INF 2140 PARALLELL-PROGRAMMERING (Parallel Programming)

INF2140 lecture 1

Jan 18, 2012

INF2140 Crew

- Lecturers
  - *Einar Broch Johnsen*, office 8460, email: einarj@ifi.uio.no
  - *Olaf Owe*, office 8463, email: olaf@ifi.uio.no
- Teaching assistant (gruppelærer)
  - *Joakim Bjørk*, office 8163, email: joakimbj@ifi.uio.no, phone: 22 85 04 75

  All from the *PMA group*, 8th floor OJD's building
- Course webpage: http://www.uio.no/studier/emner/matnat/ifi/INF2140/

Time and Place

- Lectures
  Wed. 10:15 - 12:00, Room 2423 (Store Aud.)
- Groups (starts next week)

group	day	time	place
1	Monday	14:15-16	OJD 3443 Datastue Chill
2	Tuesday	10:15-12	OJD 3443 Datastue Chill
103	Tuesday	12.15-14	OJD 3443 Datastue Chill
104	Friday	10:15-12	OJD 2443 Datastue Modula

- Evaluation
  - Three compulsory assignments which must be approved.
    - First one handed out end of next week.
  - Final exam, June 15

Plan Today

- Practical matters
- Overview of the course and motivation
- Introduction
- Repetition: Concurrency in Java

- New Course
  - Why?
  - Who?
  - How?
  - What?

Course content

The course provides a systematic and practical approach to designing, analyzing and implementing parallel programs, with regard to tightly cooperating concurrent threads as well as distributed and object-oriented systems. The topics covered include threads and interaction; interference, exclusion and synchronization; deadlock, safety and liveness properties; message passing; concurrent software architectures; and dynamic and timed systems. The course uses state models and Java programs to introduce and illustrate key concepts and techniques.

Learning outcomes

After completing this course, you will be able to

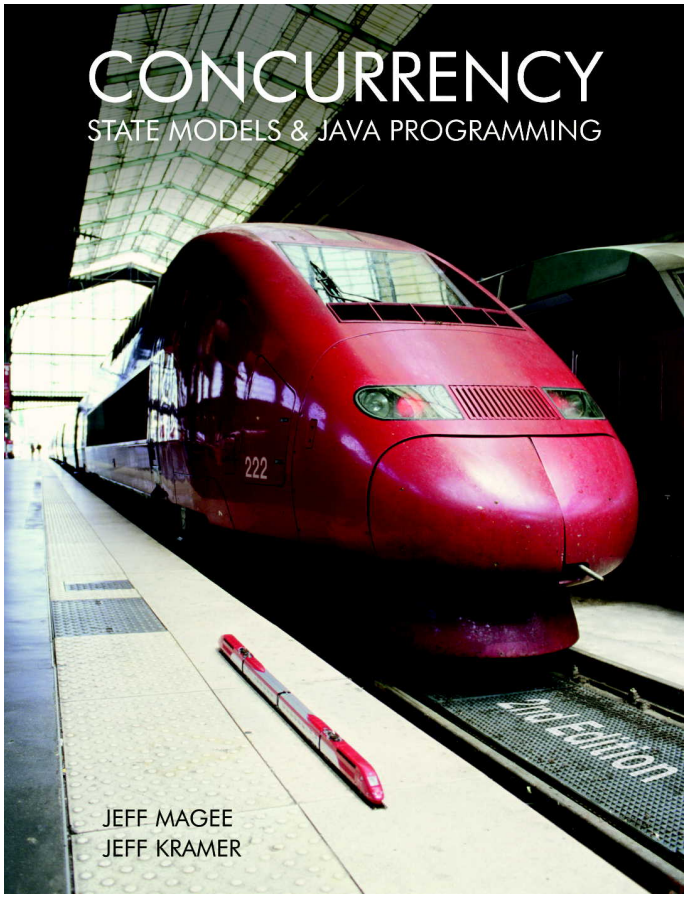
- design, analyze, and program parallel object-oriented systems.
- you will know the most important concepts and techniques for parallel programming
- you will know what are the problems which arise in parallel programming
- you will know what techniques you can use to solve these problems.

Book

Concurrency: State Models & Java Programs, 2nd Edition

Jeff Magee & Jeff Kramer

Wiley



Course Outline

- The main basic Concepts, Models, Practice
  1. Processes and Threads
  2. Concurrent Execution
  3. Shared Objects & Interference
  4. Monitors & Condition Synchronization
  5. Deadlock
  6. Safety and Liveness Properties
  7. Model-based Design
- Advanced topics ...
  1. Dynamic systems
  2. Message Passing
  3. Concurrent Software Architectures
  4. Timed Systems
  5. Program Verification
  6. Logical Properties

Web based course material

http://www.wileyeurope.com/college/magee

- Java examples and demonstration programs
- State models for the examples
- Tool
  - Labelled Transition System Analyser (**LTSA**) for modeling concurrency, model animation and model property checking.
    - installed on all "termstue" machines.
    - command *ltsa*

Motivation

- What
- Why

What is a Concurrent Program?

- A *sequential* program has a single thread of control.
- A *concurrent* program has multiple threads of control allowing it to perform multiple computations in parallel and to control multiple external activities which occur at the same time.

Concurrent and Distributed Software?

Interacting, concurrent software components of a system:

single machine $\longrightarrow$ *shared memory interactions*

multiple machines $\longrightarrow$ *network interactions*

Why Concurrent Programming?

- Performance gain from multiprocessing hardware
  - parallelism.
- Increased application throughput
  - an I/O call need only block one thread.
- Increased application responsiveness
  - high priority thread for user requests.
- More appropriate structure
  - for programs which interact with the environment, control multiple activities and handle multiple events.

Do I need to know about concurrent programming?

Concurrency is widespread but error prone.

- Therac-25 computerised radiation therapy machine

  Concurrent programming errors contributed to accidents causing deaths and serious injuries.
- Mars Rover

  Problems with interaction between concurrent tasks caused periodic software resets reducing availability for exploration.

Simple Example

- process 1: x := x + 1 (x shared variable)
- process 2: x := x - 1 (x shared variable)

Final result? Depending on the order of read and write operations on x, assuming read and write are atomic (and do not interfere).

Simple Example

There are 4 atomic x-operations: Process 1 reads x (R1), writes to x (W1). Process 2 reads x (R2), writes to x (W2). R1 must happen before W1 and R2 before W2, so these operations can be sequenced in 6 ways:

R1	R1	R1	R2	R2	R2
W1	R2	R2	R1	R1	W2
R2	W1	W2	W1	W2	R1
W2	W2	W1	W2	W1	W1
0	-1	1	-1	1	0

We see that the final value of x is -1, 0, 1. The program is thus non-deterministic: the result can vary from execution to execution.

Calculation of number of possible executions

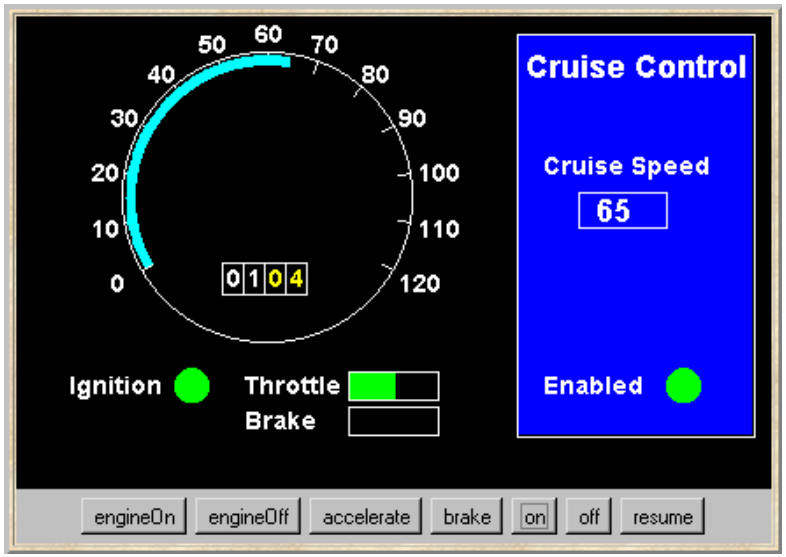
If we have 3 processes, each with a given number of atomic operations, we have the following number of possible executions:

process 1	process 2	process 3	number of executions
2	2	2	90
3	3	3	1680
4	4	4	34 650
5	5	5	756 756

Different executions can lead to different final states. Impossible, even for quite simple systems, to test every possible execution! For n processes with m atomic statements each, the formula for number of different executions is

$$\frac{(n * m)!}{m!^n}$$

a Cruise Control System



When the car ignition is switched on and the **on** button is pressed, the current speed is recorded and the system is enabled: *it maintains the speed of the car at the recorded setting.*

Pressing the brake, accelerator or **off** button disables the system. Pressing **resume** re-enables the system.

Is the system safe?

Would testing be sufficient to discover all errors?

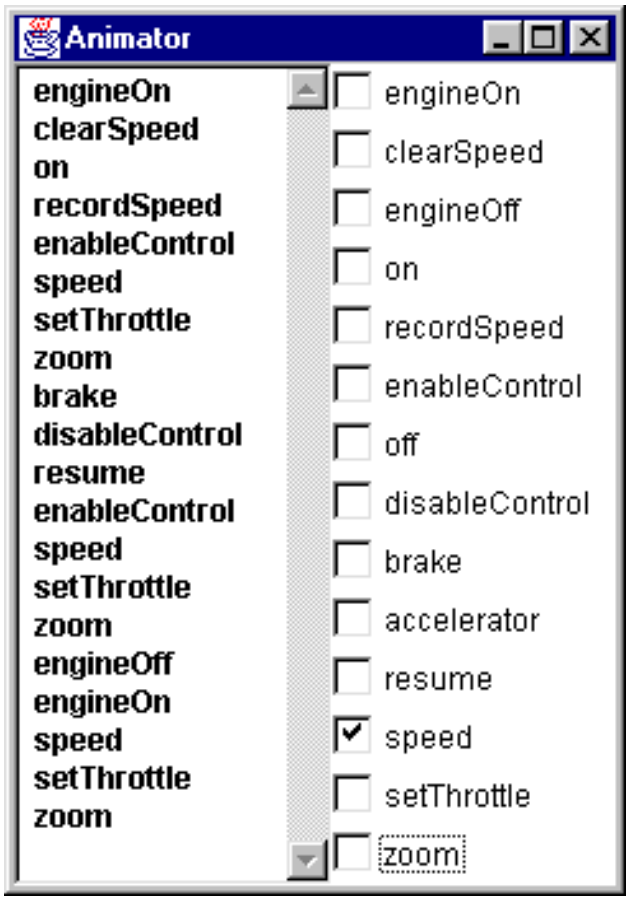
Models

A model is a simplified representation of the real world. Engineers use models to gain confidence in the adequacy and validity of a proposed design.

- focus on an aspect of interest – concurrency
- model animation to visualise a behaviour
- mechanical verification of properties (safety & progress)

Models are described using state machines, known as Labelled Transition Systems **LTS**. These are described textually as finite state processes (**FSP**) and displayed and analysed by the **LTSA** analysis tool.

Modeling the Cruise Control System



LTSA Animator to step through system actions and events.

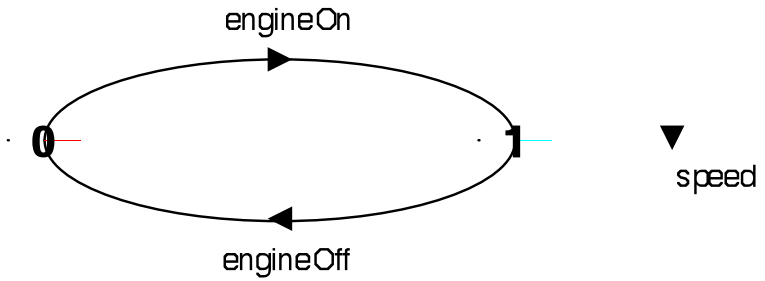


Figure: **LTS** of the process that monitors speed.

Later chapters will explain how to construct models such as this so as to perform animation and verification.

Programming practice in Java

Java is

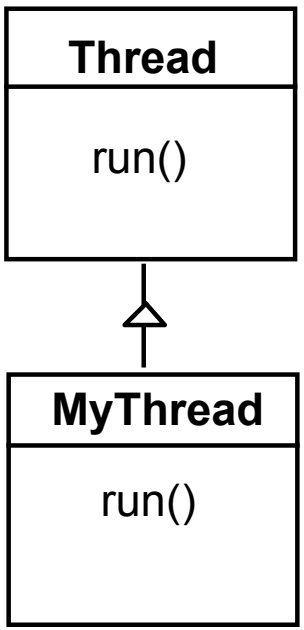
- widely available, generally accepted and portable
- provides sound set of concurrency features

Hence Java is used for all the illustrative examples, the demonstrations and the exercises. Later chapters will explain how to construct Java programs such as the Cruise Control System.

"Toy" problems are also used as they exemplify particular aspects of concurrent programming problems!

Threads in Java

A Thread class manages a single sequential thread of control. Threads may be created and deleted dynamically.

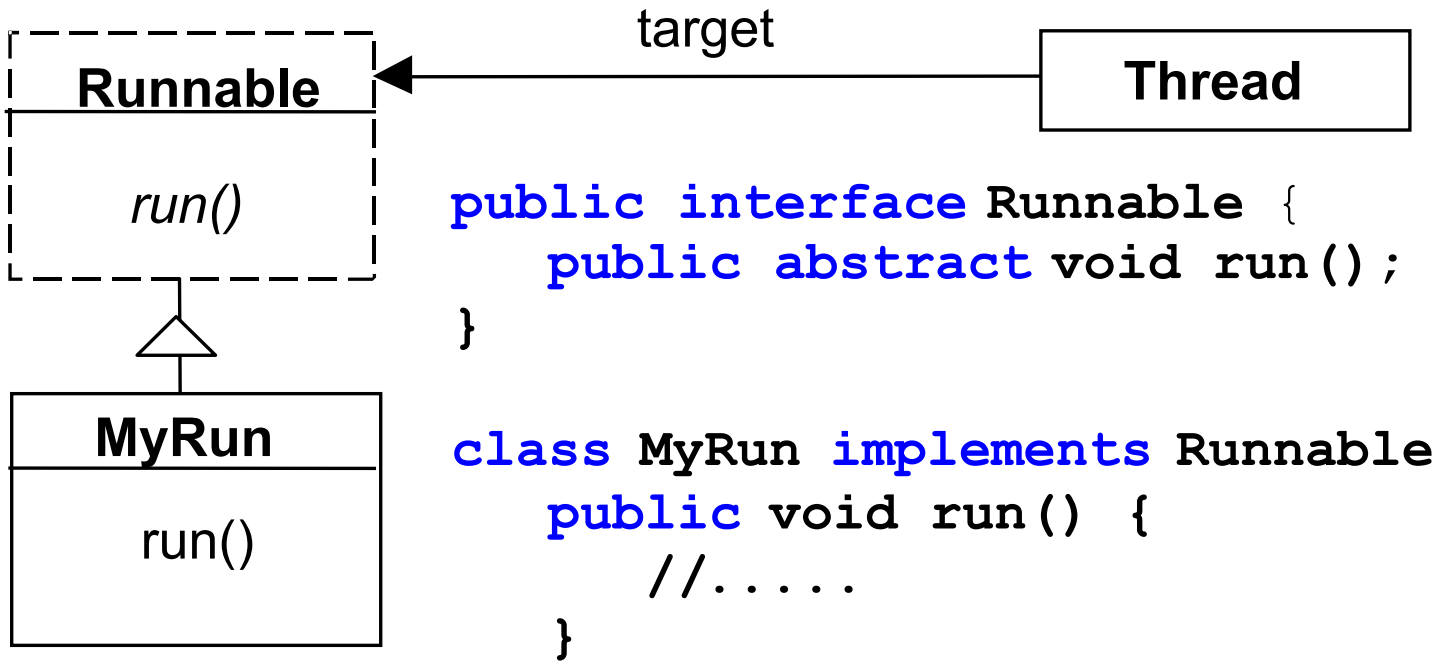


The Thread class executes instructions from its method `run()`. The actual code executed depends on the implementation provided for `run()` in a derived class.

```
class MyThread extends Thread {
  public void run() {
    //......
  }
}
  // Creating a thread object:
  Thread a = new MyThread();
```

Threads in Java

Since Java does not permit multiple inheritance, we often implement the `run()` method in a class not derived from `Thread` but from the interface `Runnable`.



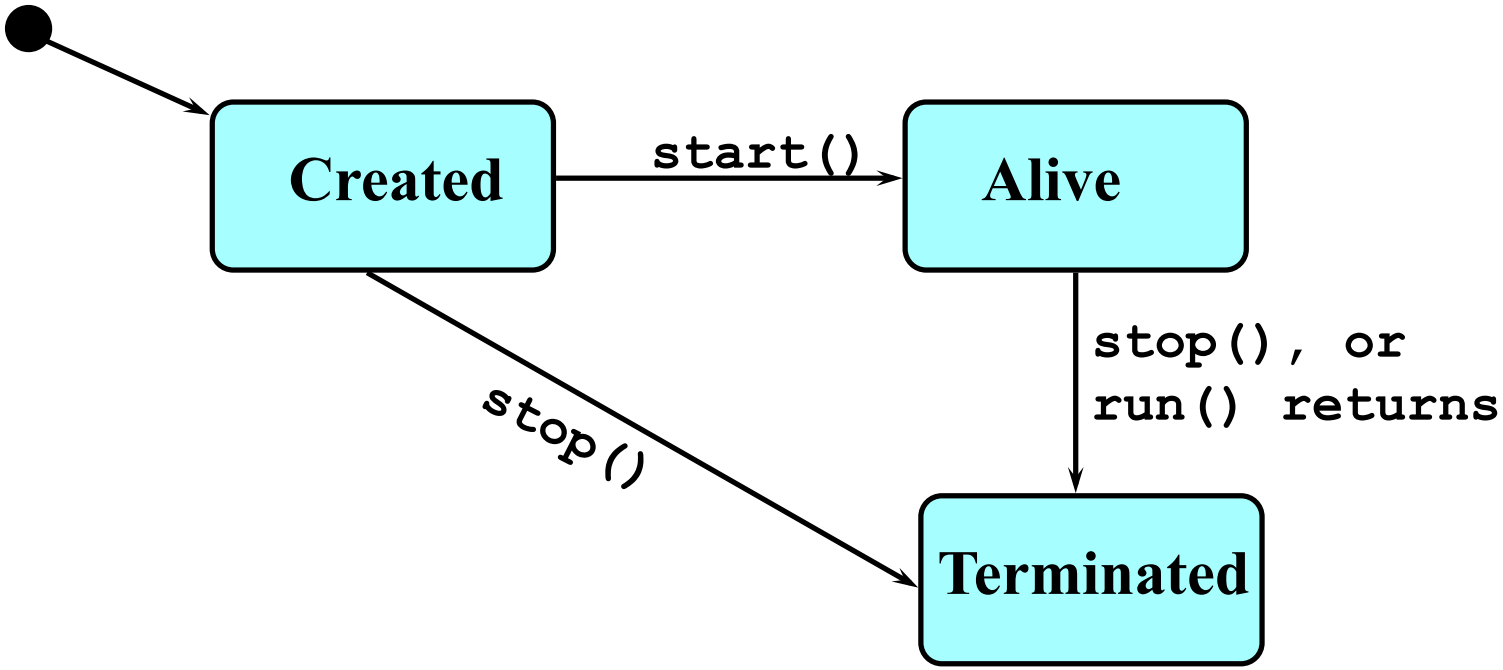

```
public interface Runnable {
    public abstract void run();
}

class MyRun implements Runnable{
    public void run() {
        //.....
    }
}
```

```
Thread b = new Thread(new MyRun());
```

thread life-cycle in Java

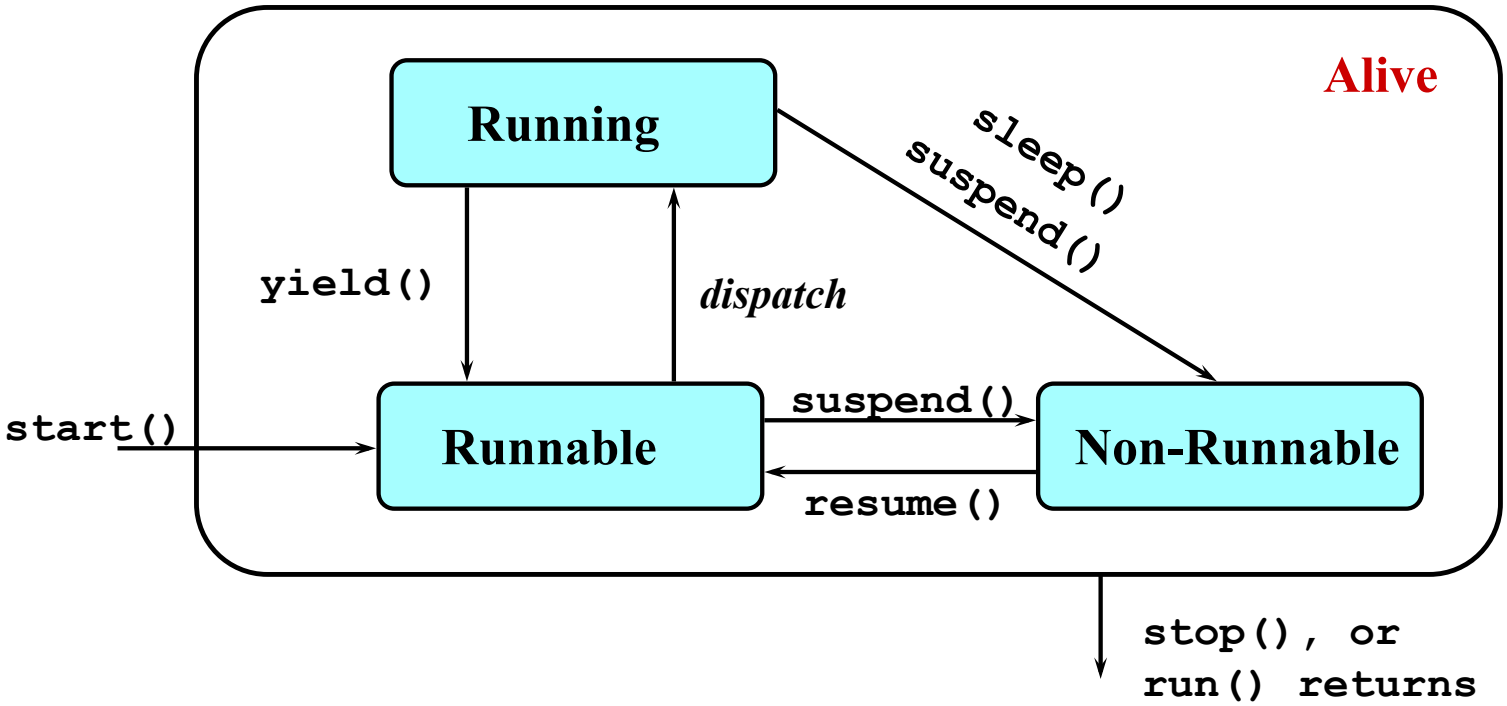
An overview of the life-cycle of a thread as state transitions:



- `start()` causes the thread to call its `run()` method.
- The predicate `isAlive()` can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted.

thread alive states in Java

Once started, an alive thread has a number of substates :



Also, `wait()` makes a thread non-Runnable, and `notify()` makes it Runnable (used in later chapters).

Note: `suspend`, `resume`, `stop` are deprecated (not recommended).

Course objective

This course is intended to provide a sound understanding of the *concepts*, *models* and *practice* involved in designing concurrent software.
The emphasis on principles and *concepts* provides a thorough understanding of both the problems and the solution techniques. *Modeling* provides insight into concurrent behavior and aids reasoning about particular designs. Concurrent programming in **Java** provides the programming *practice* and experience.

Summary

- Concepts
  - we adopt a model-based approach for the design and construction of concurrent programs
- Models
  - we use finite state models to represent concurrent behavior.
- Practice
  - we use Java for constructing concurrent programs.

Examples are used to illustrate the concepts, models and demonstration programs.