Scientific Programming Using Object-Oriented Languages
Module 8: Threads

Aims of Module 8:

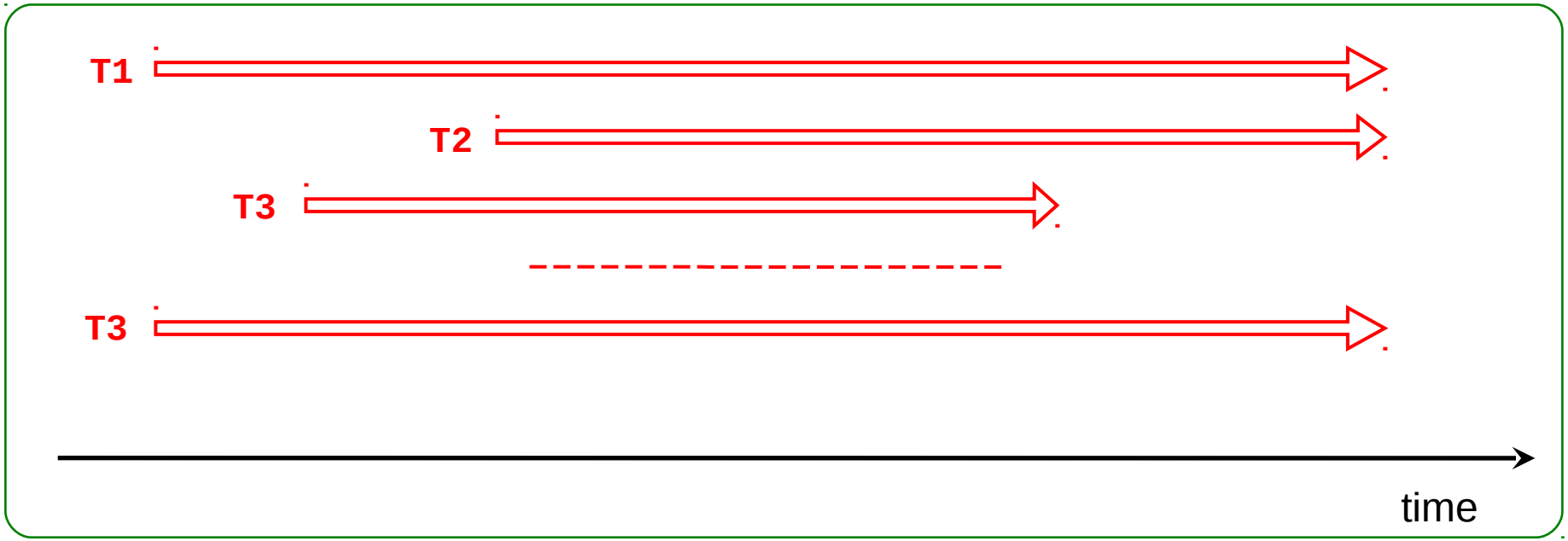
- Understand usage of *threads* and threading.
- Write programs that perform multiple tasks in parallel
- Understand some of the additional pitfalls that can occur in multithreaded code.

Concurrency

- A program may need to do several things at once: *concurrency*.
  - e.g. A web browser may be downloading a web page, at the same time as handling user input from a keyboard or mouse.
  - Can't easily do this in a single *thread* of control.
  - Your programs so far, when reading (or waiting for) user input from the keyboard, could not perform any other task at the same time.
- It may also be possible to complete a task *faster* by performing different parts of it at the same time on different processors (or cores): *parallelism*.
  - Can do this with multithreaded applications, but there are other approaches.
- There are many pitfalls in multithreaded programming: if you do need to do this for a particular application you will certainly need to learn many more details than are given in this course.

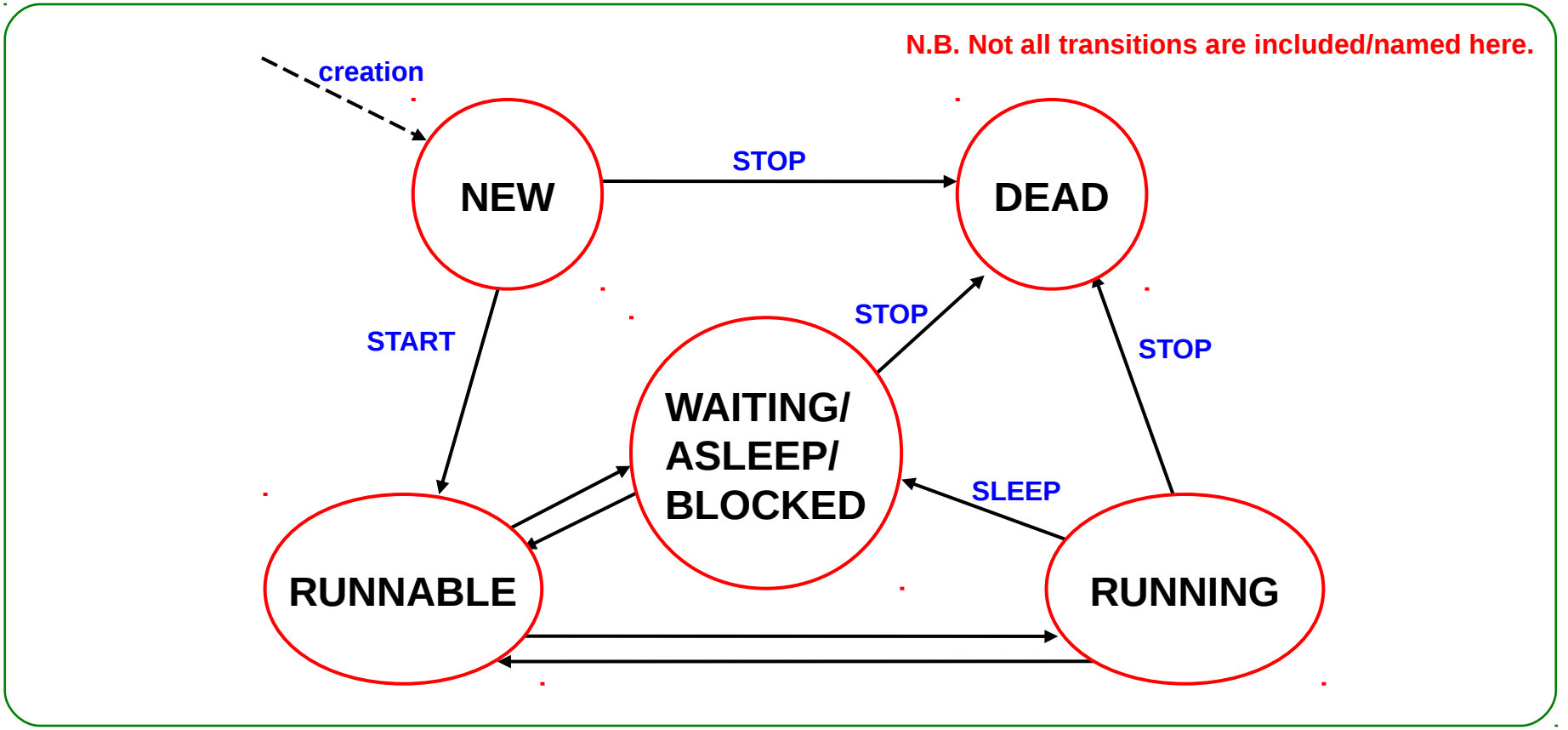
Threads

- A *thread* is a single sequential flow of control within a program.
- A program can comprise any number of threads at different times.



- You do not need to be concerned with “where” the different threads run: they may run on different processors, different processor cores or may simply be time-shared on a single CPU.

Lifecycle of a Thread (Simplified)



Threads: Implementation

- Methods to allow threaded execution in a class:
  - Extend the class **`Thread`**. *(not recommended in this course)*
  - Make our classes implement the **`Runnable`** interface and its method **`run()`**:

```
public class MyTask implements Runnable {
   public void run() {
      System.out.println("This thread is now running");
   }
}
```

- Pass to an instance of class **`Thread`** to execute **`run()`** method as separate thread:

```
MyTask task = new MyTask();
Thread thread = new Thread(task);
thread.start();
```

Calls `run()` method of `threadedObj`

Threads: Implementation short-cuts

A **`Runnable`** implementation doesn't need a name:

```
Runnable task = new Runnable() {
    public void run() {
        System.out.println("This thread is now running");
    }
};
Thread thread = new Thread(task);
thread.start();
```

We don't cover lambda expressions in this course, but they provide an even more compact implementation:

```
Thread thread = new Thread(() -> {
    System.out.println("This thread is now running");
});
thread.start();
```

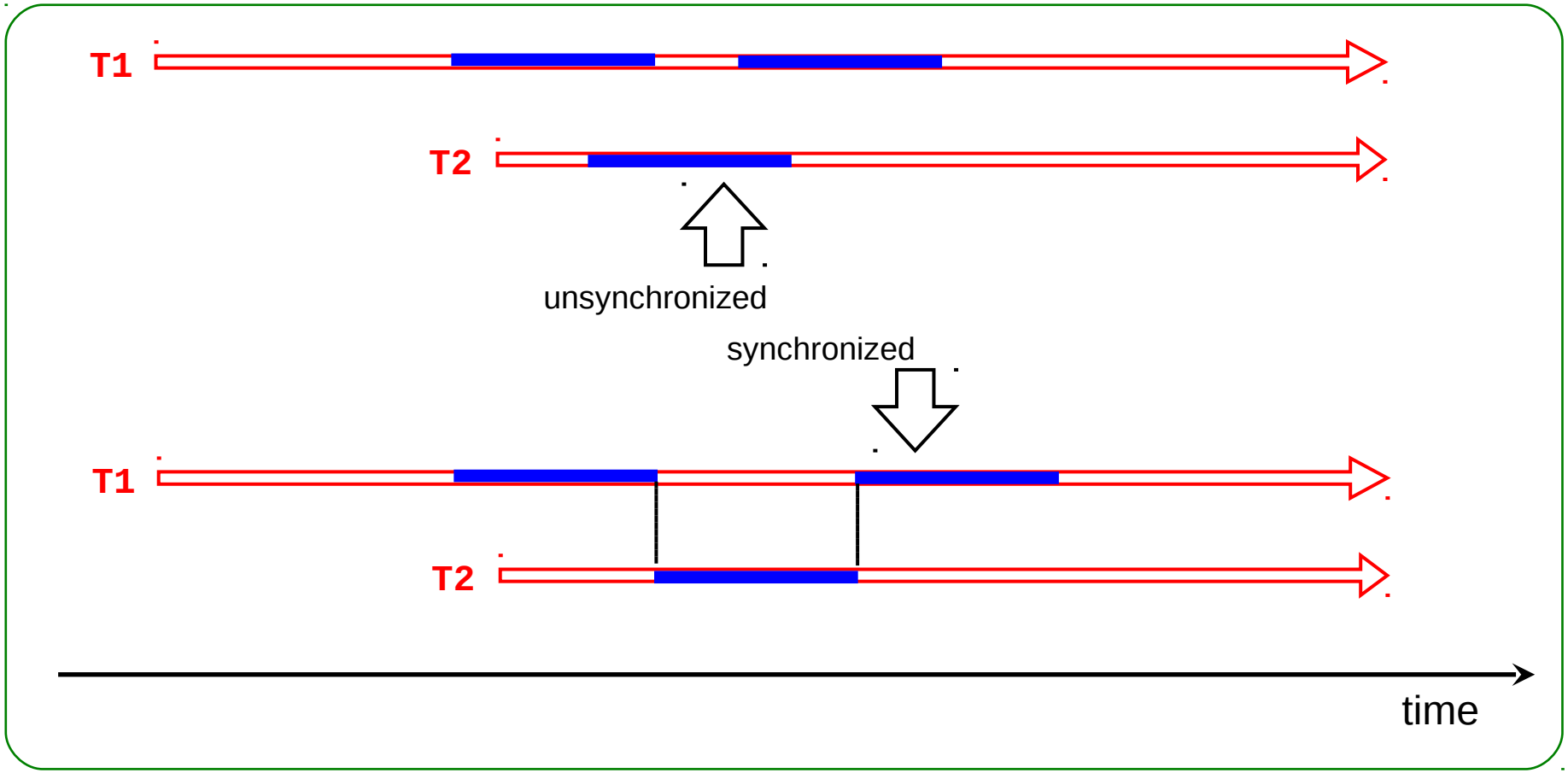
Problems with Multithreaded Code

- A crucial difference between a multithreaded program and the simpler case of multiple programs running independently in parallel, is that the different threads share memory space.
- This can create problems:
  - Instructions in parallel threads can be executed in any time-order.
  - Parallel threads may access the same resource (e.g. an object) at the same time, giving unexpected results or even corrupted data.
- There are a number of techniques involved in ensuring multithreaded programs perform as expected.
- Making objects immutable where possible is helpful: an immutable object can be accessed safely from multiple threads.

Synchronization

- Where we cannot make an object immutable, its state can be changed by a method on one thread and thus have an effect on a method in a different thread.
- This causes *indeterminacy*: the program may behave differently depending on the order of events on different threads.
- More seriously, the object may be caught in an inconsistent state, e.g. some of its member variables may have been updated but not others, resulting in incorrect results.
- We have to ensure only one thread at a time can access critical regions of the code.
- We can achieve this with *synchronization*.
- If a method is declared **`synchronized`**, then only one thread at a time can run that method on a given object, or a given class in the case of a static method.
- See notes for further details.

Synchronization: Schematic



Summary

- Threads allow multiple pieces of code to be executed simultaneously.
- Most scientific programming can be carried out with single-threaded programs.
- However, for resource-intensive tasks, multi-threading is becoming more common, particularly as the number of cores in modern CPUs continues to increase.
- Threads are important in the next module on Graphics.
- This module is assessed through coursework only and is not included in the final exam.
- You have until Monday 11th December to finish the Module 8 exercises.