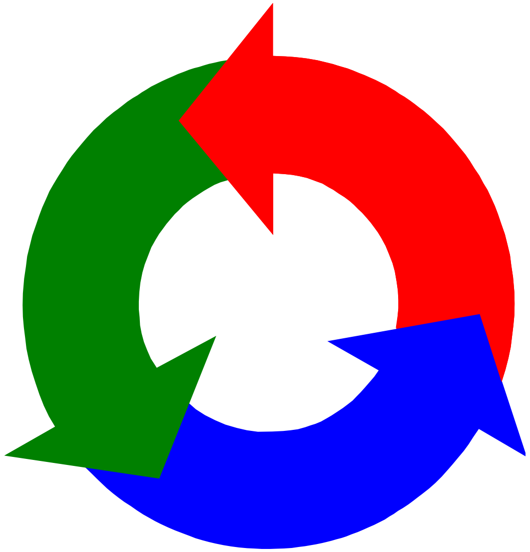
Chapter 2

Processes & Threads



concurrent processes

We structure complex systems as sets of simpler activities, each represented as a **sequential process**. Processes can overlap or be concurrent, so as to reflect the concurrency inherent in the physical world, or to offload time-consuming tasks, or to manage communications or other devices.

Designing concurrent software can be complex and error prone. A rigorous engineering approach is essential.

Concept of a process as a sequence of actions.

↓

Model processes as finite state machines.

↓

Program processes as threads in Java.

processes and threads

Concepts: processes - units of sequential execution.

Models: **finite state processes (FSP)**
to model processes as sequences of actions.
labelled transition systems (LTS)
to analyse, display and animate behavior.

Practice: Java threads

2.1 Modelling Processes

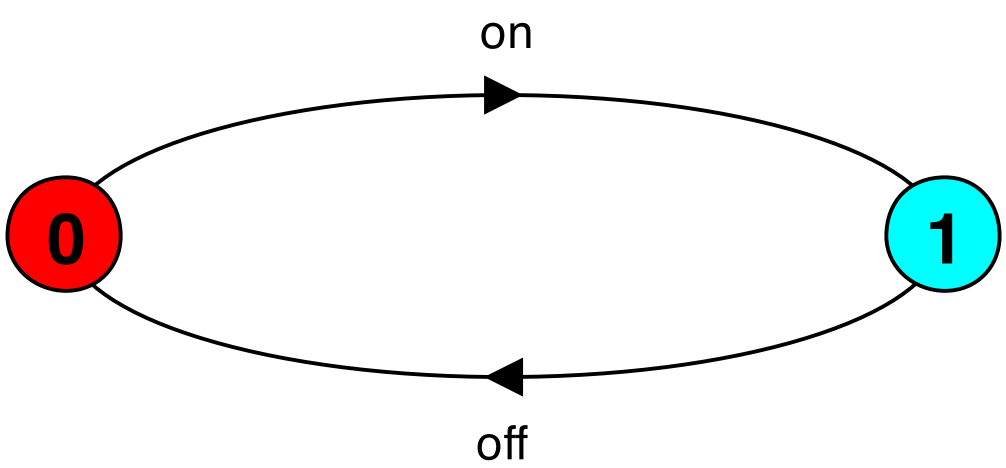
Models are described using state machines, known as Labelled Transition Systems LTS. These are described textually as finite state processes (FSP) and displayed and analysed by the ***LTSA*** analysis tool.

- LTS - graphical form
- FSP - algebraic form

LTSA and an FSP quick reference are available at http://www-dse.doc.ic.ac.uk/concurrency/

modelling processes

A process is the execution of 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→ ……….

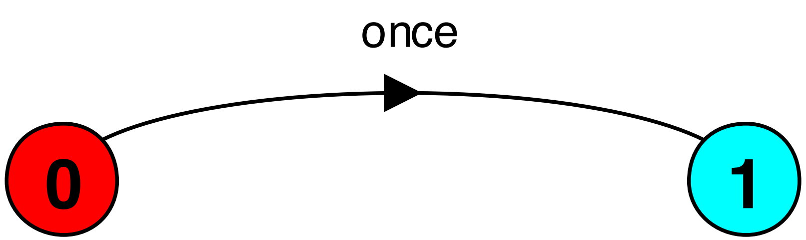
a sequence of actions or *trace*

Can finite state models produce infinite traces?

FSP - action prefix

If **x** is an action and **P** a process then **(x-> P)** describes a process that initially engages in the action **x** and then behaves exactly as described by **P**.

```
ONESHOT = (once -> STOP).
```



ONESHOT state machine

(terminating process)

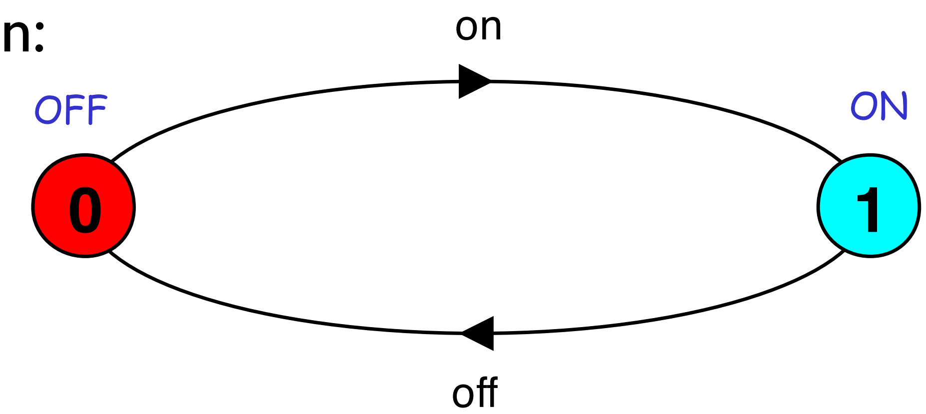
Convention: actions begin with lowercase letters

PROCESSES begin with uppercase letters

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)).
```

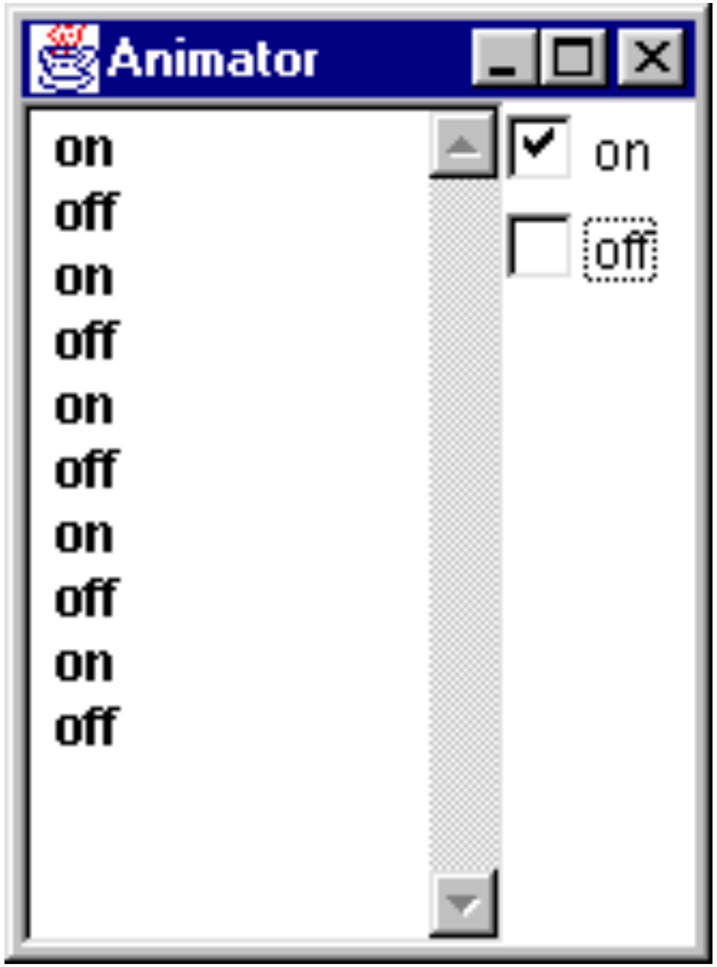
And again:

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

Scope:

OFF and ON are local subprocess definitions, local to the SWITCH definition.

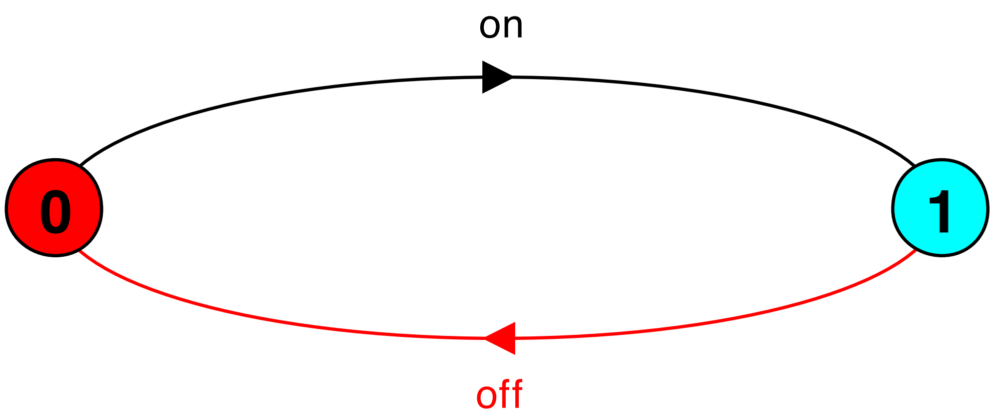
animation using LTSA



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.

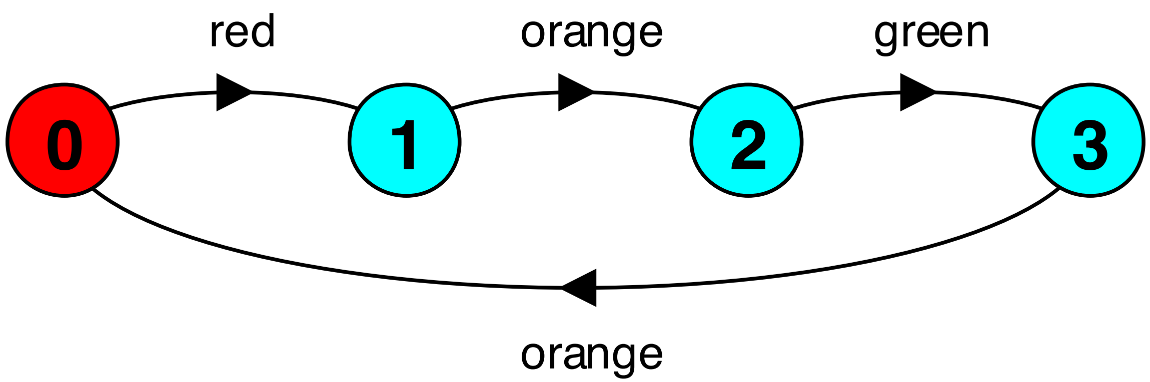


FSP - action prefix

FSP model of a traffic light :

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

LTS generated using *LTSA*:



Trace:

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

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?

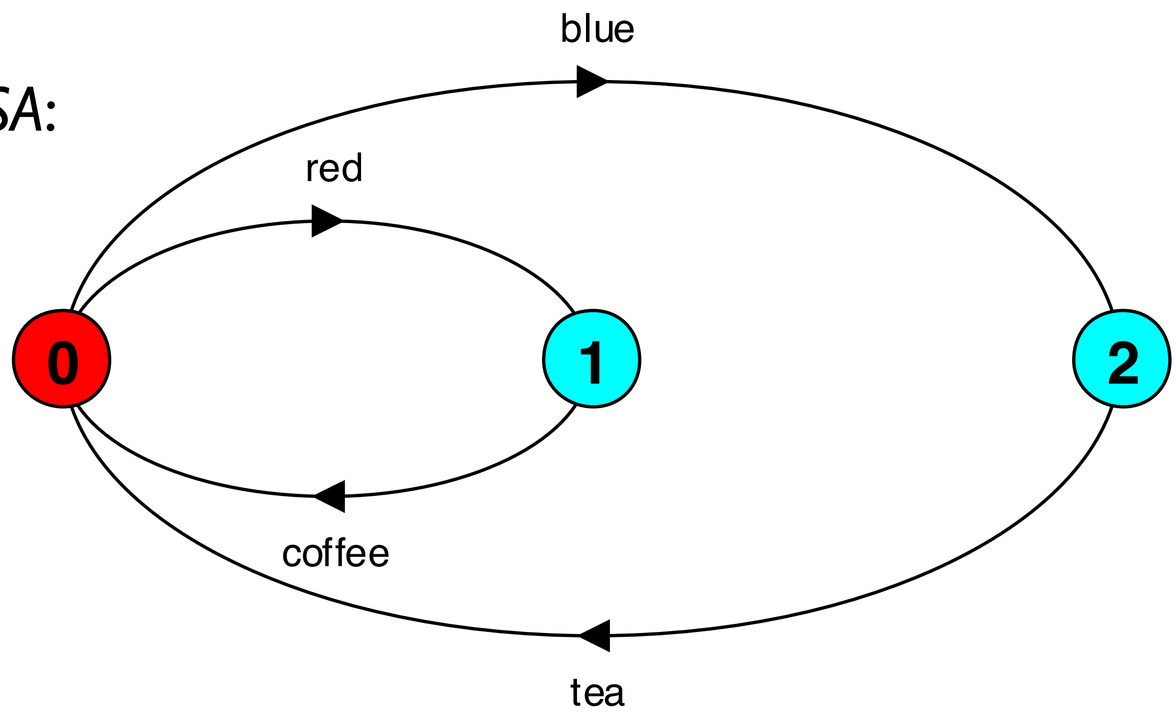
FSP - choice

FSP model of a drinks machine :

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

input?
output?

LTS generated using *LTSA*:



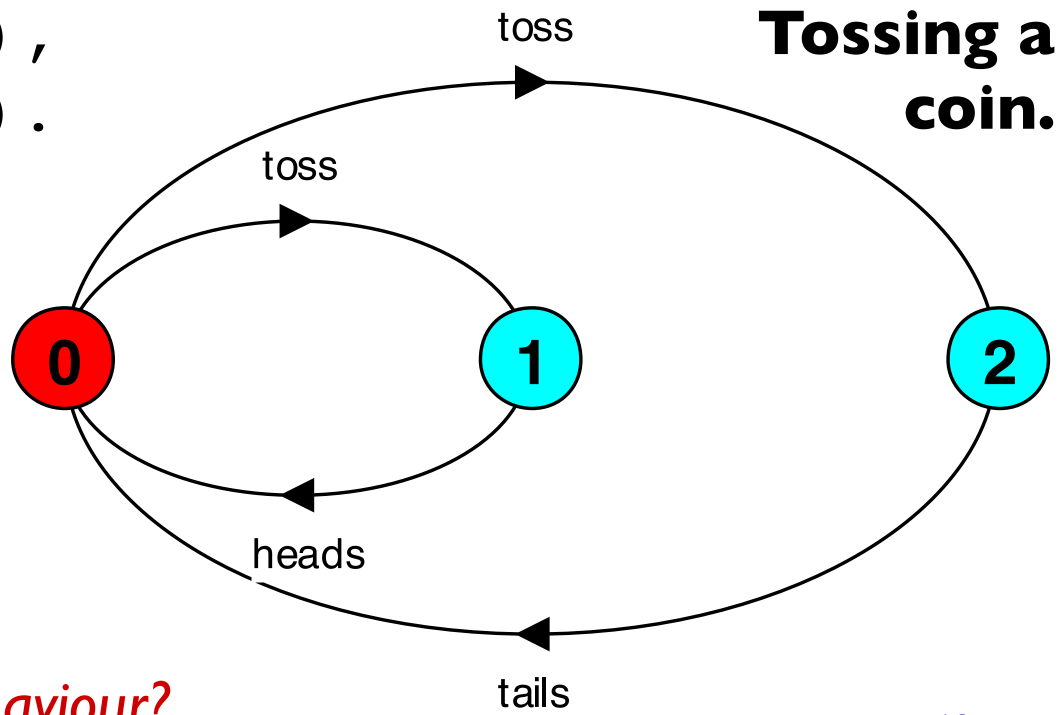
Possible traces?

Non-deterministic choice

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

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

Tossing a coin.



Possible traces?

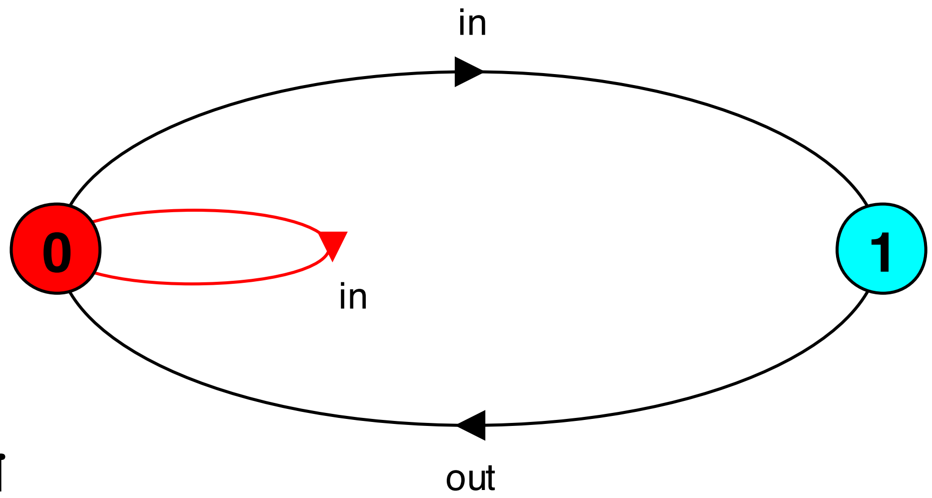
Could we make this deterministic and trace equivalent?

Would it really have equivalent behaviour?

Modelling failure

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
       ).
```

Deterministic?

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).
```

equivalent to

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

Local indexed process definitions are equivalent to process definitions for each index value

or using a process parameter with default value:

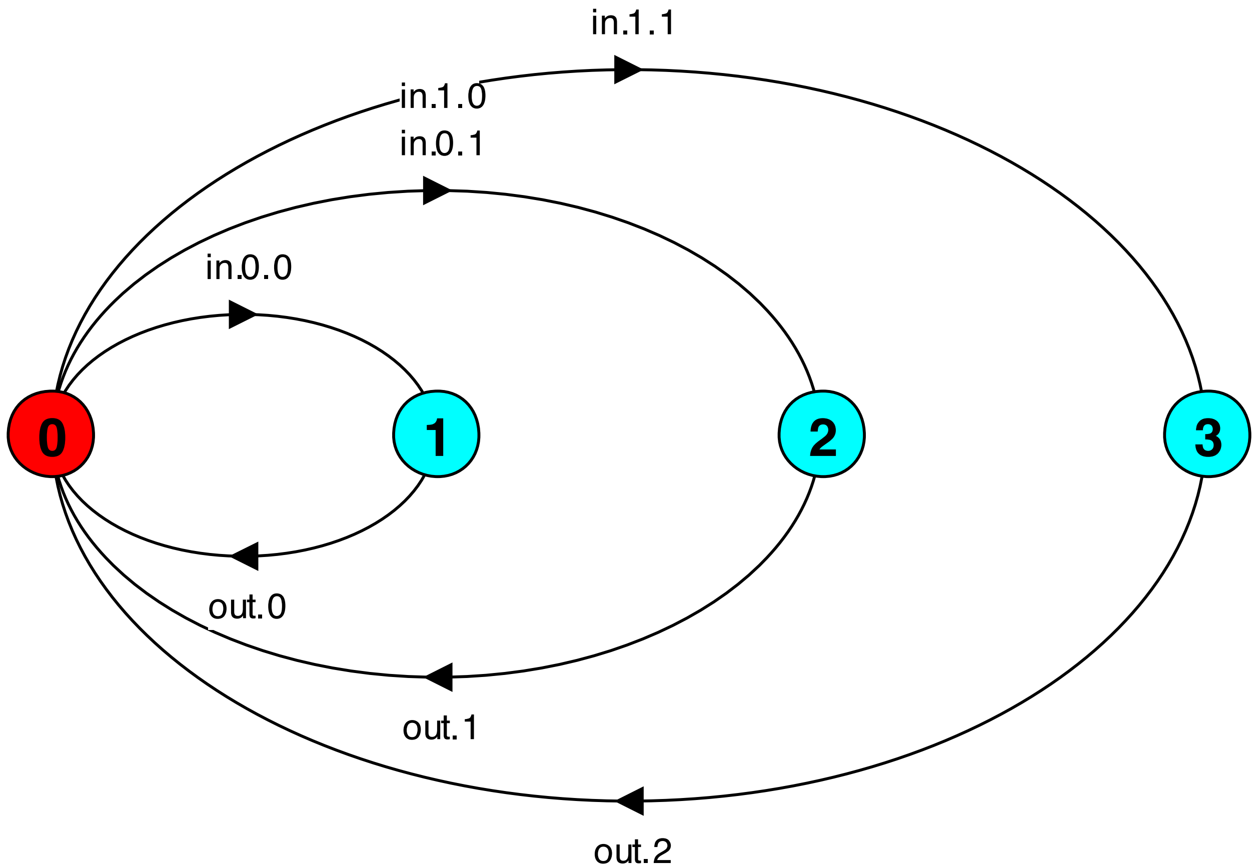
```
BUFF(N=3) = (in[i:0..N]->out[i]->BUFF).
```

FSP - indexed processes and actions

indexed actions generate LTS labels of the form

action.index

or *action[index]*



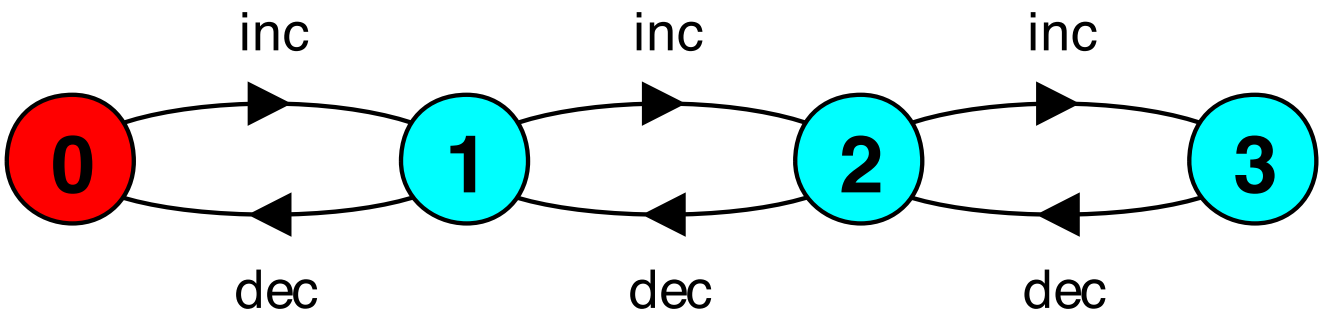
index expressions to model calculation:

```
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.

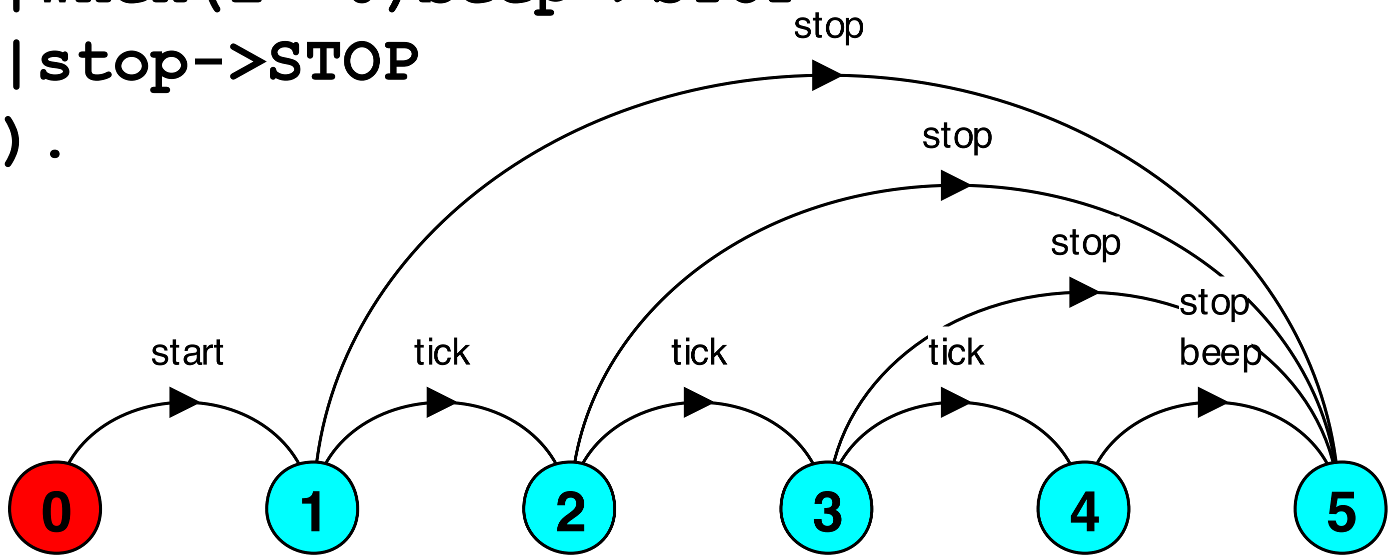
```
COUNT (N=3)    = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
               |when(i>0) dec->COUNT[i-1]
               ).
```



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) doanything->P).
```

Answer:

`STOP`

FSP - process alphabets

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

Process alphabets are implicitly defined by the actions in the process definition.

The alphabet of a process can be displayed using the LTSA alphabet window.

```
Process:
    COUNTDOWN
Alphabet:
    { beep,
      start,
      stop,
      tick
    }
```

FSP - process alphabet extension

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 to control interaction between processes)

Revision & Wake-up Exercise

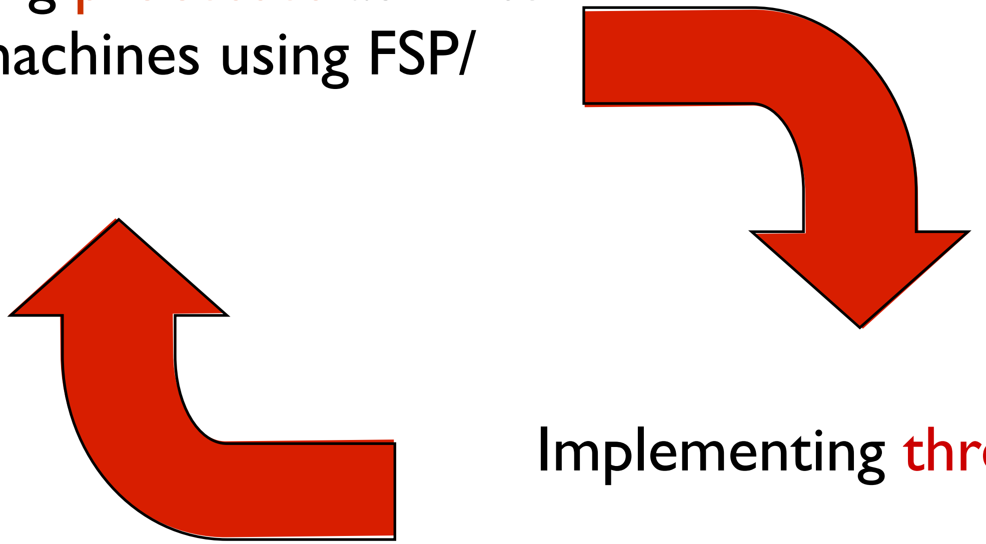
In FSP, model a process FILTER, that filters out values greater than 2 :

ie. it inputs a value v between 0 and 5, but only outputs it if v <= 2, otherwise it discards it.

```
FILTER = (in[v:0..5] -> DECIDE[v]),

DECIDE[v:0..5] = (   ?   ).
```

2.2 Implementing processes

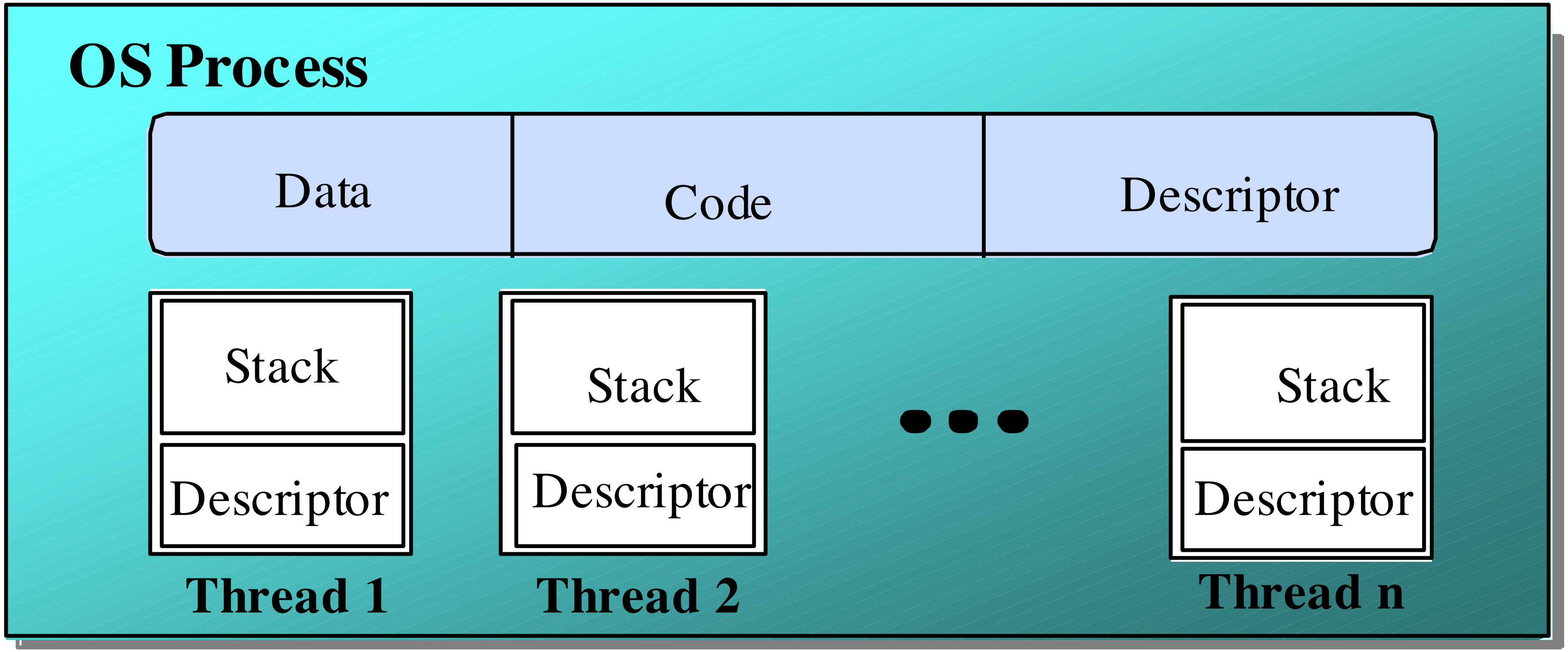
Modeling processes as finite state machines using FSP/LTS.



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.

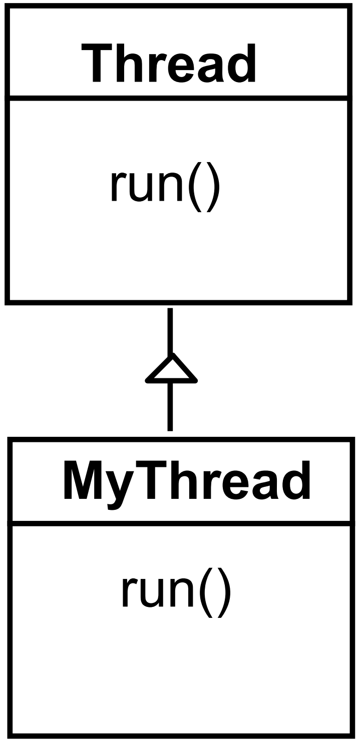
Implementing processes - the OS view



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.

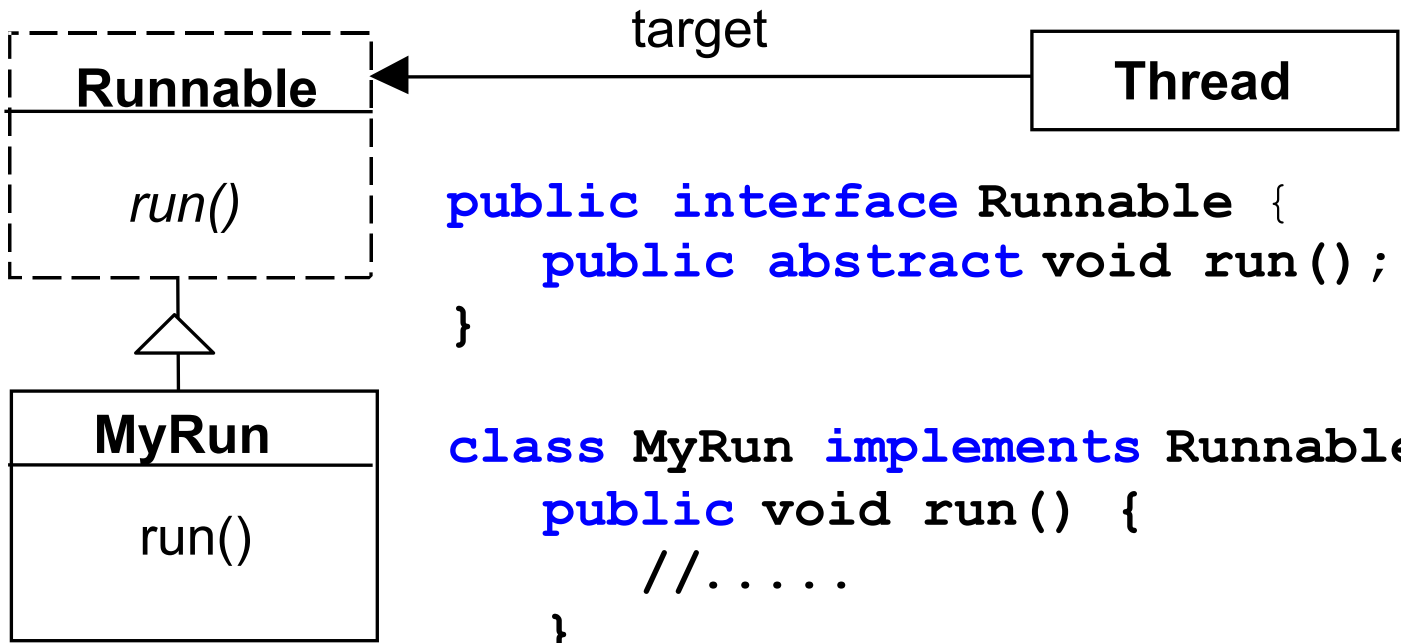
```
class MyThread extends Thread {
     public void run() {
          //......
     }
}
```

Creating and starting a thread object:

```
Thread a = new MyThread();
a.start();
```

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`**. This is more flexible and maintainable.



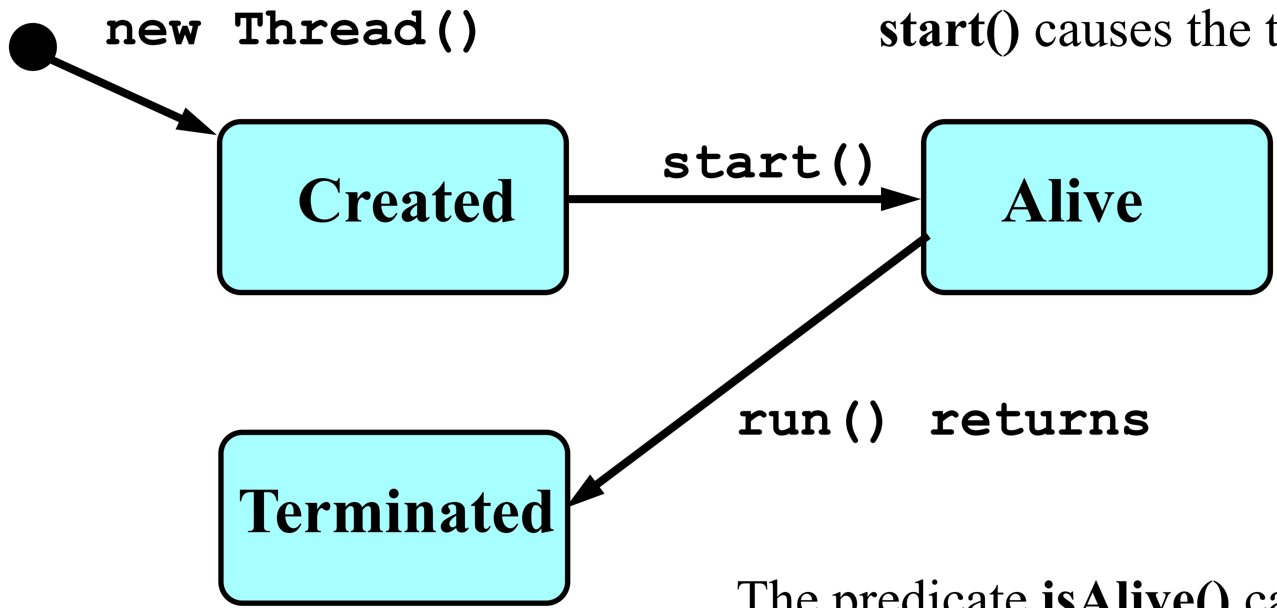

```
public interface Runnable {
    public abstract void run();
}

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

Creating and starting a thread object:

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

thread life-cycle in Java

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

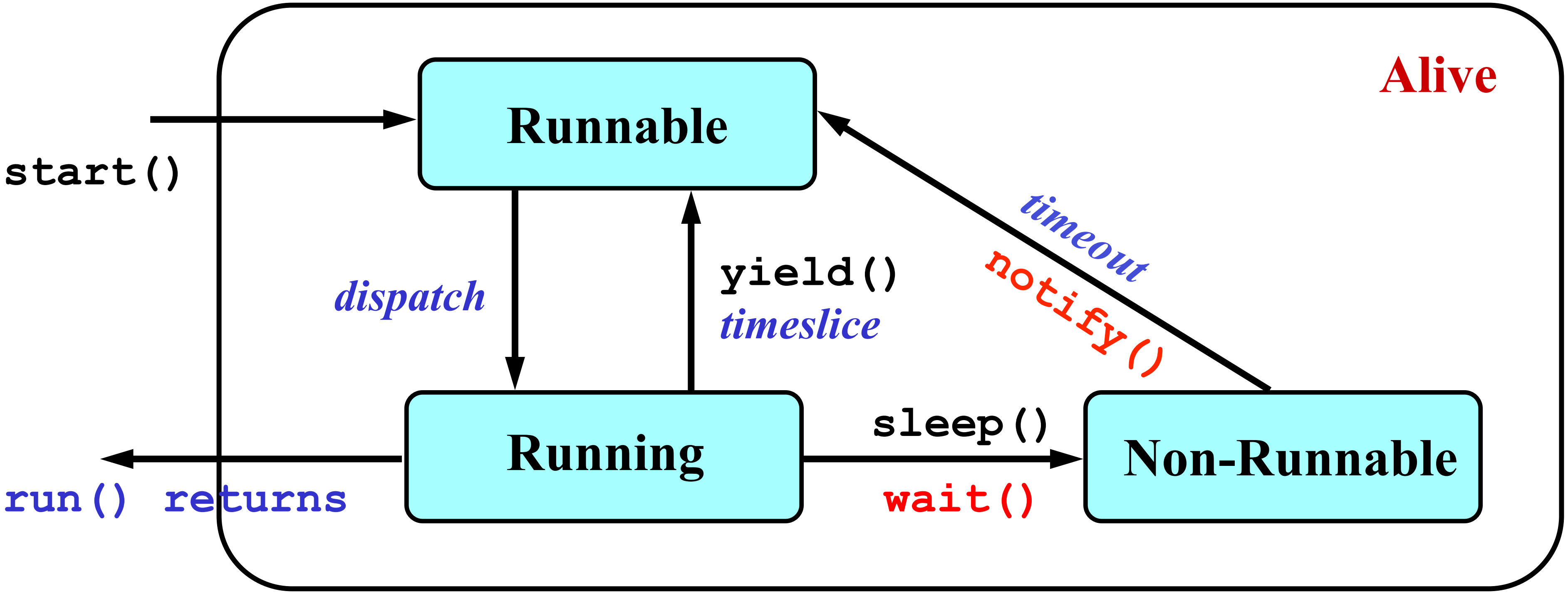


start() causes the thread to call its run() method.

The predicate **isAlive()** can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted (cf. mortals).

thread **alive** states in Java

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



wait() makes a Thread Non-Runnable (Blocked), **notify()** can, and **notifyAll()** does, make it Runnable (described in later chapters).

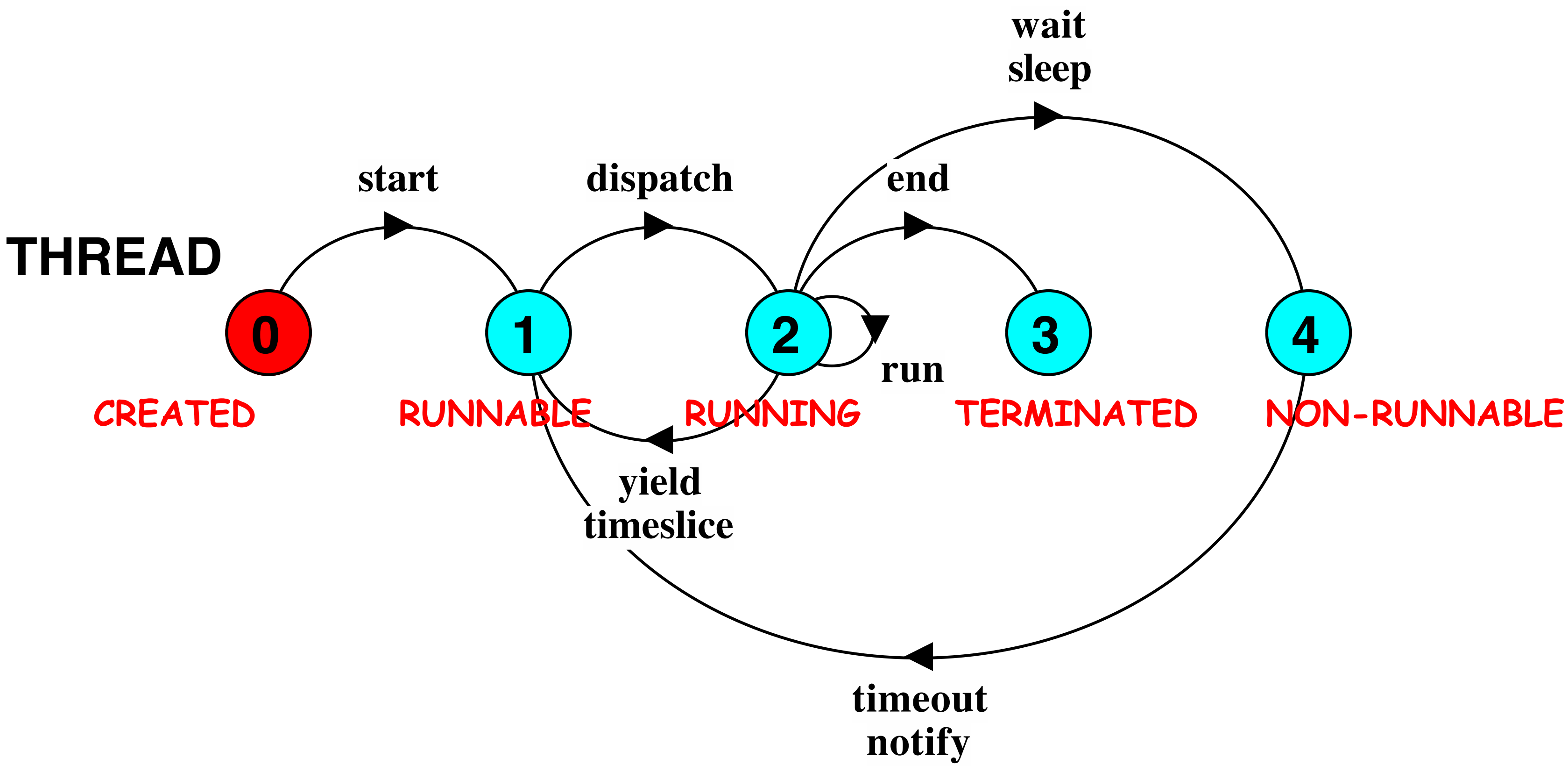
interrupt() interrupts the Thread and sets interrupt status if Running/Runnable, otherwise raises an exception (used later).

Java thread lifecycle - an FSP specification

```
THREAD       = CREATED,
CREATED      = (start           ->RUNNABLE),
RUNNABLE     = (dispatch        ->RUNNING),
RUNNING      = ({sleep,wait}    ->NON_RUNNABLE
               |{yield,timeslice}->RUNNABLE
               |end             ->TERMINATED
               |run             ->RUNNING),
NON_RUNNABLE = ({timeout,notify}->RUNNABLE),
TERMINATED   = STOP.
```

Dispatch*, *timeslice*, *end*, *run, and ***timeout*** are not methods of class Thread, but model the thread execution and scheduler .

Java thread lifecycle - an LTS specification



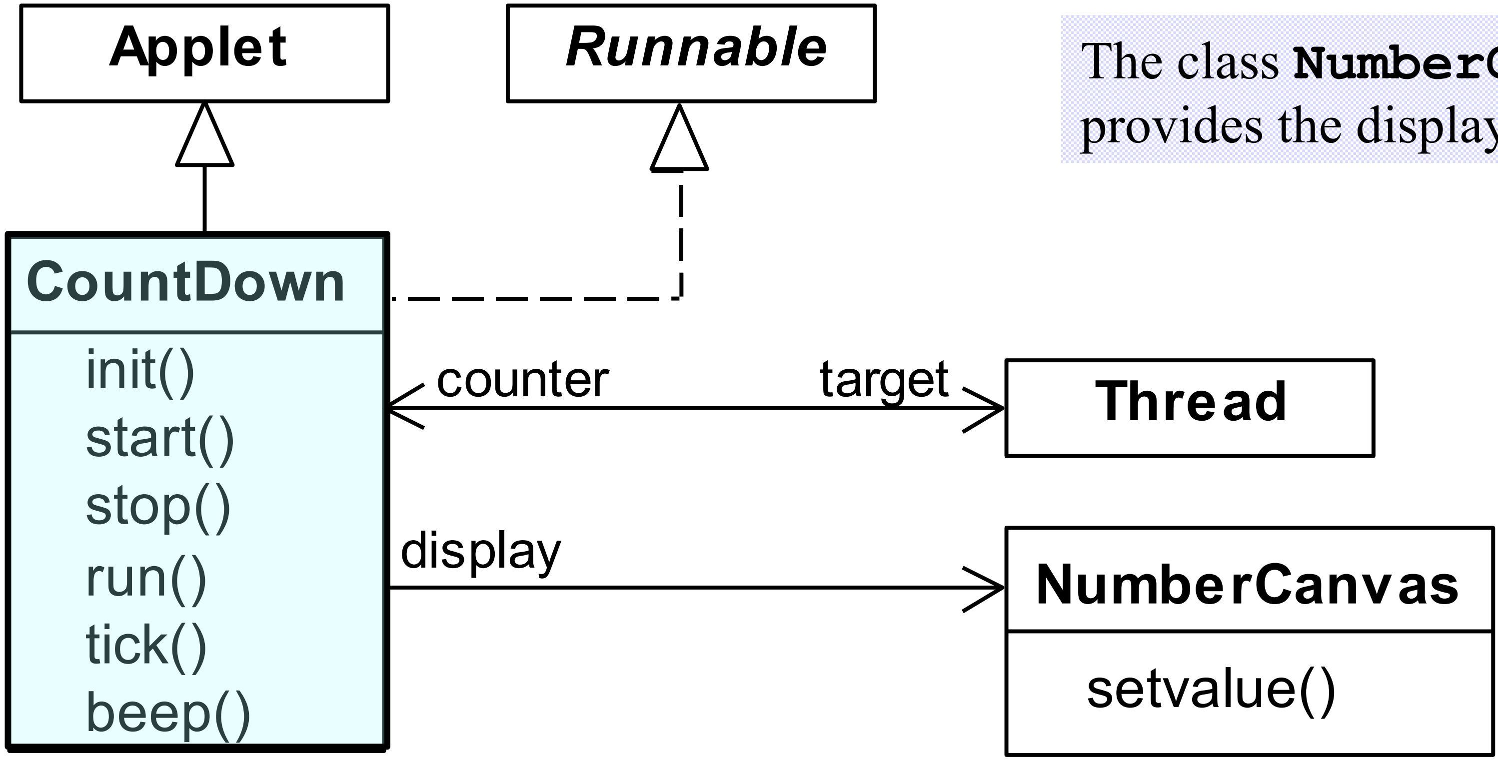
States 0 to 4 correspond to **`CREATED, RUNNABLE, RUNNING, TERMINATED`** and **`NON-RUNNABLE`** respectively.

CountDown timer example

```
COUNTDOWN (N=3)   = (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 timer - class diagram



The class **`NumberCanvas`** provides the display canvas.

The class **`CountDown`** derives from **`Applet`** and contains the implementation of the **`run()`** method which is required by **`Thread`**.

CountDown class

```
public class CountDown extends Applet
                       implements Runnable {
  Thread counter; int i;
  final static int N = 10;
  AudioClip beepSound, tickSound;
  NumberCanvas display;

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

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 (counter == null) return;
     if (i>0)  { tick(); --i; }
     if (i==0) { beep(); return;}
   }
 }
```

COUNTDOWN Model

`start ->`

`stop ->`

`COUNTDOWN[i]` process

`recursion` as a `while` loop

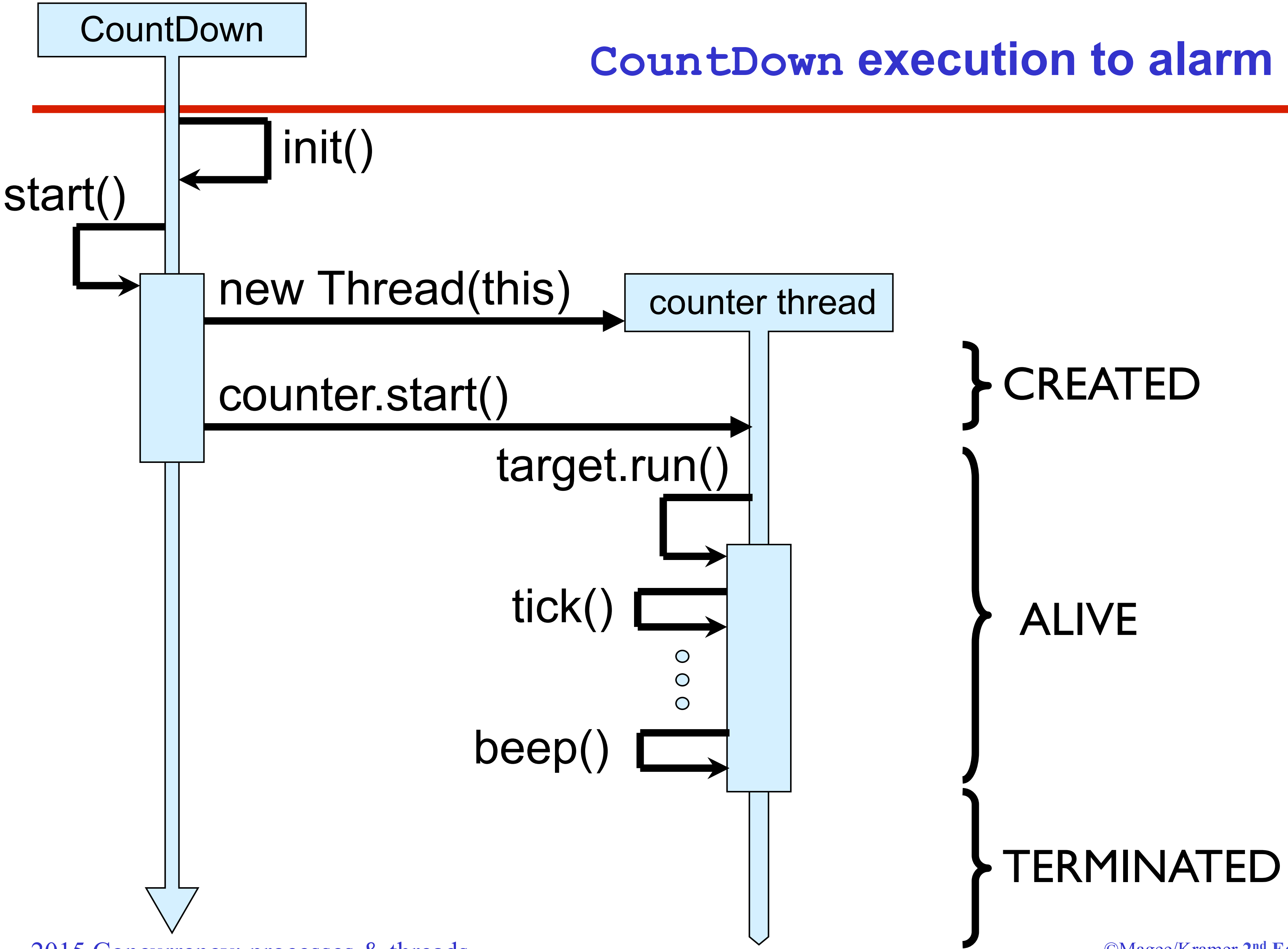
`STOP`

`when(i>0) tick -> CD[i-1]`

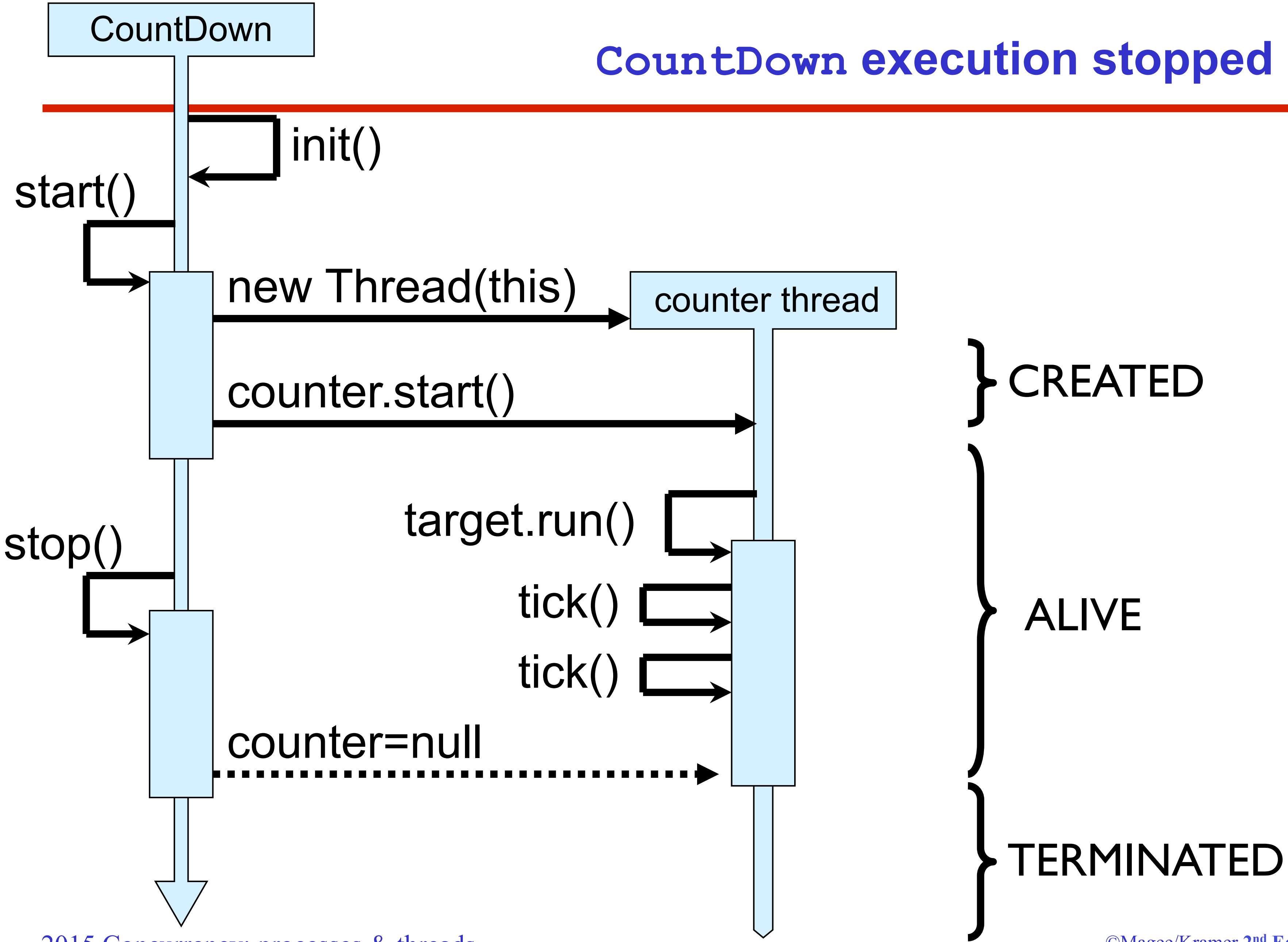
`when(i==0)beep -> STOP`

`STOP` when `run()` `returns`

CountDown execution to alarm



`CountDown` execution stopped



Summary

- Concepts
  - **process** - **unit of concurrency, execution of a program**
- Models
  - **LTS to model processes as state machines - sequences of atomic actions**
  - **FSP to specify processes using prefix "->", choice " | " and recursion.**
- Practice
  - **Java threads* to implement processes.**
  - **Thread lifecycle - created, running, runnable, non-runnable, terminated.**

* see also java.util.concurrency
* cf. POSIX pthreads in C