Programming in Java

Threads and multi-threading — the basics

KLM

Department of Computer Science and Information Systems
Birkbeck, University of London

keith@dcs.bbk.ac.uk

Multi-processing

- Modern operating systems are multiprocessing
- They appear to do more than one thing at a time
- Three general approaches:
  - Cooperative multiprocessing
  - Preemptive multiprocessing
  - Really having multiple processors

Birkbeck UNIVERSITY OF LONDON

Mulit-threading

- Multi-threading programs *appear* to do more than one thing at a time
- Same ideas as multiprocessing, but within a single program
- More efficient than multiprocessing
- Java tries to hide the underlying multiprocessing implementation

Why multithreading?

- Allows you to do more than one thing at once
  - Play music on your computers CD player
  - Download several files in the background
  - while you are writing a letter
- Multithreading is essential for animation
  - One thread does the animation
  - Another thread responds to user inputs

Birkbeck UNIVERSITY OF LONDON

Threads

- A *Thread* is a single flow of control
- When you step through a program, you are following a *thread*
- Your previous programs all had one thread
- A `Thread` is an `Object` you can create and control(-ish)

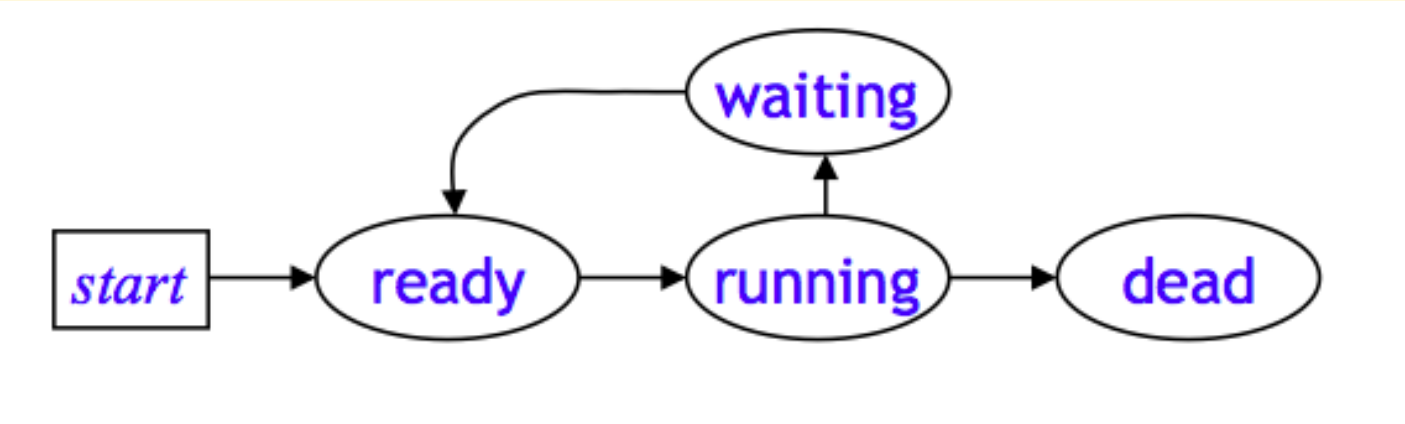
Sleeping

- Every program uses at least one `Thread`
- For example `Thread.sleep(int milliseconds);` (remember - a millisecond is 1/1000 of a second)
- ```
  try {
    Thread.sleep(1000);
    ...
  }
  catch (InterruptedException e) {
  ...
  }
  ```
- `sleep` only works for the current `Thread`
  ```

Lifecycle of a thread

- A Thread can be in one of four states:
  1. **Ready**: all set to run
  2. **Running**: actually doing something
  3. **Waiting**, or `blocked`: needs something
  4. `Dead`: will never do anything again
- State names vary across textbooks
- You have some control, but the Java scheduler has more

State transitions



Thread creation

- You can extend the `Thread` class:

  ```
  class Animation extends Thread {}
  ```

  Limiting, since you can only extend one class
- Or you can implement the `Runnable` interface:

  ```
  class Animation implements Runnable {}
  ```

  requires `public void run()`

Usually the second approach is to be preferred for most programs

Extending the `Thread` class

```
class Animation extends Thread {
  @Override
  public void run() {
     // code for this thread
  }

  // Anything else you want in this class
}
```

- `Animation anim = new Animation();`
  A newly created `Thread` is in the *Ready* state
- To start the `anim` thread running, call `anim.start();`
- `start()` is a request to the scheduler to run the thread — it may not happen right away
- The thread should eventually enter the *Running* state

Implementing `Runnable`

- `class Animation implements Runnable`
- The `Runnable` interface requires a `run()` method
  This is (effectively) the "main" method of your new thread
- `Animation anim = new Animation();`
- `Thread myThread = new Thread(anim);`
- To start the thread running, call `myThread.start();`
  Note: you do not write the `start()` method -- its provided by Java
- `start()` is a request to the scheduler to run the thread — it may not happen right away

Starting a thread

- Every thread has a `start()` method
- *Do not* write or override `start()`
- You *call* `start()` to request a thread to run
- The scheduler then (eventually) calls `run()`
- You must supply a `public void run()` method
- This is where you put the code that the thread is going to execute

Birkbeck UNIVERSITY OF LONDON

Summary I

```
class Animation extends Thread {
    public void run( ) {
        while (okToRun) { ... }
    }
}

Animation anim = new Animation( );
anim.start( );
```

Summary II

```
class Animation extends Screen
      implements Runnable {
    public void run( ) {
        while (okToRun) { ... }
    }
}

Animation anim = new Animation( );
Thread myThread = new Thread(anim);
myThread.start();
```

Things a thread can do...

- `Thread.sleep(milliseconds)`
- `yield()`
- `Thread me = currentThread();`
- `int myPriority = me.getPriority();`
- `me.setPriority(NORM_PRIORITY);`
- `if (otherThread.isAlive()) { ... }`
- `join(otherThread);`

Things a thread should NOT do...

- The thread controls its own destiny!
- Use any of the deprecated methods:
  - `myThread.stop()`
  - `myThread.suspend()`
  - `myThread.resume()`
- The above were thought to be a good idea — it turned out to be a very bad idea

An example — `SimpleThreads` I

- The following example brings together some of these concepts
- `SimpleThreads` consists of two threads
- The first is the `main` thread that every Java application has
- The `main` thread creates a new thread from the `Runnable` object, `MessageLoop`, and waits for it to finish
- If the `MessageLoop` thread takes too long to finish, the main thread interrupts it

An example — `SimpleThreads` II

- The `MessageLoop` thread prints out a series of messages
- If interrupted before it has printed all its messages the `MessageLoop` thread prints a message and exits
- and now the code...

A problem. . .

- Suppose we have

  ```
  int k = 0;
  ```

  and a thread which contains

  ```
  k = k + 1;
  ```

- If we also have a thread which contains

  ```
  System.out.println(k);
  ```

  then, if both threads are running at the same time:
- What gets printed as the value of `k`?
- This, *race condition*, is a trivial example of what is, in general, a very difficult problem

Enter synchronisation. . . I

You can *synchronise* on an object:

- `synchronized (obj){...code that uses/modifies obj...}`
- No other code can use or modify this object at the same time
- Notice that `synchronized` is being used as a *statement*

Enter synchronisation... II

You can *synchronise* on a method (uses `this`):

- `synchronized void addOne(arg1, arg2, ...) { code }`
- Only one `synchronized` method in a class can be used at one time (other methods can be used simultaneously)

Enter synchronisation... III

Synchronisation is a tool, not a solution — multithreading is in general a very hard problem

Questions