G52CON: Concepts of Concurrency

Lecture 2 Processes & Threads

Brian Logan
School of Computer Science
bsl@cs.nott.ac.uk

Outline of this lecture

- Java implementations of concurrency
- process and threads
- a simple `ParticleApplet` example
- ways of creating `Thread` objects in Java:
  - extending the `Thread` class; and
  - implementing the `Runnable` interface
- the lifecycle of a `Thread`:
  - starting a new `Thread`,
  - while the `Thread` is running; and
  - shutting it down

Implementations of Concurrency

We can distinguish two main types of implementations of concurrency:

- **shared memory**: the execution of concurrent processes by running them on one or more processors all of which access a shared memory —processes communicate by reading and writing shared memory locations; and

- **distributed processing**: the execution of concurrent processes by running them on separate processors which don't share memory— processes communicate by message passing.

Java Implementations of Concurrency

Java supports both shared memory and distributed processing implementations of concurrency:

- **shared memory**: multiple user threads in a single Java Virtual Machine—threads communicate by reading and writing shared memory locations; and

- **distributed processing**: via the `java.net` and `java.rmi` packages—threads in different JVMs communicate by message passing or (remote procedure call)

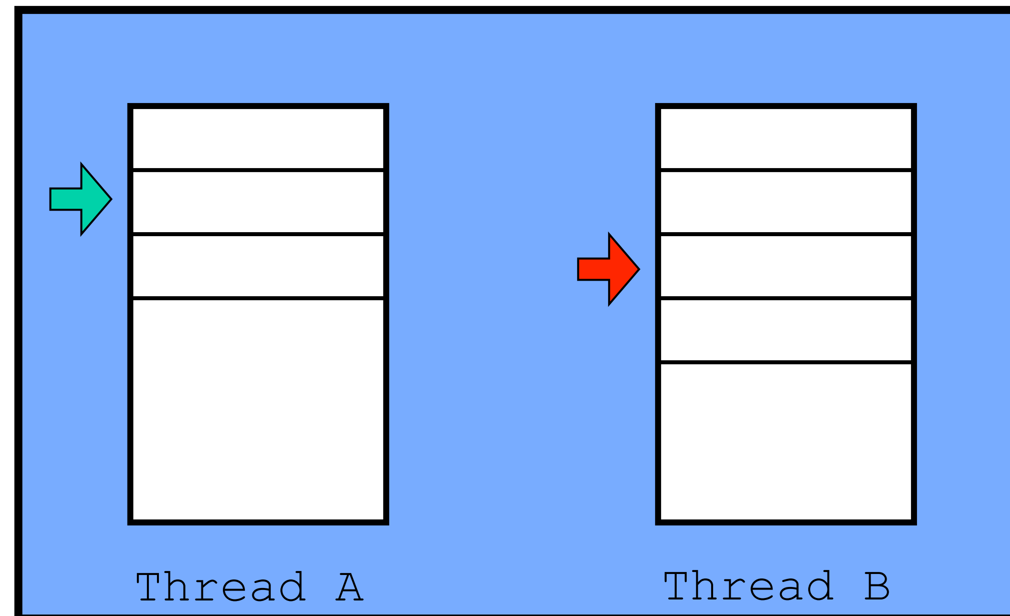
G52CON covers both approaches

Processes and Threads

- A *process* is any thread of execution or control, e.g.:
  - part of a concurrent program (lightweight process)
  - programs running in different address spaces on the same processor (heavyweight or OS processes)
  - running on a different processor or on a different computer
- A *thread* is a process which forms part of a concurrent program
  - threads execute within a *shared address space*
  - a *Java thread* is a process running within a JVM (JVM is generally run as a heavyweight or OS process)

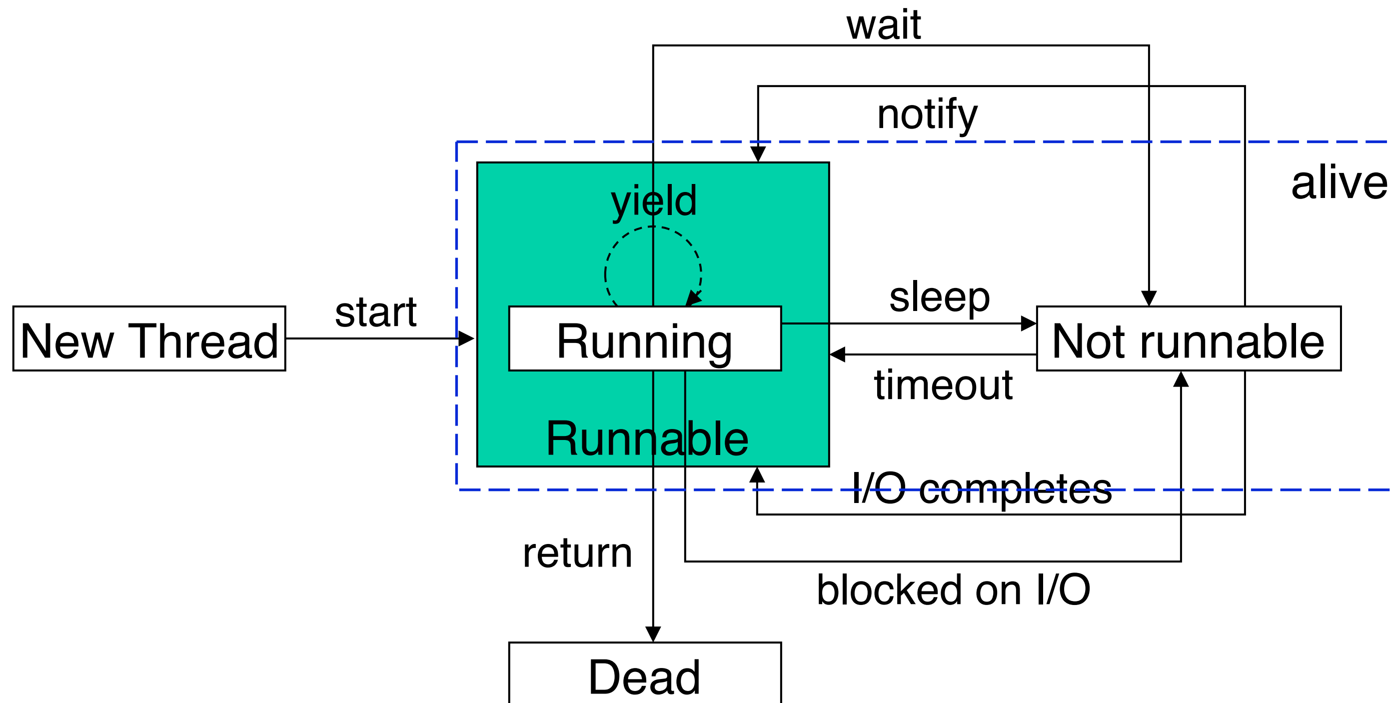
Threads in Java

A *thread* is a single sequential flow of control within a Java program.



Within the JVM, the *threads* comprising a Java program are represented by instances of the `Thread` class.

`Thread` lifecycle



A Simple Example: `ParticleApplet`

`ParticleApplet` creates *n* `Particle` objects, sets each particle in autonomous 'continuous' motion, and periodically updates the display to show their current positions:

- each `Particle` runs in its own Java `Thread` which computes the position of the particle; and
- an additional `ParticleCanvas` `Thread` periodically checks the positions of the particles and draws them on the screen.
- in this example there are at least 12 threads and possibly more, depending on how the browser handles applets.

`ParticleApplet`

There are three classes:

- `Particle`: represents the position and behaviour of a particle and can draw the particle at its current position;

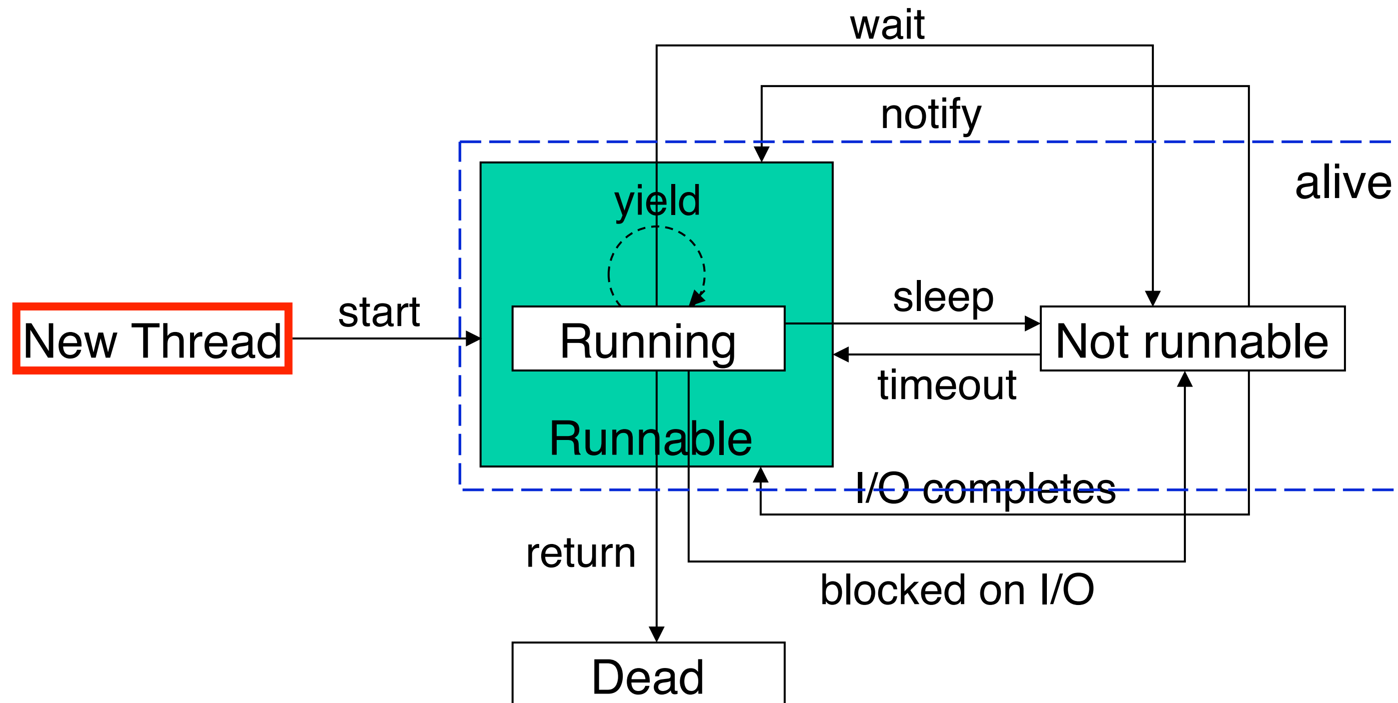
- `ParticleCanvas`: provides a drawing area for the `Particles`, and periodically asks the `Particles` to draw themselves; and

- `ParticleApplet`: creates the `Particles` and the canvas and sets the `Particles` in motion.

See also Lea (2000), chapter 1 for an alternative implementation.

© Brian Logan 2014

`Thread` lifecycle: creation



Creating `Threads`

There are two ways to create a thread:

- extending the `Thread` class and overriding its `run()` method; or
- defining a class which implements the `Runnable` interface and its `run()` method

  ```
  public interface java.lang.Runnable {
      void run();
  }
  ```

  and passing the `Runnable` object to the `Thread` constructor.

The `Thread` class implements the `Runnable` interface.

Extending the `Thread` class

```
class Particle extends Thread {
    protected int x, y;
    protected final random rng = new Random(this.hashCode());

    // constructor etc…

    public void run() {
        try {
            for(;;) {
                move();
                sleep(100);
            }
        } catch (InterruptedException e) {
            return;
        }
    }
    // other methods ...
```

`Particle` class continued

```
    public synchronized void move() {
        x += (rng.nextInt() % 10);
        y += (rng.nextInt() % 10);
    }

    public void draw(Graphics g) {
        int lx, ly;
        synchronized(this) { lx = x; ly = y; }
        g.drawRect(lx, ly, 10, 10);
    }
}
```

Implementing `Runnable`

```
class ParticleCanvas extends Canvas implements Runnable {
    private Particle[] particles = new Particle[0];

    // constructor etc ...

    public void run() {
        try {
            for(;;) {
                repaint();
                Thread.sleep(100);
            }
        }
        catch (InterruptedException e) { return; }
    }

    // other methods ...
```

`ParticleCanvas` class continued

```
    protected synchronized void getParticles() {
        return particles;
    }

    // called by Canvas.repaint();
    public void paint(Graphics g) {
        Particle[] ps = getParticles();

        for (int i = 0; i < ps.length(); i++)
            ps[i].draw(g);
    }
}
```

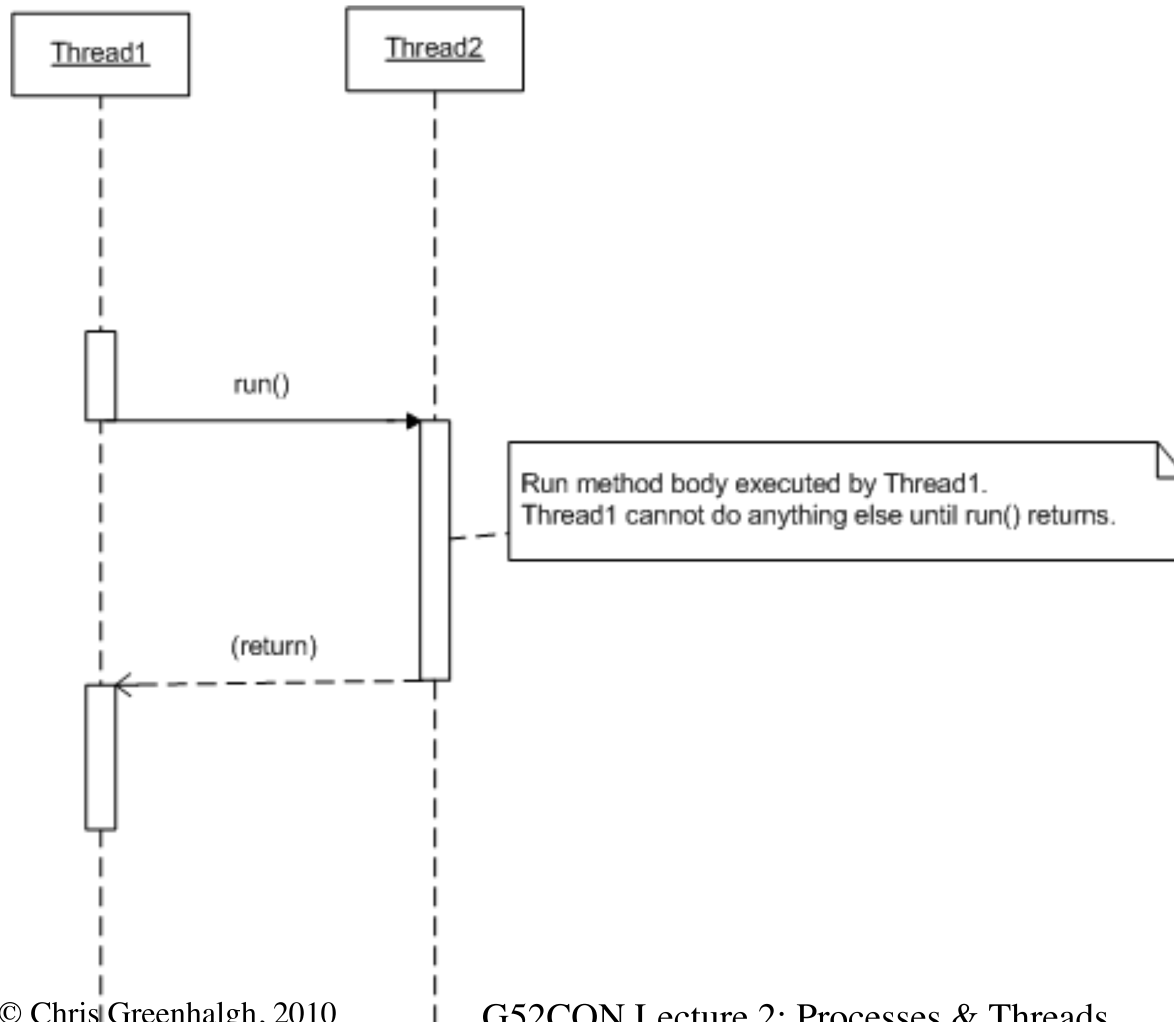
Particle threads

```
public class ParticleAppletA extends Applet {

    protected final ParticleCanvas canvas = new ParticleCanvas(400);
    protected Particle[] particles; // null when not running
    protected Thread canvasThread;

     // ParticleApplet start method
    public synchronized void start() {
        int n = 10; // just for demo

        if (particles == null) { // bypass if already started
            particles = new Particle[n];
            for (int i = 0; i < n; ++i) {
                particles[i] = new Particle(200, 200);
                particles[i].setName("Particle Thread " + i);
                particles[i].start();
            }
            canvas.setParticles(particles);
             // continued ...
```

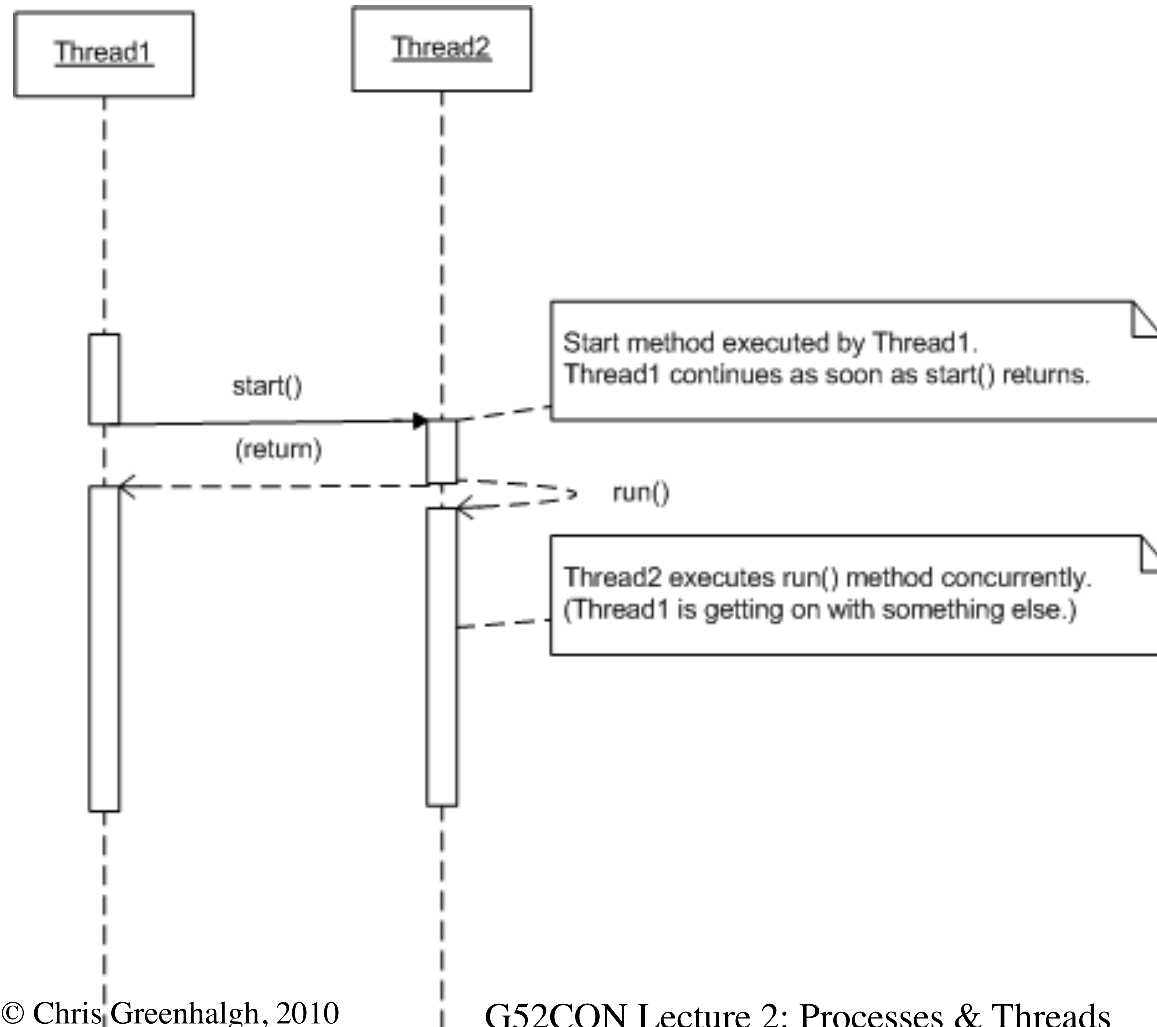
ParticleCanvas thread

```
public class ParticleAppletA extends Applet {

    protected final ParticleCanvas canvas = new ParticleCanvas(400);
    protected Particle[] particles; // null when not running
    protected Thread canvasThread;

     // ParticleApplet start method ...
    public synchronized void start() {
         int n = 10; // just for demo

         if (particles == null) { // bypass if already started

        // code to start particles omitted …

             canvasThread = new Thread(canvas);
             canvasThread.setName("Canvas Thread");
             canvasThread.start();
         }
    }
```

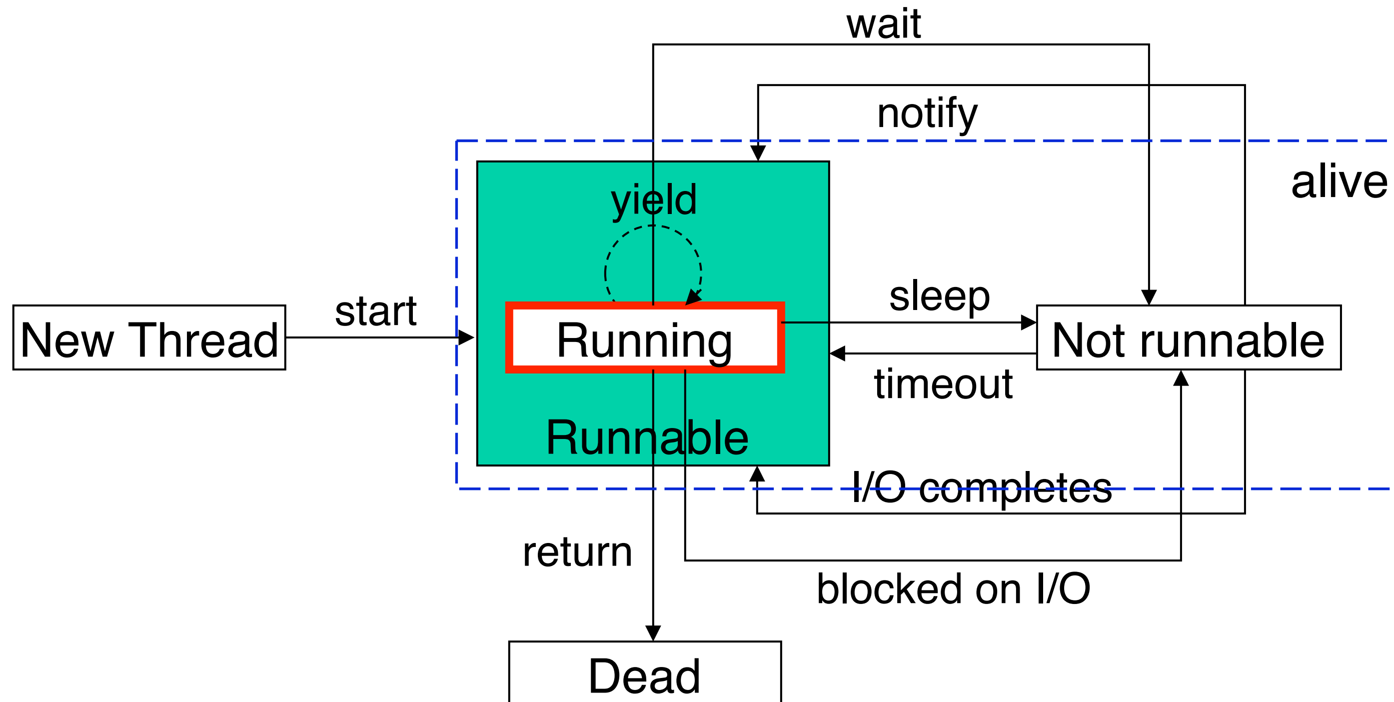
Calling `run()` ... (wrong!)



Calling `start()`... (right!)



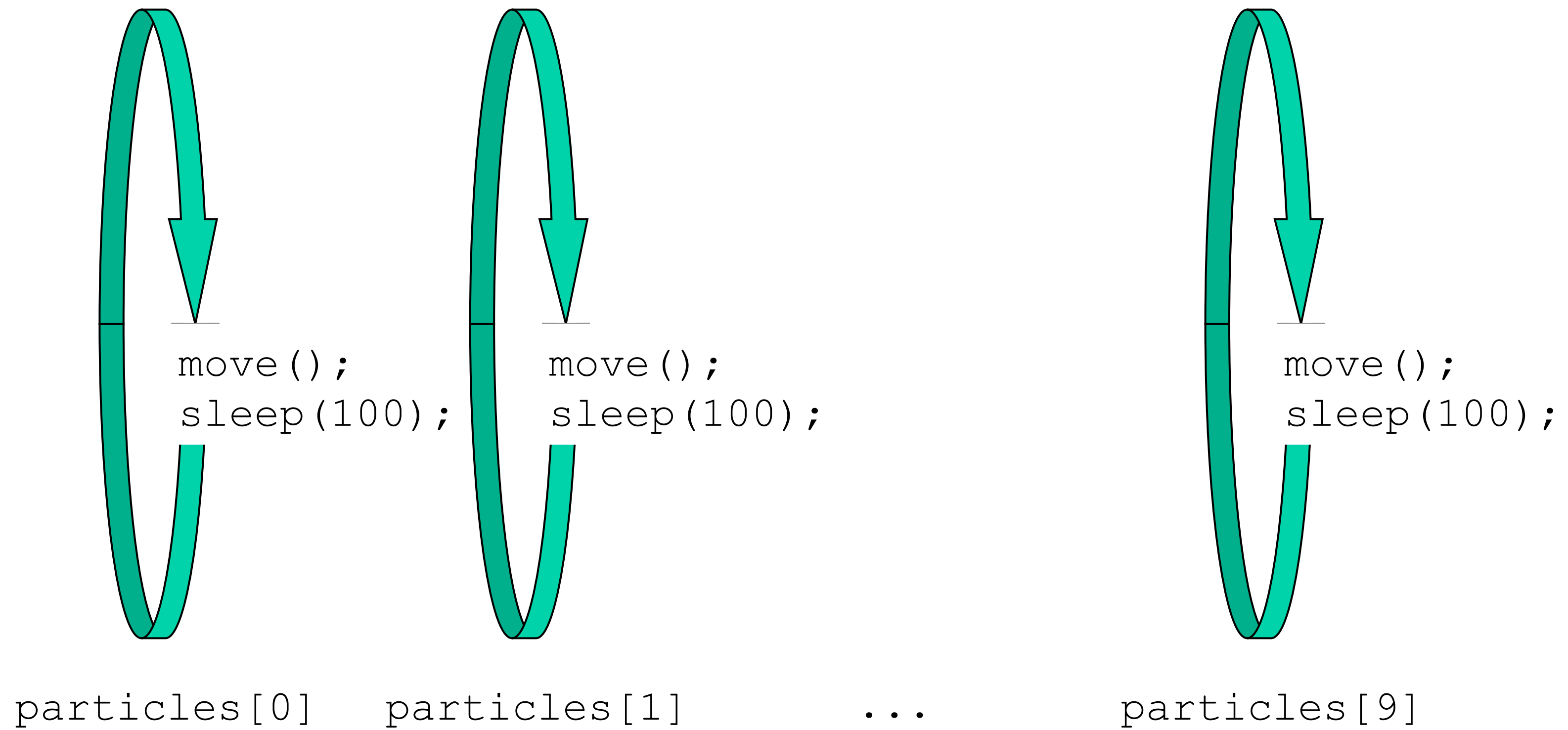
`Thread` lifecycle: running



Particle.run()

```
class Particle extends Thread {
    // fields, constructor etc…
    public void run() {
        try {
            for(;;) {
                move();
                sleep(100);
            }
        }
        catch (InterruptedException e) { return; }
    }
    // other methods …
}
```

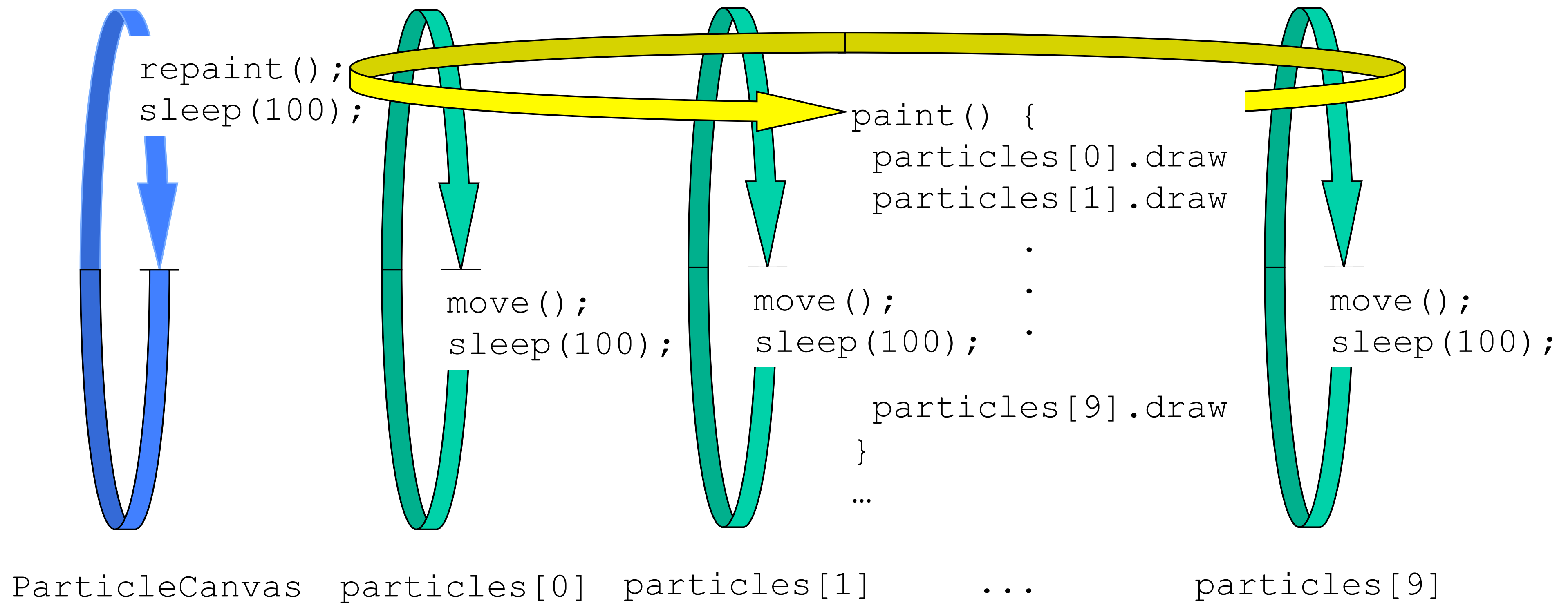
`Particle` threads



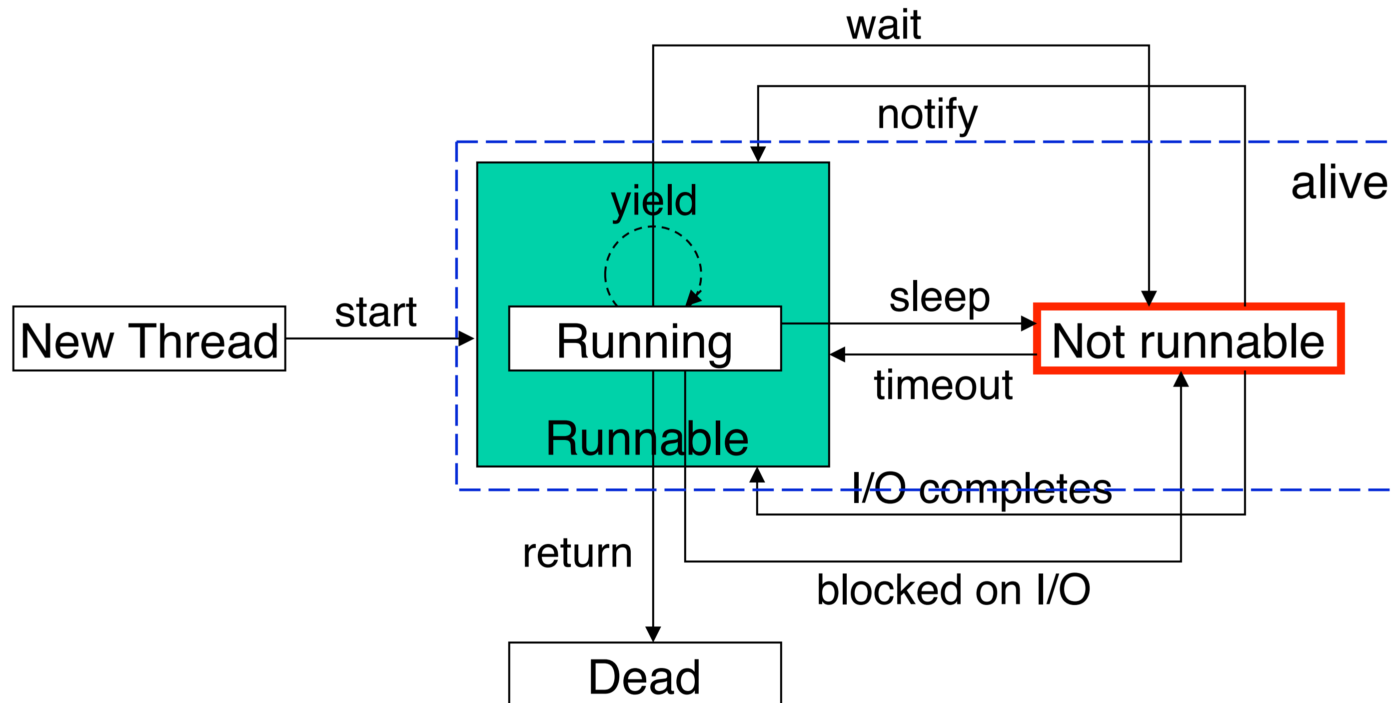
ParticleCanvas.run()

```
class ParticleCanvas extends Canvas implements Runnable {
    // fields, constructor etc …
    public void run() {
        try {
            for(;;) {
                repaint();
                Thread.sleep(100);
            }
        }
        catch (InterruptedException e) { return; }
    }
    // other methods …
}
```

`ParticleCanvas` & AWT event threads



`Thread` lifecycle: not runnable



The not runnable state

A running `Thread` becomes not runnable when:

- it calls `sleep()` to tell the scheduler that it no longer wants to run;
- it blocks for I/O; or
- it blocks in `wait()` for condition synchronisation.

Examples of not runnable

- `Particle` threads become not runnable when they `sleep()`
- the `ParticleCanvas` thread becomes not runnable when it calls `sleep()`
- we'll return to `wait()` and condition synchronisation in later lectures …

Scheduling methods

The `Thread` class provides the following `static` scheduling methods:

- `sleep(long msecs)`: causes the current thread to suspend for at least `msecs` milliseconds.

- `yield()`: requests that the JVM to run any other runnable but non-running thread rather than the current thread.

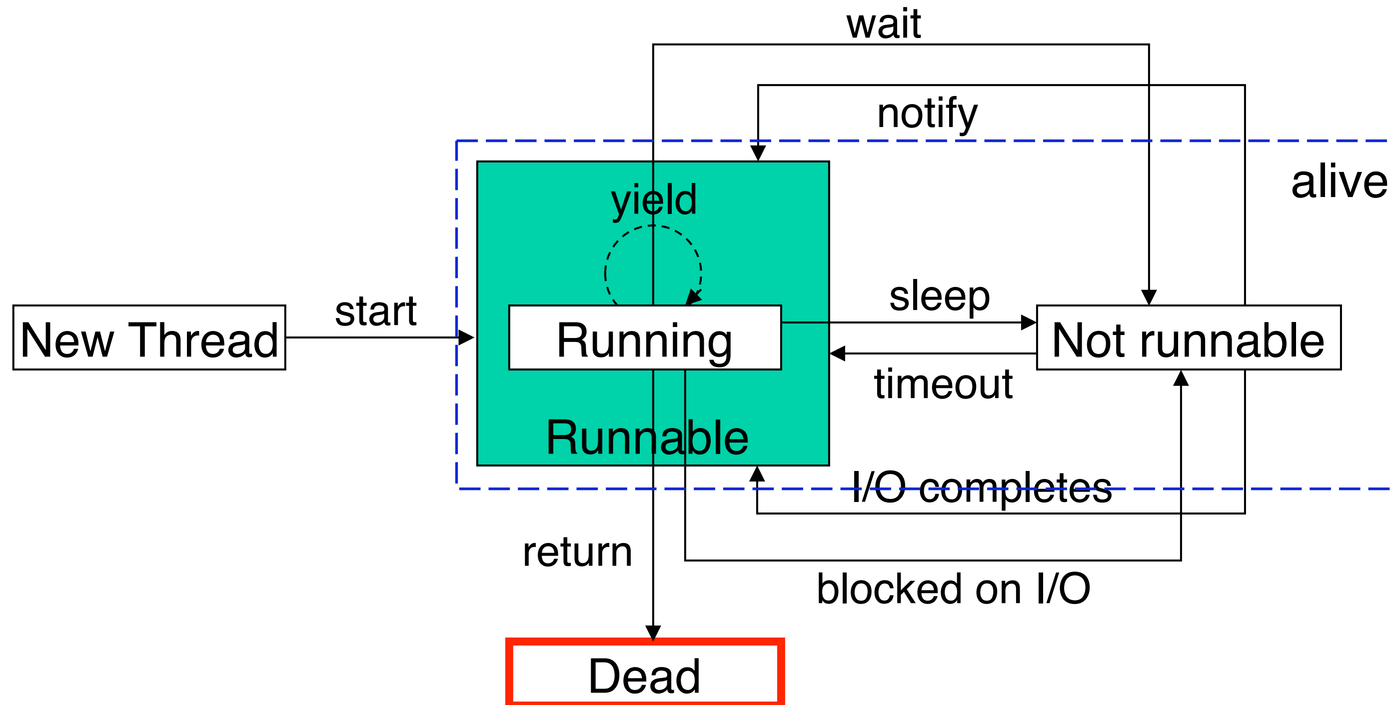
`Thread` priorities

`Threads` have *priorities* which heuristically influence schedulers:

- each thread has a priority in the range `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`
- by default, each new thread has the same priority as the thread that created it---the initial thread associated with a main method by default has priority `Thread.NORM_PRIORITY`
- the current priority of a thread can be accessed by the method `getPriority` and set via the method `setPriority`.

When there are more runnable threads than CPUs, a scheduler is generally biased in favour of threads with higher priorities.

`Thread` lifecycle: cancellation



Thread termination

A thread terminates when its `run()` method completes:

- either by returning normally; or
- by throwing an unchecked exception (`RuntimeException`, `Error` or one of their subclasses)

`Thread`s are not restartable—invoking `start()` more than once results in an `InvalidThreadStateException`.

Thread cancellation

There are several ways to get a thread to stop:

- when the thread's `run()` method returns;
- call `Thread.stop()` --- *this is a **bad** idea*, as it doesn't allow the thread to clean up before it dies; or
- `interrupt()` the thread.

A multi-threaded program will continue to run until its last (non-daemon) thread terminates.

Interrupting a `Thread`

Each `Thread` object has an associated boolean interruption status:

- `interrupt()`: sets a running thread's interrupted status to *true*
- `isInterrupted()`: returns *true* if the thread has been interrupted by `interrupt()`

A thread can periodically check its interrupted status, and if it is *true*, clean up and exit.

`Thread` (checked) exceptions

Threads which are blocked in calls `wait()` and `sleep()` aren't runnable, and can't check the value of the interrupted flag

- interrupting a thread which is waiting or sleeping aborts the thread and throws an `InterruptedException`
- if the interrupt flag is set *before* entering `sleep` or `wait` the thread immediately throws an `InterruptedException`

```
synchronized <method or block>
    try {
        wait()|sleep()
    } catch (InterruptedException e) {
        // clean up and return (interrupted status false)
    }
```

Stopping the `ParticleApplet`

```
// ParticleApplet stop method (not Thread.stop) …
public synchronized void stop() {
    // Bypass if already stopped …
    if (particles != null) {
        for (int i = 0; i < particles.length; ++i)
            particles[i].interrupt();
        particles = null;
        canvasThread.interrupt();
        canvasThread = null;
    }
}
```

Stopping the `Particles`

```
// Particle run method …

public void run() {
    try {
        for(;;) {
            move();
            sleep(100);
        }
    }
    catch (InterruptedException e) { return; }
}
```

The Next Lecture

Synchronisation

Suggested reading:

- Andrews (2000), chapter 2, sections 2.1, chapter 3, section 3.1;
- Ben-Ari (1982), chapter 2.