Model-Based Approach to Designing Concurrent Systems (Part One)

Kenneth M. Anderson

University of Colorado, Boulder

CSCI 5828 — Lecture 12 — 02/18/2010

Credit Where Credit is Due

- Portions of these slides drawn from the course materials developed by Jeff Magee and Jeff Kramer for their excellent book
  - Concurrency: State Models and Java Programming, 2nd Ed.
- Portions are thus copyright © John Wiley & Sons, Ltd. 2006

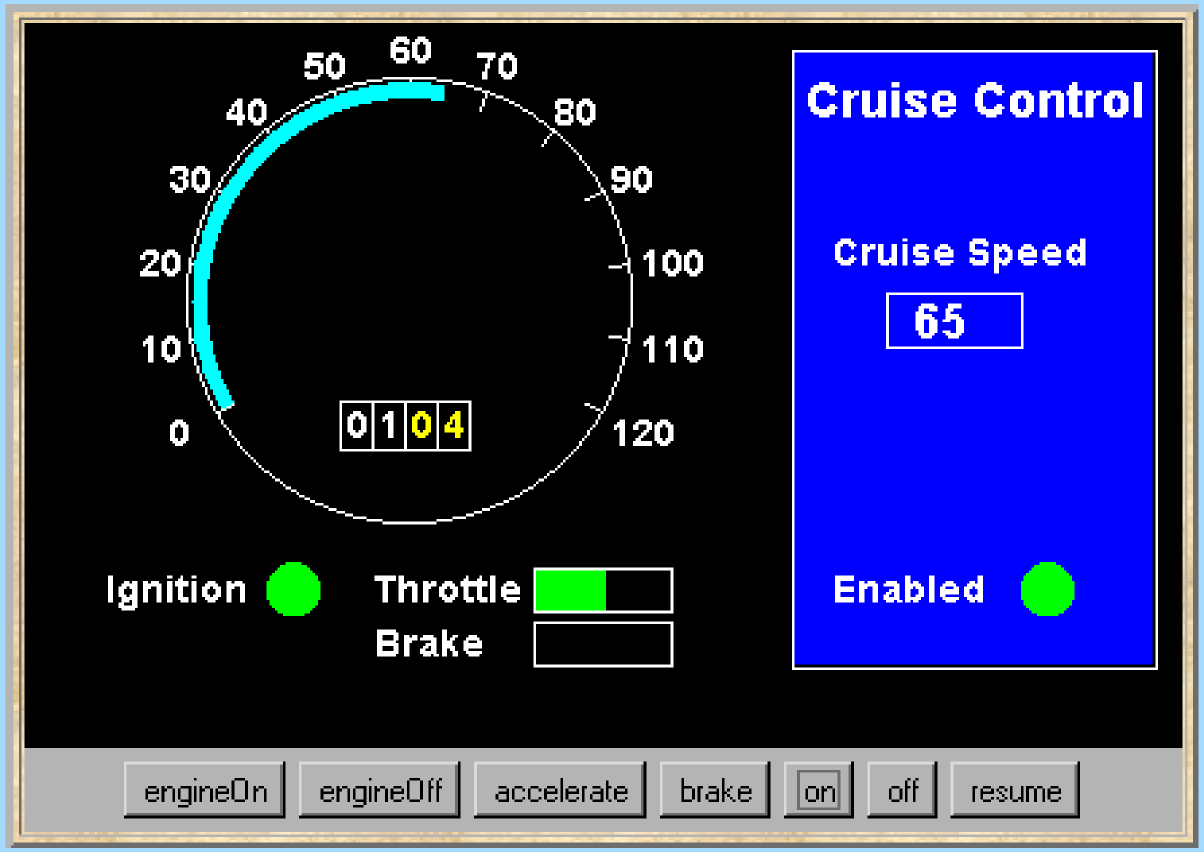
Goals

- Review material from chapters 1, 2 from the optional textbook (Concurrency: State Models and Java Programming by Magee and Kramer)
  - Present a model-based approach to designing concurrent systems
    - What do we mean by model-based software engineering?
    - Examine fundamental approach used in this book:
      - Concepts, Modeling, Practice
  - Finite State Processes and Labelled Transition Systems

More on the Authors: "The Two Jeffs"

- Jeff Kramer
  - Dean of the Faculty of Engineering and Professor of Distributed Computing at the Department of Computing at Imperial College London
  - ACM Fellow; Editor of IEEE's Transactions on Software Engineering
  - Winner of numerous software engineering awards including best paper and outstanding research awards
- Jeff Magee
  - Professor at the Department of Computing at Imperial College London
  - Long time member of the SE community with more than 70 journal and conference publications!
- This book is based on their SE research into modeling concurrency over the past 20 years

Ex.: Cruise Control System



Two Threads: Engine and Control

Is the system safe?
Would testing reveal all errors?
How many paths through system?

- Requirements
  - Controlled by three buttons
    - on, off, resume
  - When ignition is switched on and on button pressed, current speed is recorded and system maintains the speed of the car at the recorded setting
  - Pressing the brake, the accelerator, or the off button disables the system
  - Pressing resume re-enables the system

Models to the Rescue!

- To answer, we need a model of the concurrent behavior of the system and then we need to analyze it
  - This is one benefit of models, they focus on one particular aspect of the world and ignore all others
- Consider the **model** on the front of the Concurrency book
  - The picture shows a real-world train next to its model
  - Depending on the model, you can ask certain questions and get answers that reflect the answers you would get if you asked “the real system”

Models to the Rescue!

- For the train model, you might be able to ask
  - What color is the train? How long is it? How many cars does it have?
- But not
  - What’s the train’s maximum speed?
  - How does it behave when a car derails?

Models, continued

- A model is a simplified representation of the real world
  - A model airplane, e.g., used in wind tunnels, models only the external shape of the airplane
  - The reduction in scale and complexity achieved by modeling allows engineers to analyze properties of the model
  - The earliest models were physical (like our model train)
    - modern models tend to be mathematical and analyzed by computers

Models, continued

- Engineers use models to gain confidence in the adequacy and validity of a proposed design
  - focus on an aspect of interest — concurrency
  - can animate model to visualize a behavior
  - can analyze model to verify properties
- Models support hypothesis testing
  - we make observations and test against our model’s predictions
  - if predictions match observations, we gain confidence in the model; otherwise, we update model and try again

Models for Concurrency

- When modeling concurrency
  - our book makes use of a type of finite state machine known as a labeled transition system (LTS)
    - LTS == Model
  - These machines are described textually with a specification language called finite state processes (FSP)
    - FSP == Specification Language
      - Used to generate an instance of an LTS

Models for Concurrency

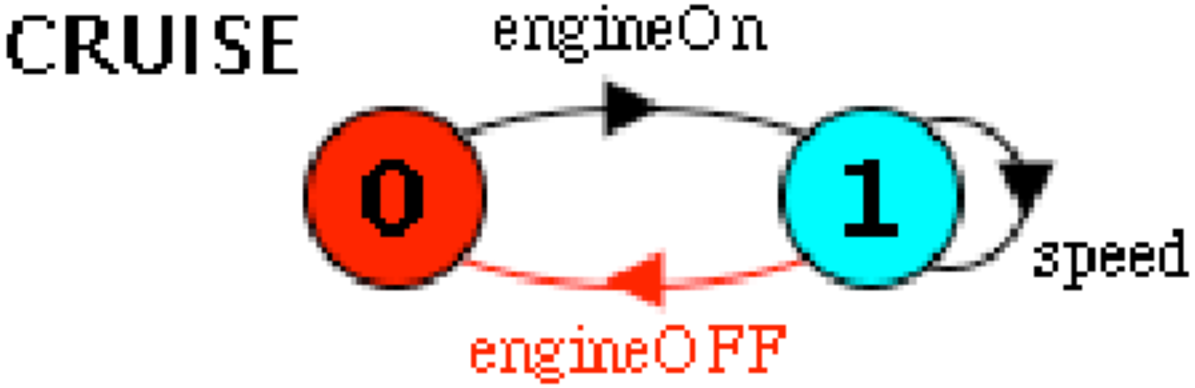
- These machines can be displayed and analyzed by an analysis tool called LTSA
    - Note: LTSA requires a Java 2 run time system, version 1.5.0 or later
    - On Windows and Mac OS systems, you can run the LTSA tool by double clicking on its jar file
    - Note: Its not the most intuitive piece of software, but once you “grok it”, it provides all of the advertised functionality

Modeling the Cruise Control System

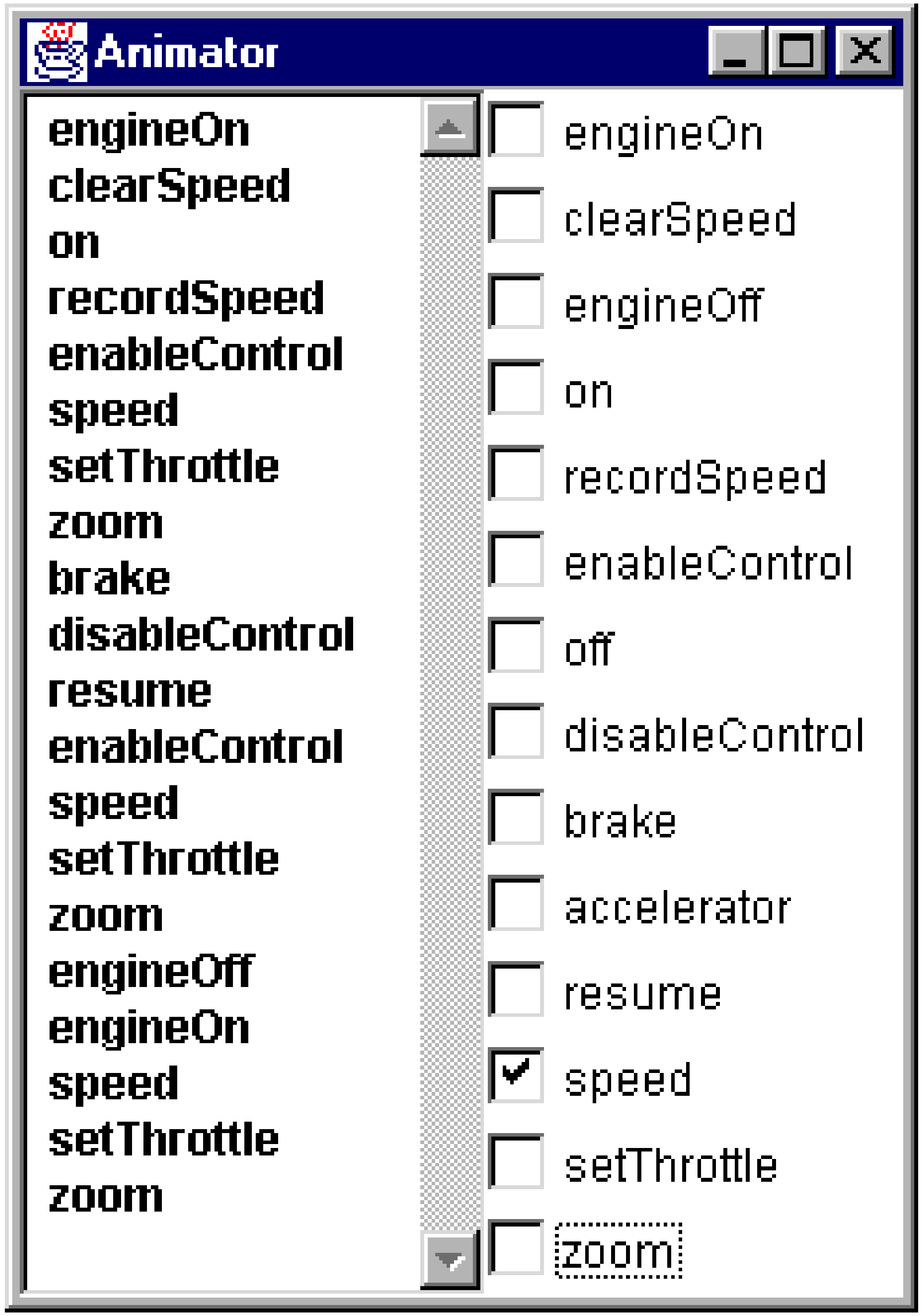
- We won’t model the entire system
  - lets look at a simplified example
- Given the following specification

```
CRUISE = (engineOn -> RUNNING),
RUNNING = (speed -> RUNNING | engineOFF -> CRUISE).
```

- We can generate a finite state machine that looks like this



LTSA



```
engineOn
clearSpeed
on
recordSpeed
enableControl
speed
setThrottle
zoom
brake
disableControl
resume
enableControl
speed
setThrottle
zoom
engineOff
engineOn
speed
setThrottle
zoom
```

☐ engineOn
☐ clearSpeed
☐ engineOff
☐ on
☐ recordSpeed
☐ enableControl
☐ off
☐ disableControl
☐ brake
☐ accelerator
☐ resume
☑ speed
☐ setThrottle
☐ zoom

- LTSA allows us to enter specifications and generate state machines like the ones on the previous slide
- It can also be used to “animate” or step through the state machine
- Lets see a demo
- Note: animation at left shows the problem we encountered before with the cruise control system

LTSA, continued

- Using a modeling tool, like LTSA, allows us to understand the concurrent behaviors of systems like the cruise control system, BEFORE they are implemented
    - This can save a lot of time and money, as it is typically easier to test and evolve a model's behavior than it is to implement the system in a programming language

Applying Concepts/ Models via Programming

- The optional textbook uses Java to enable practice of these concepts
- Java is
  - widely available, generally accepted, and portable
  - provides sound set of concurrency features
- Java is used for all examples, demo programs, and homework exercises in the optional textbook

Summary So Far

- Concepts
  - We adopt a model-based approach for the design and construction of concurrent programs
- Models
  - finite state machines to represent concurrent behavior
- Practice
  - Book uses Java for constructing concurrent programs
- We will be presenting numerous examples to illustrate concepts, models and demonstration programs

Modeling Sequential 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 and/or other devices
- Designing concurrent software can be complex/error prone
    - A rigorous engineering approach is essential

Overall Approach

Concept of a process as
a sequence of actions

Model processes as
finite state machines

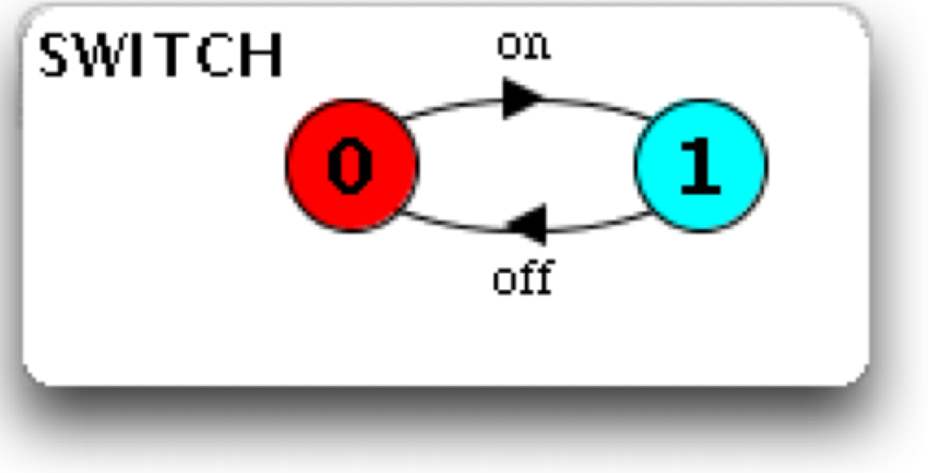
Program processes as
threads in Java

Modeling Processes

- Models are described using state machines
  - Labeled Transition System (LTS)
- Described textually as finite state processes (FSP)
- They are displayed and analyzed by the LTSA tool
- Summary
  - FSP: textual form
  - LTS: data structure
  - LTSA: visualizer and analyzer

Modeling Processes

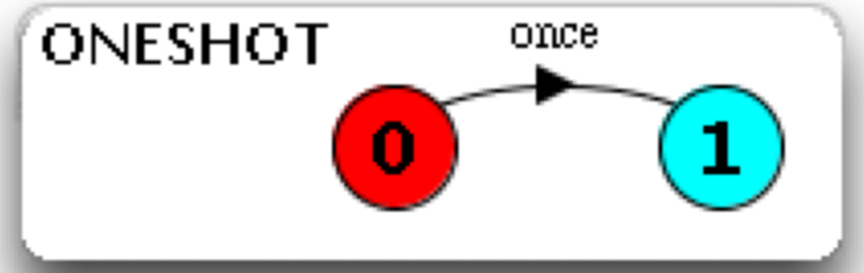
- A process is the execution of a sequential program. It is modeled as a finite state machine that moves from state to state by executing a sequence of **atomic** actions
- To the right is a “light switch”
  - it has two states and two actions
  - what does state zero represent?



  - A **trace** is a sequence of actions
    - For the light switch: on → off → on → off → on …

Specifying a process

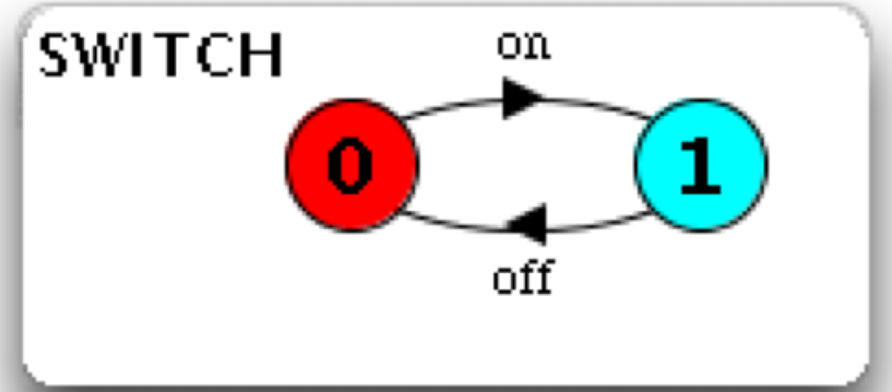
- FSP — action prefix
  - If x is an action and P a process then (x → P) describes a process that initially engages in the action **x** and then behaves exactly as described by **P**. i.e. (x → P) is also a process.
  - ONESHOT = (once -> STOP).
  - STOP is a predefined process that tells LTSA to halt.



  - ONESHOT is a process; it executes “**once**” before halting
- Convention: actions begin with lowercase letters; PROCESSES use all uppercase letters

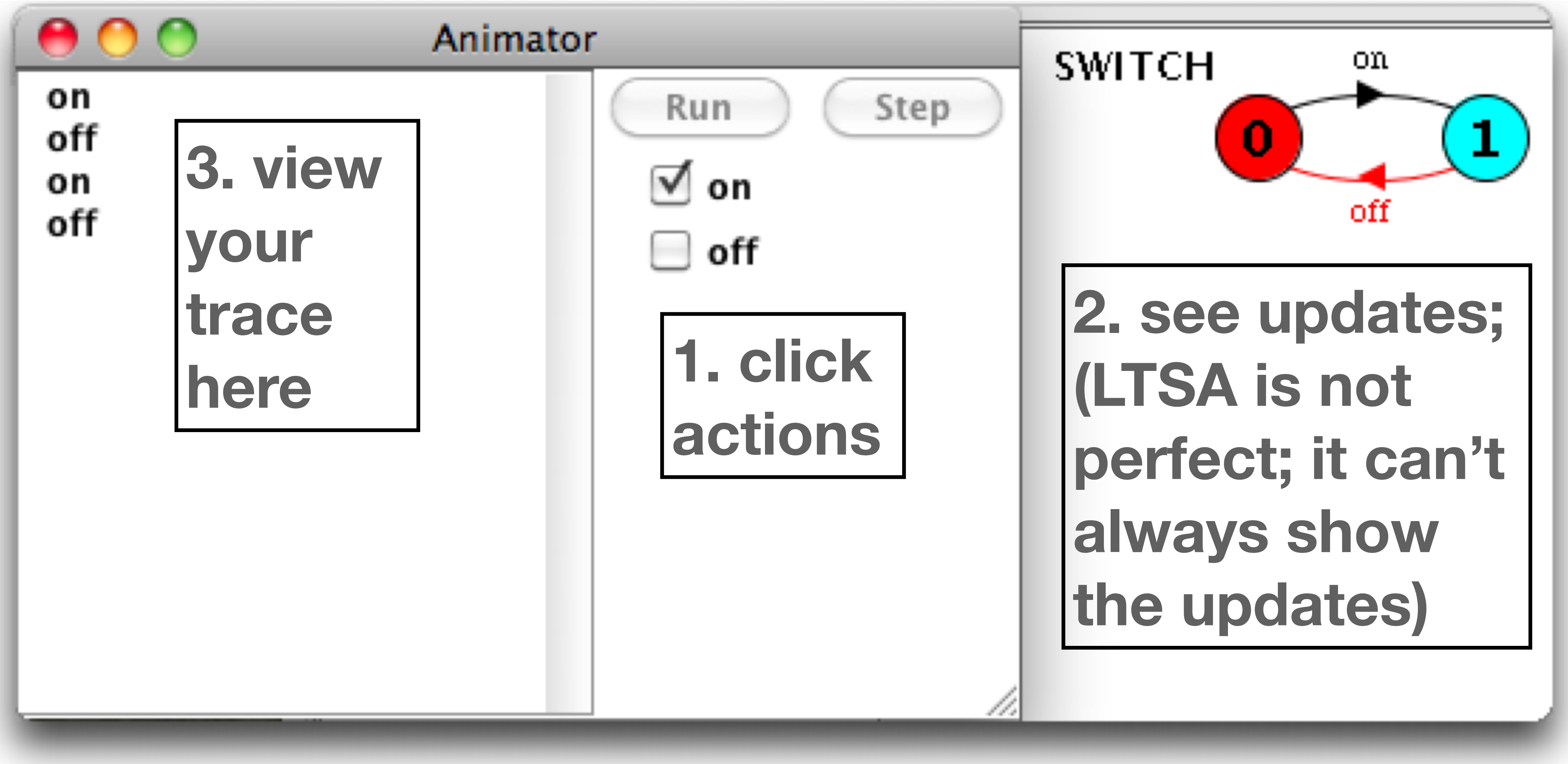
Repetitive behavior

- Repetitive behavior uses recursion:
  - SWITCH = OFF,
    OFF = (on -> ON),
    ON = (off -> OFF).
- You can apply substitution
  - SWITCH = OFF,
  - OFF = (on -> (off -> OFF)).
- And again, to get a succinct definition
  - SWITCH = (on -> off -> SWITCH).



All three produce the above LTS

Animation



```
on
off
on
off
```

☑ on
☐ off

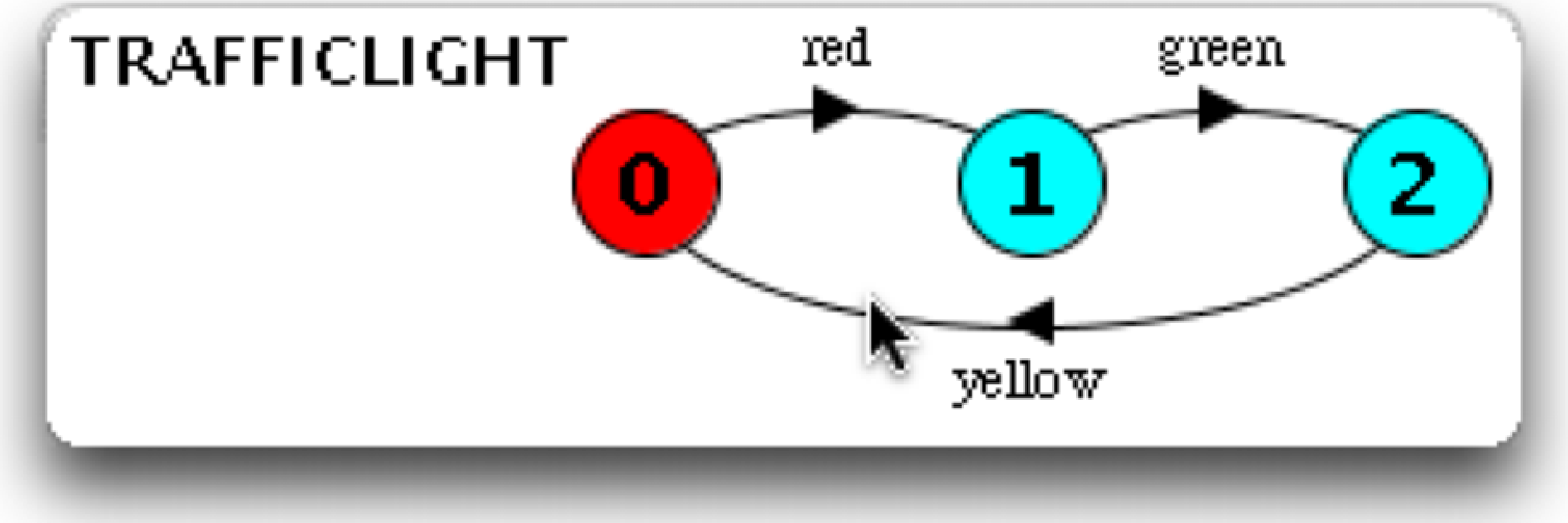
3. view your trace here

1. click actions

2. see updates; (LTSA is not perfect; it can’t always show the updates)

Simple Example

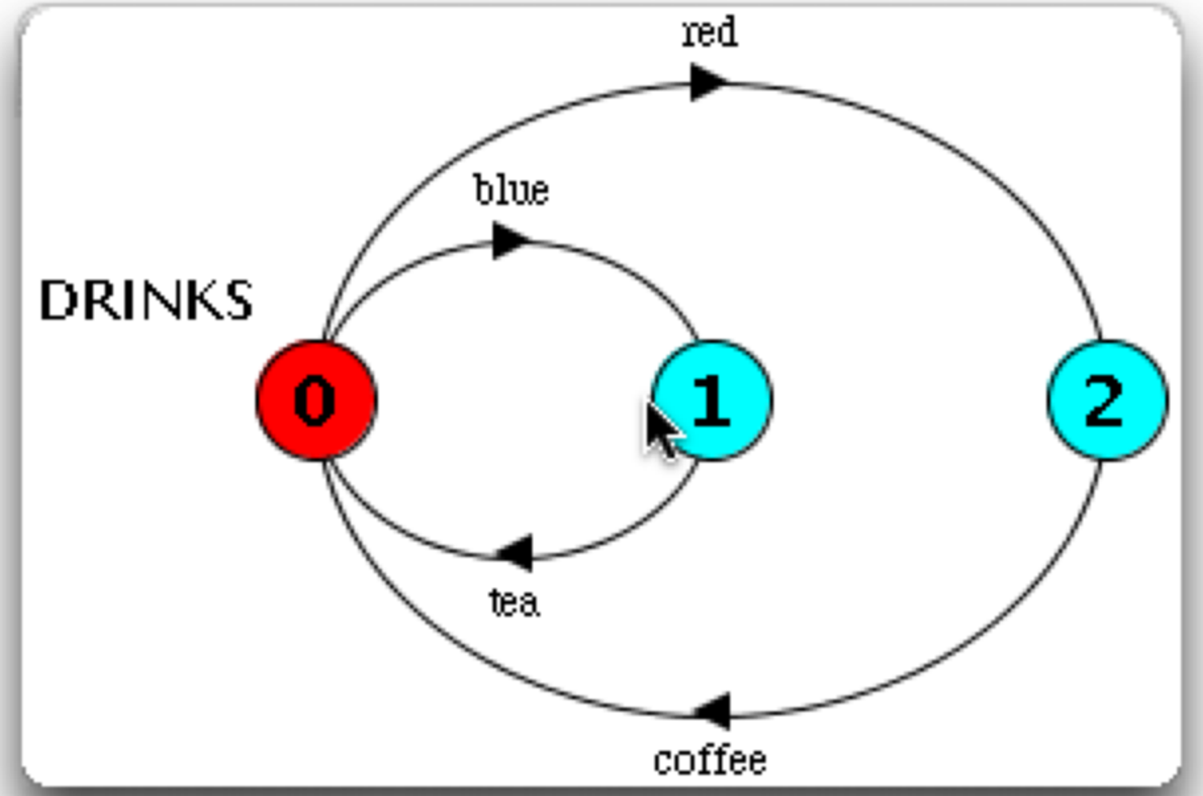
- `TRAFFICLIGHT = (red -> green -> yellow -> TRAFFICLIGHT).`



- Trace
  - red → green → yellow → red → green → yellow → …

Adding Choice

- If x and y are actions then (x → P | y → Q) is a process which initially engages in either of the actions x or y.
- `DRINKS = (red -> coffee-> DRINKS |blue ->tea -> DRINKS).`
- red and blue are considered **input** actions; coffee and tea are **output** actions

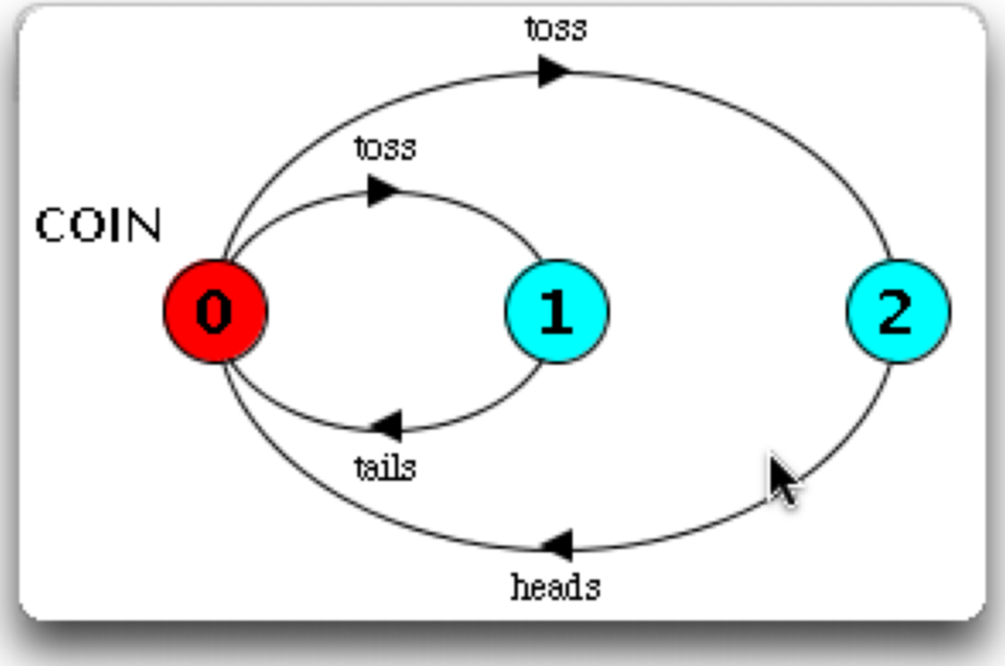


An input action is one which participates in a choice; someone has to select an action before the process can go on.

Nondeterministic Choice

- Process (x → P | x → Q) describes a process which engages in **x** and then behaves as either P or Q.
  - As you can see, we have the same action on multiple branches

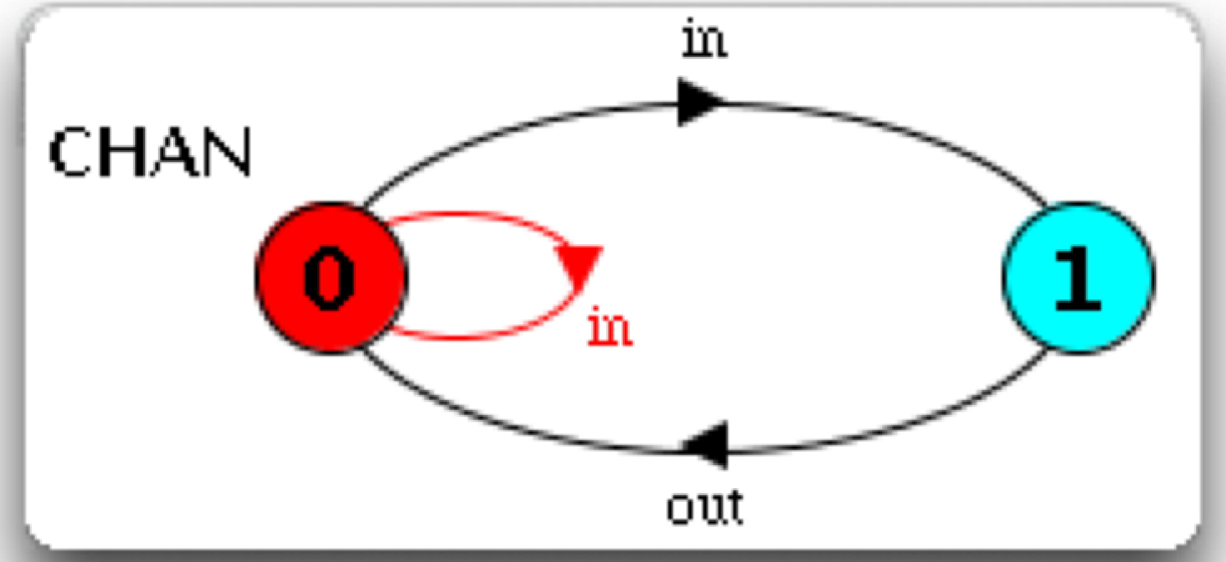
  -
    ```
    COIN = (toss -> HEADS | toss -> TAILS),
    HEADS = (heads -> COIN),
    TAILS = (tails -> COIN).
    ```
- Tossing a coin.
- In this case, LTSA will randomly select a branch to execute.



Modeling Failure

- We can use nondeterminism to model failure
  - Here we want to model a communication channel that is sometimes unreliable; an input can sometimes fail to produce an output

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

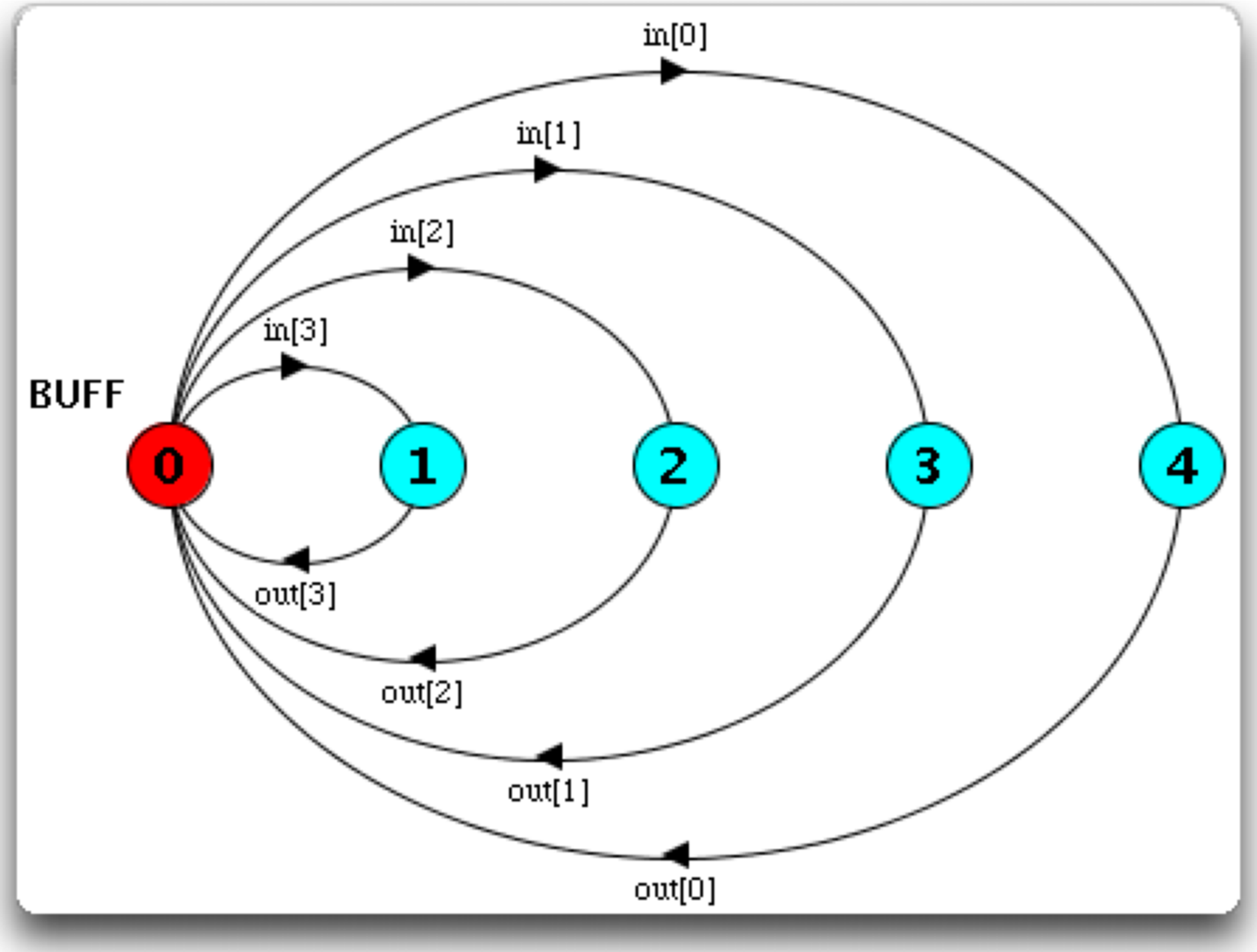


Adding modeling power

- In order to increase the power of our models, we can add indexes to both actions and processes
- We can add an index to an action, like this…
  - `in[i : 0 .. 3]`
- …which requires us to pick a value for the index when we execute the action
- The index can then be referenced in later actions, carrying the value we picked
  - `out[i]`

Example

- `BUFF = (in[i: 0..3] -> out[i] -> BUFF).`
- Single slot buffer
  - what goes in
  - must come out



indexes are shortcuts

- Note, this:
  - `BUFF = (in[i: 0..3] -> out[i] -> BUFF).`
- is equivalent to this:

-
  ```
  BUFF = (in[0]->out[0]->BUFF
         |in[1]->out[1]->BUFF
         |in[2]->out[2]->BUFF
         |in[3]->out[3]->BUFF).
  ```
- indexed actions simply expand to all possible choices behind the scenes

Magic Numbers

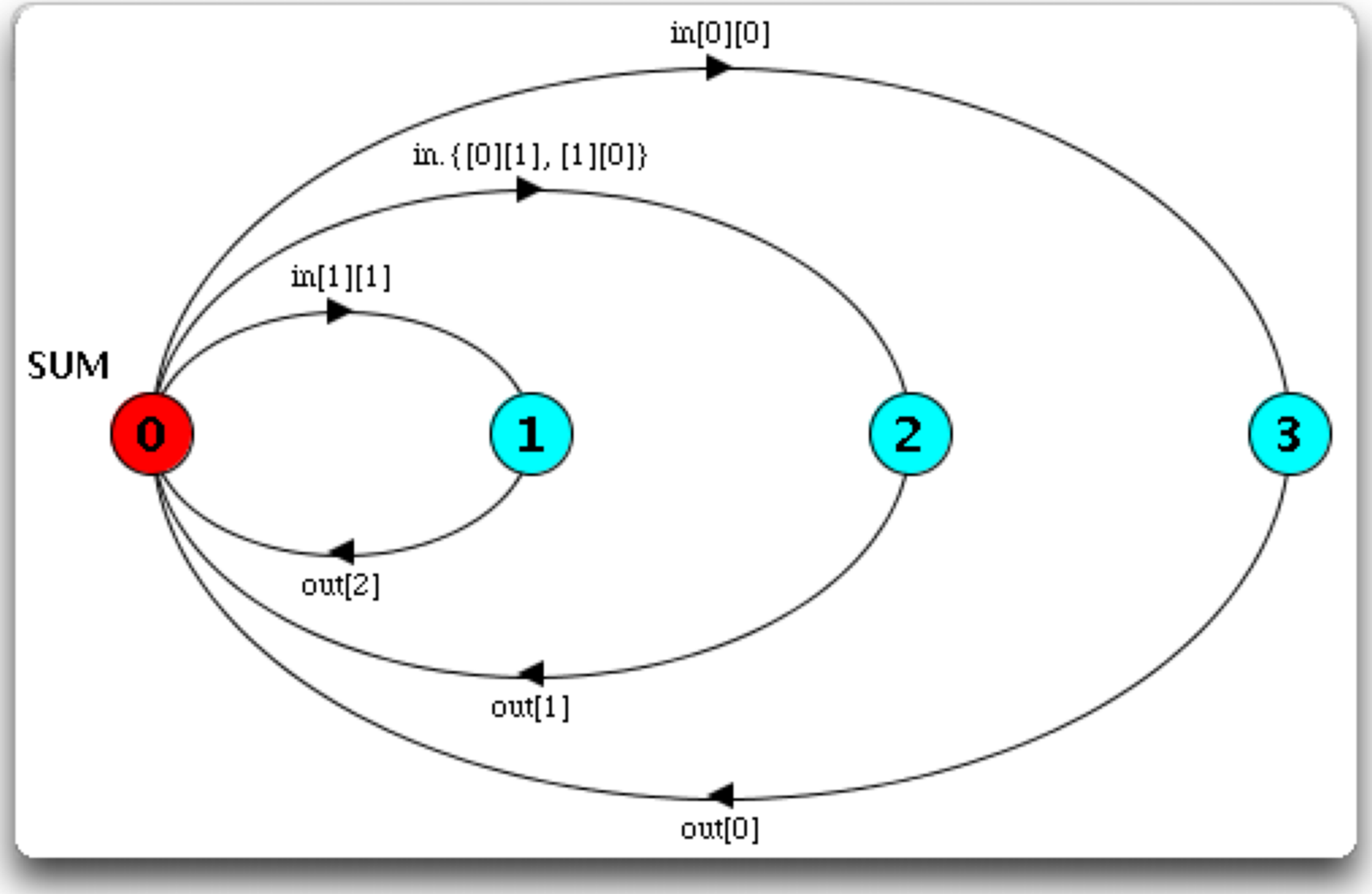
- In this process
  - `BUFF = (in[i: 0..3] -> out[i] -> BUFF).`
- “3” is a magic number
- We can add flexibility to our models via indexed processes
  - `BUFF(N=3) = (in[i:0..N]->out[i]-> BUFF).`
- Now we can change N to whatever value we need

Computation

- Indexes can be used 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).
    ```
- Here, our choices for indexes a and b influence the starting value s for process TOTAL; a + b is calculated and passed to TOTAL, setting the value for index s

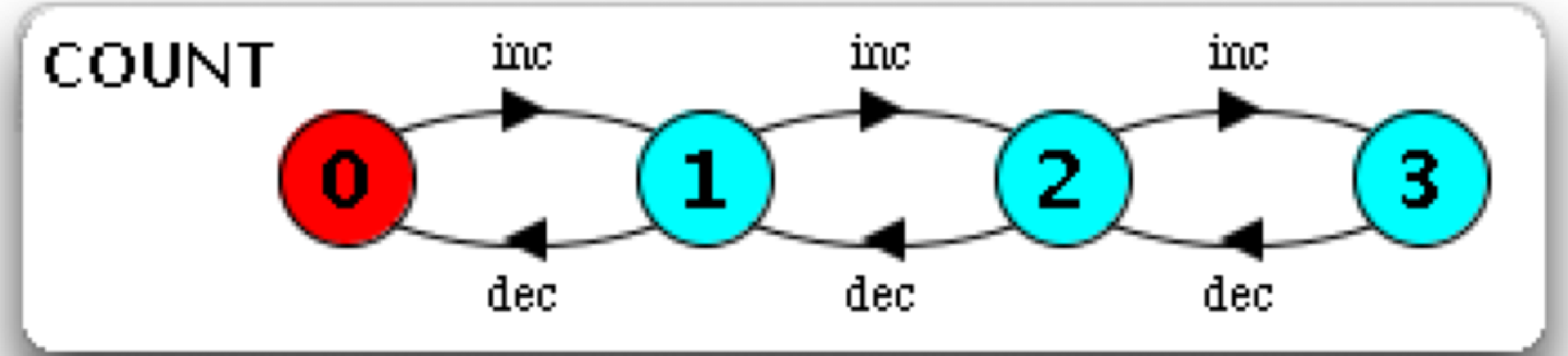
LTS for SUM



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 only **y** can be selected.

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



Process Alphabets

- The alphabet of a process is the set of actions in which it can engage; LTSA can show a process alphabet on request
- Process alphabets are implicitly defined by the actions in the process definition.
-
```
COUNTDOWN (N=3)   = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
  (  when(i>0) tick->COUNTDOWN[i-1]
   | when(i==0)beep->STOP
   | stop->STOP).
```
- The alphabet of COUNTDOWN is “start”, “tick”, “beep”, and “stop”

Implementing Models

- Implementing a model is typically straightforward

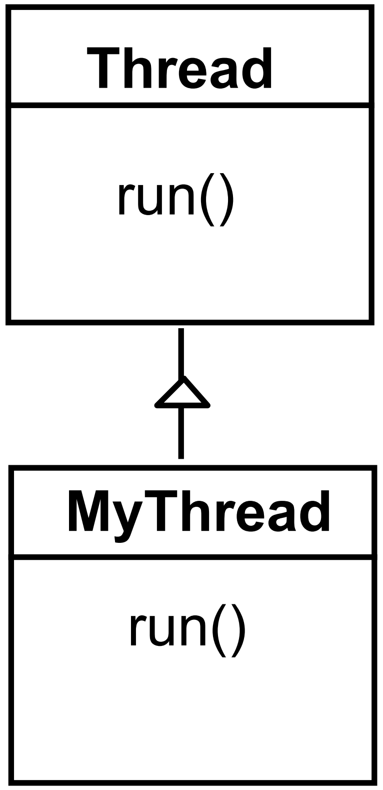
```
public void start() {
    counter = new Thread(this);
    i = N; counter.start();
  }
  public void stop() {
    counter = null;
  }
  public void run() {
    while(true) {
      if (counter == null) return;
      if (i>0)  { tick(); --i; }
      if (i==0) { beep(); return;}
    }
  }
```

Implementation of COUNTDOWN

imagine this placed inside of a class that implements Runnable

threads in Java

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



Thread

run()

MyThread

run()

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

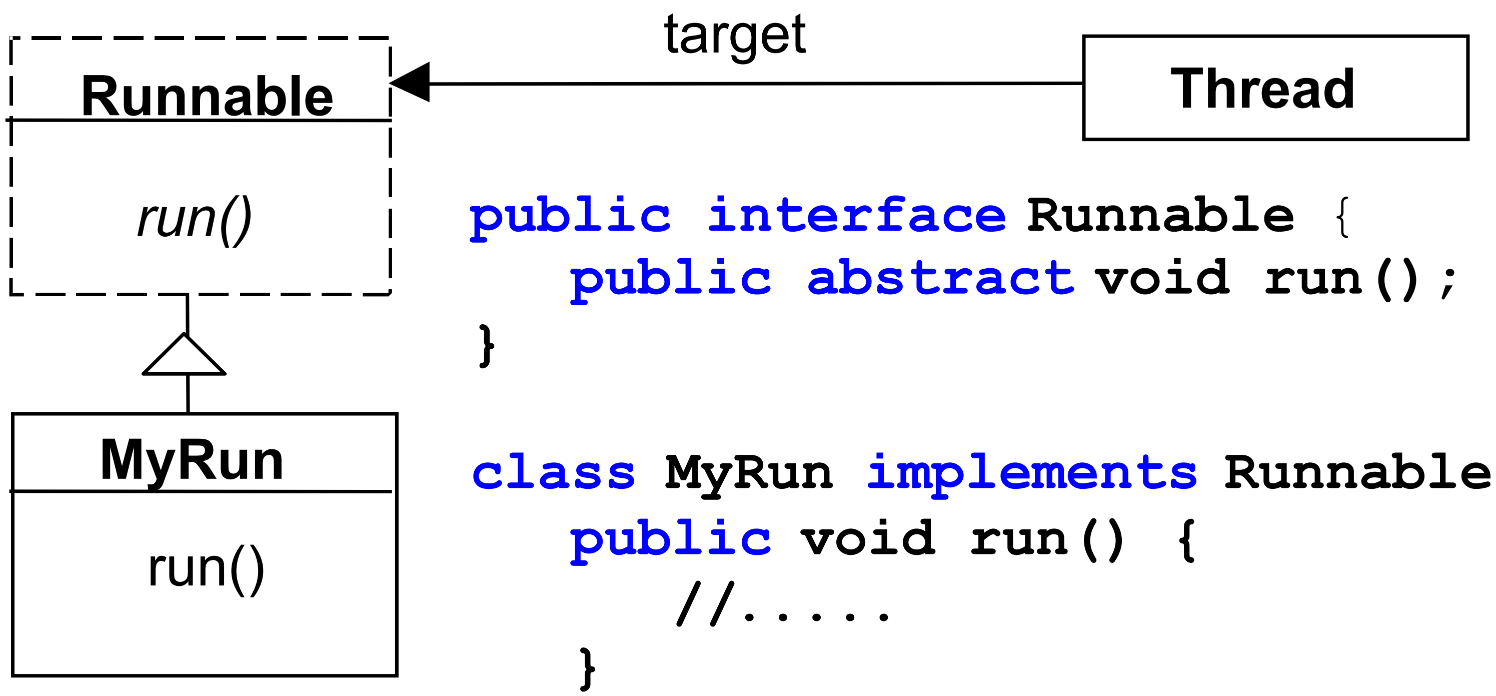
```
class MyThread extends Thread {
     public void run() {
          //......
     }
}
```

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.



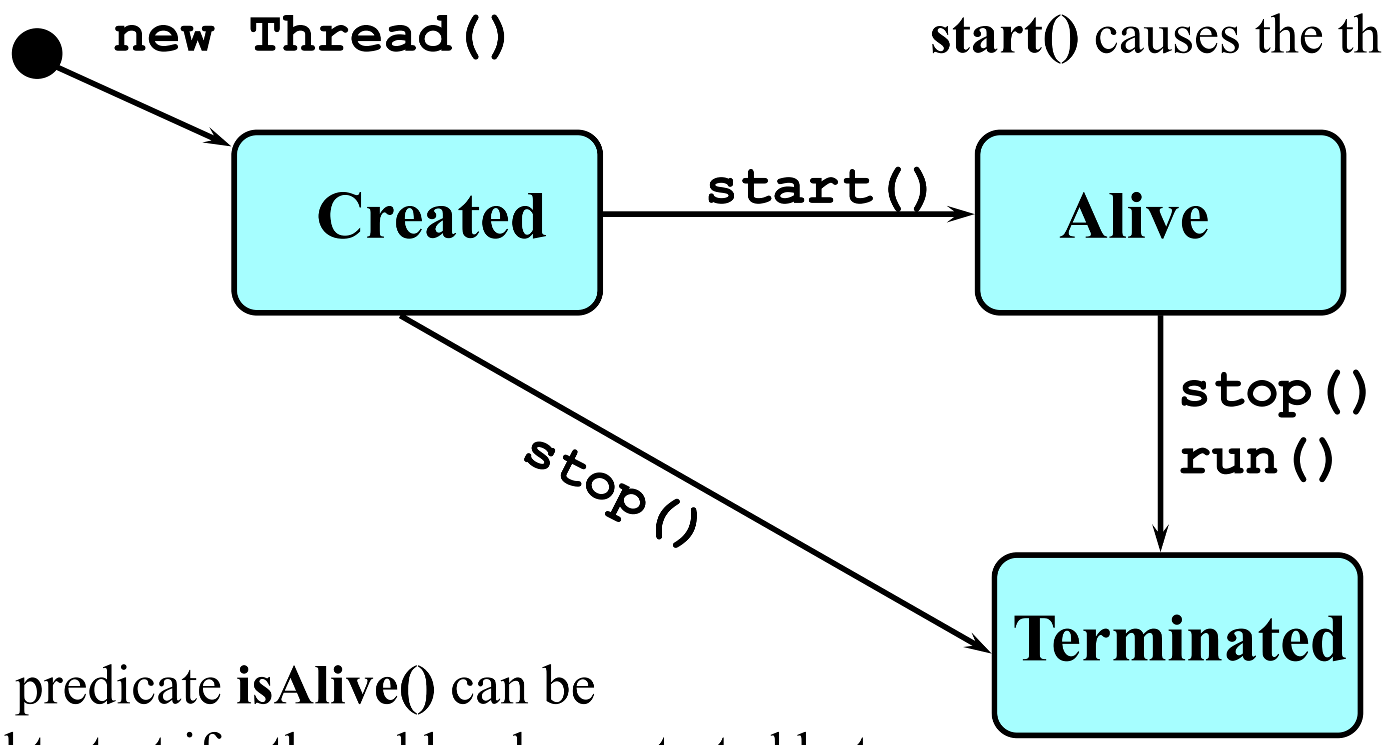
```
public interface Runnable {
    public abstract void run();
}

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:



`new Thread()`

Created

`start()`

Alive

`stop(), or`
`run() returns`

`stop()`

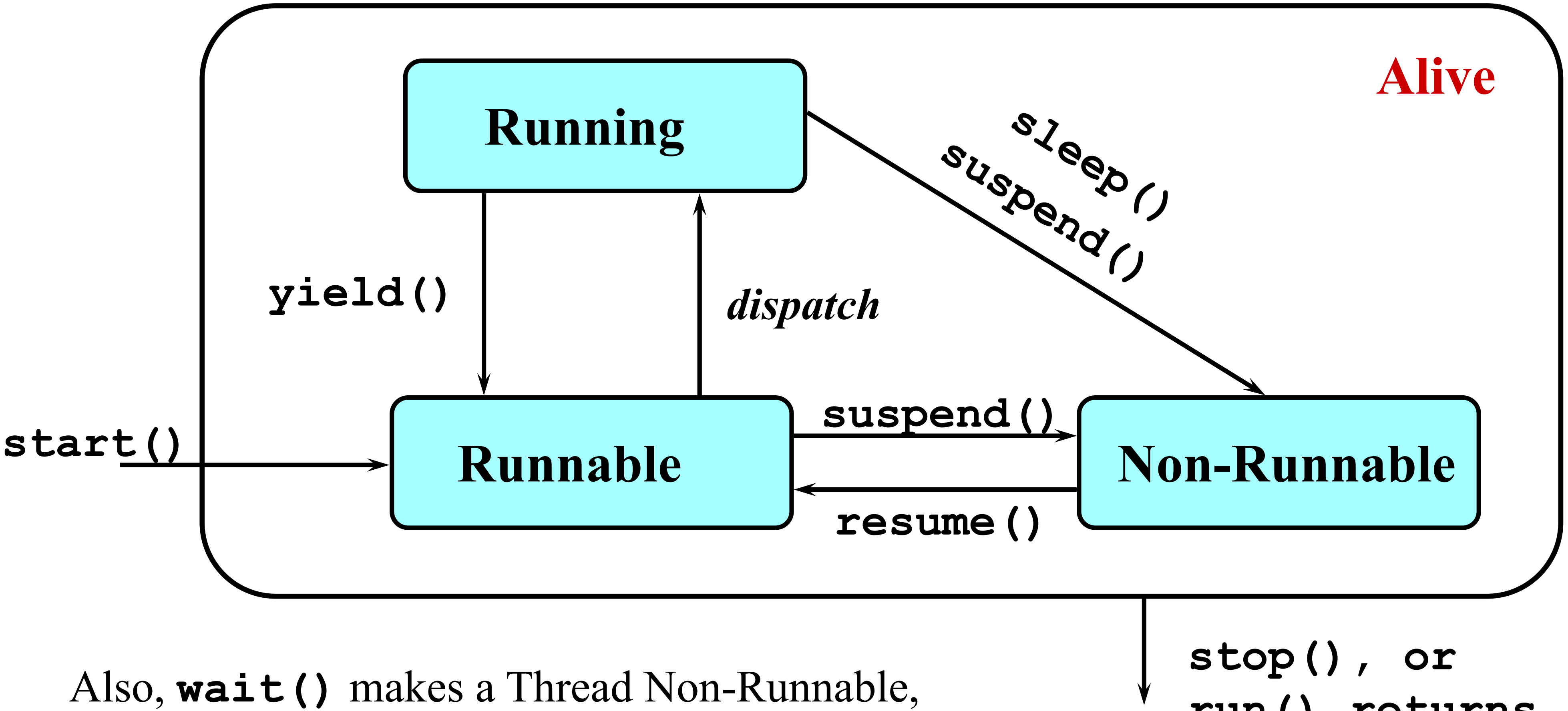
Terminated

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 :

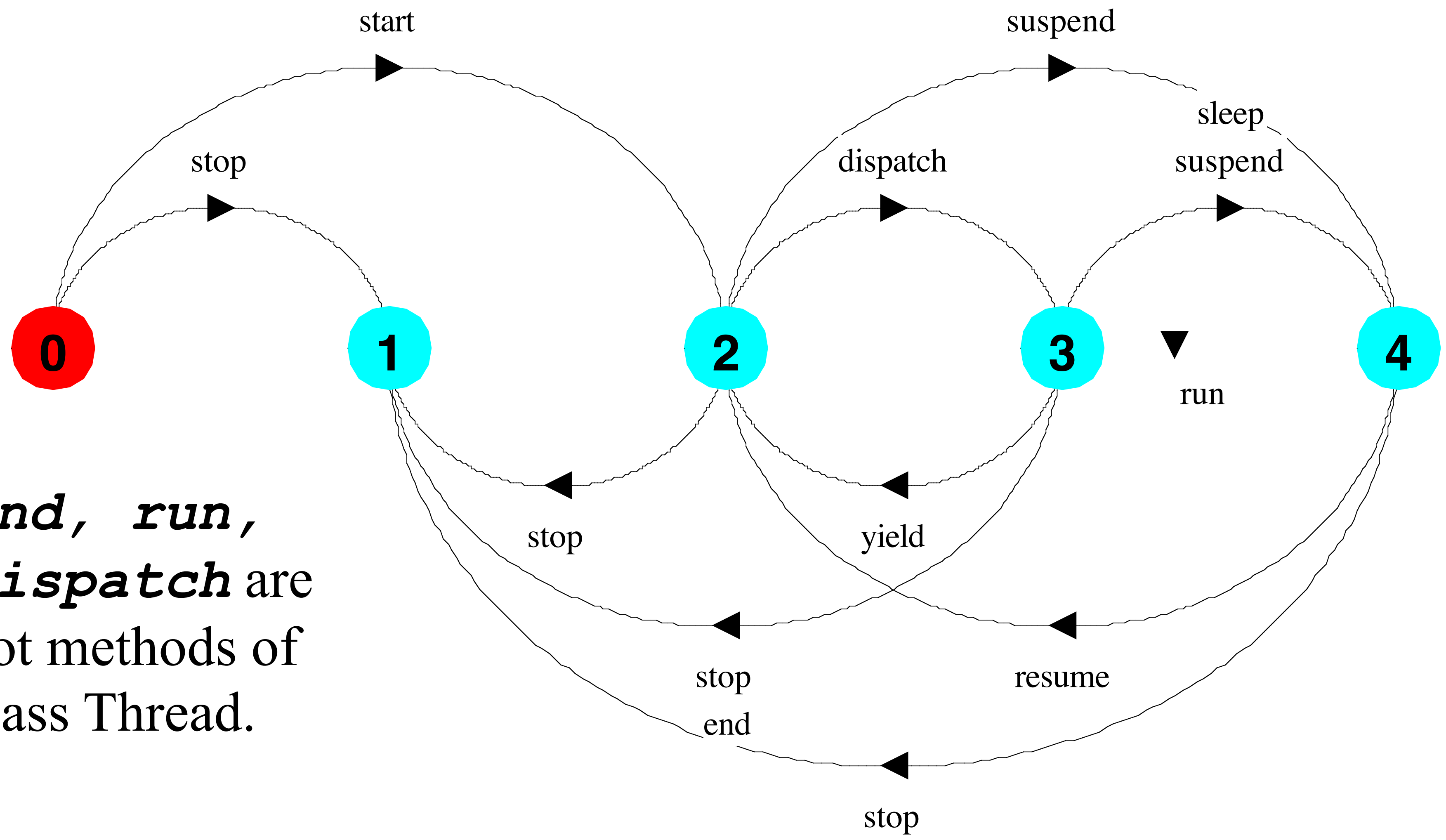


Also, **`wait()`** makes a Thread Non-Runnable,
and **`notify()`** makes it Runnable
(used in later chapters).

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.

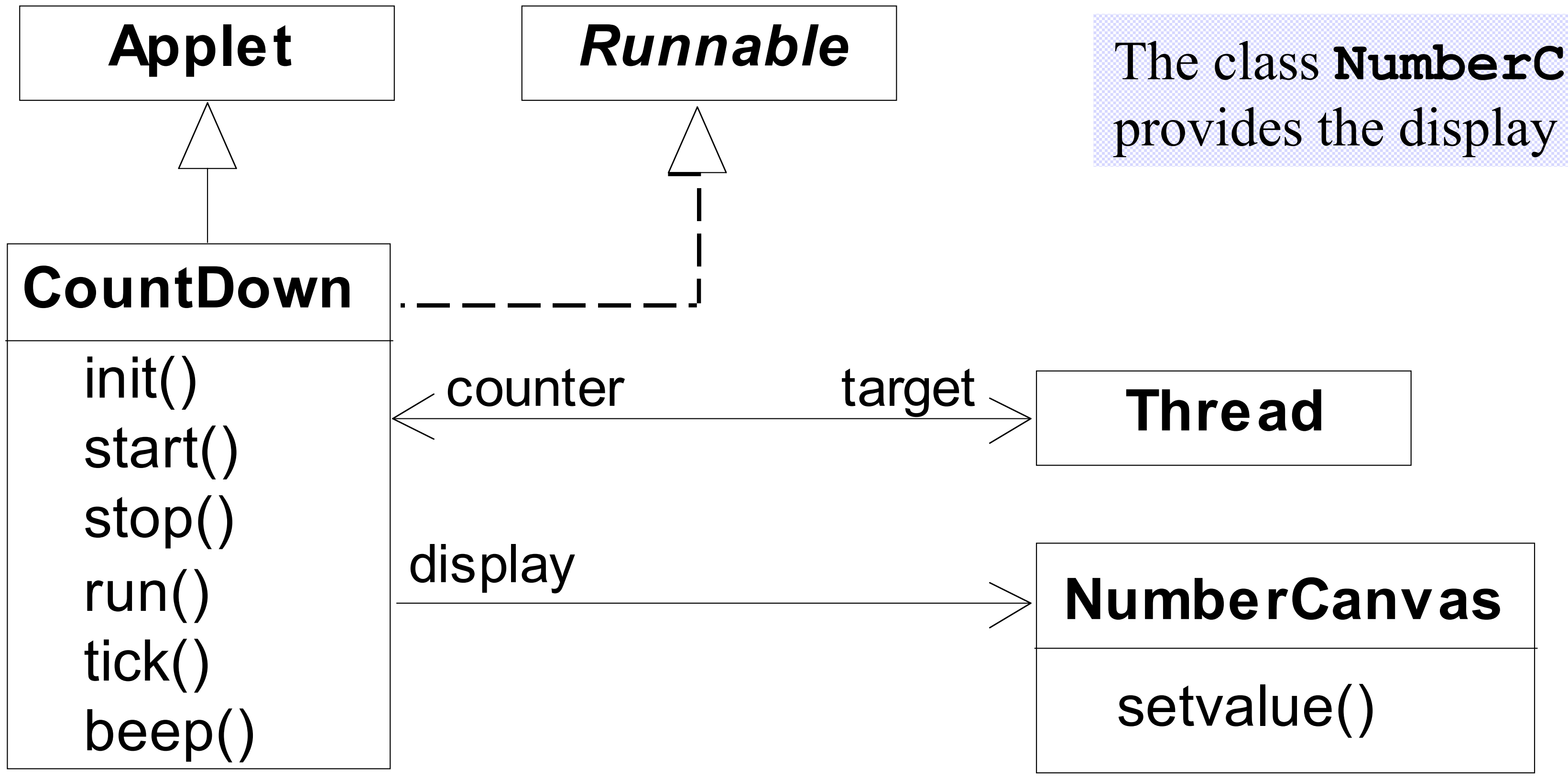
States 0 to 4 correspond to **`CREATED, TERMINATED, RUNNABLE, RUNNING,`** and **`NON-RUNNABLE`** respectively.

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

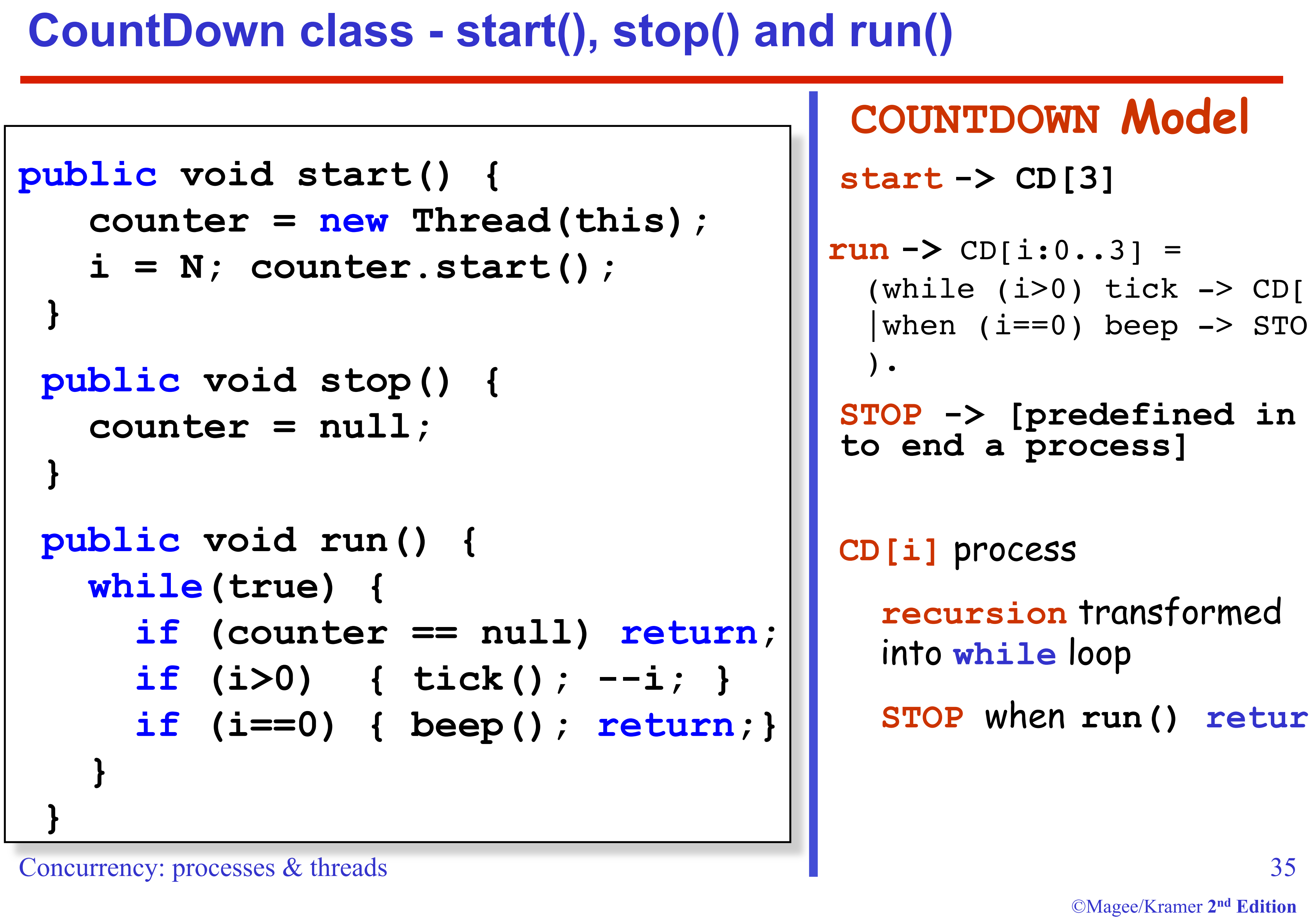


The class **`NumberCanvas`** provides the display canvas.

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

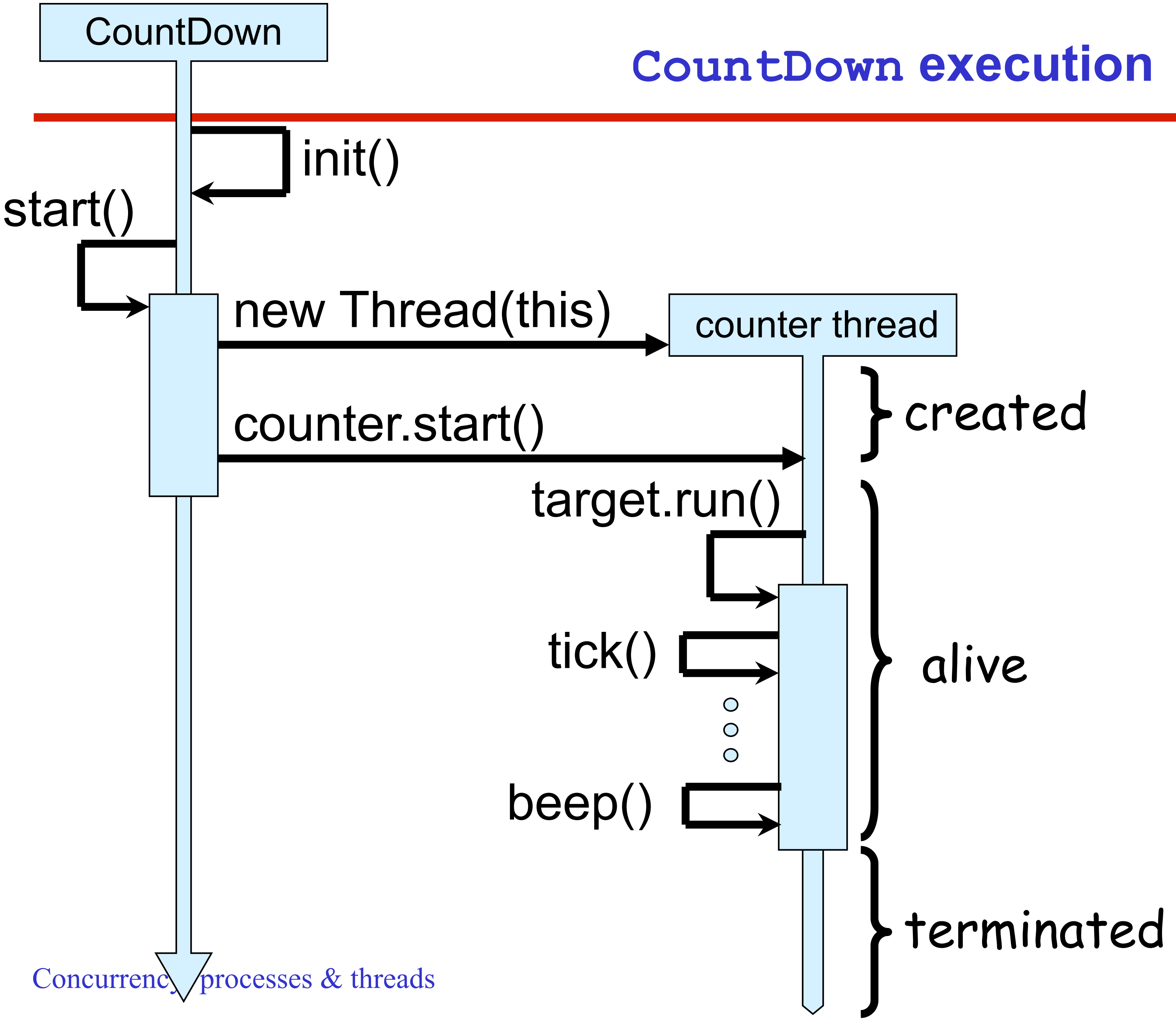
```
start -> CD[3]

run -> CD[i:0..3] =
  (while (i>0) tick -> CD[i-1]
  |when (i==0) beep -> STOP
  ).

STOP -> [predefined in FSP
to end a process]
```

CD[i] process

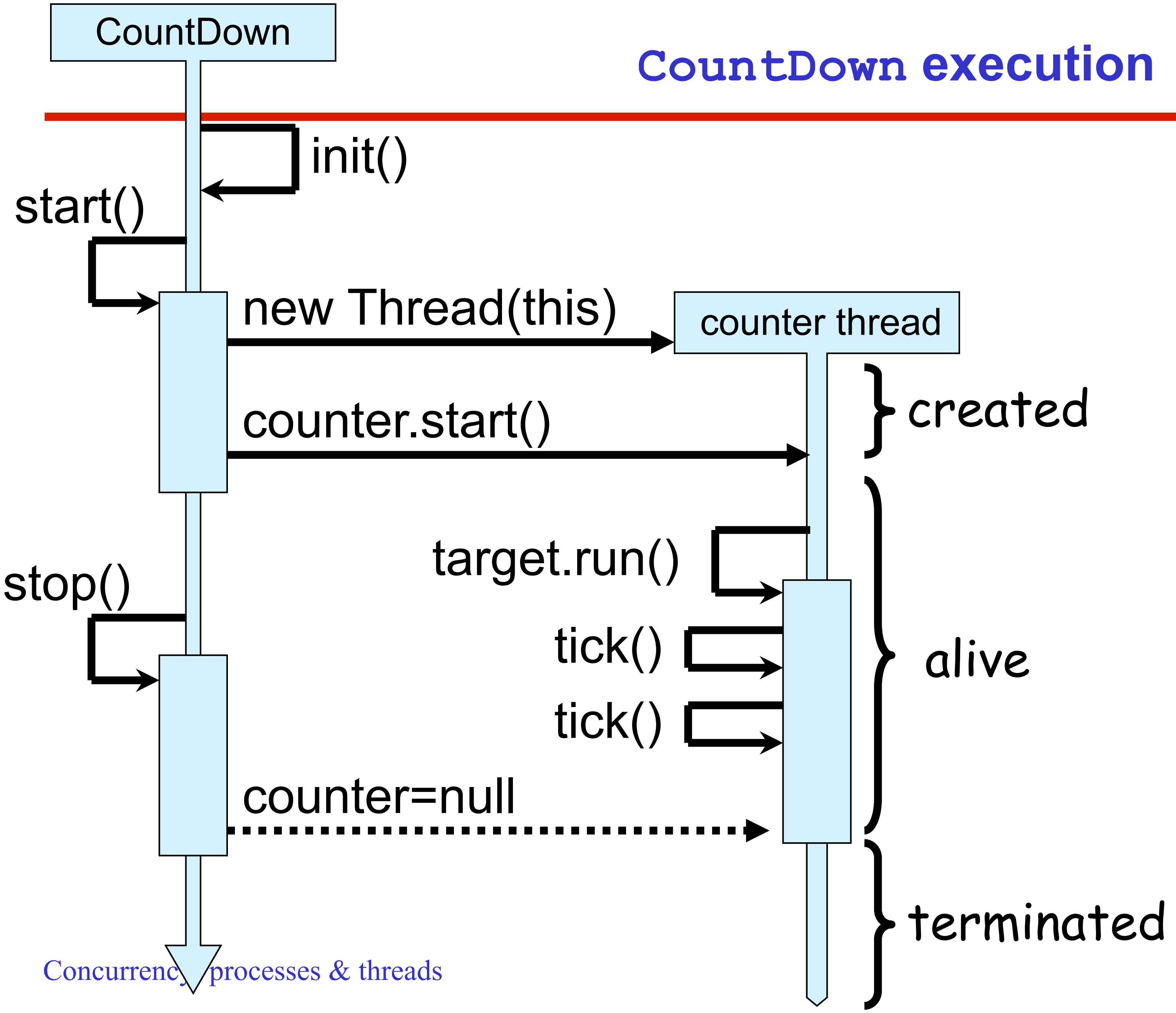
recursion transformed into while loop

STOP when run() returns

`CountDown` execution



`CountDown` execution



Wrapping Up

- Introduced the syntax of FSP and showed how to use it to create finite state machines that model single threaded processes
  - actions, choices, guarded choices, action/process indexes
- Learned about LTSA and how to use it

- In our next lecture, we'll see how to model multiple concurrent processes and their interactions

Coming Up

- Lecture 13: Model-Based Approach to Designing Concurrent Systems, Part 2
- Lecture 14 will be a review for the Midterm
    - Chapters 1-6 of Pilone & Miles
    - Chapters 1-4 of Breshears
    - Lectures 12 and 13