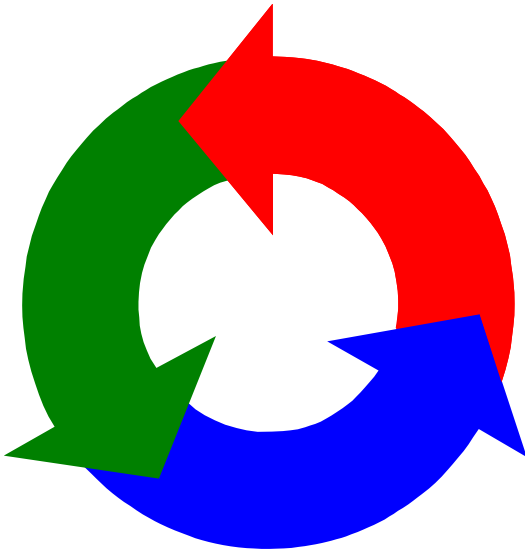
Chapter 2

Processes & Threads



Concurrent Processes

- Complex systems are structured 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
- 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:

- use Finite State Processes (FSP) to code processes as sequences of *actions*.
- use Labelled Transition System (LTS) to model processes
- use Labelled Transition System Analyser (LTSA) to analyze, display, animate behaviour.

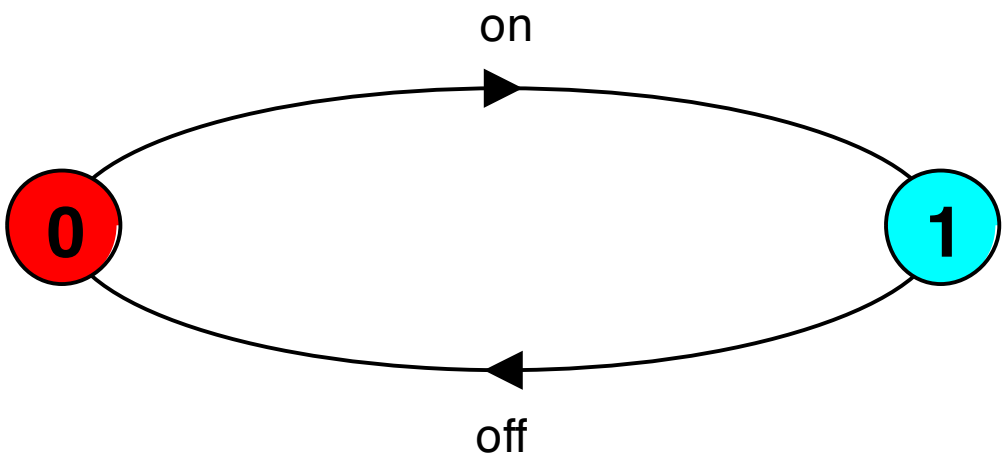
Practice: Java threads

2.1 Modelling Processes

- Models are described using state machines, known as Labelled Transition System (**LTS**):
  - **coded** as finite state process (**FSP**) in algebraic form
  - **displayed** and **analysed** by the Labelled Transition System Analyser (***LTSA***) analysis tool in graphical form

Modeling Processes

- **A Process:**
  - is the execution of a sequential program.
  - is modelled as a finite state machine which transits from state to state by executing a sequence of atomic **actions—on, off**.



a light switch LTS

`on→off→on→off→on→off→ .....`

a sequence of actions or *trace*

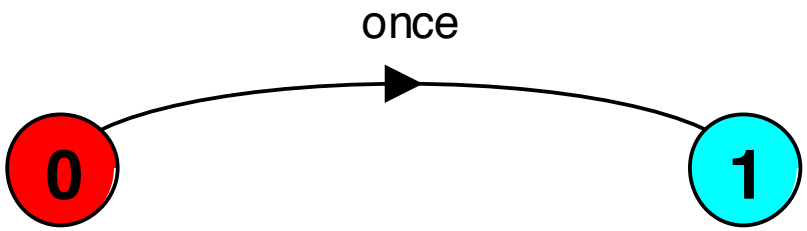
Finite State Process (FSP) - Action prefix

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).
```

FSP for ONESHOT



LTS for ONESHOT

(terminating process)

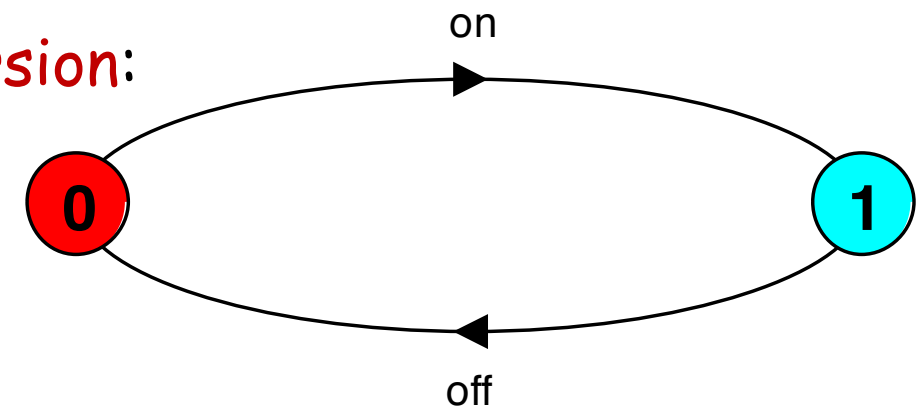
Convention: actions begin with lowercase letters

PROCESSES begin with uppercase letters

Finite State Process (FSP) - Action prefix & recursion

Repetitive behaviour uses recursion:

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



Substitute ON to get a more brief definition:

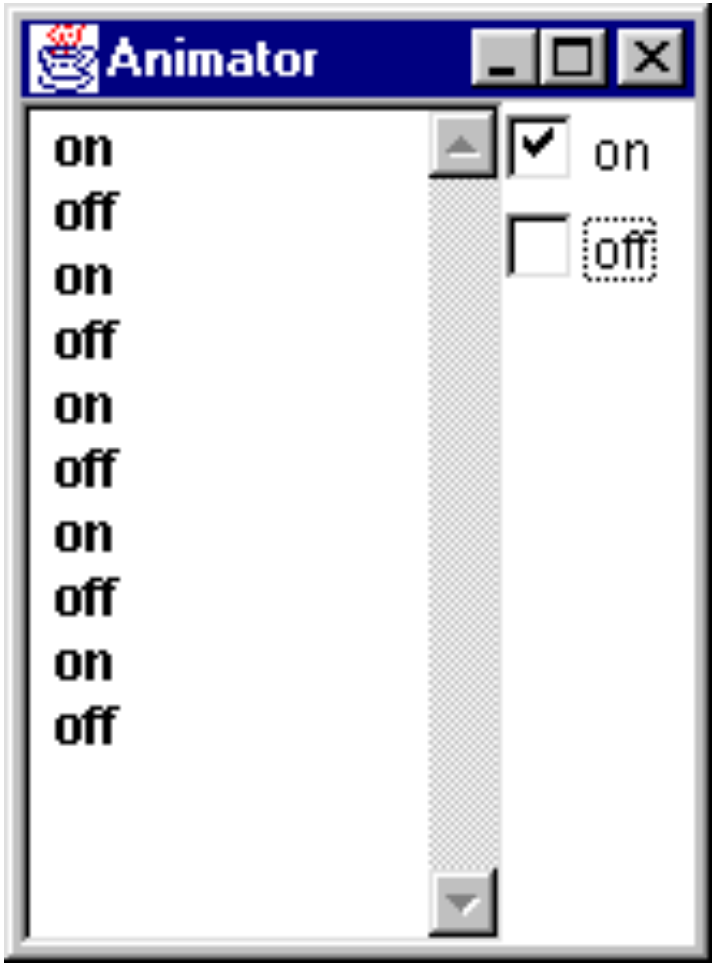
```
SWITCH = OFF,
OFF    = (on -> (off -> OFF) ).
```

And again substitute OFF :

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

Note: ON and OFF are local processes representing state(1) and state(0), respectively.

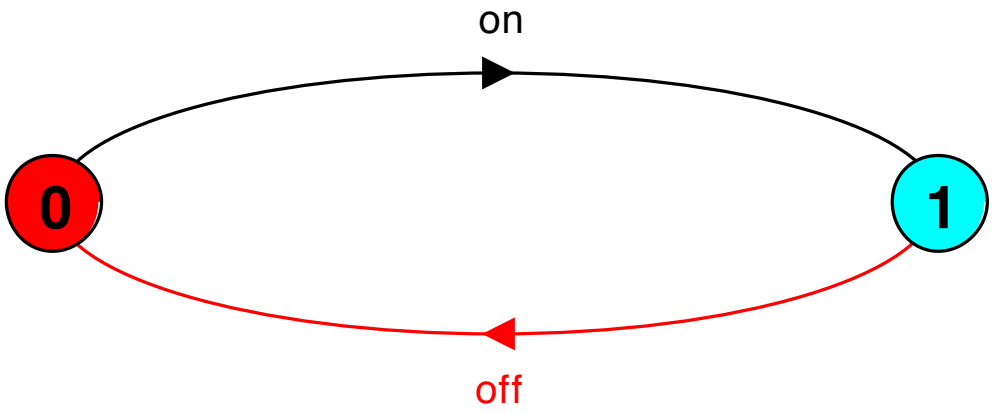
Labelled Transition System Analyser (LTSA)



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

Ticked actions in LTSA are eligible for selection.

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

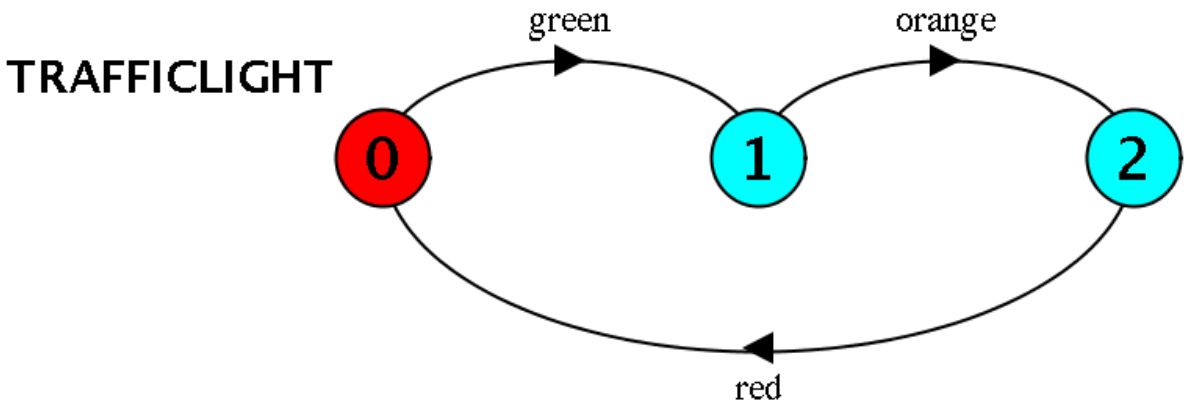


Finite State Process (FSP) – Action prefix & recursion

FSP model/coding of a traffic light :

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

LTS generated using *LTSA*:



Trace:

green→orange→red→green→orange→red ...

Finite State Process (FSP) - Choice

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

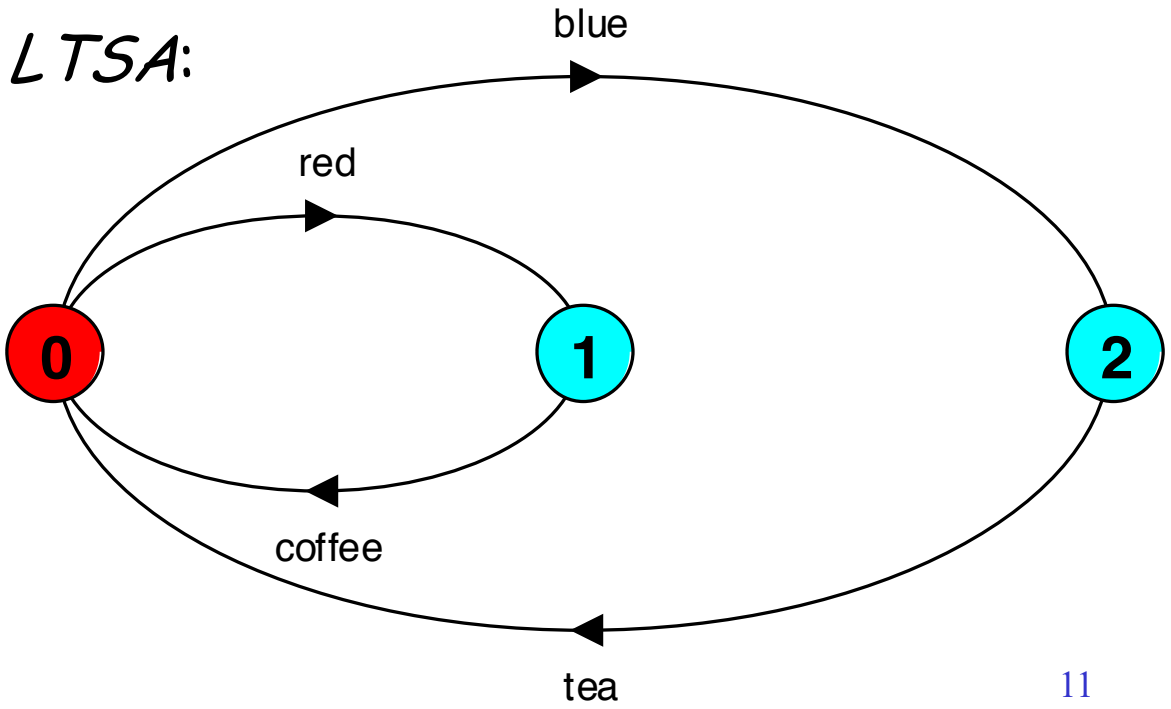
Who or what makes the choice?

Finite State Process (FSP) - Choice

FSP model of a drinks machine :

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

LTS generated using *LTSA*:

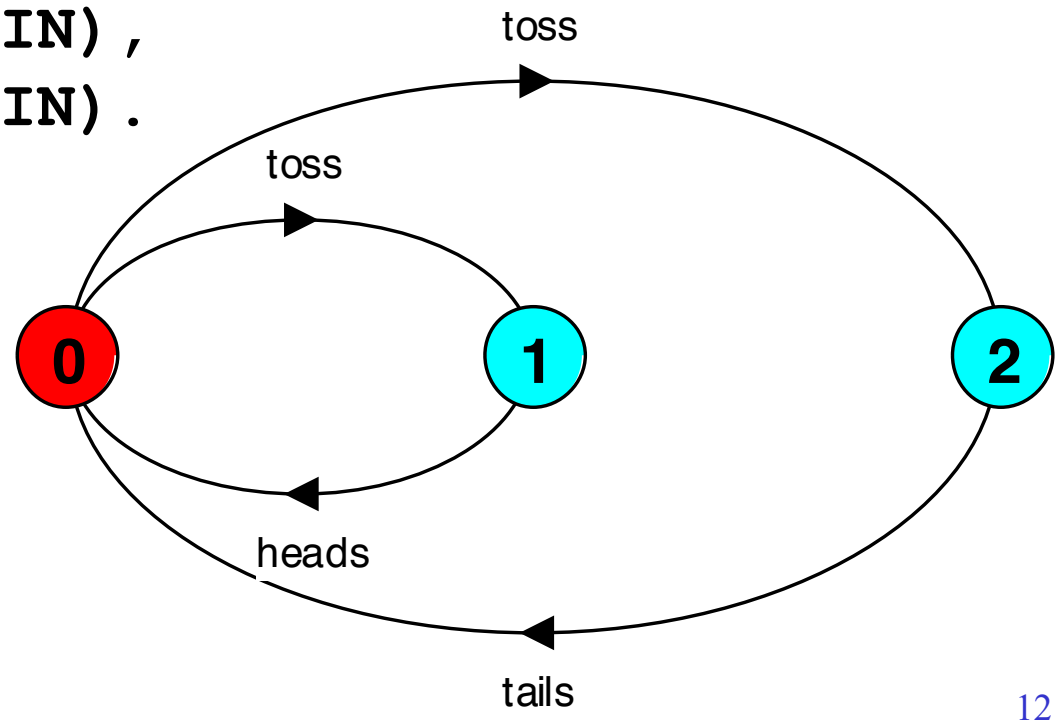


Possible traces?

Non-Deterministic choice

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

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



Tossing a coin.

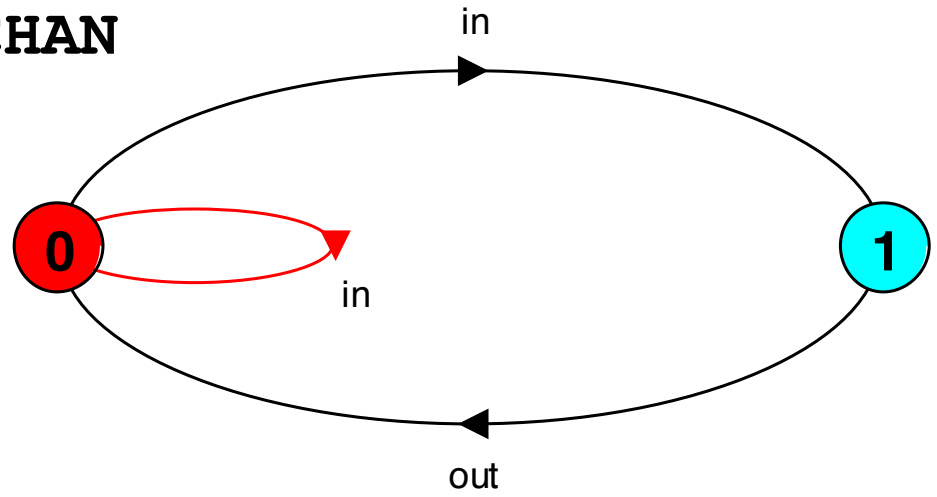
Possible traces?

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
        ).
```



Finite State Process – Indexed Processes and Actions

Single slot buffer that takes an input 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
       ).
```

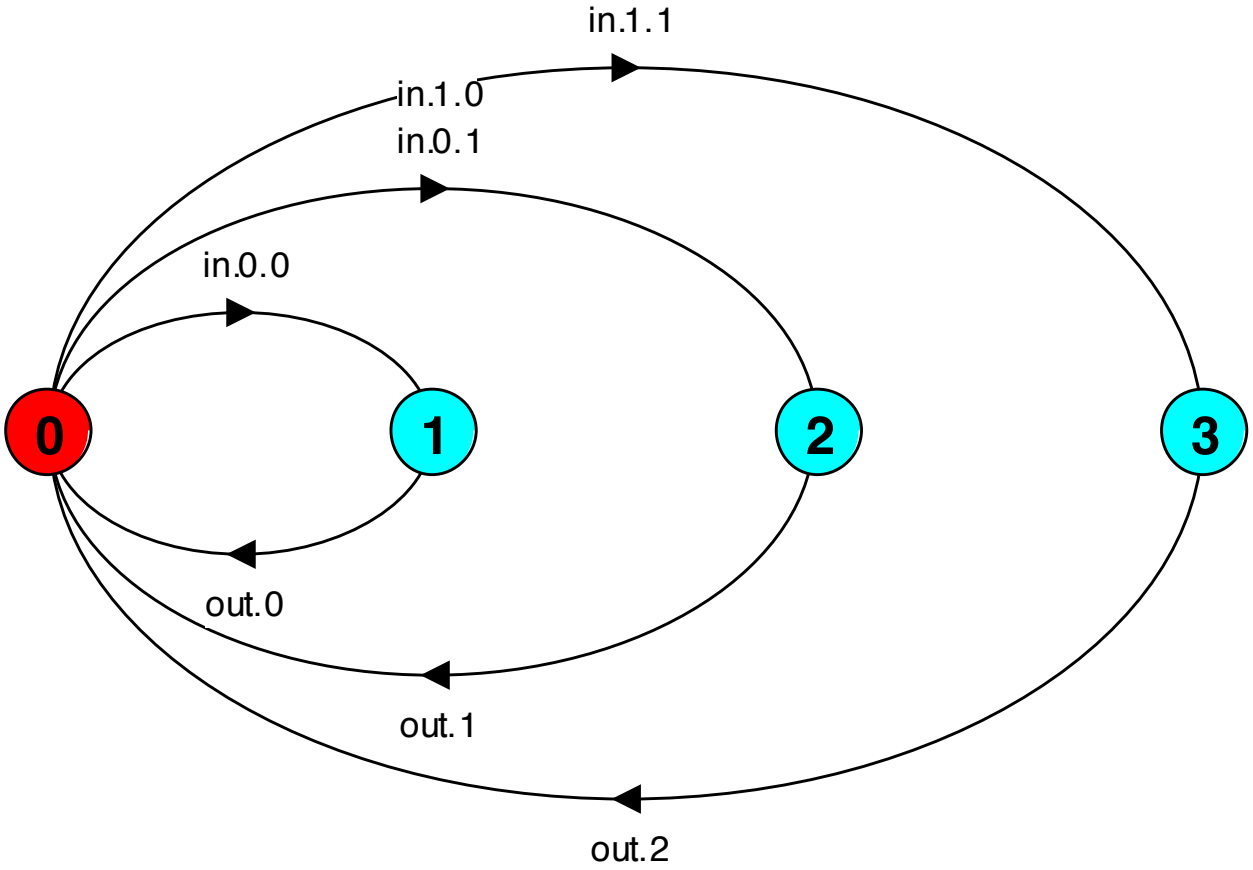
indexed actions generate labels of the form *action.index in LTS*

or using a process parameter with a default value:

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

Finite State Process – Indexed Processes and Actions

index expression to model calculation:

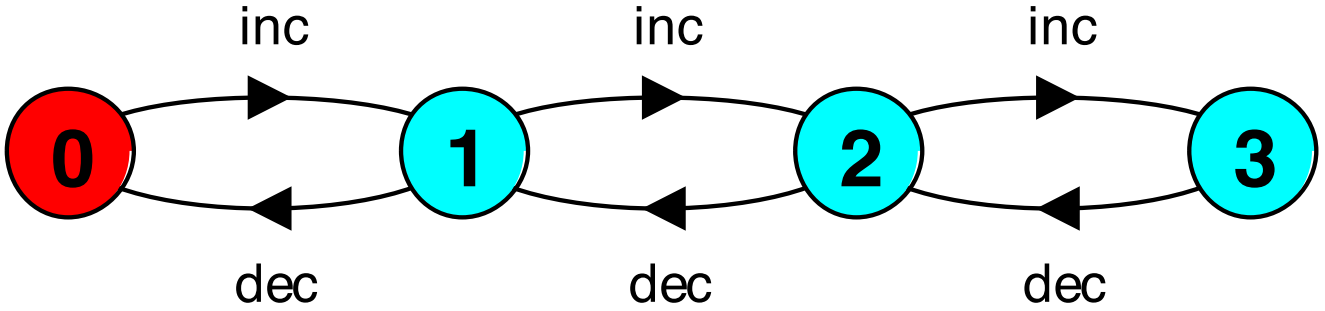



```
const N = 1
range T = 0..N
range R = 0..2*N

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

Finite State Process (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** or **y** **cannot** be chosen.

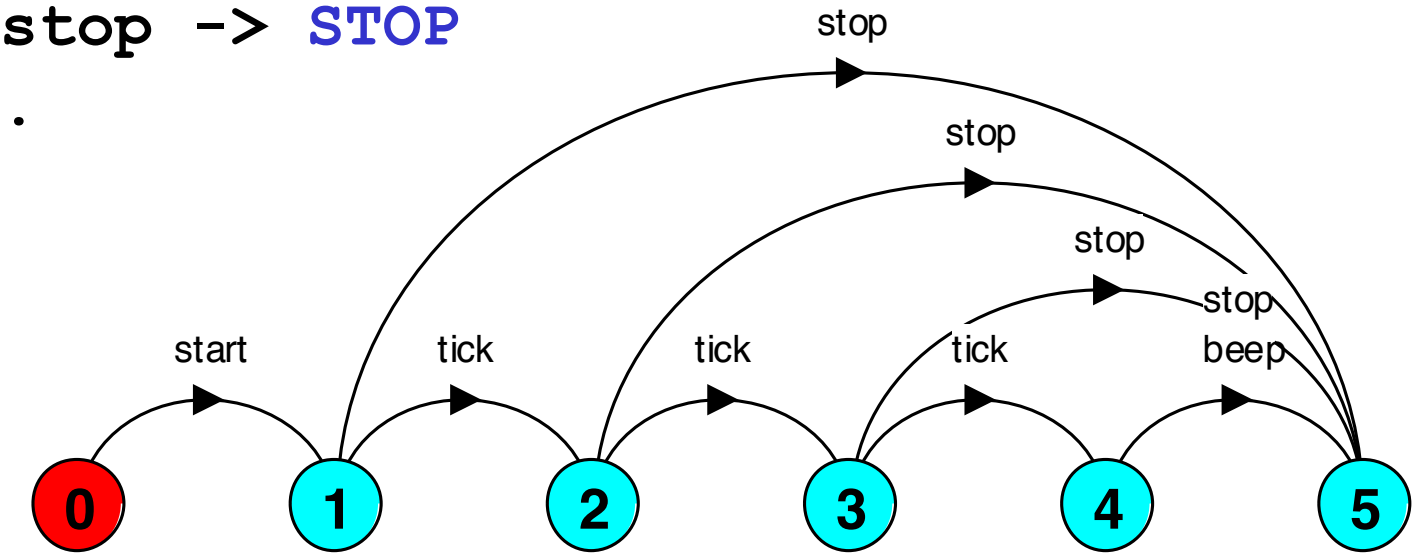
```
COUNT  (N=3)     = COUNT[0],
COUNT[i:0..N] = (when(i<N)  inc->COUNT[i+1]
                | when(i>0)  dec->COUNT[i-1]
                ).
```



Finite State Process (FSP) – Guarded Actions

A countdown timer beeps after N ticks, or can be stopped after it starts

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



Finite State Process (FSP) – Guarded Actions

What is the following FSP process equivalent to?

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

Finite State Process (FSP) – Guarded Actions

What is the following FSP process equivalent to?

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

Answer:

STOP (i.e., end of process)

Finite State Process (FSP) – Process Alphabet

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

Process alphabet is 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
     }
```

Finite State Process (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]}`

Alphabet extension (e.g., `+{write[0..3]`) is used when the set of actions in the alphabet is larger than the set of actions referenced in its definition.

Finite State Process (FSP) – Filter

In FSP, model a process FILTER for the following behavior:

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

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

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

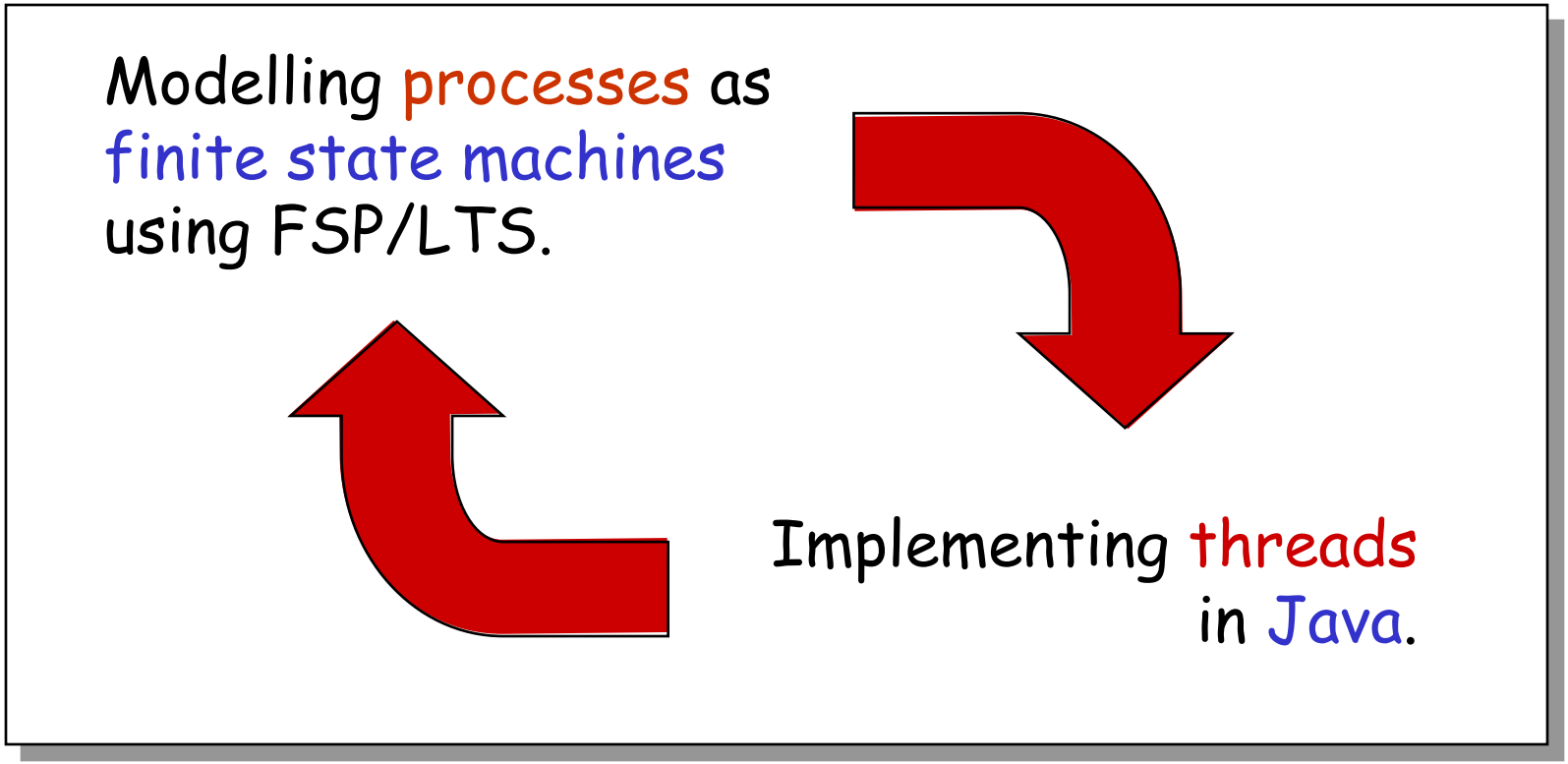
Finite State Process (FSP) – Filter

In FSP, model a process FILTER for the following behavior:

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

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

DECIDE[v:0..5] = ( when (v < 3) out[v]-> FILTER
                 | when (v > 2) discard -> FILTER
               ).
```

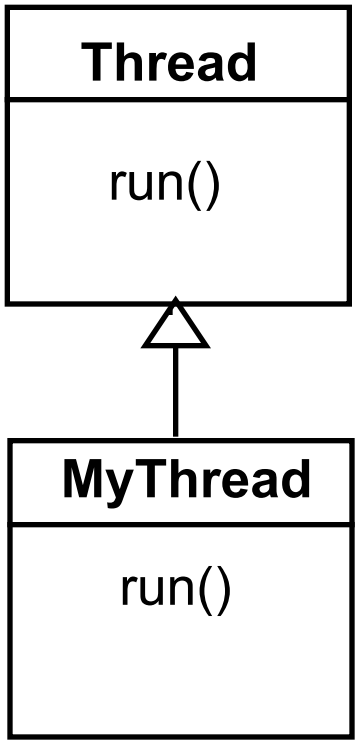
2.2 Implementing processes



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

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**.

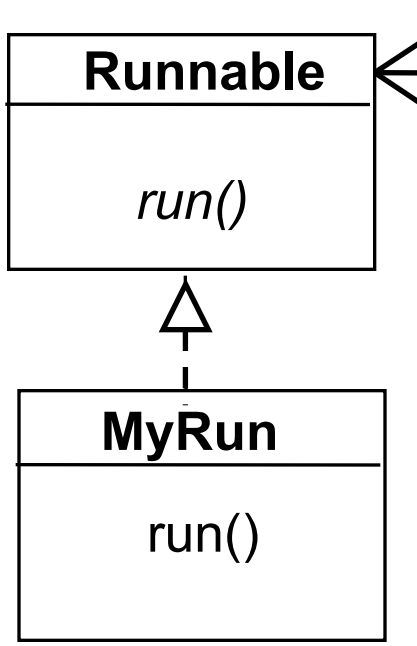
```
class MyThread extends Thread {
      public void run() {
            //......
      }
}
```

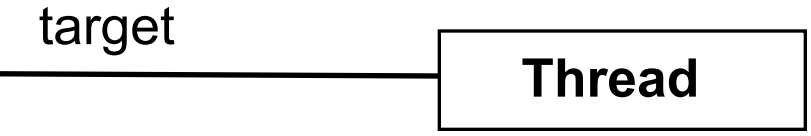
Creating a thread object:

```
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**.





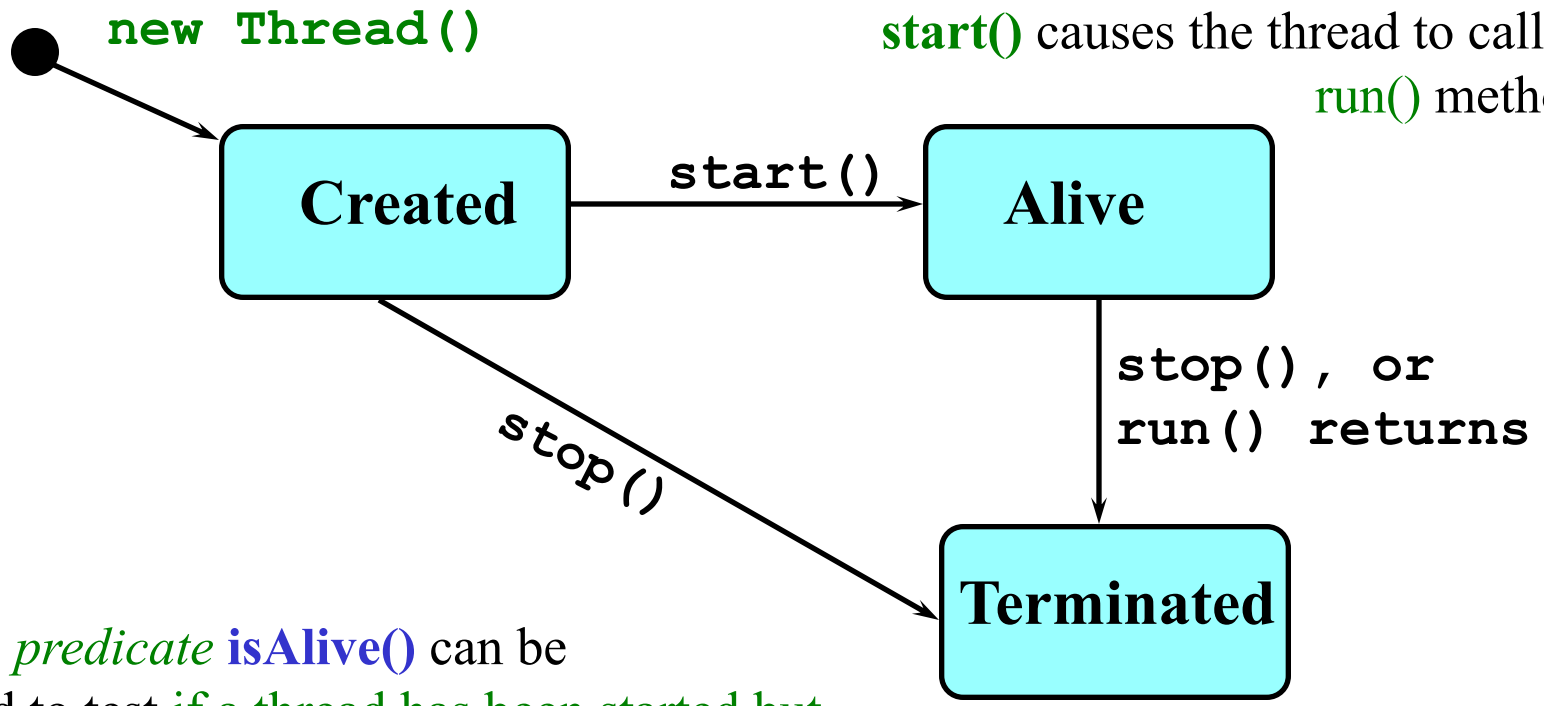

```
public interface Runnable {
    public abstract void run();
}

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

Creating a thread object:

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

Thread life-cycle in Java

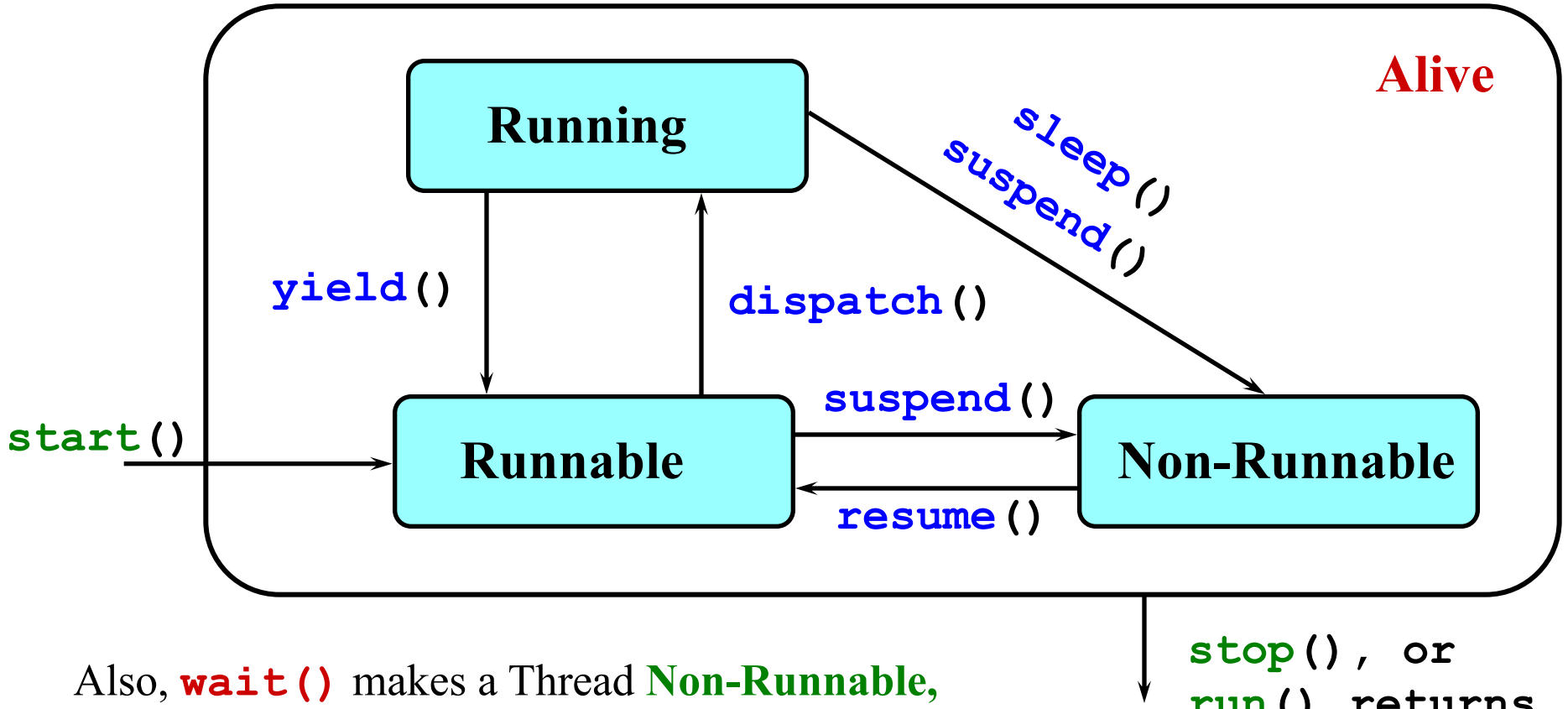
An overview of the **life-cycle of a thread** as state transitions:



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

Thread Alive States in Java

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

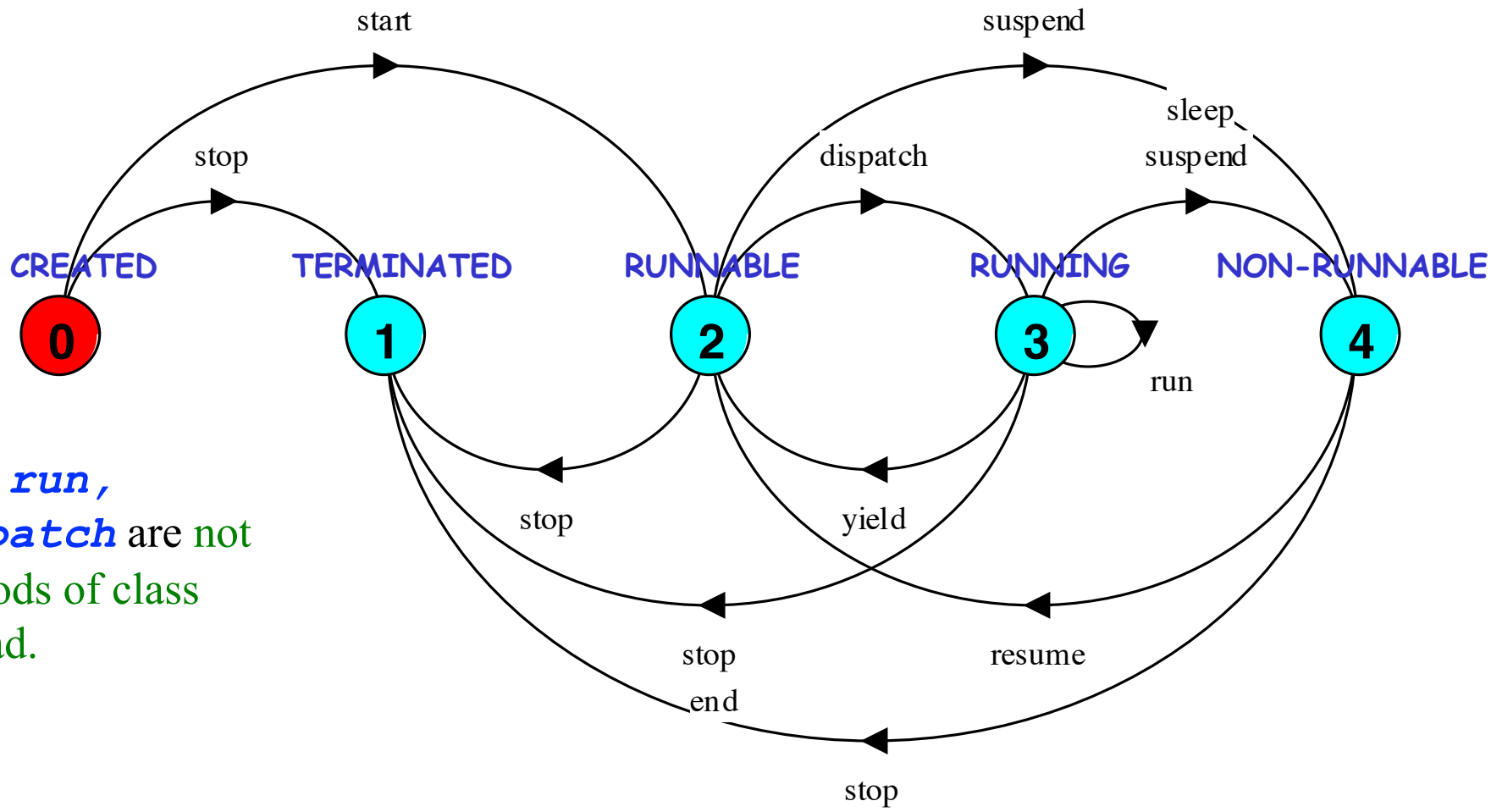


Also, `wait()` makes a Thread **Non-Runnable,**
and `notify()` makes it **Runnable**
(used in later chapters).

Java thread lifecycle - an FSP specification

```
THREAD          = CREATED,
CREATED         = (start                ->RUNNABLE
                   |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 lifecycle - an FSP specification



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

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

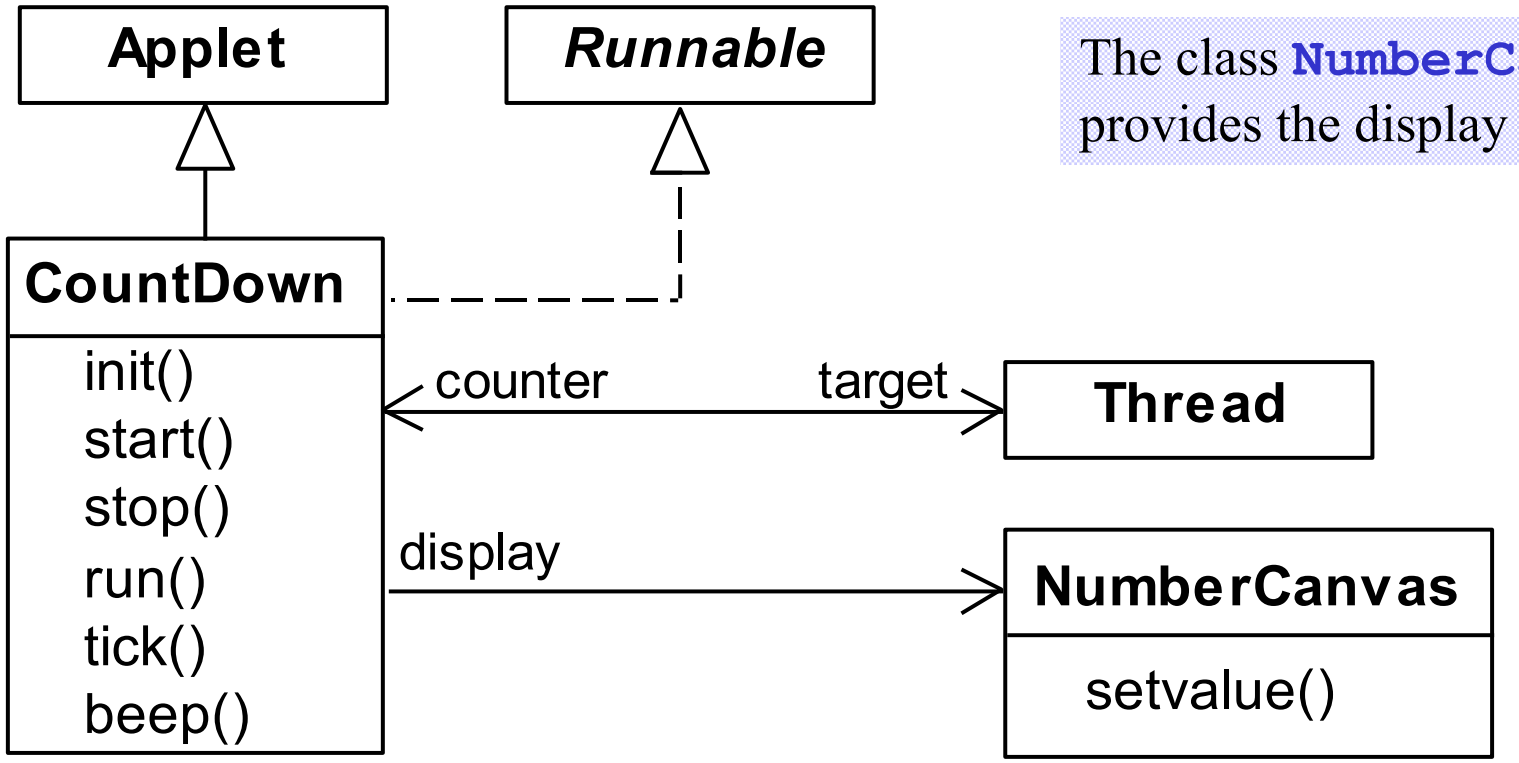
end: action of run() returning or exiting.

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() {...}
}
```

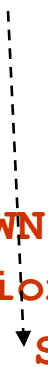
CountDown class - start(), stop() and run()

```
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 ->



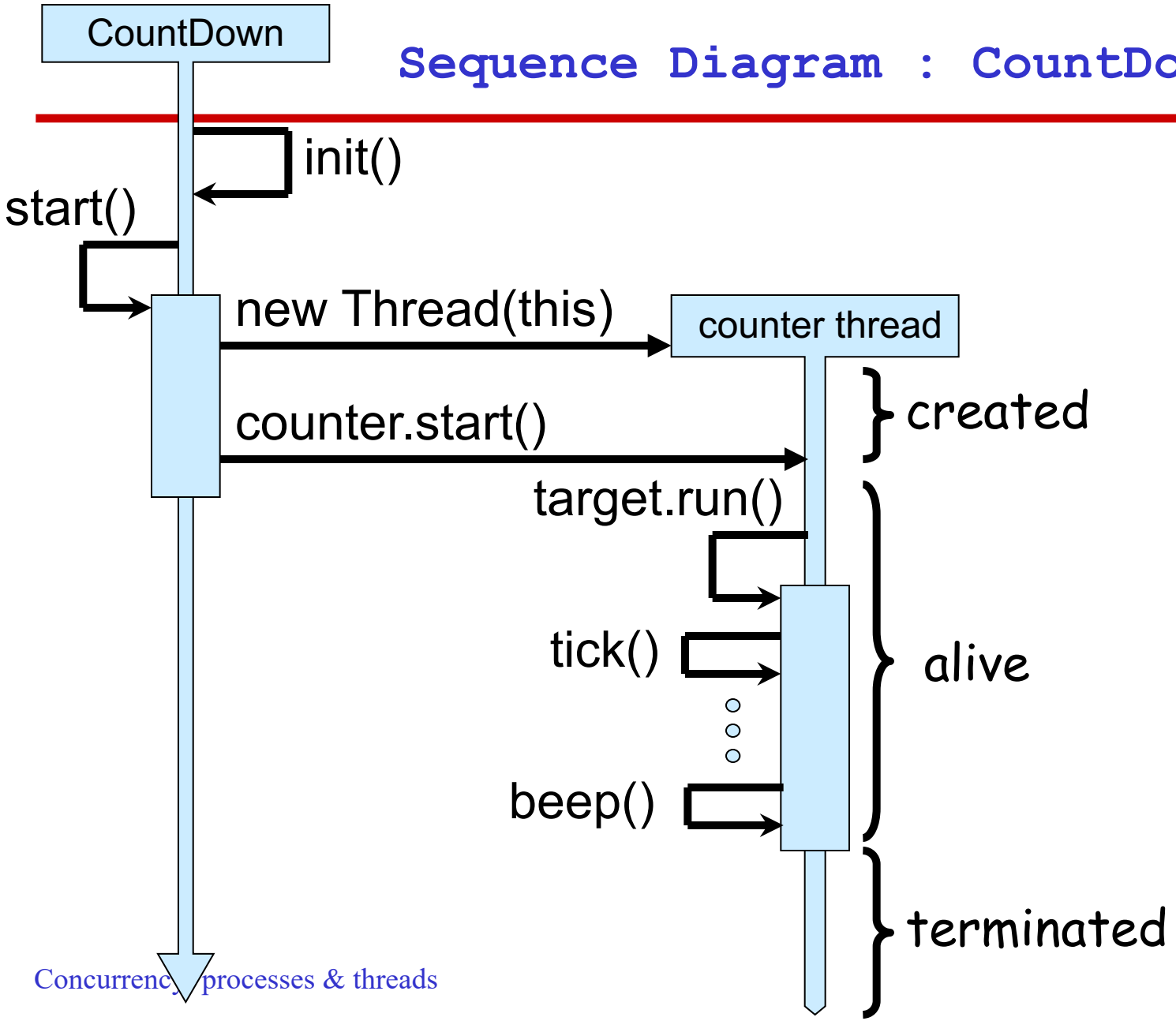
COUNTDOWN[i] process
recursion as a while loop

STOP

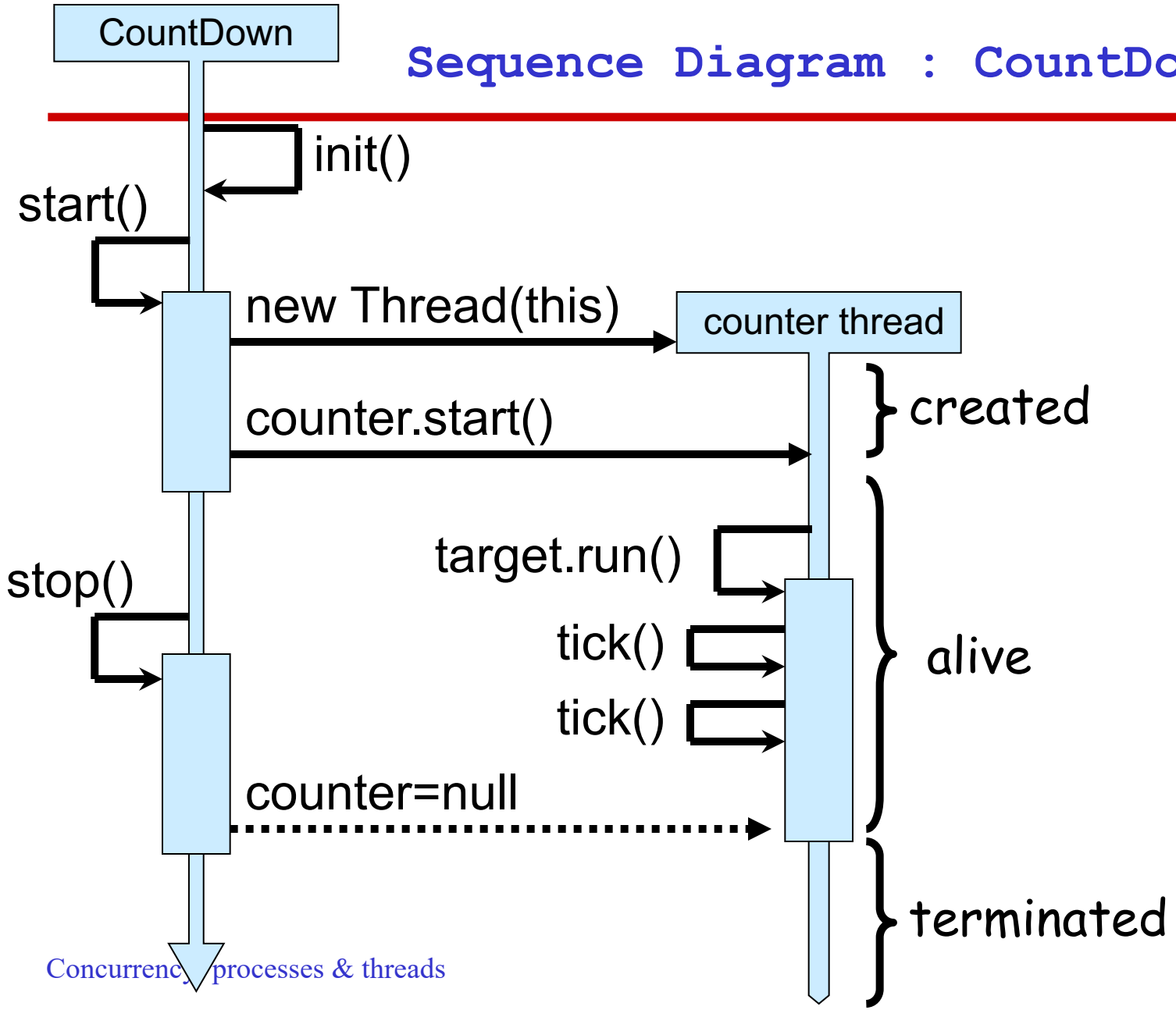
when(i>0) tick -> CD[i-1]
when(i==0)beep -> STOP

STOP when run() returns

Sequence Diagram : CountDown



Sequence Diagram : CountDown



Summary

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