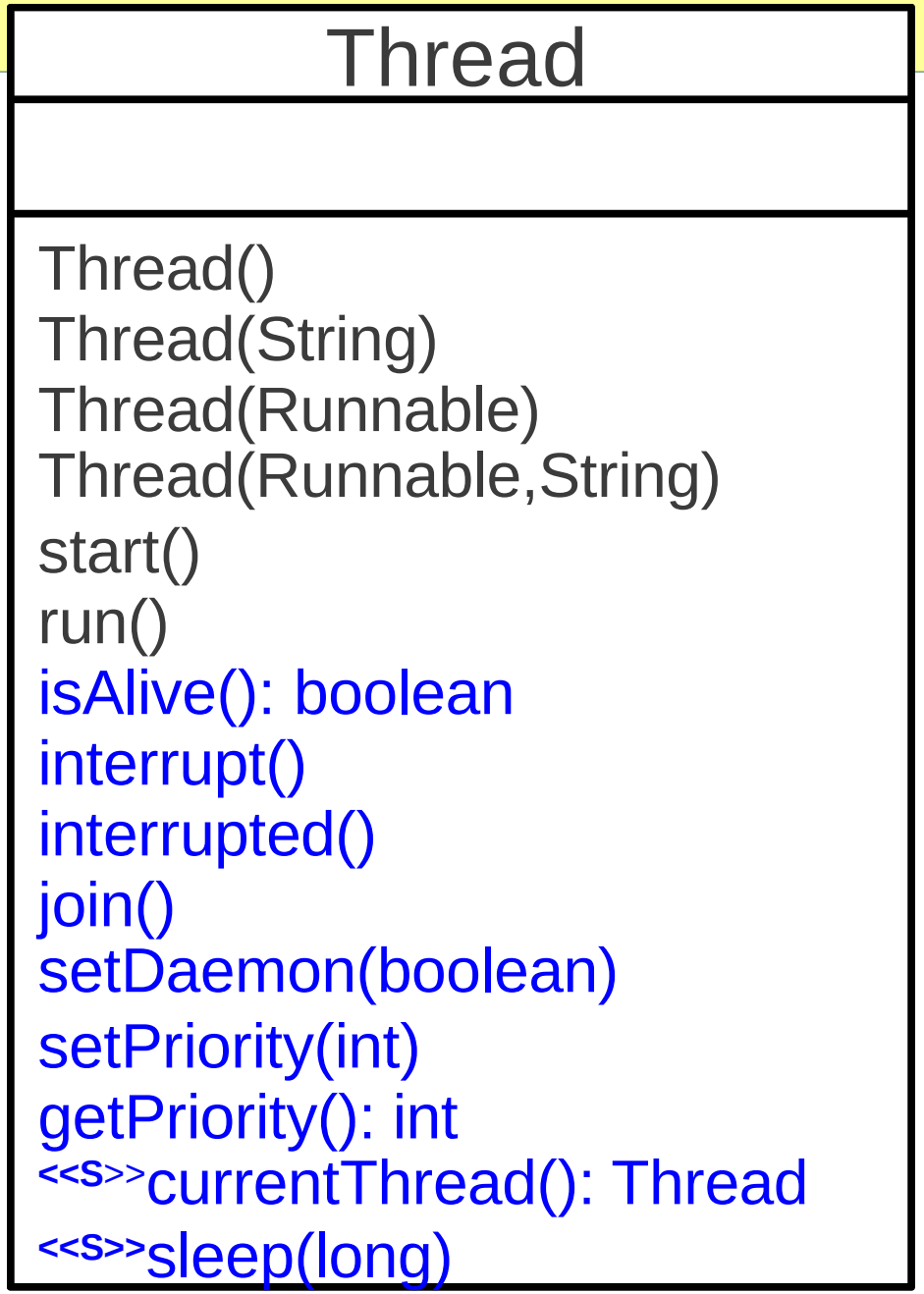
Example: CountPrimesThread

```
public class ThreadTest1 {
   private static class CountPrimesThread extends Thread {
     //int id;

     public CountPrimesThread(int id) {
       super("Thread " + id);
       //this.id = id;
     }

     @Override
     public void run() {
       long startTime = System.currentTimeMillis();
       int count = countPrimes(2, 1000000);
       long elapsedTime = System.currentTimeMillis() - startTime;
       System.out.println(getName()+" counted "+count+
        " primes in " + (elapsedTime / 1000.0) + " seconds.");
     }
   }

  //...
}
```

Example: ThreadTest1

```
public class ThreadTest1 {
   //....omitted ...
  /**
   * Start several CountPrimesThreads. The number of threads
   * is specified by the user.
   */
   public static void main(String[] args) {
     int numberOfThreads = 0;
     //....initialise numberOfThreads...
     CountPrimesThread[] worker = new
                          CountPrimesThread[numberOfThreads];

     for (int i = 0; i < numberOfThreads; i++)
       worker[i] = new CountPrimesThread(i);

     for (int i = 0; i < numberOfThreads; i++)
       worker[i].start();

     System.out.println("Threads have been created and started.")
   }

  //....omitted ...
}
```

class java.lang.Thread (full)



Thread methods

- **`join()`**
  The calling thread goes into a waiting state. It remains in a waiting state until the referenced thread terminates.
- **`join(long millis)`**
  Waits at most millis milliseconds for this thread to terminate.
- **`setDaemon(boolean on)`**
  Daemon thread is a low-priority thread (as opposed to a user thread).

  JVM will exit when all user threads terminate even if there's still a daemon thread running.

Thread interruption

An interrupt is **an indication to a thread that it should stop what it is doing and do something else**. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.

- `interrupt()`

  Interrupt the referenced thread (i.e. send interruption signal to the reference thread).

InterruptedException

Some methods (such as `Thread.sleep()`) are designed to throw `InterruptedException` when the thread receives interruption signal.

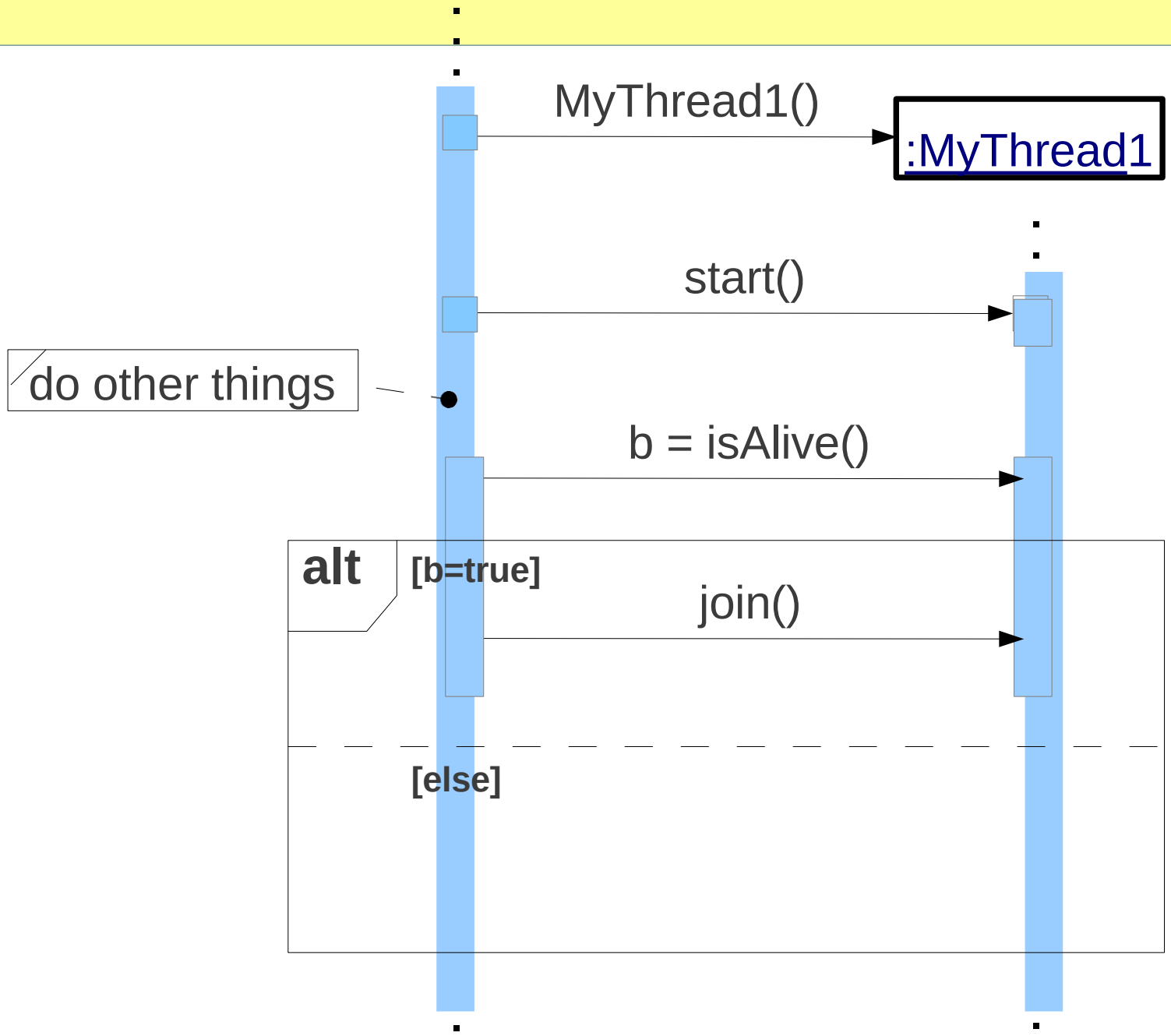
```
for (int i = 0; i < importantInfo.length; i++) {
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}
```

Handle Thread interruption

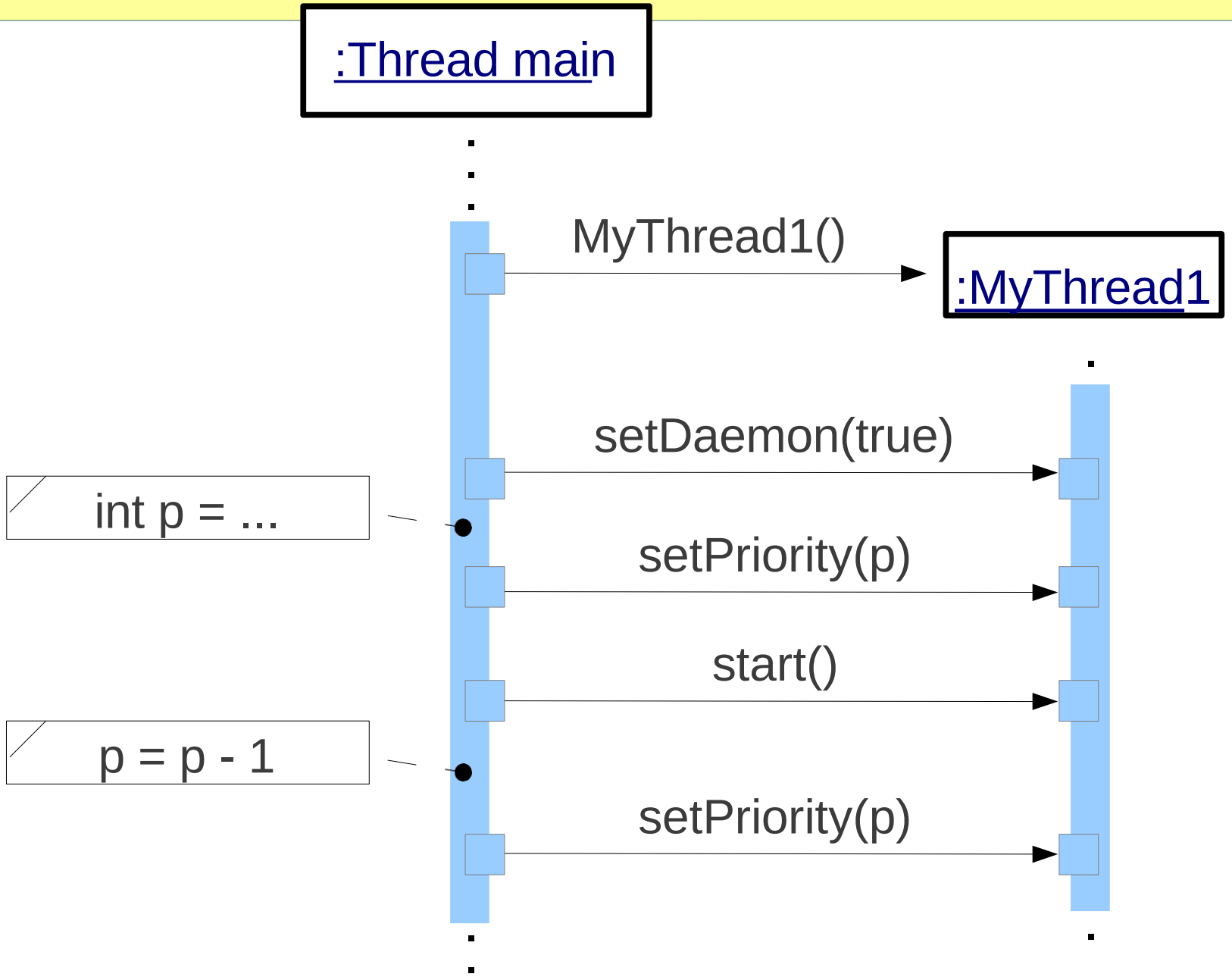
In other cases, we need to regularly check if our thread has received an interruption signal or not.

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

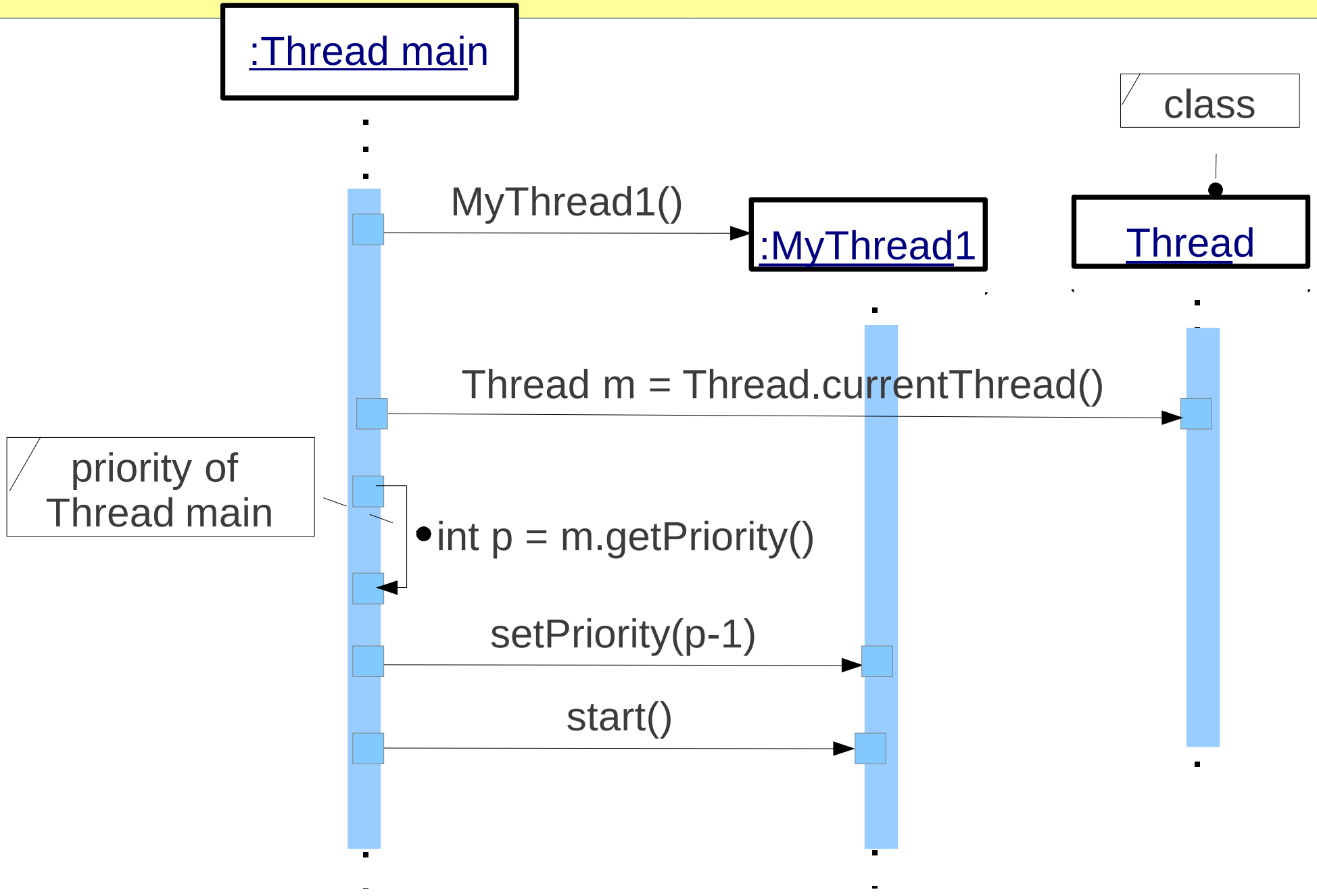
isAlive() & join()



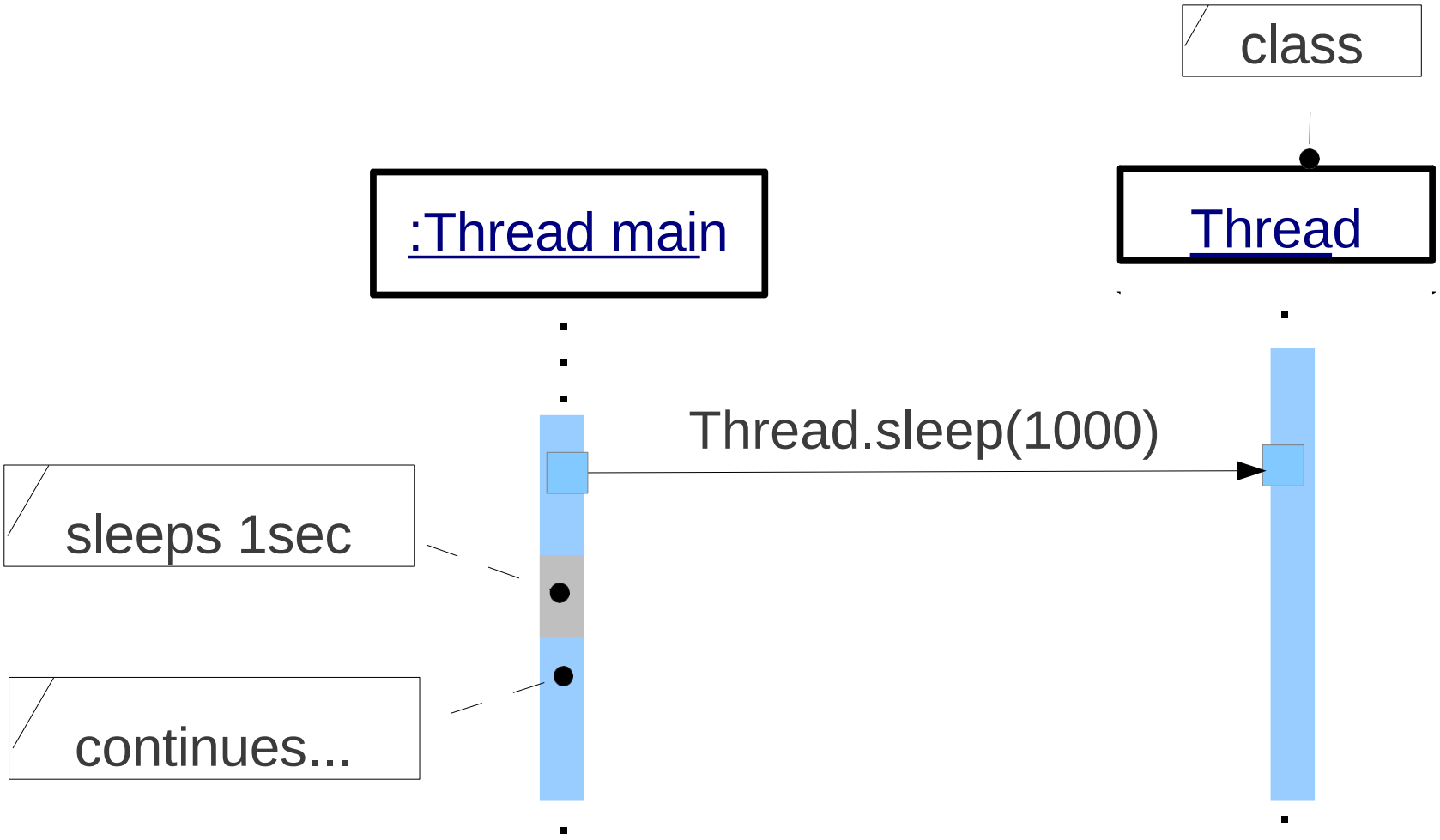
setDaemon() & setPriority()



<<static>> currentThread()



<<static>> sleep()



Mutual exclusion

- A control mechanism to protect a shared resource from threads
- Protected resource is defined using `synchronized` keyword
- Threads agree to obtain exclusive access to the resource:
  - also using `synchronized`
  - one thread has access, other threads wait for turns

Example: ThreadTest2 (1)

```
public class ThreadTest2 {

   //....omitted...
   /**
    * The total number of primes found....
    */
   private static int total;

   /**
    * Adds x to total.
    */
   synchronized private static void addToTotal(int x) {
      total = total + x;
      System.out.println(total + " primes found so far.");
   }

  //....omitted...

}
```

Example: ThreadTest2 (2)

```
public class ThreadTest2 {

   //....omitted...

  private static class CountPrimesThread extends Thread {
    int count = 0;
    int min, max;

    public CountPrimesThread(int min, int max) {
      this.min = min;
      this.max = max;
    }

    @Override
    public void run() {
      count = countPrimes(min, max);
      System.out.println("There are " + count + " primes
              between " + min + " and " + max);
      addToTotal(count);
    }
  }
}
```

Thread management

- To reduce overhead, thread objects are re-used for different tasks
  - tasks need to follow design choice (2)
- Thread objects are created in a **thread pool**
- Tasks are placed in a **queue**
- A thread
  - receives task from queue (or is blocked if empty)
  - processes the task
  - when finished terminate or repeats

Example: TaskQueue

```
public class TaskQueue {
   private static ConcurrentLinkedQueue<Runnable> taskQueue;

   public static void main(String[] args) {
      taskQueue = new ConcurrentLinkedQueue<Runnable>();

      int numTasks = 2;
      MyTask mt;
      for (int i = 1; i <= numTasks; i++) {
        mt = new MyTask(i);
        taskQueue.add(mt);
      }

      int threadCount = 6;
      WorkerThread t;
      for (int i = 1; i <= threadCount; i++) {
        t = new WorkerThread();
        t.start();
      }
   }
}
```

Example: MyTask

```
public class TaskQueue {
   //....omitted...
   private static class MyTask implements Runnable {
      private int id;
      public MyTask(int id) {
         this.id = id;
      }
      public void run() {
         int count = (int) (Math.random()*1000);
         for (int i = 0; i < count; i++) {
            System.out.print(id);
         }
      }
   }
   //....omitted...
}
```

Example: WorkerThread

```
public class TaskQueue {
   //....omitted...
   private static class WorkerThread extends Thread {
     public void run() {
       System.out.printf("Thread %s (id=%d) is started with "+
           priority %d%n",getName(), getId(), getPriority());
       while (true) {
         Runnable task = taskQueue.poll();
         if (task != null) {
           System.out.println("executing task: "+

((MyTask)task).id);
           task.run();
         } else {
           break;
         }
       }
     }
   }
}
```

Applications

- **Timer**
  - Thread.sleep(...)
- GUI
- Networking