Lecture 1: Introduction, Processes & Threads

Teacher

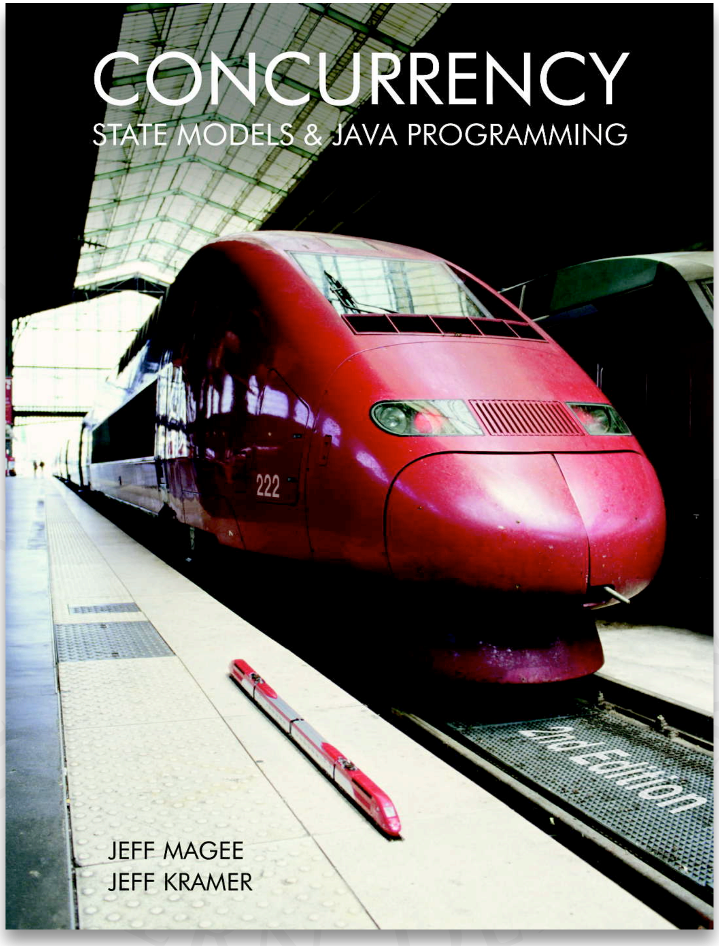
Peter Schneider-Kamp
<petersk@imada.sdu.dk>

Teaching Assistants

Christian Østergaard Lautrup Nørskov (S7)
Mathias Wulff Svendsen (S17)
Jakob Lykke Andersen (S1)

Textbook

[M&K] Concurrency: State Models & Java Programs (2nd edition). Jeff Magee & Jeff Kramer. Wiley. 2006, ISBN: 0-470-09355-2

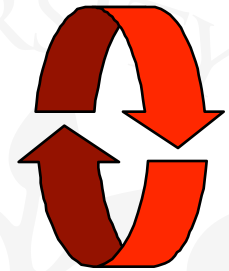


Course Home Page

http://imada.sdu.dk/~petersk/DM519/

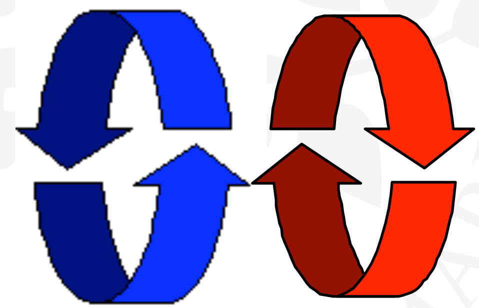
What is a Concurrent Program?

A *sequential* program has a single thread of control.



A *concurrent* program has multiple threads of control:

- perform multiple computations in parallel
- control multiple external activities occurring simultaneously.



Why Concurrent Programming?

UNIVERSITY OF SOUTHERN DENMARK

More appropriate program structure

- Concurrency reflected in program

Performance gain from multiprocessing HW

- Parallelism

Increased application throughput

- An I/O call need only block one thread

Increased application responsiveness

- High-priority thread for user requests

Concurrency is much Harder

Harder than sequential programming:

- Huge number of possible executions
- Inherently non-deterministic
- Parallelism conceptually harder

Consequences:

- Programs are harder to write(!)
- Programs are harder to debug(!) (Heisenbugs)
- Errors are not always reproducible(!)
- New kinds of errors possible(!):
  - Deadlock, starvation, priority inversion, interference, …

Solution: Model-based Design

Model: a simplified representation of the real world.

- focus on *concurrency aspects*

Design *abstract model*

Decompose **model**

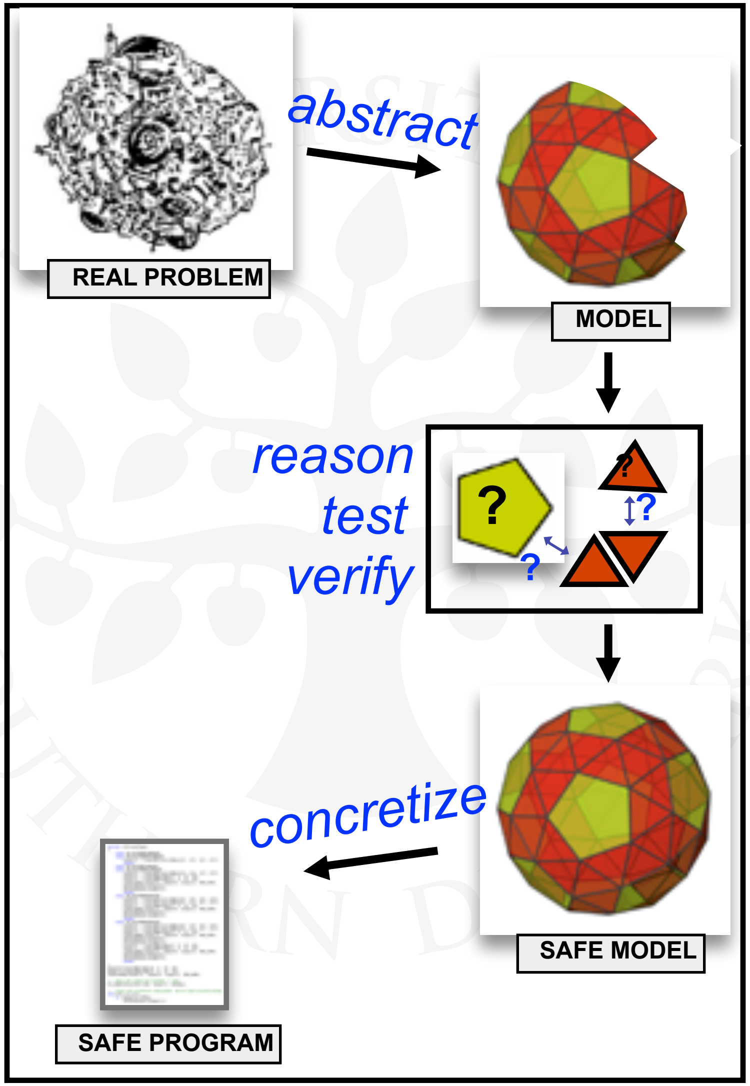
Reason/Test/Verify model

- individual parts and whole

Recompose **insights**

- make model safe

Implement *concrete program*



What you will be able to do after the course

Construct models from specifications of concurrency problems

Test, analyze, and compare models' behavior

Define and verify models' safety/liveness properties (using tools)

Implement models in Java

Relate models and implementations

How to achieve them?

UNIVERSITY OF SOUTHERN DENMARK

Lectures

Theoretical exercises during the discussion sections

Practical exercises in your study groups

Evaluation: Graded project exam

- mid-quarter deadline for model (March 14)
- end-quarter deadline for implementation & report (April 18)

Concurrent Processes

We structure complex systems as sets of simpler activities, each represented as a (sequential) process

Processes can be concurrent

Designing concurrent software:
- **complex** and **error prone**

Concept: process ~ sequences of actions

Model: process ~ Finite State Processes (FSP)

Practice: process ~ Java thread

Modelling Processes

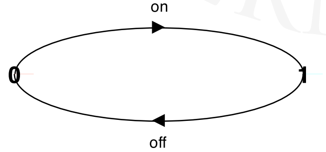
Models are described using state machines, known as **Labelled Transition Systems** (LTS)

These are described textually as **Finite State Processes** (FSP)

Analysed/Displayed by the **LTS Analyser** (LTSA)

- FSP - algebraic form
- LTS - graphical form

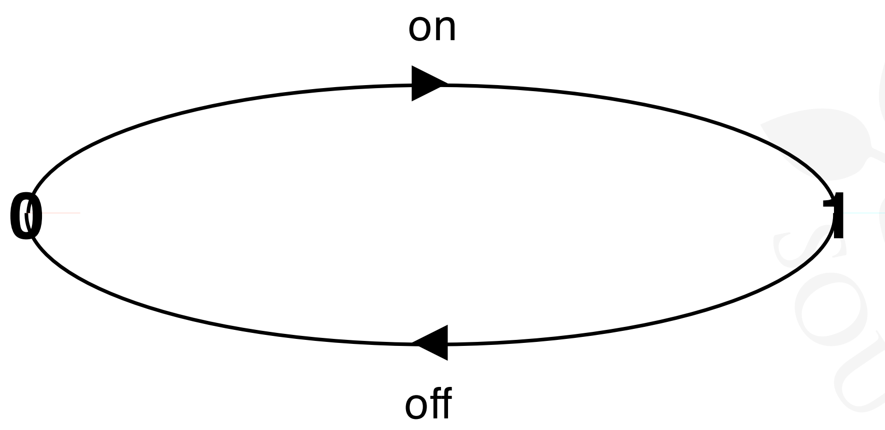
```
SWITCH = OFF,
OFF    = (on -> ON),
ON     = (off-> OFF).
```



Modelling Processes

A **process** is modelled by a sequential program.

It is modelled as a **finite state machine** which transits from state to state by executing a sequence of atomic actions.



a light switch
LTS

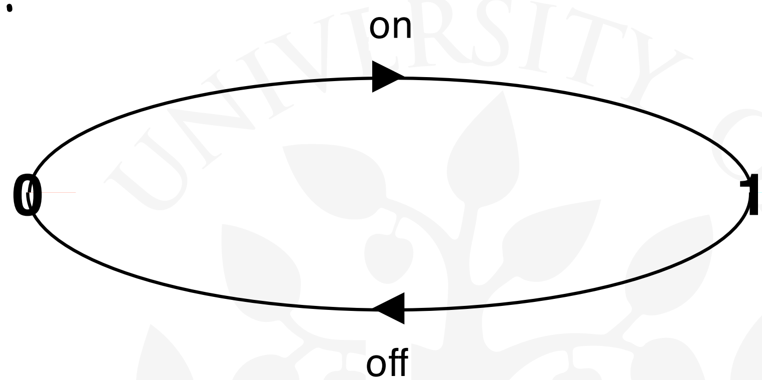
`on→off→on→off→on→off→ ……….`

a sequence of actions or **trace**

FSP - action prefix & recursion

Repetitive behaviour uses recursion:

```
SWITCH = OFF,
OFF    = (on -> ON),
ON     = (off-> OFF).
```



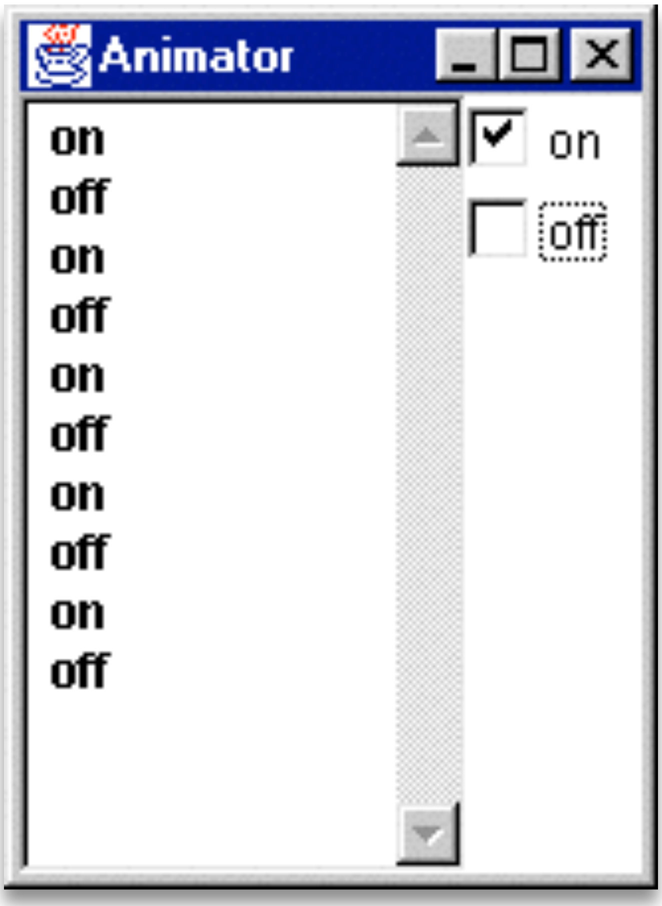
Substituting to get a more succinct definition:

```
SWITCH = OFF,
OFF    = (on ->(off->OFF)).
```

Again?:

```
SWITCH = (on->off->SWITCH).
```

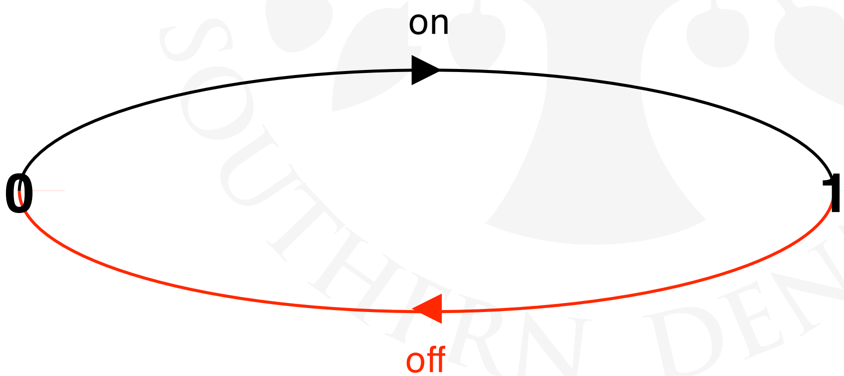
Animation using LTSA



The LTSA animator can be used to produce a **trace**.

Ticked actions are eligible for selection.

In the LTS, the last action is highlighted in red.

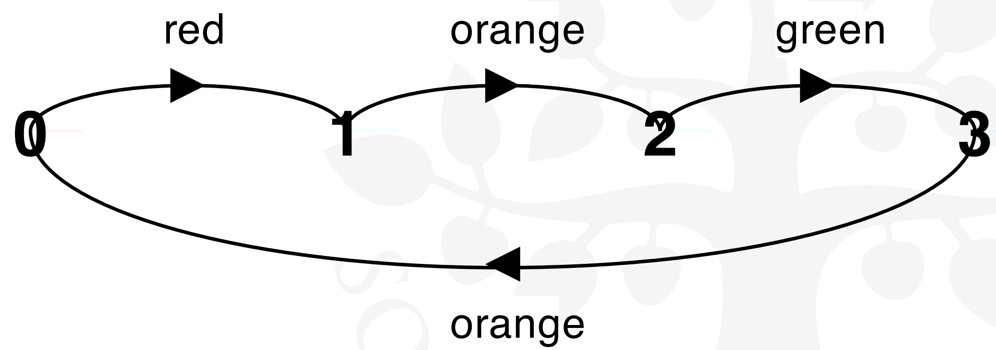


FSP - action prefix

FSP model of a traffic light:

```
TRAFFICLIGHT = (red->orange->green->orange
                         -> TRAFFICLIGHT).
```

LTS?



Trace(s)?

red→orange→green→orange→red→orange→green …

What would the LTS look like for?:

```
T = (red->orange->green->orange->STOP).
```

FSP - choice

If **x** and **y** are actions then **(x-> P | y-> Q)** describes a process which initially engages in either of the actions **x** or **y**. After the first action has occurred, the subsequent behavior is described by **P** if the first action was **x**; and **Q** if the first action was **y**.

Who or what makes the choice?

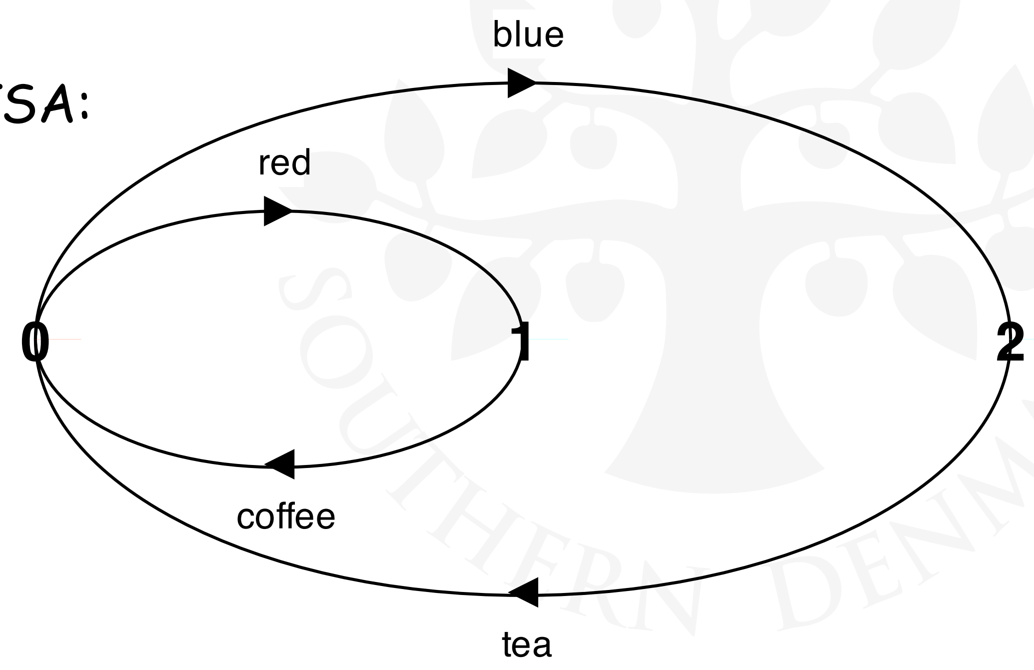
Is there a difference between input and output actions?

FSP - choice

FSP model of a drinks machine :

```
DRINKS = (red->coffee->DRINKS
         |blue->tea->DRINKS
         ).
```

LTS generated using LTSA:



Possible traces?

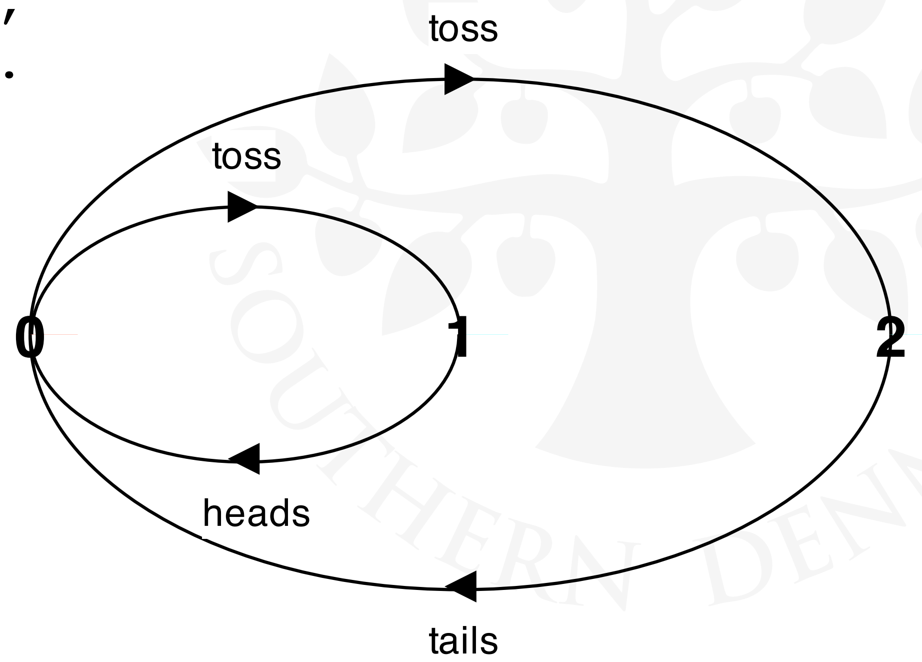
Non-deterministic choice

> Process (x -> P | x -> Q) describes a process which engages in **x** and then **non-deterministically** behaves as either P or Q.

```
COIN = (toss->HEADS|toss->TAILS),
HEADS= (heads->COIN),
TAILS= (tails->COIN).
```

Tossing a coin.

LTS?



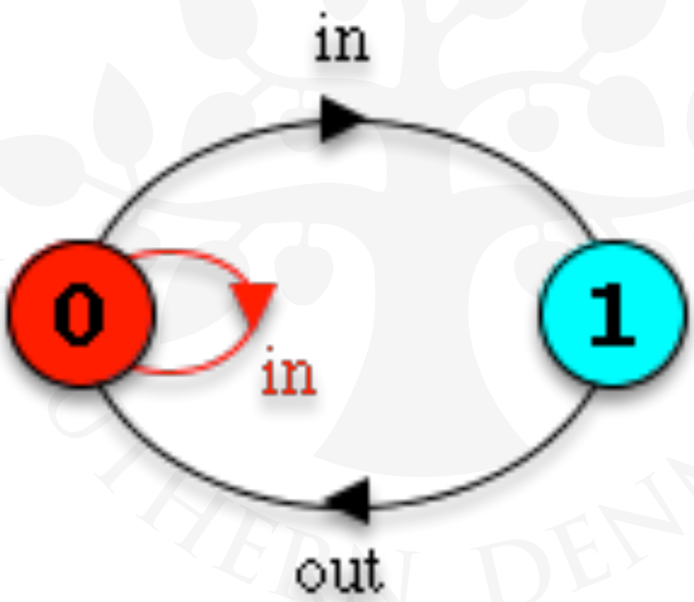
Possible traces?

Example: Modelling unreliable communication channel

How do we model an unreliable communication channel which accepts **in** actions and if a failure occurs produces no output, otherwise performs an **out** action?

Use non-determinism...:

```
CHAN = (in->CHAN
       |in->out->CHAN
       ).
```



FSP - indexed processes and actions

Single slot buffer that inputs a value in the range 0 to 3 and then outputs that value:

```
BUFF = (in[i:0..3]->out[i]-> BUFF).
```

Define then Use (as in programming languages)

Could we have made this process w/o using the indices?

```
BUFF = (in_0->out_0->BUFF
       |in_1->out_1->BUFF
       |in_2->out_2->BUFF
       |in_3->out_3->BUFF
       ).
```

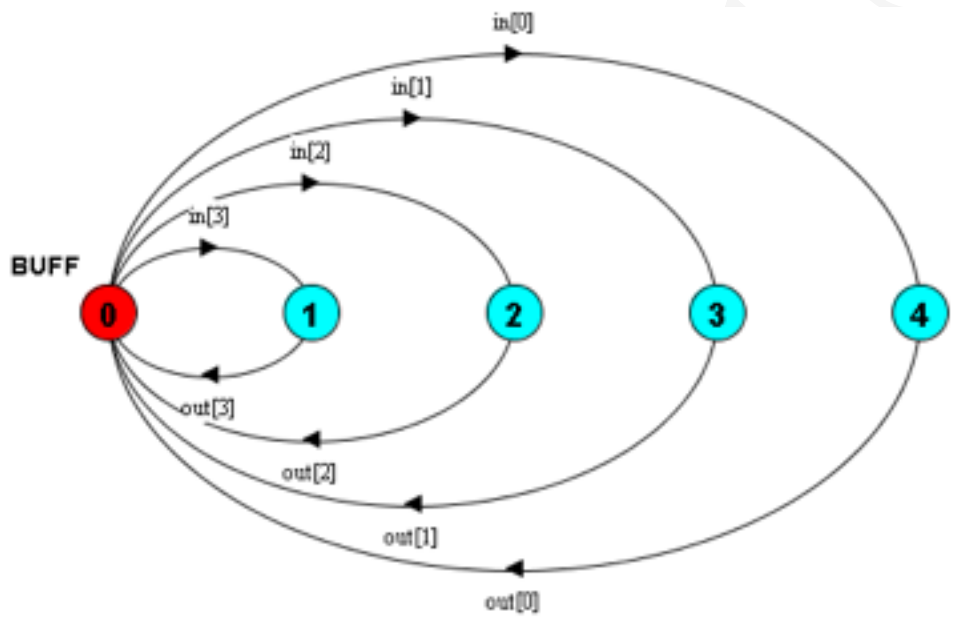
...or...:

```
BUFF = (in[0]->out[0]->BUFF
       |in[1]->out[1]->BUFF
       |in[2]->out[2]->BUFF
       |in[3]->out[3]->BUFF
       ).
```

Indices (cont'd)

```
BUFF = (in[i:0..3]->out[i]-> BUFF).    or

BUFF = (in[0]->out[0]->BUFF
       |in[1]->out[1]->BUFF
       |in[2]->out[2]->BUFF
       |in[3]->out[3]->BUFF).
```

LTS?



FSP - indexed processes and actions (cont'd)


```
BUFF = (in[i:0..3]->out[i]-> BUFF).
```

equivalent to

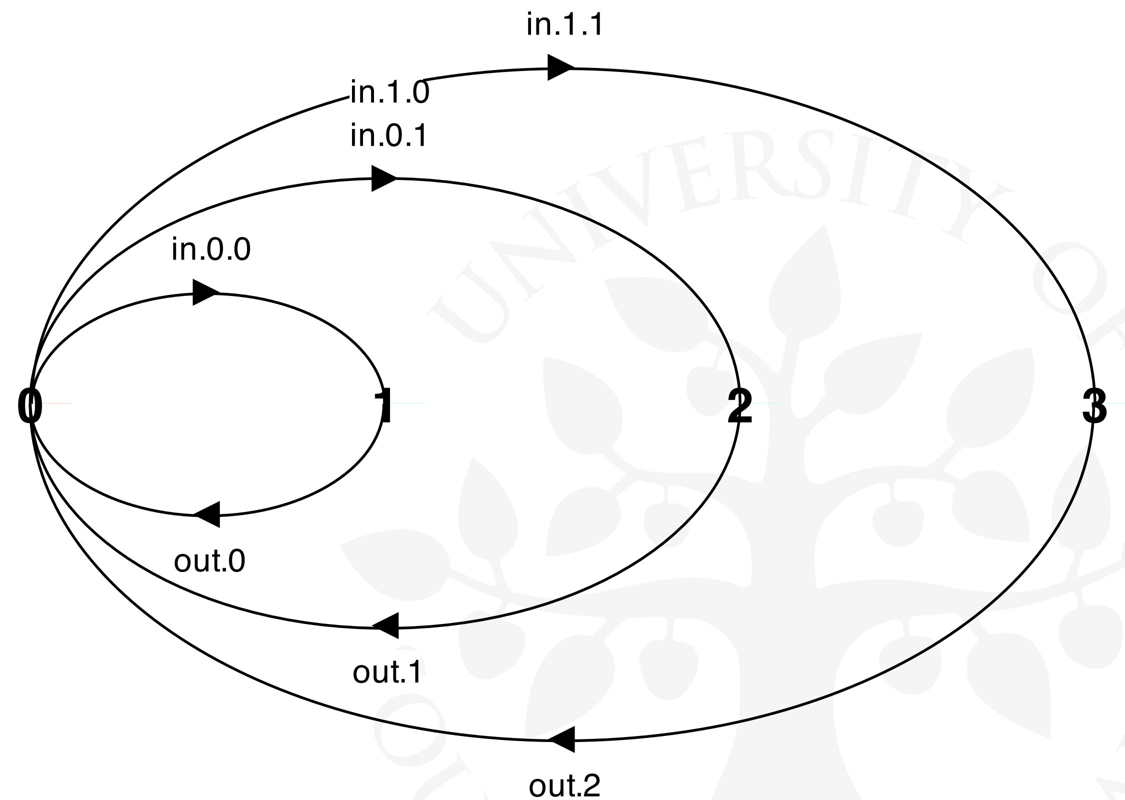
```
BUFF        = (in[i:0..3]->OUT[i]),
OUT[i:0..3] = (out[i]->BUFF).
```

equivalent to

```
BUFF        = (in[i:0..3]->OUT[i]),
OUT[j:0..3] = (out[j]->BUFF).
```

FSP - constant & addition

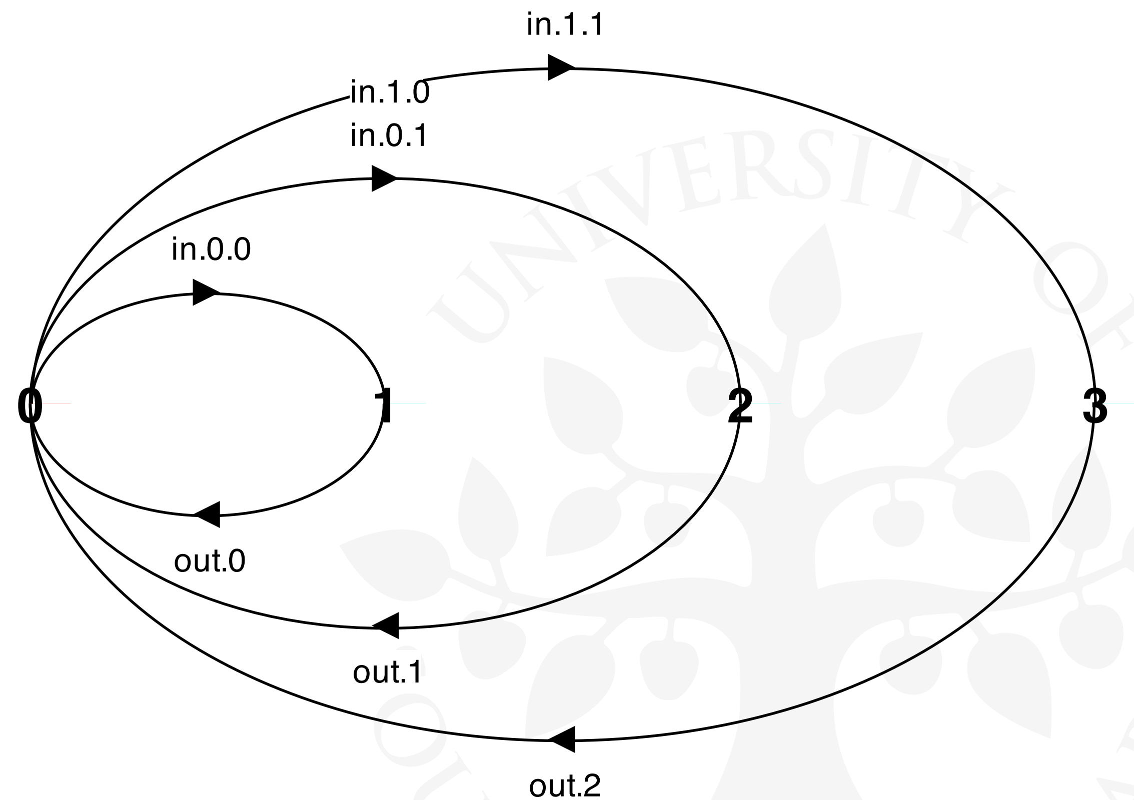
index expressions to model calculation:




```
const N = 1

SUM          = (in[a:0..N][b:0..N]->TOTAL[a+b]),
TOTAL[s:0..2*N] = (out[s]->SUM).
```

FSP - constant & range declaration

index expressions to model calculation:



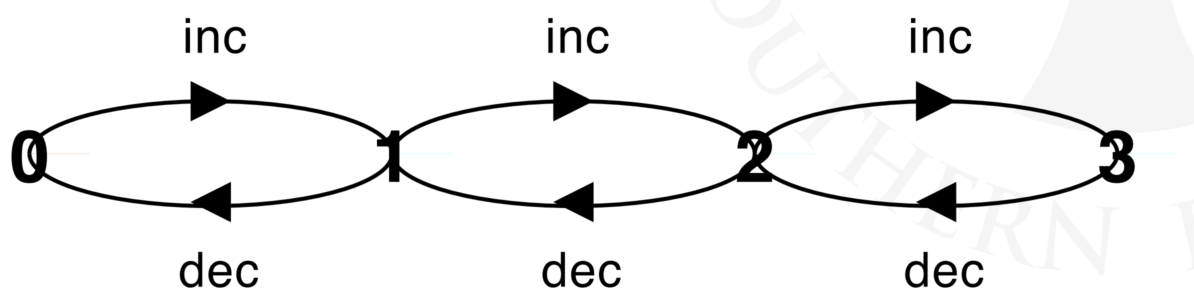

```
const N = 1
range T = 0..N
range R = 0..2*N

SUM          = (in[a:T][b:T]->TOTAL[a+b]),
TOTAL[s:R]   = (out[s]->SUM).
```

FSP - guarded actions

The choice (**when** B x -> P | y -> Q) means that when the guard B is true then the actions x and y are both eligible to be chosen, otherwise if B is false then the action x cannot be chosen.

```
COUNT (N=3)          = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
                |when(i>0) dec->COUNT[i-1]
                ).
```

LTS?

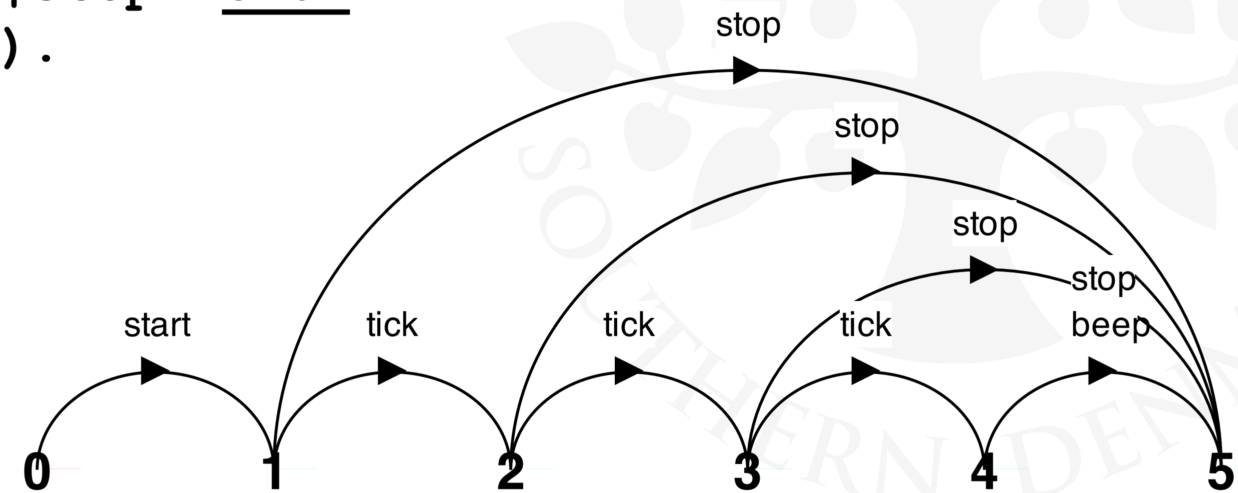


Could we have made this process w/o using the guards?

FSP - guarded actions

A countdown timer which beeps after N ticks, or can be stopped.

```
COUNTDOWN (N=3)    = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
            (when(i>0) tick->COUNTDOWN[i-1]
            |when(i==0)beep->STOP
            |stop->STOP
            ).
```



FSP - guarded actions

What is the following FSP process equivalent to?

```
const False = 0
P = (when (False) do_anything->P).
```

Answer:

STOP

FSP - process alphabets

The **alphabet** of a process is the set of actions in which it can engage.

Alphabet extension can be used to extend the **implicit** alphabet of a process:

```
WRITER = (write[1]->write[3]->WRITER)
                +{write[0..3]}.
```

Alphabet of **WRITER** is the set **{write[0..3]}**

(we make use of alphabet extensions in later chapters)

Practice

Threads in Java

2.2 Implementing processes

Modelling processes as finite state machines using FSP/LTS.

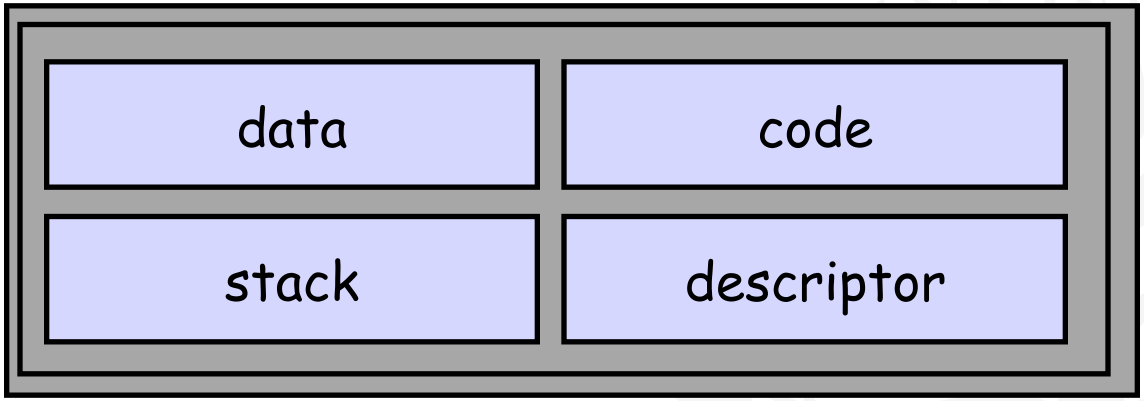
Implementing threads in Java.

Note: to avoid confusion, we use the term **process** when referring to the models, and **thread** when referring to the implementation in Java.

One Process

University of Southern Denmark

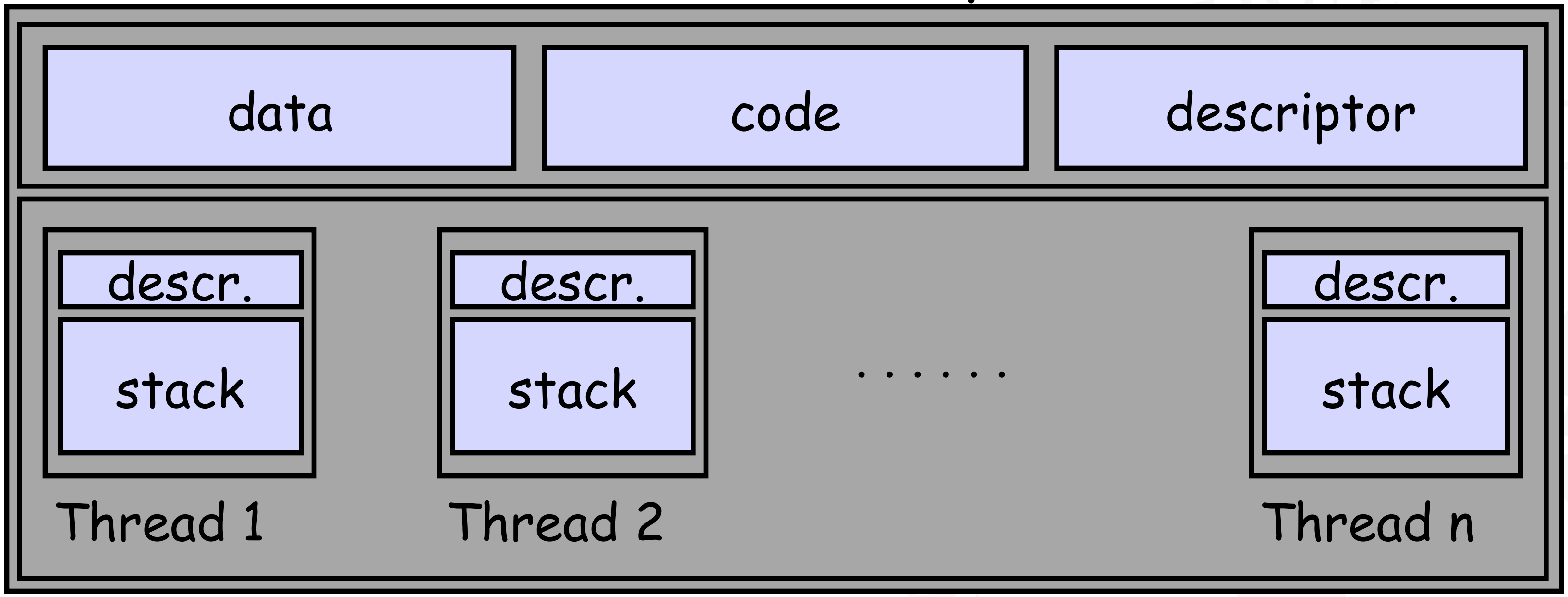
- Process:



- Data: The heap (global, heap allocated data)
- Code: The program (bytecode)
- Stack: The stack (local data, call stack)
- Descriptor: Program counter, stack pointer, …

Implementing processes - the OS view

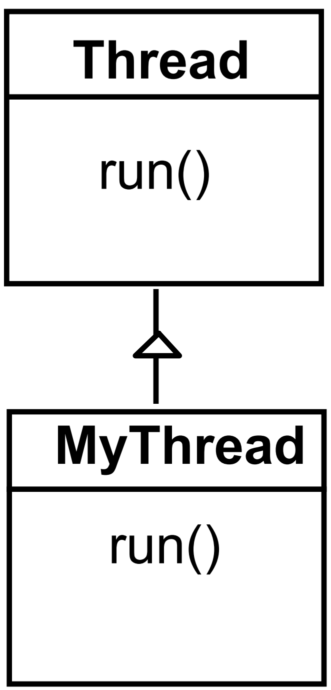
A multi-threaded process



A (heavyweight) process in an operating system is represented by its code, data and the state of the machine registers, given in a descriptor. In order to support multiple (lightweight) **threads of control**, it has multiple stacks, one for each thread.

Threads in Java

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



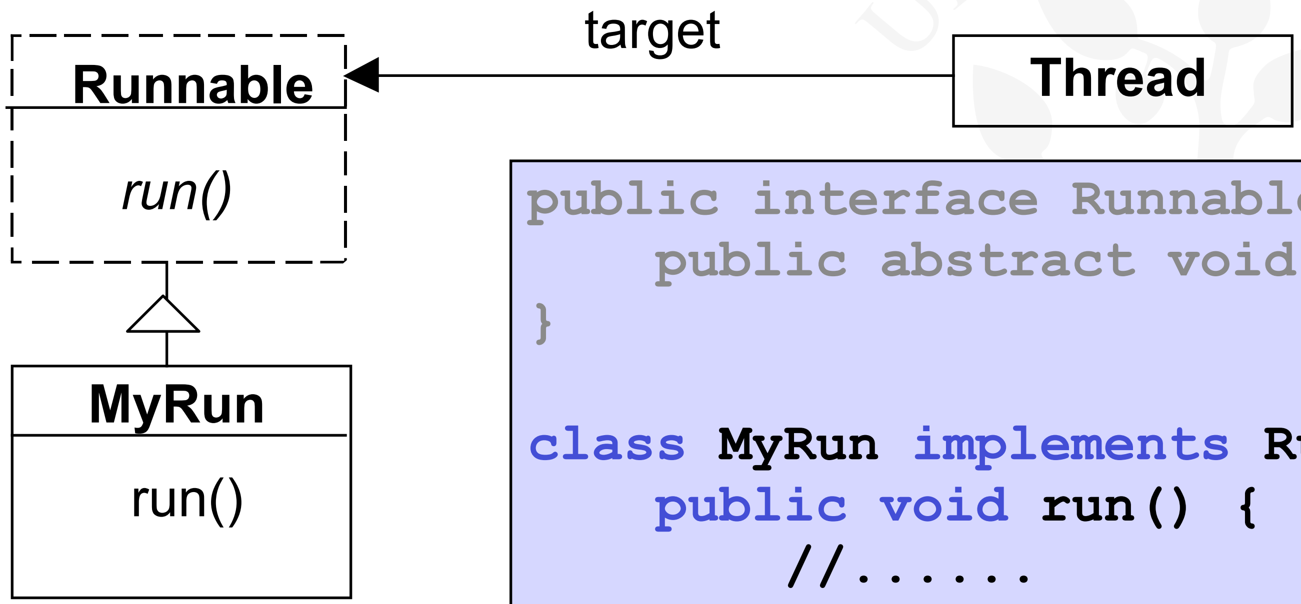
The Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

```
class MyThread extends Thread {
    public void run() {
        //......
    }
}
```

```
Thread x = new MyThread();
```

Threads in Java (cont'd)

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.

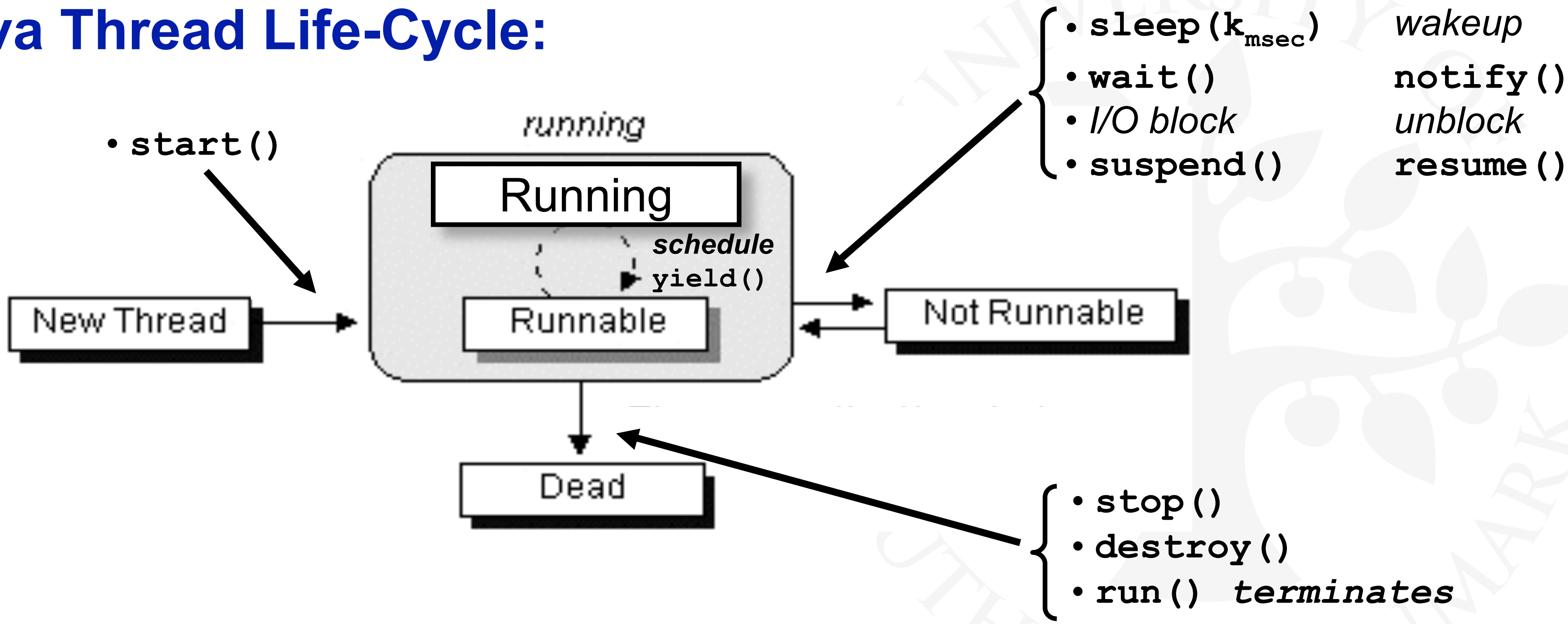



```
public interface Runnable {
    public abstract void run();
}

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

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

Thread Life-Cycle

Java Thread Life-Cycle:

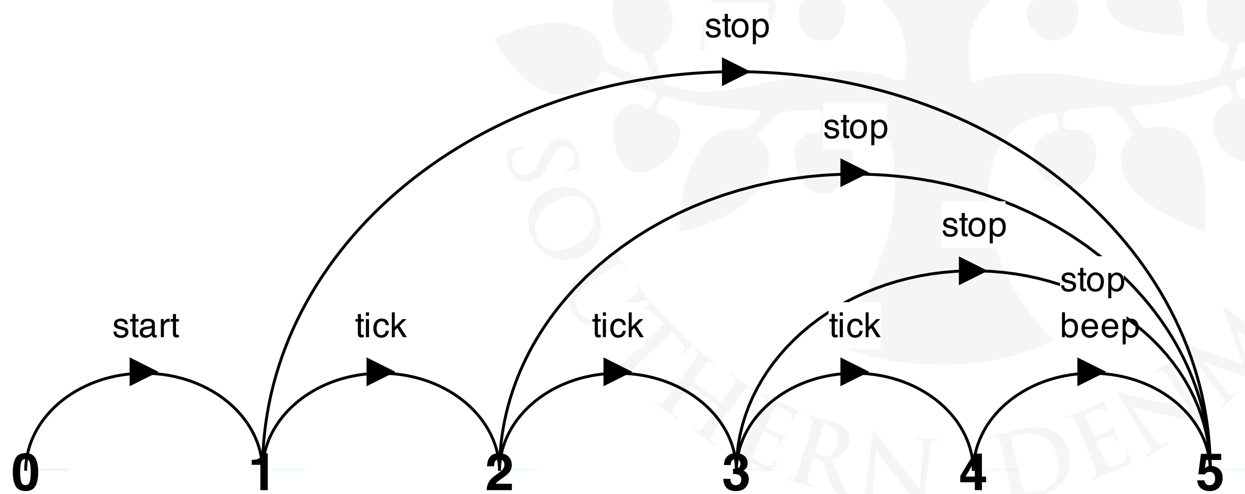


Example: Countdown timer

Model <-> Implementation

CountDown timer example

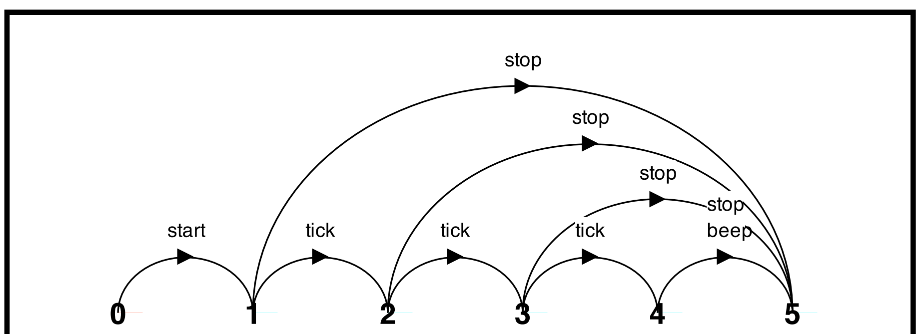
```
const N = 3
COUNTDOWN = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
            (when(i>0)  tick->COUNTDOWN[i-1]
            |when(i==0) beep->STOP
            |stop->STOP
            ).
```



Implementation in Java?

CountDown class


```java
public class CountDown implements Runnable {
    Thread counter;
    int i;
    final static int N = 3;

    public void run()          { ... }
    public void start()        { ... }
    public void stop()         { ... }
    protected void tick()      { ... }
    protected void beep()      { ... }
}
```




```
const N = 3
COUNTDOWN = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
        (when(i>0)  tick->COUNTDOWN[i-1]
        |when(i==0) beep->STOP
        |stop->STOP
        ).
```

CountDown class - start(), stop() and run()

```
public void start() {
   counter = new Thread(this);
   i = N; counter.start();
}

public void stop() {
   counter = null;
}

public void run() {
   while(true) {
     if (i>0)  { tick(); --i; }
     if (i==0) { beep(); return;}
     if (counter == null) return;
   }
}
```

COUNTDOWN Model

```
start -> CountDown[N]
```

```
stop -> STOP
```

COUNTDOWN[i] process

recursion as a while loop

```
when(i>0)  tick -> CD[i-1]
when(i==0)beep -> STOP
stop->STOP
```

STOP ~ run() *terminates*

CountDown class – the output actions: tick() and beep()


```
protected void tick() {
    <<emit tick sound>>
    try {
        Thread.sleep(1000);
    } catch(InterruptedException iex){
        // ignore (in this toy-example)
    }
}

protected void beep() {
    <<emit beep sound>>
}
```

Summary

Concepts

- process - unit of concurrency, execution of a program

Models

- LTS (Labelled Transition System) to model processes as state machines - sequences of atomic actions
- FSP (Finite State Process) to specify processes using prefix “->”, choice ” | ” and recursion

Practice

- Java threads to implement processes
- Thread lifecycle
  (created, running, runnable, non-runnable, terminated)

Near Future

Lecture Friday:

- M&K: Chapter 3

Discussion Sections & Study Groups

- Details are in Weekly Note 1