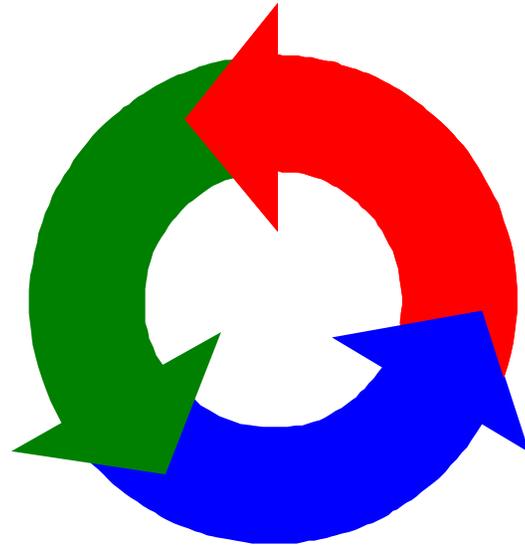


# Concurrency

---

## 2 – Processes and Threads



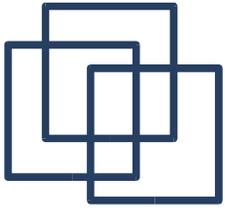
Alexandre David

*adavid@cs.aau.dk*

Credits for the slides:

Claus Brabrand

Jeff Magee & Jeff Kramer



# Concurrent Processes

---

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

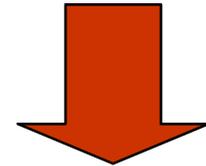
*Processes can be **concurrent***

*Designing concurrent software:*

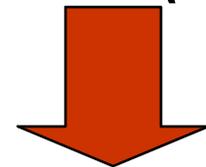
*- **complex** and **error prone***

*We need rigorous engineering approach!*

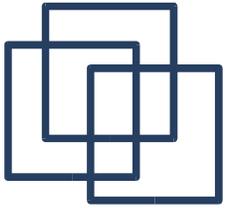
**Concept:** process ~  
sequence of actions



**Model:** process ~  
**Finite State  
Processes (FSP)**



**Practice:** process ~  
**Java thread**



# Processes and Threads

---

*Concepts: Processes - units of sequential execution*

*Models:*      **Finite State Processes (FSP)**

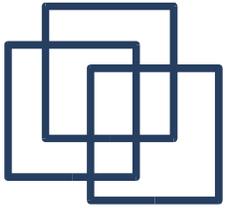
*to model processes as sequences of actions*

**Labelled Transition Systems (LTS)**

*to analyse, display, and animate behaviour*

**Abstract** *model of execution*

*Practice:*      *Java threads*



# Modeling Processes

---

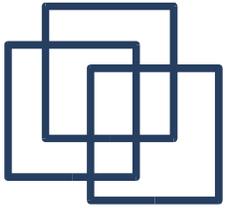
*Models are described using state machines, known as **Labelled Transition Systems (LTS)***

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

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

◆ **LTS** - graphical form

◆ **FSP** - algebraic form



# FSP - STOP

---

**STOP** is the inactive process, doing absolutely nothing.

**FSP:**

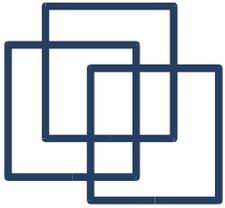
**INACTIVE = STOP.**

*INACTIVE* state machine

*(terminating process)*

**LTS:**

0



# FSP – Action Prefix

If  $\mathbf{X}$  is an action and  $\mathbf{P}$  a process then  $(\mathbf{X} \rightarrow \mathbf{P})$  describes a process that initially engages in the action  $\mathbf{X}$  and then behaves exactly as described by  $\mathbf{P}$ .

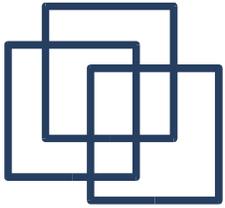
**FSP:**  $\text{ONESHOT} = (\text{once} \rightarrow \underline{\text{STOP}})$ .

**LTS:**



*Convention: actions begin with lowercase letters*

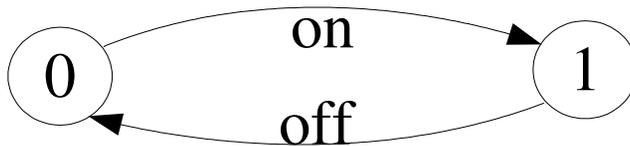
*PROCESSES begin with uppercase letters*



# Modeling Processes

---

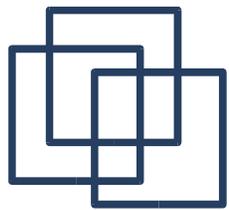
*A process is the execution of a sequential program. It is modelled as a finite state machine which transits from state to state by executing a sequence of atomic actions.*



*a light switch* **LTS**

**on->off->on->off->on->off-> ...**

*a sequence of actions or*  
**trace**



# FSP – Action Prefix and Recursion

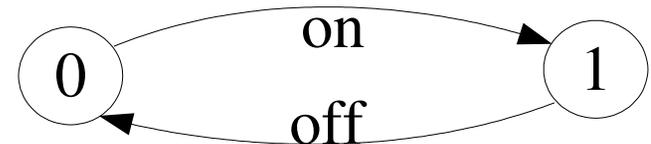
---

*Repetitive behaviour uses recursion:*

**SWITCH = OFF ,**

**OFF = (on -> ON) ,**

**ON = (off-> OFF) .**



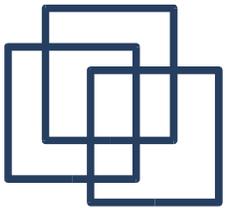
*Substituting to get a more succinct definition:*

**SWITCH = OFF ,**

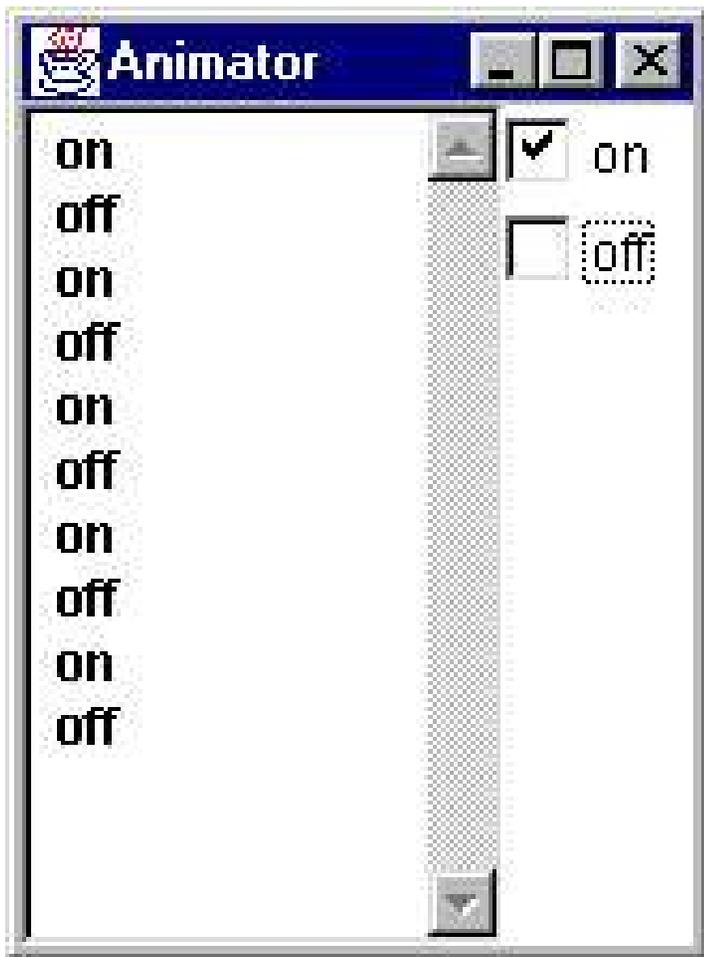
**OFF = (on -> (off->OFF)) .**

*Again?:*

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



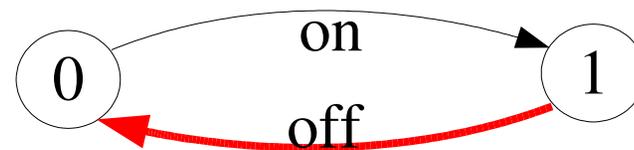
# Animation using LTSA

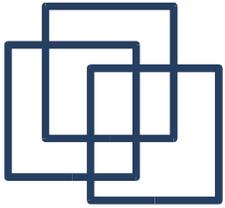


*The LTSA animator can be used to produce a trace.*

*Ticked actions are eligible for selection.*

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





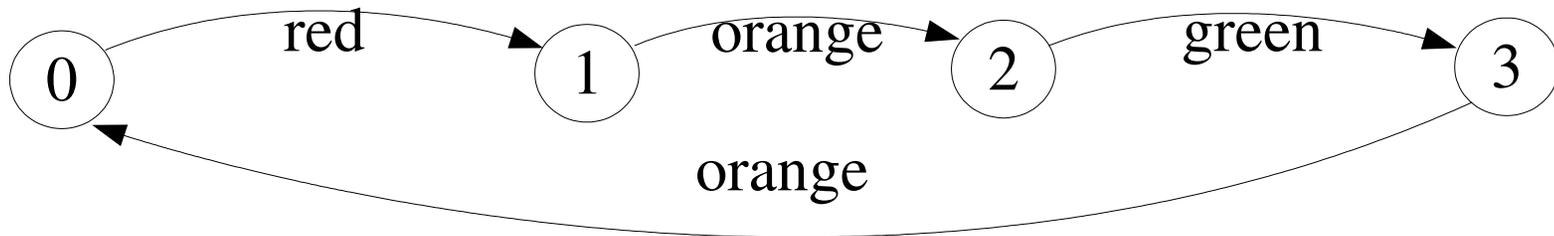
# FSP – Action Prefix

---

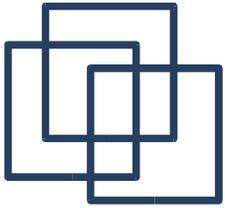
*FSP model of a traffic light:*

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

*LTS?*



*Trace(s)?*



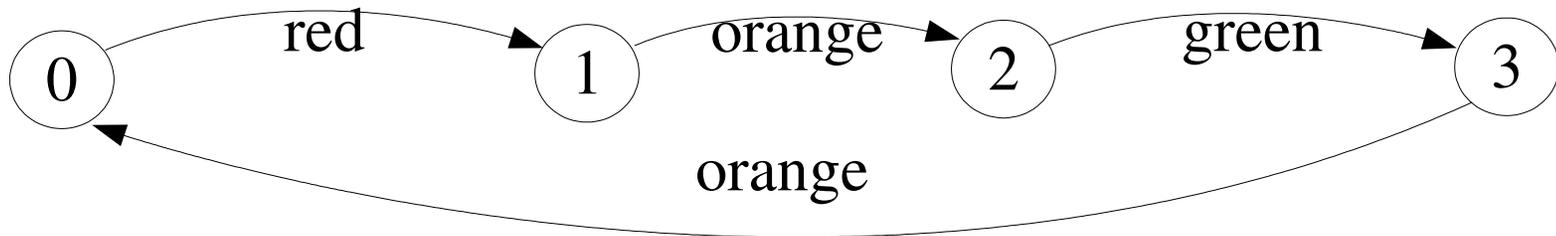
# FSP – Action Prefix

---

*FSP model of a traffic light:*

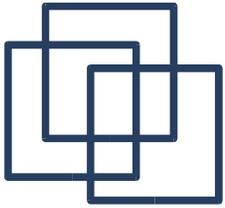
**TRAFFICLIGHT = (red->orange->green->orange  
-> TRAFFICLIGHT).**

*LTS?*



*Trace(s)?*

**red->orange->green->orange->red->orange->...**



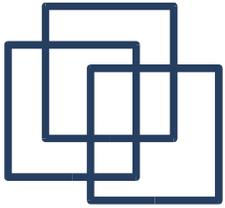
# FSP - Choice

---

*If  $\mathbf{X}$  and  $\mathbf{y}$  are actions then  $(\mathbf{x} \rightarrow \mathbf{P} \mid \mathbf{y} \rightarrow \mathbf{Q})$  describes a process which initially engages in either of the actions  $\mathbf{X}$  or  $\mathbf{y}$ . After the first action has occurred, the subsequent behaviour is described by  $\mathbf{P}$  if the first action was  $\mathbf{X}$ ; and  $\mathbf{Q}$  if the first action was  $\mathbf{y}$ .*

*Who or what makes the choice?*

*Is there a difference between input and output actions?*

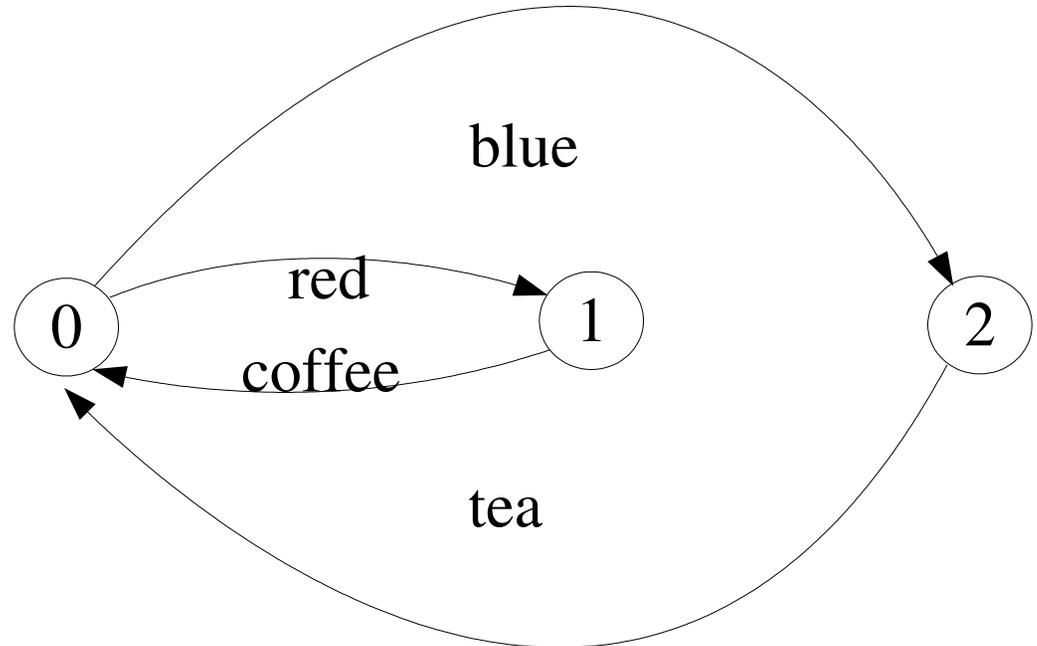


# FSP - Choice

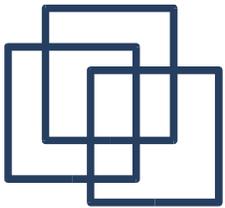
*FSP model of a drinks machine :*

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

*LTS generated using LTSA:*



*Possible traces?*



# Non-deterministic Choices

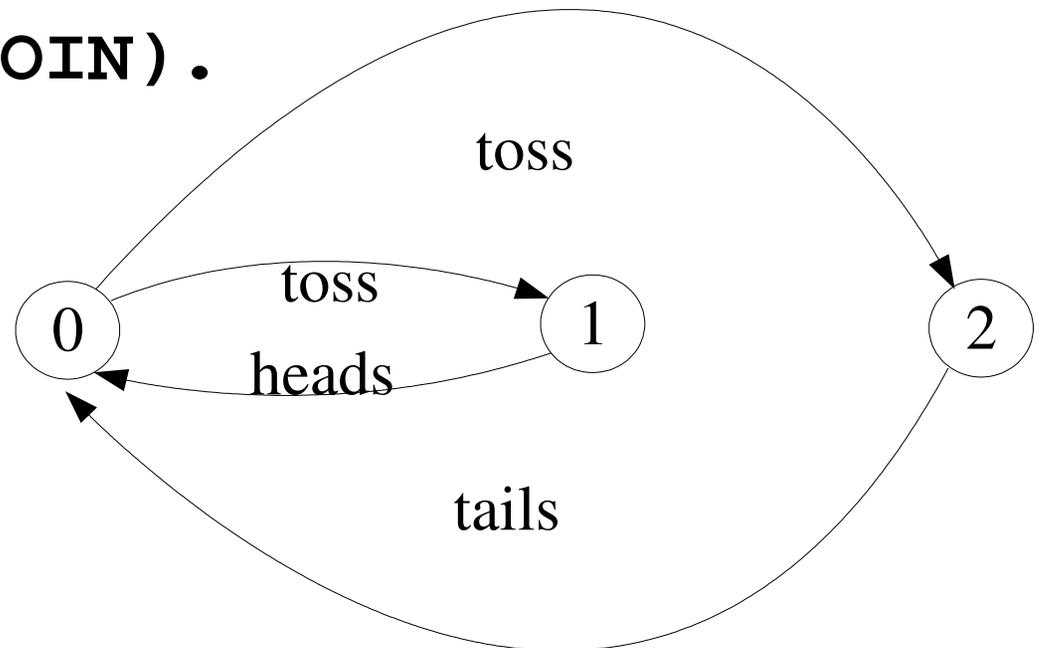
*Process  $(\chi \rightarrow P \mid \chi \rightarrow Q)$  describes a process which engages in  $\mathbf{X}$  and then behaves as either  $P$  or  $Q$ .*

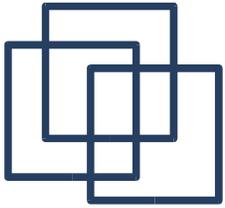
**COIN = (toss  $\rightarrow$  HEADS  $\mid$  toss  $\rightarrow$  TAILS) ,**  
**HEADS = (heads  $\rightarrow$  COIN) ,**  
**TAILS = (tails  $\rightarrow$  COIN) .**

***Tossing a coin.***

*LTS?*

*Possible traces?*



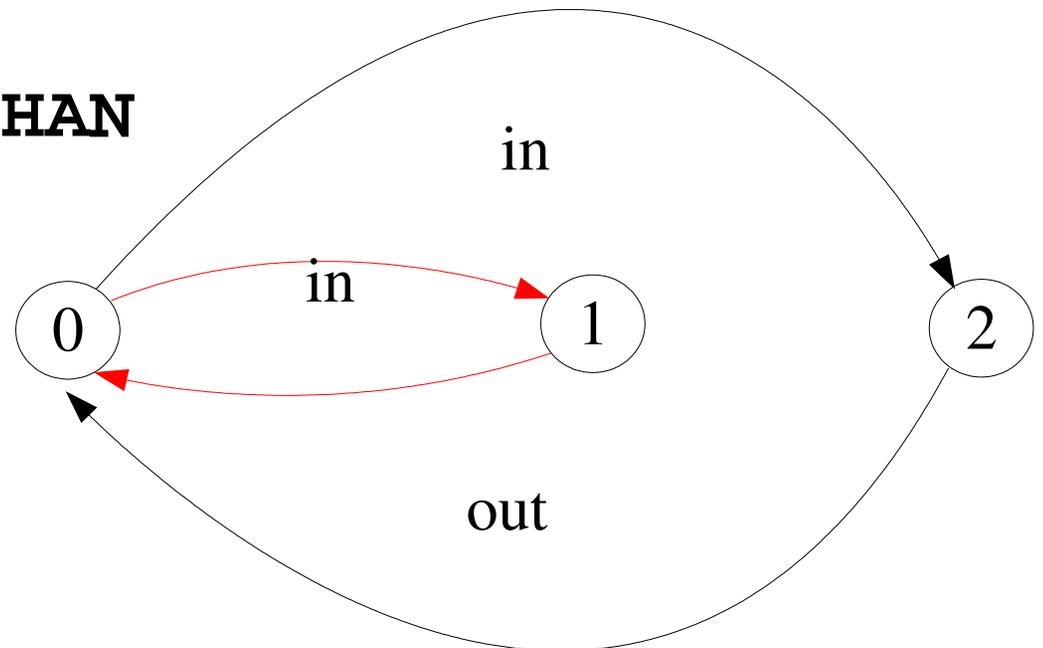


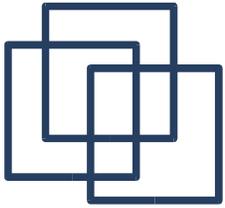
# Example

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

---

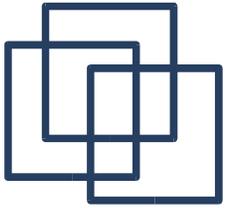
*Single slot buffer that inputs a value in the range 0 to 3 and then outputs that value:*

*equivalent to*

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

*or using a [process parameter](#) with default value:*

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



## Cont.

---

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

*equivalent to*

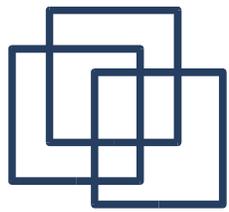
**BUFF = (in[i:0..3]->OUT[i]),**

**OUT[i:0..3] = (out[i]->BUFF).**

*equivalent to*

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

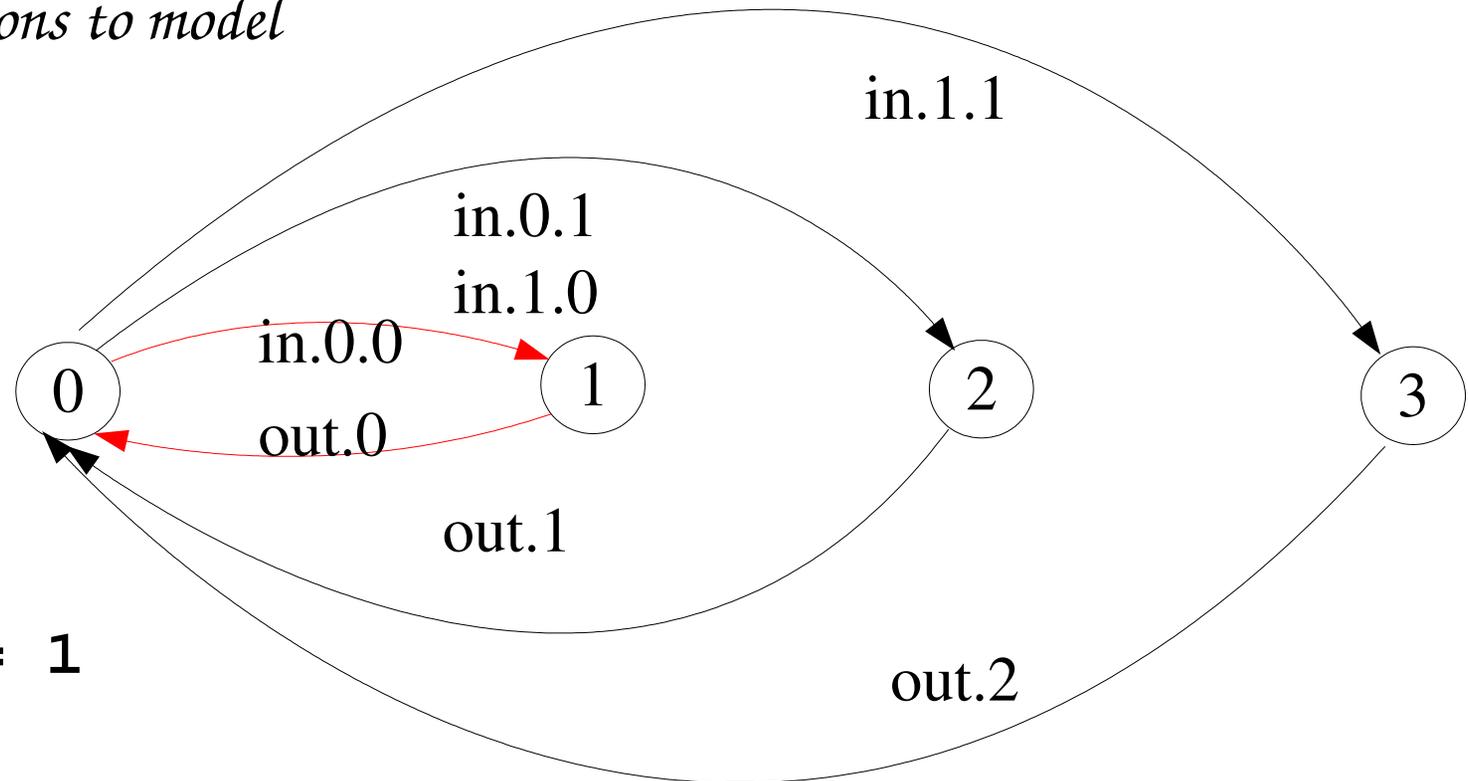
**OUT[j:0..3] = (out[j]->BUFF).**



# FSP – Constant and Addition

*index expressions to model*

*calculation:*



**const**  $N = 1$

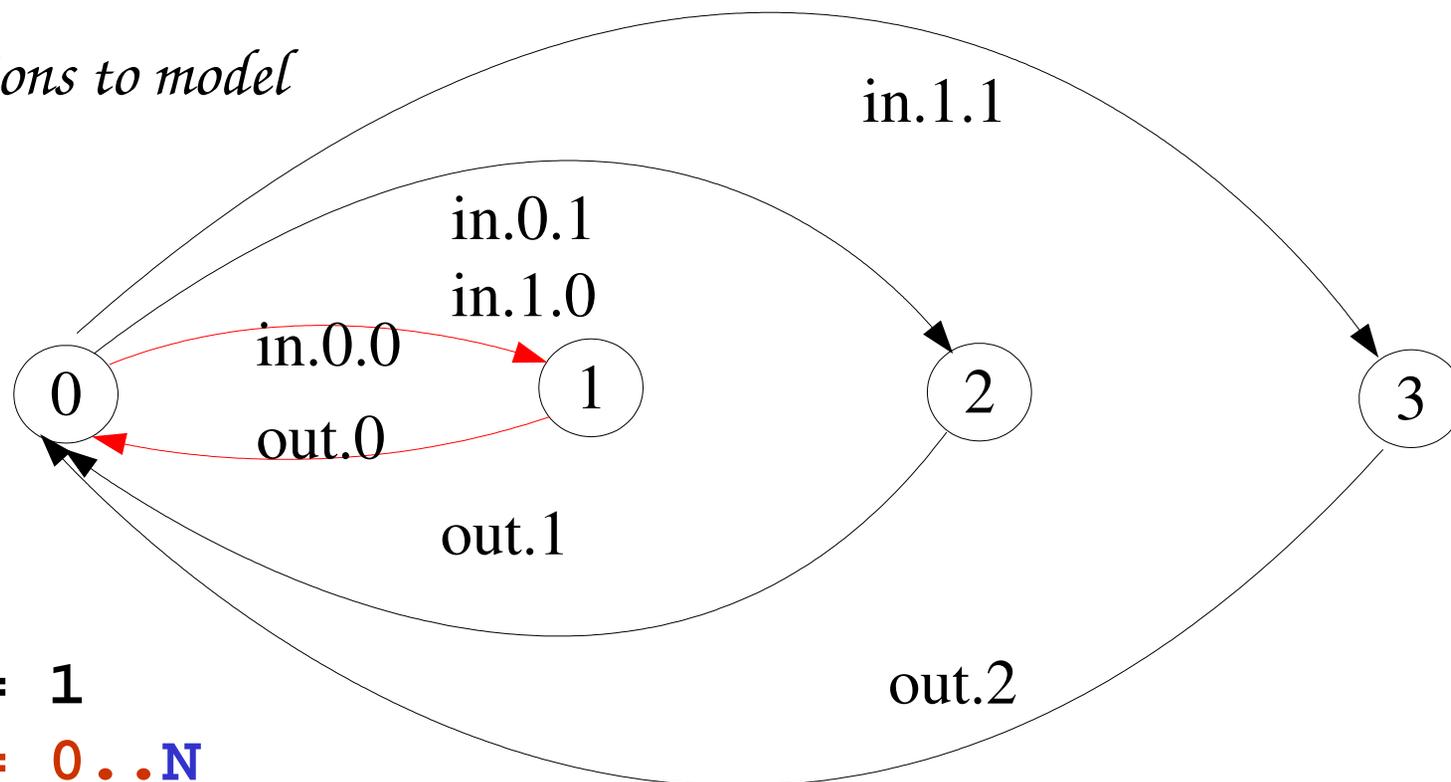
**SUM** = (in[a:0..N][b:0..N]->TOTAL[a+b]),  
**TOTAL**[s:0..2\*N] = (out[s]->SUM).



# FSP – Constant and Range Declaration

*index expressions to model*

*calculation:*



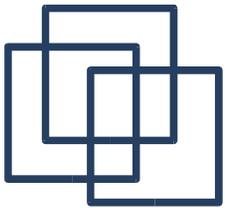
**const**  $N = 1$

**range**  $T = 0..N$

**range**  $R = 0..2*N$

**SUM** = (in[a:T][b:T]->TOTAL[a+b]),

**TOTAL[s:R]** = (out[s]->SUM).

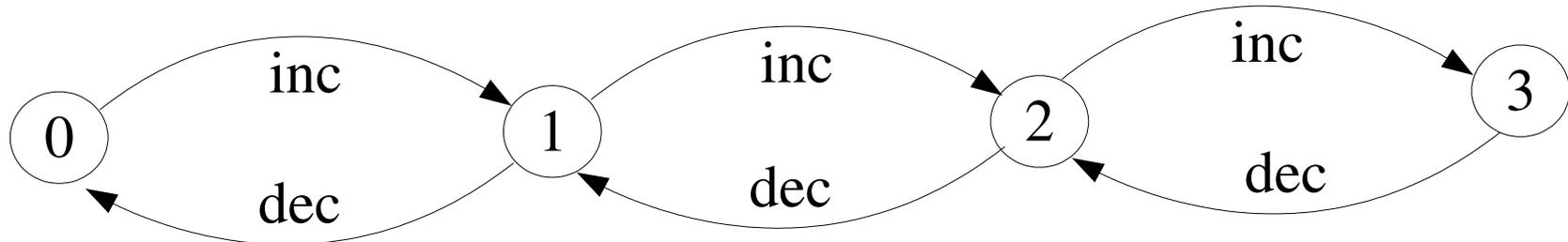


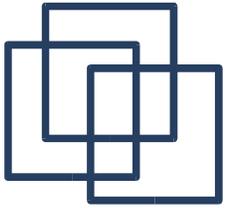
# FSP – Guarded Actions

The choice (**when**  $B \ x \rightarrow P \mid y \rightarrow Q$ ) 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?*

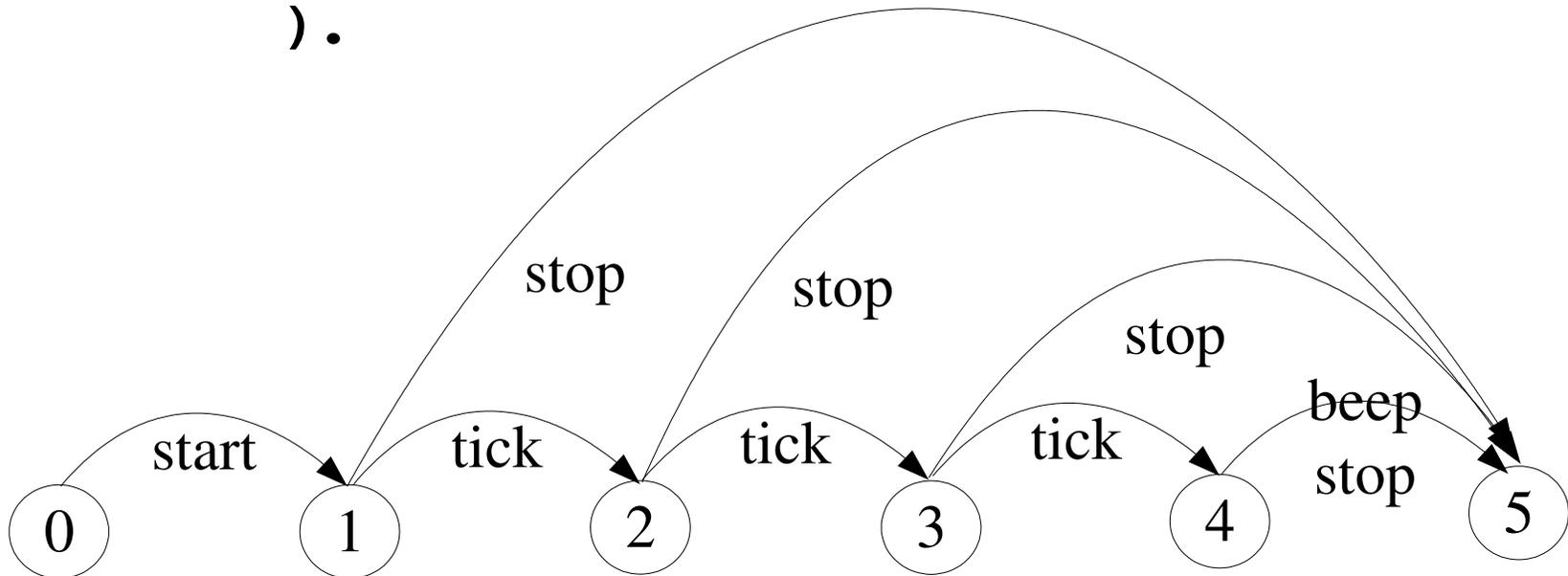


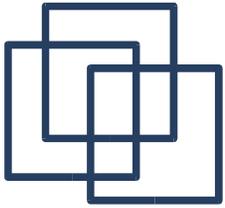


# FSP – Guarded Actions

*A countdown timer which beeps after  $\mathcal{N}$  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

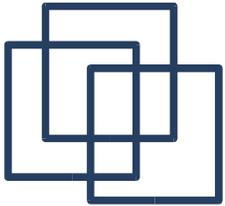
---

*What is the following FSP process equivalent to?*

**const False = 0**

**P = (when (False) doanything->P).**

*Answer:*



# FSP – Guarded Actions

---

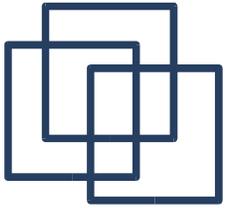
*What is the following FSP process equivalent to?*

```
const False = 0
```

```
P = (when (False) doanything->P).
```

*Answer:*

STOP



# FSP – Process Alphabets

---

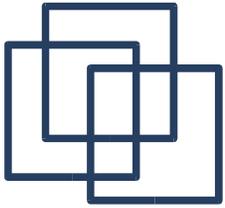
*The alphabet of a process is the set of actions in which it can engage.*

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

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

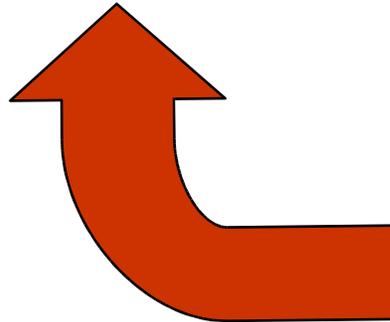
*Alphabet of **WRITER** is the set {write[0..3]}*

*(we make use of alphabet extensions in later chapters)*

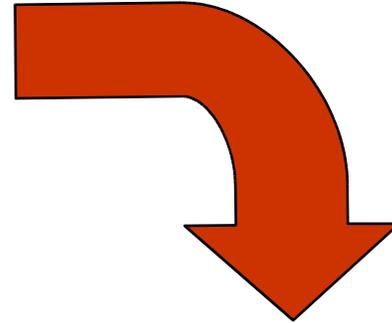


# Implementing Processes

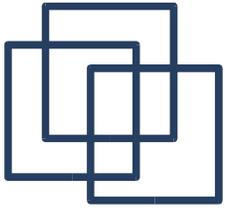
*Modelling **processes** as finite state machines using FSP/LTS.*



*Implementing **threads** in Java.*



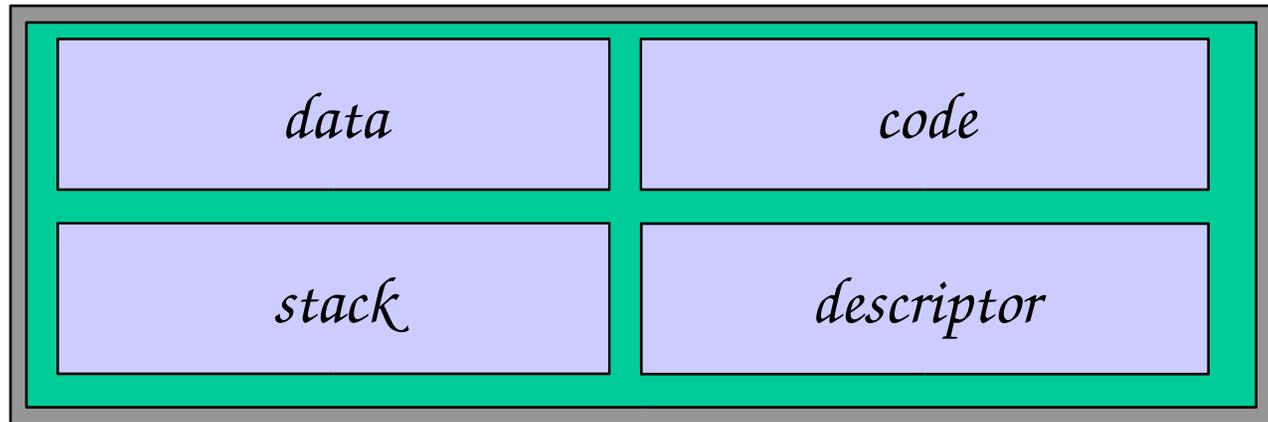
**Note:** to avoid confusion, we use the term **process** when referring to the models, and **thread** when referring to the implementation in Java.



# Process

---

## \* 1 Process:



\* Data: the heap (global, heap allocated data)

\* Code: the program (bytecode)

\* Stack: the stack (local data, call stack)

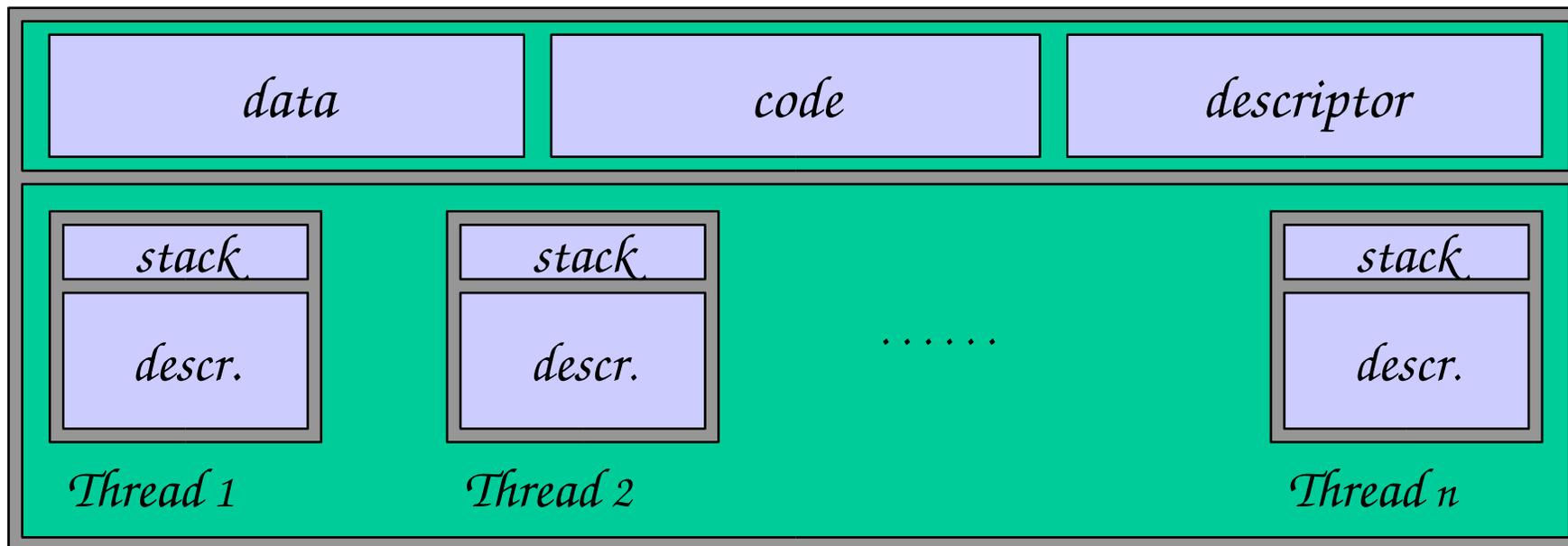
\* Descriptor: program counter, stack pointer, ...



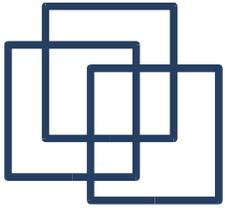
# Implementing Processes: the OS View

---

*A multi-threaded process*



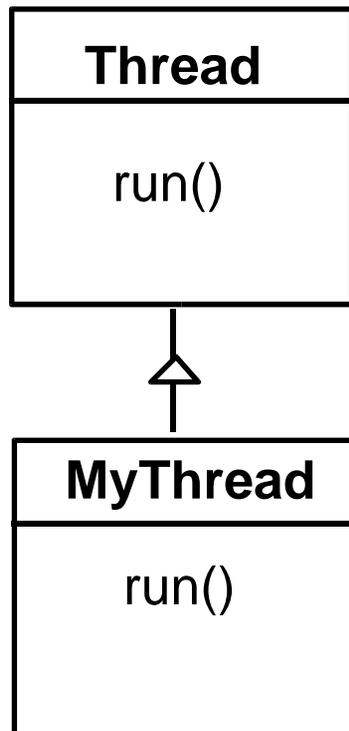
*A (heavyweight) process in an operating system is represented by its code, data and the state of the machine registers, given in a descriptor. In order to support multiple (lightweight) **threads of control**, it has multiple stacks, one for each thread.*



# Threads in Java

---

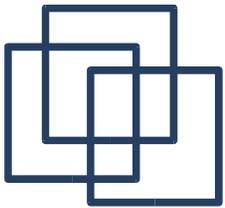
*A Thread class manages a single sequential thread of control. Threads may be created and deleted dynamically.*



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

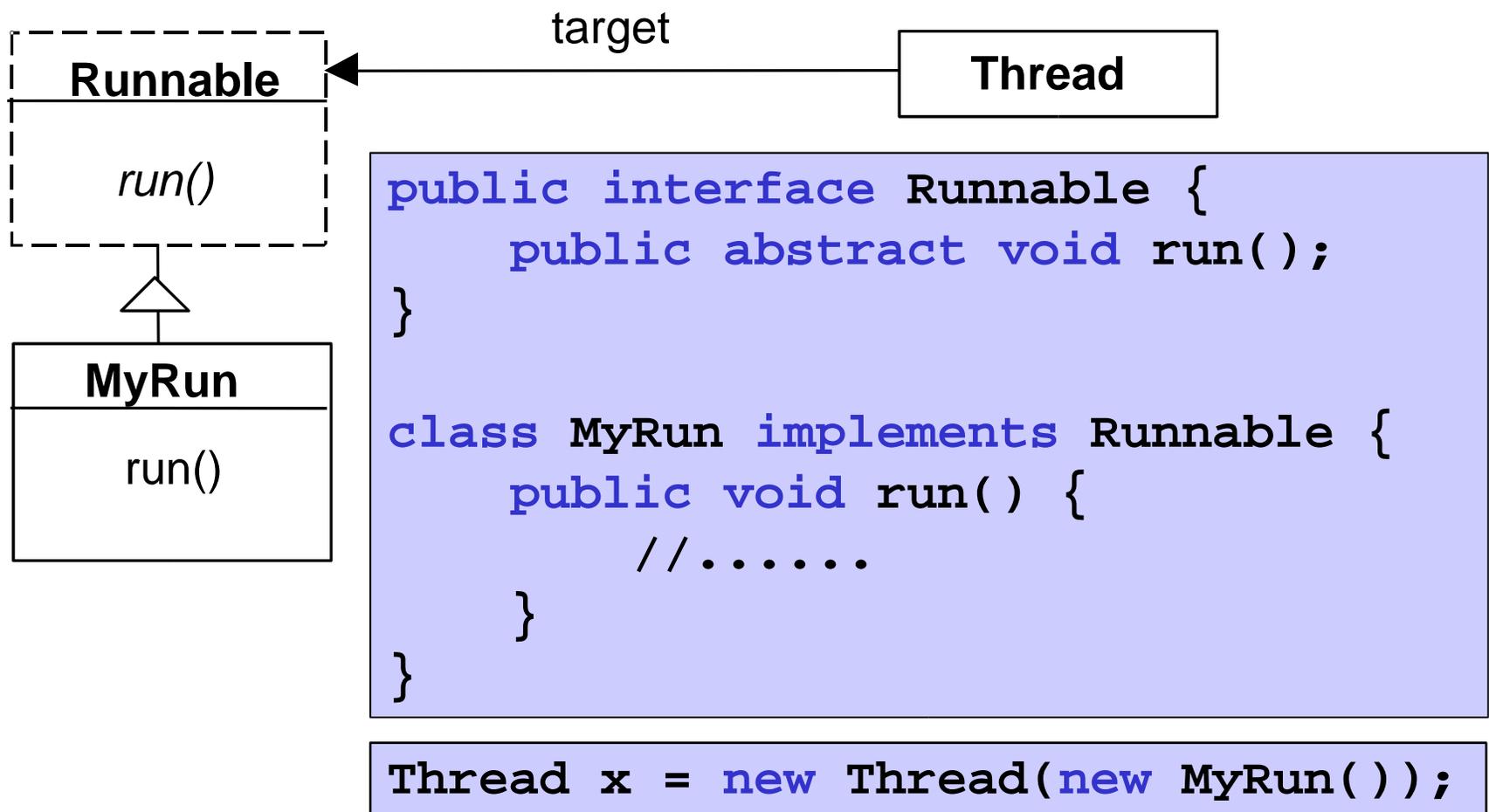
```
class MyThread extends Thread {
    public void run() {
        //.....
    }
}
```

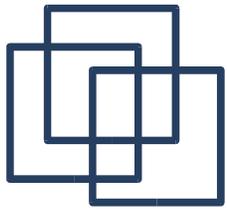
```
Thread x = new MyThread();
```



# Cont.

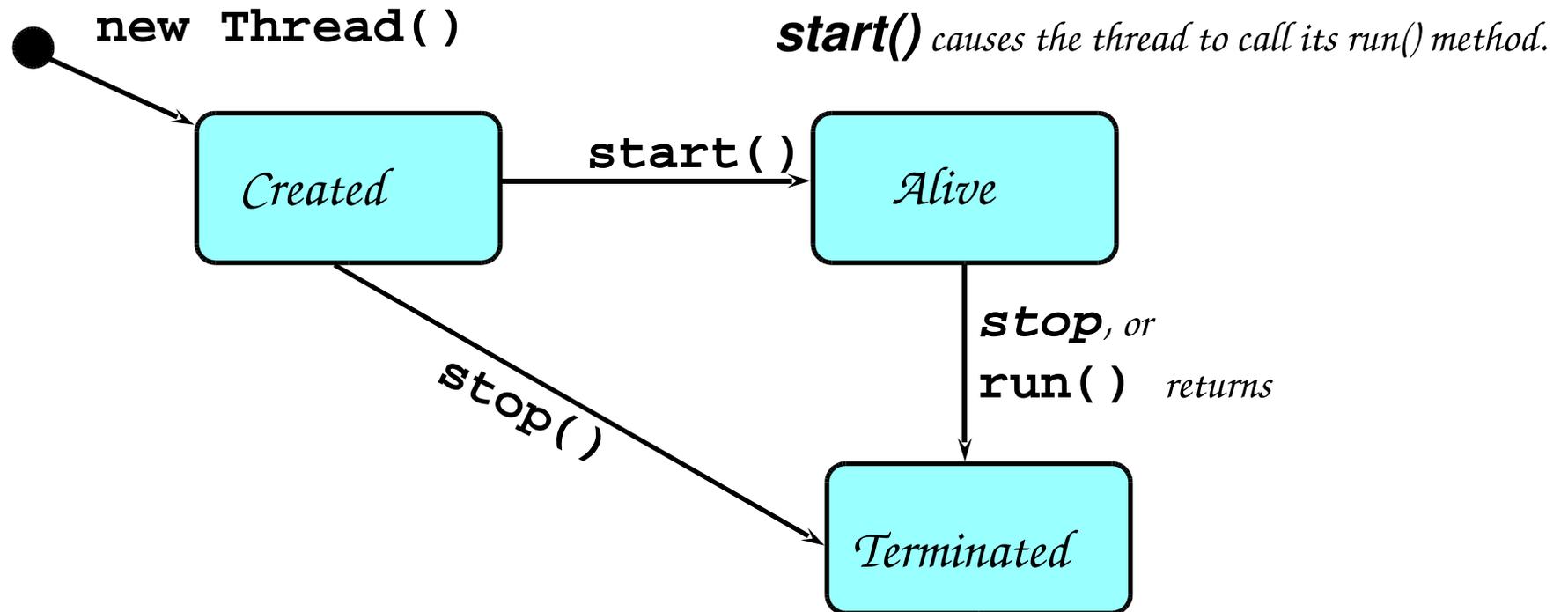
Since Java does not permit multiple inheritance, we often implement the **run()** method in a class not derived from *Thread* but from the interface *Runnable*.



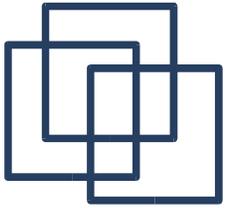


# Thread Life-cycle in Java

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

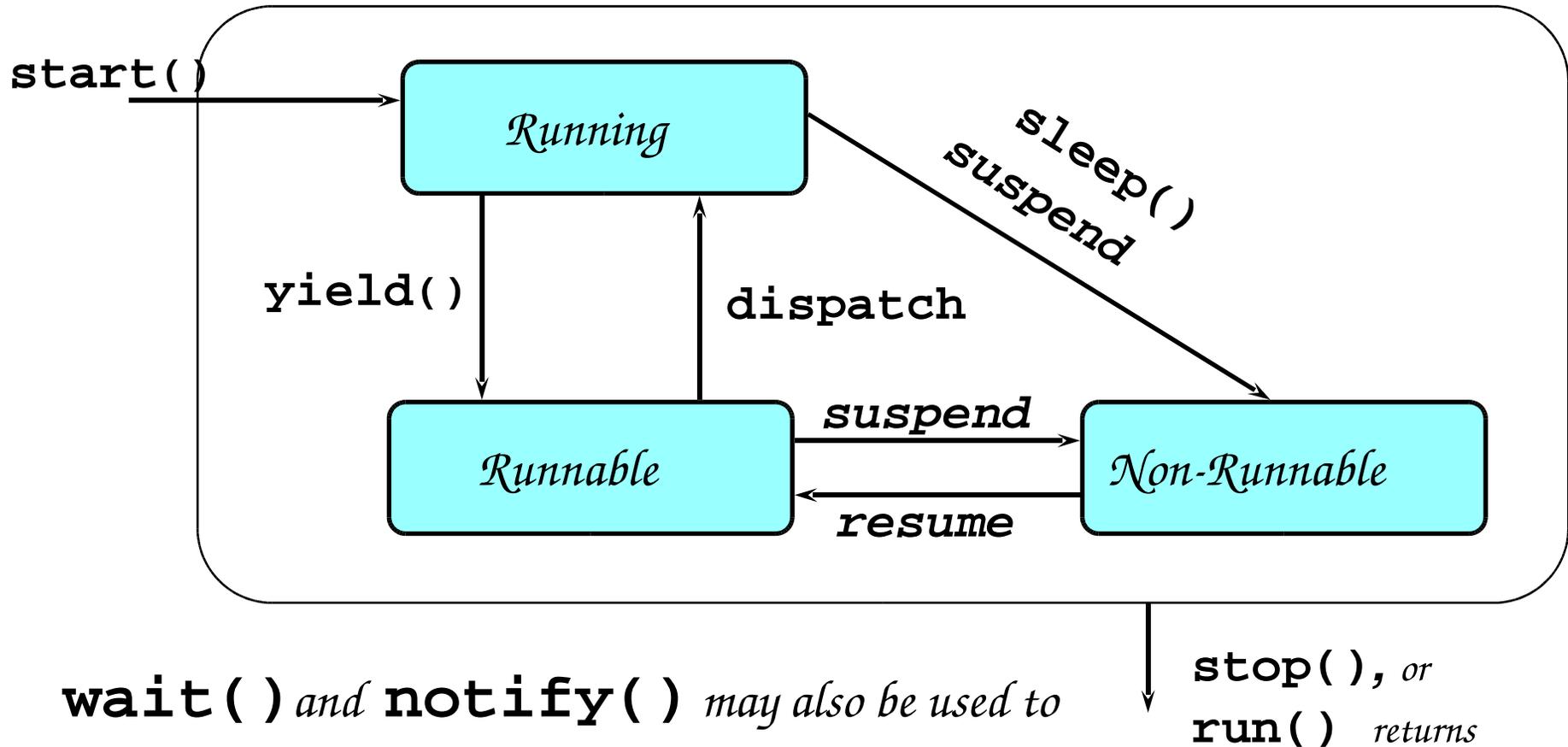


*The predicate **isAlive()** can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted (see *mortals*).*



# Cont. *Alive* States

Once started, an **alive** thread has a number of sub-states :

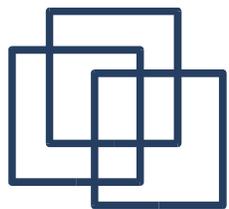


**wait()** and **notify()** may also be used to change between *Runnable* and *Non-Runnable*

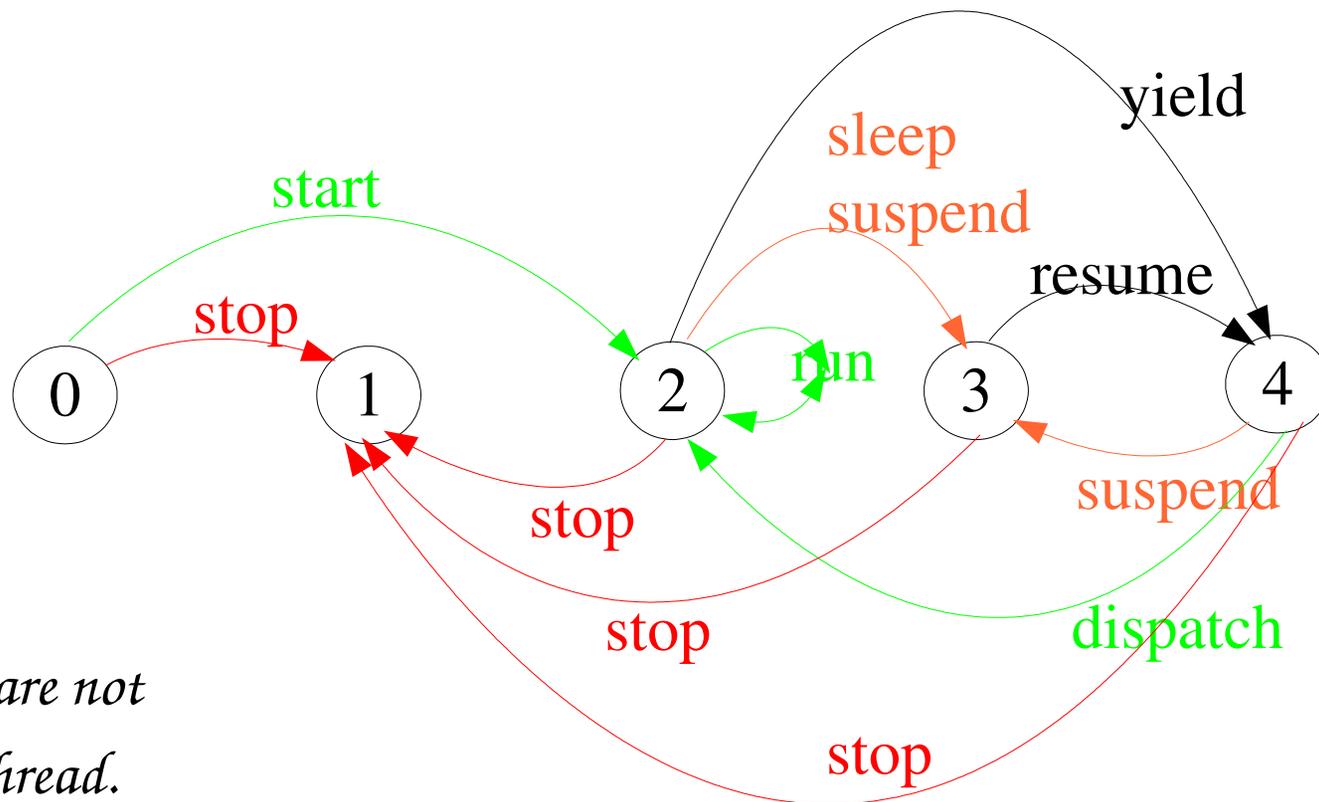


# Java Thread Life-cycle: FSP

```
THREAD          = CREATED,
CREATED         = (start          ->RUNNING
                  |stop           ->TERMINATED),
RUNNING        = ({suspend,sleep}->NON_RUNNABLE
                  |yield          ->RUNNABLE
                  |{stop,end}    ->TERMINATED
                  |run            ->RUNNING),
RUNNABLE       = (suspend        ->NON_RUNNABLE
                  |dispatch     ->RUNNING
                  |stop         ->TERMINATED),
NON_RUNNABLE   = (resume        ->RUNNABLE
                  |stop         ->TERMINATED),
TERMINATED     = STOP.
```

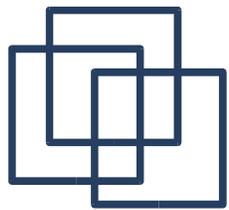


# Java Thread Life-cycle: FSP



*end, run, dispatch* are not methods of class *Thread*.

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

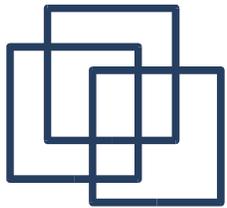


# Countdown Timer - Example

---

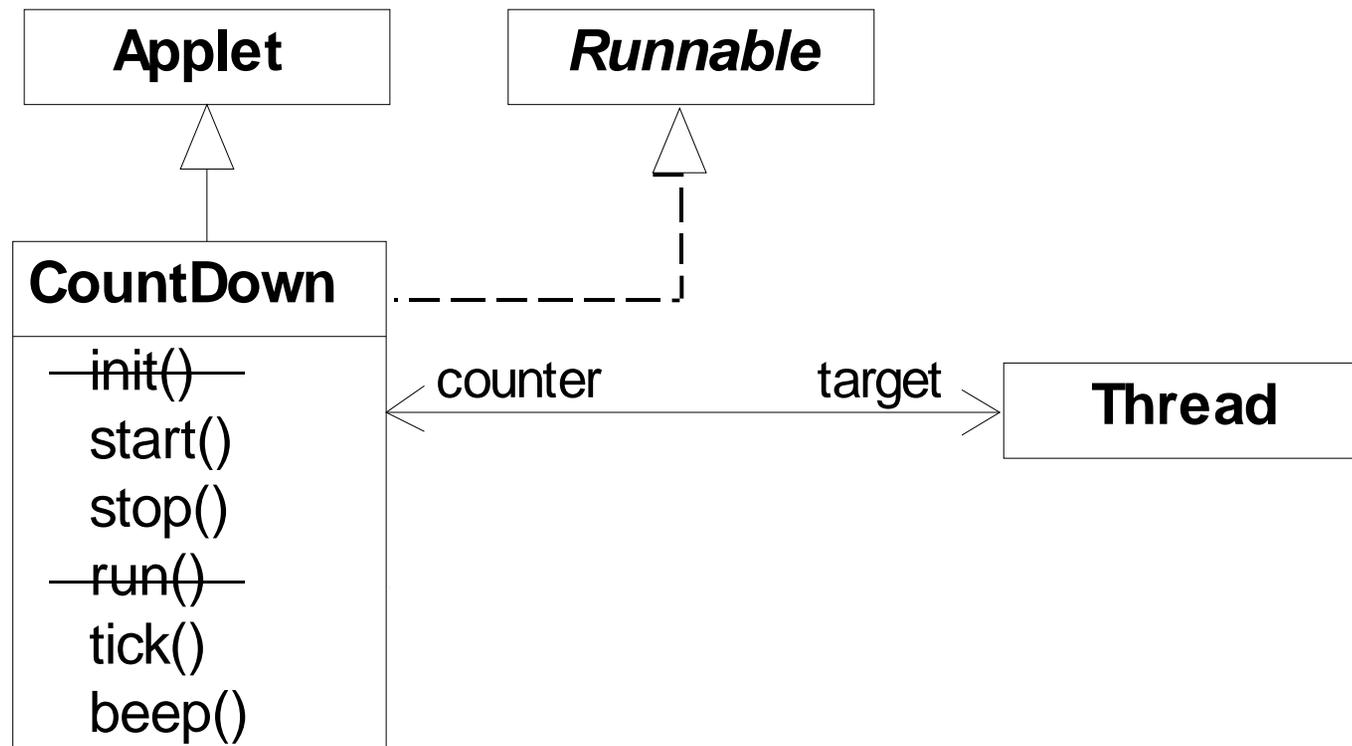
```
COUNTDOWN (N=3)      = (start->COUNTDOWN[N]),  
COUNTDOWN[i:0..N] =  
    (when(i>0) tick->COUNTDOWN[i-1]  
    | when(i==0) beep->STOP  
    | stop->STOP  
    ).
```

*Implementation in Java?*

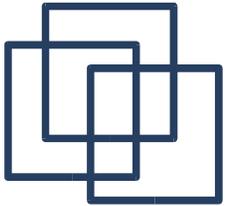


# Countdown Timer – Class Diagram

---



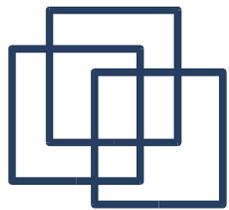
The class **CountDown** derives from **Applet** and contains the implementation of the `run()` method which is required by **Thread**.



# Countdown Timer - Class

```
public class Countdown extends Applet
    implements Runnable {
    Thread counter;
    int i;
    final static int N = 10;

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



# Class/Model of 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;}  
    }  
}
```

**start** -> CD[N]

**stop** -> STOP

**COUNTDOWN[i]** *process*

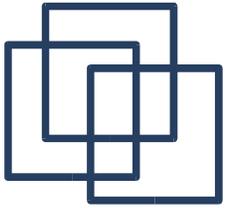
**recursion** *as a while loop*

**STOP**

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

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

**STOP** *when run() returns*



# 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 life-cycle (created, running, runnable, non-runnable, terminated)*