Robotics and Autonomous Systems

Lecture 10: Threads and Multitasking Robots

Richard Williams

Department of Computer Science
University of Liverpool

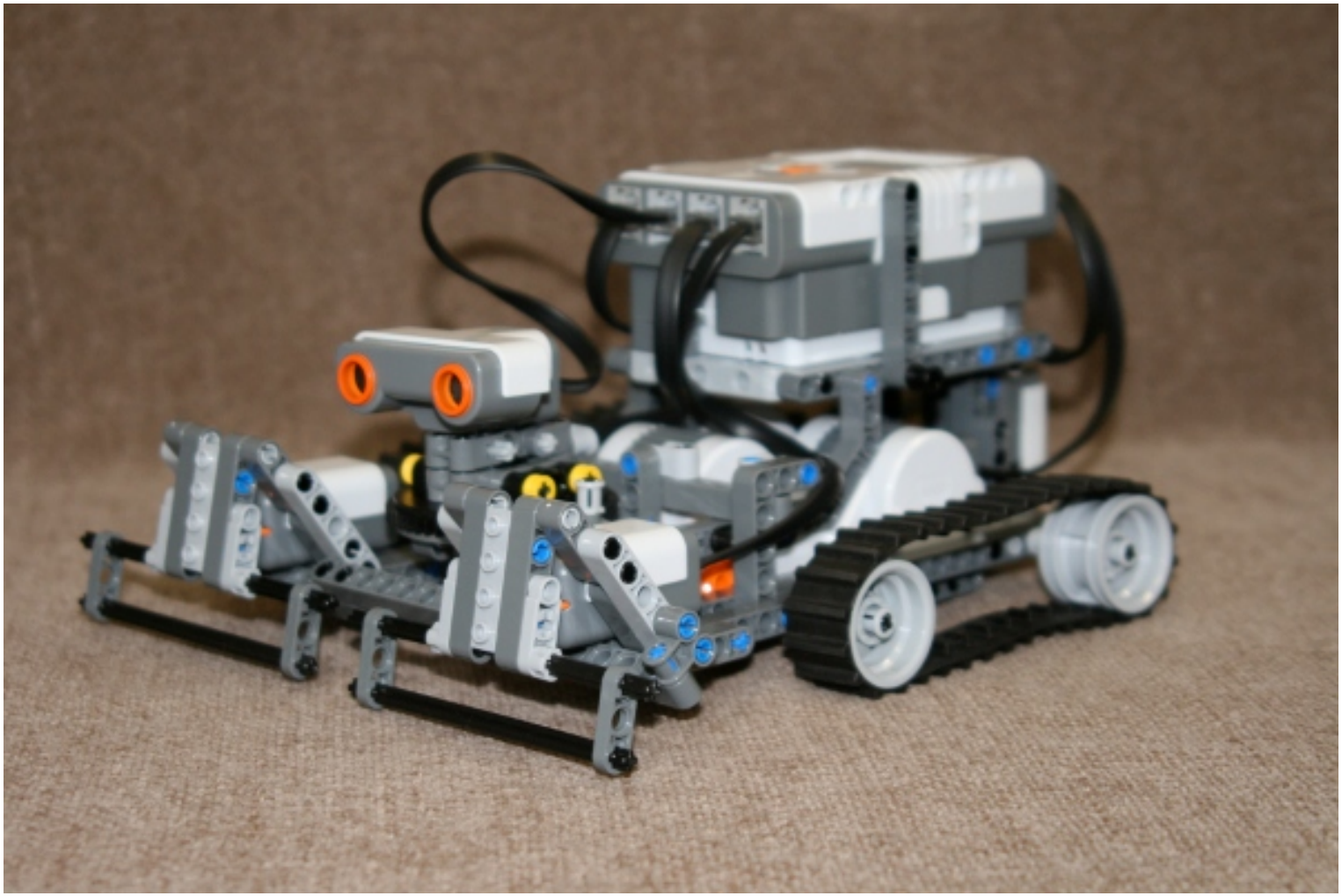
UNIVERSITY OF LIVERPOOL

Today

- Some more programming techniques that will be helpful for the assignment.
- The main subject will be multitasking
  - How to get the robot to do several things at once.
- That involves using threads.
- But we'll cover some other useful programming ideas as well.

Why Threads?

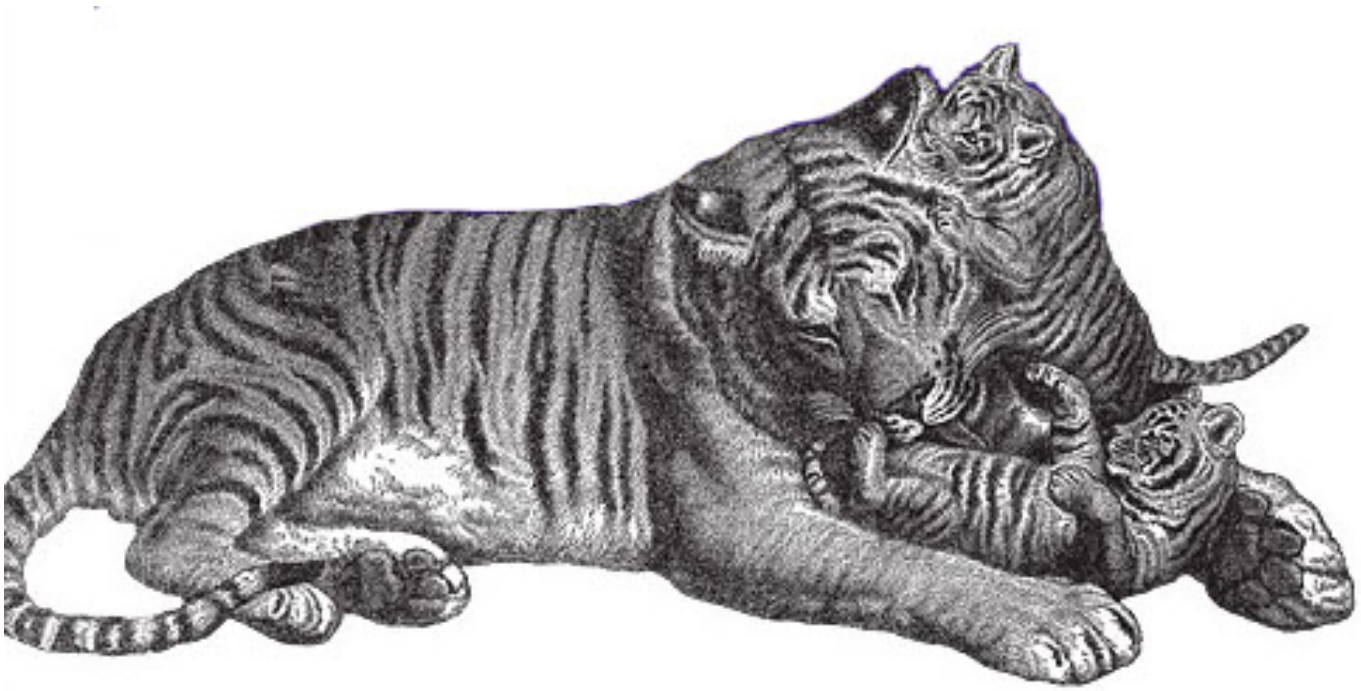
Why Threads?



Threads

- In robotics we frequently need to deal with concurrency
  - Different bits of code running more or less independently in time.
- Once upon a time these had to be separate processes.
  - Rather heavyweight.
- A more modern approach is that of threads
- These provide fine-grained concurrency within a process.
- Here we discuss the basic ideas behind the use of threads in Java.

Source



- This material is drawn from Learning Java, Patrick Niemeyer, Jonathan Knudsen, O'Reilly.

What are threads?

- A thread is a flow of control within a program.
  - Like processes, but threads within a program can easily share state.
- Threads are like lectures at UoL:
  - Separate entities
  - Share resources
  - Only one entity uses a resource at the same time
  - Need coordination to manage access to resources.
- Threads also have local data that is distinct.

`Thread` object

- All execution in Java is associated with a `Thread` object.
  - That is what `main()` launches.
- New threads are born when a new instance of:

  `java.lang.Thread`

  is created
- This object is what we manipulate to control and coordinate execution of the thread.

`Thread` object

- Two ways to handle threads.
- One way is to sub-class the `Thread` object.
  - Create your own thread which `extends` the standard thread.
- Do this by defining/over-riding the `run()` method.

  (This is what is invoked when the thread starts.)
- Second way is to create a `Runnable` object and execute it in an unmodified `Thread`
- Many Java programmers consider that using `Runnable` is better style.
- We will stick to the first.

Scheduling

- In most Java implementations, threads are time-sliced.
- ???

Scheduling

- In most Java implementations, threads are time-sliced.
- Each one runs for a while in some order (in the implementation that I use, it seems that the first thread to be started is the first one to run).
- On other Java implementations, you might get different behavior.
  - All depends on what the VM does.
- All the specification says is as follows (next slide).

Scheduling

- All threads have a priority value.
- Any time a higher priority thread becomes runnable, it preempts any lower priority threads and starts executing.
- By default, threads with the same priority are scheduled round-robin.
- This means that once a thread begins to run it continues until:
  - It sleeps due to a `sleep()` or `wait()`;
  - It waits for the lock for a `synchronized` method;
  - It blocks on I/O;
  - It explicitly yields control using `yield()`; or
  - It terminates.
- So there is no necessity for threads to be time-sliced.

Controlling threads

- There are a few methods that allow us to control the execution of threads.
- `start()` is used to start a thread running.
  - Will see an example in a bit.
- `stop()`, `suspend()` and `resume()` are deprecated.
- We will discuss `sleep()`, `wait()` and `notify()`.
- There are also `join()` and `interrupt()`.
  - Won't talk about these.

sleep()

- Sometimes we need to tell a thread to take a break.
- The method `sleep()` will do this.
  - It takes an argument that is the number of milliseconds to sleep for.
- `sleep()` is a class method of `Thread`, so it can be called either using:

```
Thread.sleep()
```

or by calling it on a specific instance of `Thread`:

```
myOwnLittleThread.sleep()
```

`sleep()`

- You may have seen this kind of thing:

  ```
  Thread.sleep(200)
  ```

  used without much explanation.
- Puts the current thread to sleep.
- Typically the one launched by `main()`.

sleep()

- Here's another example from a classic scheduling example:

```
public void run(){
  while(true){
    System.out.println("One!");
    try{
      Thread.sleep(1000);
    }
      catch(InterruptedException e){
      // Guess we won't get to sleep after all
      }
    }
}
```

`sleep()`

- Good practice to put a `sleep` in a `try/catch` structure in case the thread is interrupted during its sleep.
- You often set threads to sleep precisely because you are waiting for them to be interrupted.
- A sleeping thread can be woken up by an `InterruptedException` so we need to specify what to do if this happens.

Synchronization

- When threads share resources, they need to be synchronized
  - Otherwise things can get confusing
- Imagine two threads trying to use the same variable at the same time.
  - In the lecture example, imagine two lectures trying to use the same classroom at the same time.
  - On a robot, imagine two threads trying to use the same motor at the same time.
- Even when you take scheduling (in the Java thread sense) into account it can be an issue.
- Java provides some simple monitor-based methods for controlling access.

Synchronization

- The most basic idea is that of `synchronized` methods.
- Just add the keyword `synchronized` to the method definition:

```
public synchronized void myFunction(){
    :
}
```

- Only one thread at a time is allowed to execute any `synchronized` method of an object.
  - The object is locked.
- Other threads are blocked until they can aquire the lock on the object.

Synchronization

- Note that locks are reentrant, so a thread does not block itself.
- The `synchronized` function can call itself recursively, and it can call other synchronized methods of the same object.

`wait()` and `notify()`

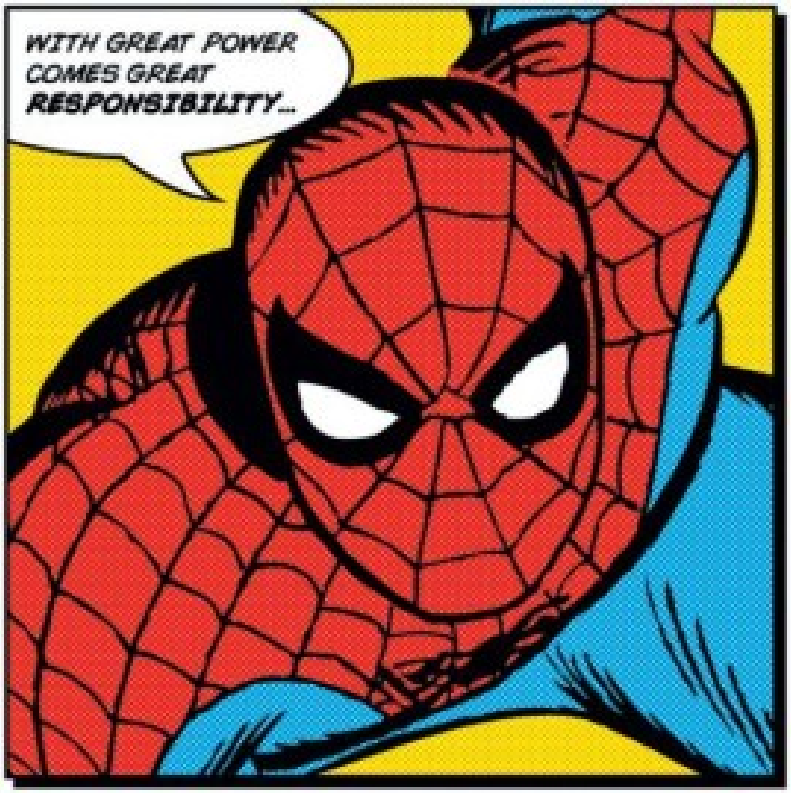
- `wait()` and `notify()` provide more direct synchronization of threads.
- When a thread executes a synchronized method that contains a `wait()`, it gives up its hold on the block and goes to sleep.
- The idea is that the thread is waiting for some necessary event to take place.
- Later on, when it wakes up, it will start to try to get the lock for the synchronized object.
- When it gets the lock, it will continue from where it left off.

`wait()` and `notify()`

- What wakes the thread up from `waiting` is a call to `notify()` on the same synchronized object.

Complex huh?

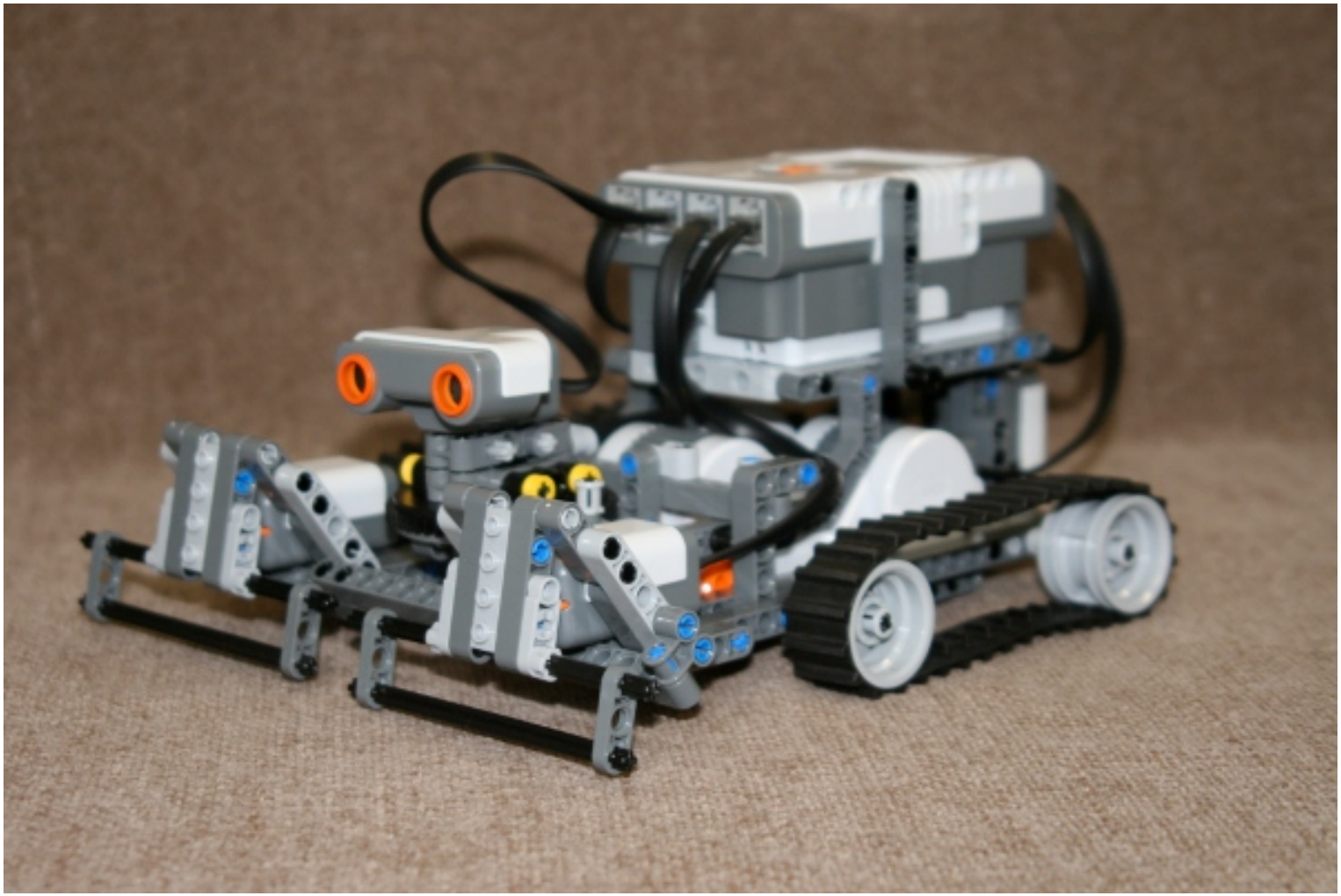
- Using threads gives you great power to perform complex operations.



- It also gives you the chance to create really incomprehensible errors.

Back to robots

- Now, how does this effect robots?



Standard Robot

- Let's define a Java class to represent our robot.
  - Two touch sensors
  - One ultrasound sensor
  - One colour sensor
  - Two drive motors

  (I'm ignoring the motor pointing the ultrasound sensor).
- Create an instance as part of a control program.

Standard Robot

- A data member for each element of the robot.

```
import lejos.nxt.*;
import lejos.nxt.addon.ColorSensorHT;

public class StandardRobot{
  private TouchSensor leftBump, rightBump;
  private UltrasonicSensor uSense;
  private ColorHTSensor cSense;
  private NXTRegulatedMotor leftMotor, rightMotor;
}
```

Standard Robot

- Each of these `private` members would need appropriate “get” and/or “set” functions.
- Get sensor values.
- Set motor values.
- Get motor values, for example `isMoving()`

Standard Robot

- Constructor sets up the data members to talk to the relevant bits of the hardware.

```
public StandardRobot(){
  leftBump = new TouchSensor(SensorPort.S2);
  rightBump = new TouchSensor(SensorPort.S1);
  uSense = new UltrasonicSensor(SensorPort.S3);
  cSense = new ColorHTSensor(SensorPort.S4);
  leftMotor = Motor.C;
  rightMotor = Motor.B;
}
```

- Good Java practice/style to set up the robot like this.
  - Independent of using threads.

Standard Robot

- Now we'll use a thread to set up a robot monitor.
- Thread that observes what the robot is doing
  - Uses the `StandardRobot` object to be able to do this.
- Reports the robot state on the screen.
- Useful debug tool.

Standard Robot

- Set up a thread which can reference a `StandardRobot`

```
public class RobotMonitor extends Thread{
  private int delay;
  public StandardRobot robot;

  public RobotMonitor(StandardRobot r, int d){
    this.setDaemon(true);
    delay = d;
    robot = r;
}
```

- Will explain the “daemon” bit next.

Daemons

- Daemons are threads providing “services” for other threads in the program.
- They run as background processes
- They serve basic functionalities upon which other threads build
- If a thread is declared Daemon, its existence does not prevent the JVM from exiting (unlike other threads).
- Methods in `java.lang.Thread`:
  - `boolean isDaemon()`
    Flags whether thread is daemon
  - `void setDaemon (Boolean on)`
    Sets the thread to be a daemon. Can only be used before the thread is created.

Not to be confused with this kind of daemon

Daemons

- Here our daemon thread reports on the robot:

```
public void run(){
  while(true){
    LCD.clear();
    LCD.drawString(robot.isLeftBumpPressed(), 0, 0);
    LCD.drawString(robot.isRightBumpPressed(), 0, 1);
    LCD.drawString(robot.getUSenseRange(), 0, 2);
    LCD.drawString(robot.getCSenseColor(), 0, 3);
    try{
      sleep(delay);
    }
    catch(Exception e){
    // We have no exception handling
    ;
}}}
```

Standard Robot

- The functions being called on `robot` are the `get` and `set` functions defined for `StandardRobot`
- See the code (on the course website) for details.

Standard Robot

- Finally we connect the monitor and an instance of `StandardRobot`

```
public class RunMonitor{
  public static void main(String [] args) throws Exception {
    StandardRobot me = new StandardRobot();
    RobotMonitor rm = new RobotMonitor(me, 400);

    rm.start();
    // Here we wait. But could be any control program
    // either inline, or as an object that is passed
    // a reference to me (the robot).
    Button.waitForAnyPress();
  }
}
```

Standard Robot

- Clearly, we could use the same style to build more complex robot controllers.
- Threads controlling different aspects of the robot:
  - Moving around
  - Avoiding obstacles
  - Preventing collisions
- All talking to the `StandardRobot` object to operate the hardware.
- All together determining what the robot does.
- We'll look more into this next time.

Summary

- This lecture looked at multi-tasking, which is handy for many robotics tasks.
- First we looked at threads, which provide a lightweight approach to multi-tasking.
- Then we looked at how threads can be used in LeJOS.
- Our example also showed how to use LeJOS in a more object-oriented way.