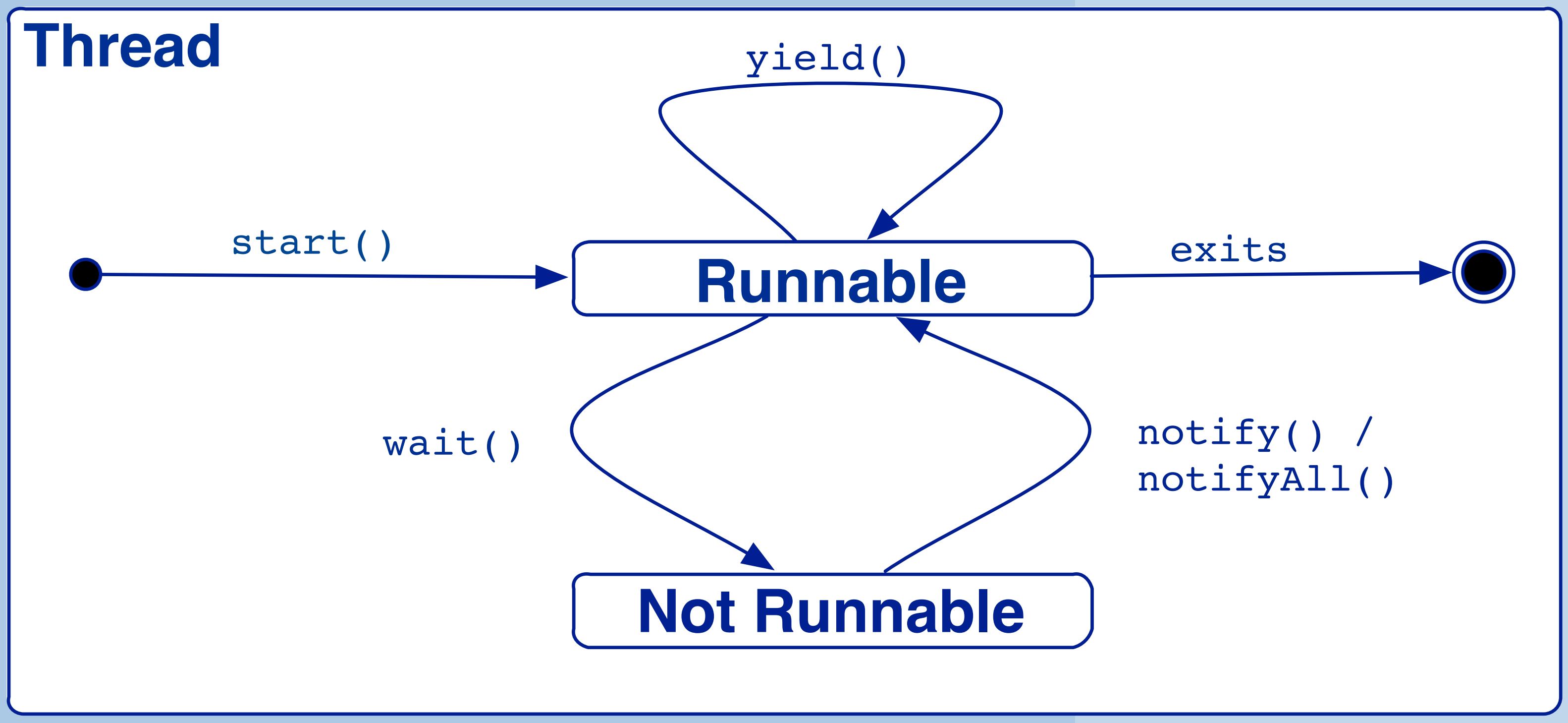
2. Concurrency and Java

Oscar Nierstrasz



Roadmap

> **Threads in Java**
> —Threads (vs. Processes)
> —Thread creation
> —Thread lifecycle

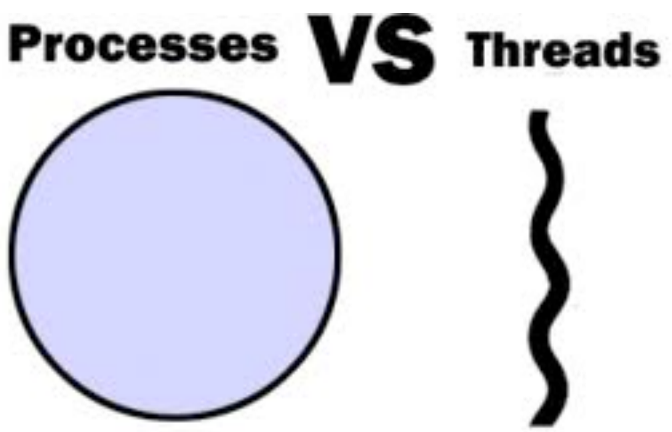
> Synchronization in Java
> —wait() and notify()

> Modelling Concurrency
> —Finite State Processes
> —Labelled Transition Systems

Threads (vs. Processes)

> Processes
  - have their own memory
  - communicate via IPC

> Threads (lightweight processes)
  - exist within a process
  - share memory and open files



Recall: processes are heavyweight and have their own memory; threads are lightweight and share the memory of the process they are contain in.

Within classical compiled languages (like C), threads are typically supported with the help of dedicated libraries. In languages that are compiled to bytecode (like Smalltalk or Java), threads are supported by the virtual machine.

Java’s concurrency model is based on monitors. An object optionally behaves like a monitor, if the `synchronized` keyword is used to synchronize its methods. Threads are created according to the fork and join model

To define a thread extend Thread

and override run()

```
public class Competitor extends java.lang.Thread {

    public Competitor(String name) {
        super(name); // Call Thread constructor
    }

    @Override
    public void run() { // What the thread actually does
        for (int km = 0; km < 5; km++) {
            System.out.println(km + " " + getName());
            try {
                sleep(100);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("DONE: " + getName());
    }
}
```

ThreadDemo

A thread is simply an object that has a `run` method. This method can be inherited from the `Thread` class and overridden, or (next slide) the object can simply implement the `Runnable` interface.

In this example `Competitor` extends `Thread` and overrides the `run` method to print a message every 100 milliseconds.

Source code of all Java examples are available from the git repo:

```
git clone git://scg.unibe.ch/lectures-cp-examples
```

... or implement Runnable

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

Since Java does not support multiple inheritance, it is impossible to inherit from both Thread and another class.

Instead, simply define:

```java
class MTUsefulStuff extends UsefulStuff
    implements Runnable ...
```

and instantiate:

```java
new Thread(new MTUsefulStuff())
```

Instantiating Threads

A Java thread can either *inherit* from java.lang.Thread, or *contain* a Runnable object:

```
public class java.lang.Thread
    extends java.lang.Object
    implements java.lang.Runnable
{
    public Thread();
    public Thread(String name);
    public Thread(Runnable target);
    public Thread(Runnable target, String name);
...
}
```

Example: A thread-based visual clock component


```
public class Clock extends Canvas implements Runnable {
    private Thread clockThread = null;

    public Clock() {
        super();
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }

    public void paintComponent(Graphics g) {
        ...
        String time = dateFormat.format(new Date());
        g2d.drawString(...);
    }
...
```

Clock

In this example, `Clock` needs to extend `Canvas`, so it cannot also extend `Thread`. The solution is to implement the `Runnable` interface.

In order to create and start a new thread in Java, one must instantiate a `Thread` or a subclass of `Thread` and then invoke the `start` method.

The `start` method will execute the `run` method in a new thread. Note that directly invoking the `run` method will *not* start a new thread, but will just execute `run` synchronously within the current thread.

In order to execute a `run` method of an object that implements `Runnable`, one simply passes that object as a parameter to a newly instantiated `Thread`, as in this example. To start a clock thread, a new `Thread` is instantiated with this (the `Runnable` clock object) as a parameter. Then the `start` method is called as usual.

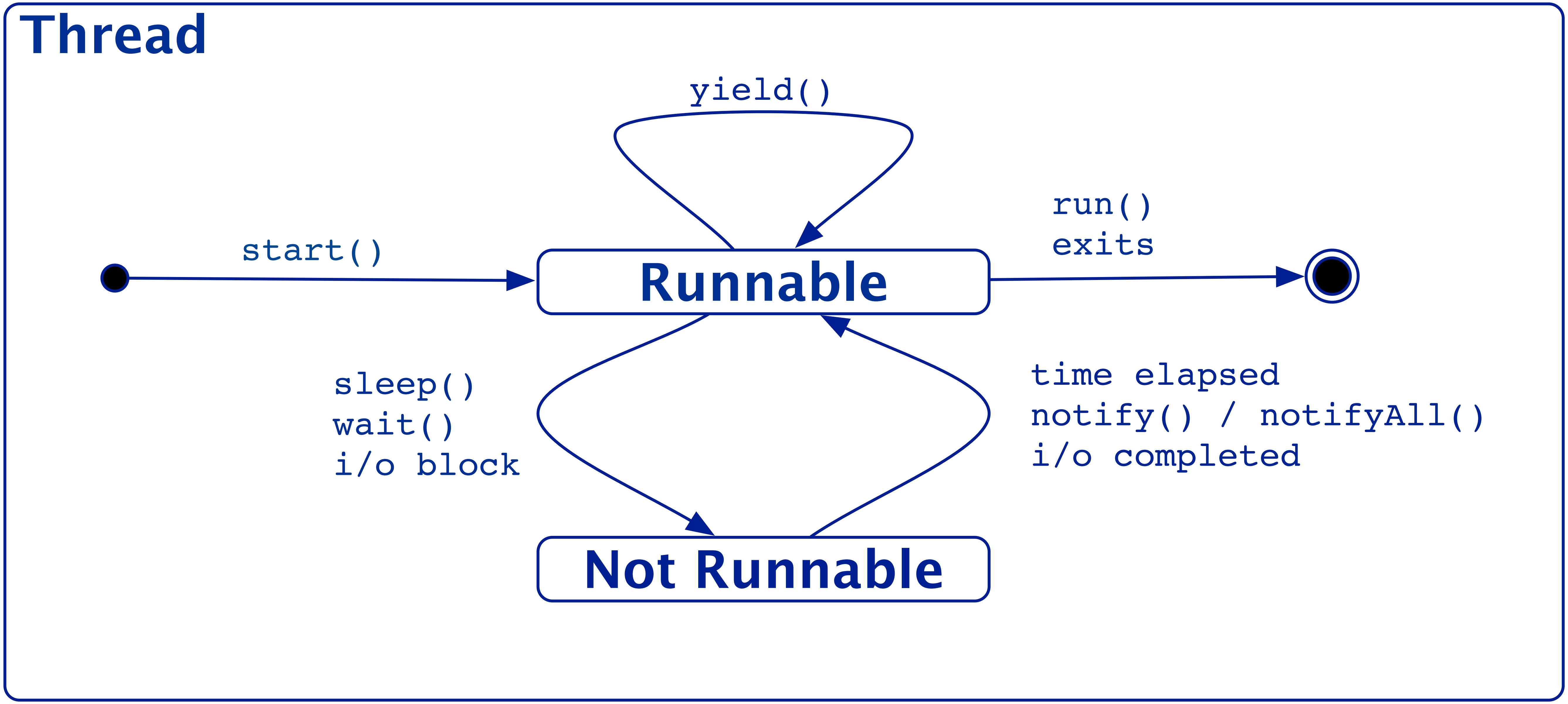
The *run()* repaints the canvas in a loop while the the thread is not null

```
...
    public void run() {
        // stops when clockThread is set to null
        while(Thread.currentThread() == clockThread) {
            repaint();
            try {clockThread.sleep(1000); }
            catch (InterruptedException e){ }
        }
    }

    public void stopThread() {
        clockThread = null;
    }
```

NB: The `stopThread` method is invoked when the stop button is pressed.

After construction...



This UML statechart illustrates the lifecycle of a Java thread. Once a `Thread` instance has been created, nothing happens until the `start` method is called. At that point the thread becomes "*runnable*". This means that the thread scheduler may schedule it to run for a period of time. If the thread explicitly *yields*, then another thread may be scheduled, but the thread remains runnable.

If, on the other hand, the thread is forced to *wait* for something, i.e., a timer event (`sleep`), a monitor synchronization condition (`wait`), or an i/o event (reading or writing), then the thread becomes *not runnable*, until the awaited event occurs.

Finally, if the run method terminates, the thread will exit.

To start a thread... call start()

A Thread's run method is never called directly but is executed when the Thread is *start()-ed:*

```
class Race5K {
     public static void main (String[] args)
{
          // Instantiate and start threads
           new Competitor("Tortoise").start();
           new Competitor("Hare").start();
     }
     …
}
```

The Racing Day!

```
0 Tortoise
0 Hare
1 Hare
1 Tortoise
2 Tortoise
2 Hare
3 Hare
3 Tortoise
4 Tortoise
4 Hare
DONE: Hare
DONE: Tortoise
```

```
0 Hare
0 Tortoise
1 Hare
1 Tortoise
2 Hare
2 Tortoise
3 Hare
3 Tortoise
4 Tortoise
4 Hare
DONE: Tortoise
DONE: Hare
```

Different runs can have different results.

No given ordering of threads is guaranteed by the JVM.

Could the output be garbled? Why? Why not?

```
0 Ha0 Tortoreise
...
```

We see that the outputs of the two threads can be arbitrarily interleaved, but only on a line-by-line basis. Why don't we see individual characters interleaved?

Roadmap

> Threads in Java
—Threads (vs. Processes)
—Thread creation
—Thread lifecycle

> **Synchronization in Java**
—wait() and notify()

> Modelling Concurrency
—Finite State Processes
—Labelled Transition Systems

Synchronization

Without synchronization, an arbitrary number of threads may run at any time within the methods of an object.

—Class invariant may not hold when a method starts!
—So can’t guarantee any post-condition!

A solution: consider *critical sections* that lock access to the object while it is running.

In sequential software, the *class invariant*, which expresses the valid states of instances of the class, is assumed to hold when the object is first created, and before and after every public method.

During a public method (or any internal private or protected method) the invariant may be temporarily invalid, while the object's state is being updated. This poses a problem for concurrent programs. Without synchronization, a thread may enter a public method when another thread is active elsewhere, and the class invariant may not hold, i.e., the object may be in an inconsistent state. As a consequence, the second method is likely to produce incorrect results.

See also: https://en.wikipedia.org/wiki/Class_invariant

Critical Section...

- ... a piece of code that accesses a shared resource (e.g. memory location) that must not be concurrently accessed by multiple threads.
- *This works as long as methods cooperate in locking and unlocking access!*

```
/* This is the critical section object
(statically allocated). */
static pthread_mutex_t cs_mutex =
PTHREAD_MUTEX_INITIALIZER;

void f()
{
    /* Enter the critical section
    -- other threads are locked out */
    pthread_mutex_lock( &cs_mutex );

    /* Do some thread-safe processing! */

    /*Leave the critical section
    -- other threads can now
    pthread_mutex_lock()  */
    pthread_mutex_unlock( &cs_mutex );
}
```

Critical section in C/C++ on Unix/Linux

A *critical section* is a piece of code that accesses (reads or writes) shared resources. Inconsistencies may arise if two threads try to access the same resources during overlapping critical sections.

A common solution is to guarantee mutual exclusion with the help of locks or semaphores. Of course *all* critical sections must be protected. If even a single critical section is not protected, two threads entering the same critical section can interfere.

Synchronized blocks

Either: synchronize an individual block within a method with respect to a shared resource:

```
public Object aMethod() {
    // unsynchronized code
    ...
    synchronized(resource)
    { // lock resource
        ...
    } // unlock resource
    ...
}
```

Critical sections can be protected in Java by declaring the relevant code as a `synchronized` block. Such a block takes as a parameter the object that is accessed as a shared resource. The resource is *locked* when the block is entered, and *unlocked* when the block ends.

Note that only a *single resource* can be passed as an argument to the synchronized block, even if multiple resources are accessed. If there are multiple critical sections, it is important that the same resource be locked to ensure consistency.

Aside: actually, an arbitrary object be used as the lock, not necessarily one of the resources accessed.

Synchronized methods

Or: declare an entire method to be *synchronized* with other synchronized methods of **`this`** object:

```
public class PrintStream extends FilterOutputStream {
    ...
    public synchronized void println(String s);
    public synchronized void println(char c);
    ...
}
```

Note: synchronized methods are a particular case of synchronizing on *this* object

It is also possible to declare an *entire method* as `synchronized`. This is just syntactic sugar for declaring the entire body of the method as a `synchronized` block, with `this` object as the shared resource to be locked.

Note that by declaring *all* public methods as being synchronized, you effectively turn the object into a *monitor*.

wait() and notify()

Synchronization must sometimes be interrupted:

```java
public class Account {
    protected long assets = 0;
    public synchronized void withdraw(int amount) {
        while (amount > assets) {
            try {
                wait();
            } catch(InterruptedException e) { }
        }
        assets -= amount;
    }
    public synchronized void deposit(int amount) {
        assets += amount;
        notifyAll();
    }
}
```

Account

NB: you must either catch or throw `InterruptedException`

This `Account` class is a *monitor*, with methods `withdraw` and `deposit` being critical sections accessing the `assets` variable as a shared resource.

Since both methods are synchronized, it is guaranteed that, no matter how many threads are attempting to execute either method, at most one thread will be running within the monitor at any point in time.

This example follows a classical structure. The `withdraw` method may not proceed if the requested amount is greater than the available assets. The method therefore tests this *guard condition* in a while loop. As long as the guard fails, the thread will wait, releasing the monitor lock, and making the thread *not runnable*.

Eventually (we hope) some thread will `deposit` some cash. At this point `notifyAll` is called (Java's equivalent to *signal*), which will cause all threads waiting on `this` object to be made runnable. Awakened threads will again check the guard condition, and possibly proceed or wait again.

What could go wrong if withdraw were to use an if statement instead of a while loop?

A waiting thread may receive an `InterruptedException`, so it is advisable to always wrap calls to wait within a try-catch block. It is not a good idea to leave the body of the catch clause empty (you should decide what is the appropriate action), but for the example programs we will generally ignore this issue.

wait and notify in action ...

```
Waiting to withdraw 100 units ...
Depositing 200 units ...
I deposited 200 units!
I withdrew 100 units!
```

```java
final Account myAccount = new Account();

new Thread() { // Withdrawing
    public void run() {
        int amount = 100;
        System.out.println("Waiting to withdraw"+amount+ "units...");
        myAccount.withdraw(amount);
        System.out.println("I withdrew " + amount + " units!");
    }
}.start();

Thread.sleep(1000); ...

new Thread() { // Depositing
    public void run() {
        int amount = 200;
        System.out.println("Depositing " + amount + " units ...");
        myAccount.deposit(amount);
        System.out.println("I deposited " + amount + " units!");
    }
}.start();
```

java.lang.Object

NB: wait() and notify() are methods rather than keywords:

```
public class java.lang.Object
{
    ...
    public final void wait()
        throws InterruptedException;
    public final void notify();
    public final void notifyAll();
    ...
}
```

While `synchronized` is a keyword, `wait` and `notify` are methods defined in the class `Object`, i.e., the root of the Java class hierarchy, and are thus inherited by all classes.

The difference between `notify` and `notifyAll` is that the latter will wake up *all* waiting threads, while the former will only wake up one. As we shall see in a later lecture, it is generally a bad idea to use `notify` instead of `notifyAll`. (You risk waking up the wrong thread.)

NB: `wait` and `notify` may *only* be called within a `synchronized` block. Attempting to do otherwise will raise an `IllegalMonitorException`. (This is a common beginner's mistake.)

Roadmap

> Threads in Java
—Threads (vs. Processes)
—Thread creation
—Thread lifecycle

> Synchronization in Java
—wait() and notify()

> **Modelling Concurrency**
—Finite State Processes
—Labelled Transition Systems

Non-determinism

> Multiple threads are rotated by the processor(s)
> A thread might be interrupted at any time
> No two runs are guaranteed to be the same

Modelling Concurrency

Because concurrent systems are ***non-deterministic***, it can be difficult to build them and reason about their properties.

A model is an *abstraction of the real world* that makes it easier to focus on the points of interest.

Approach:

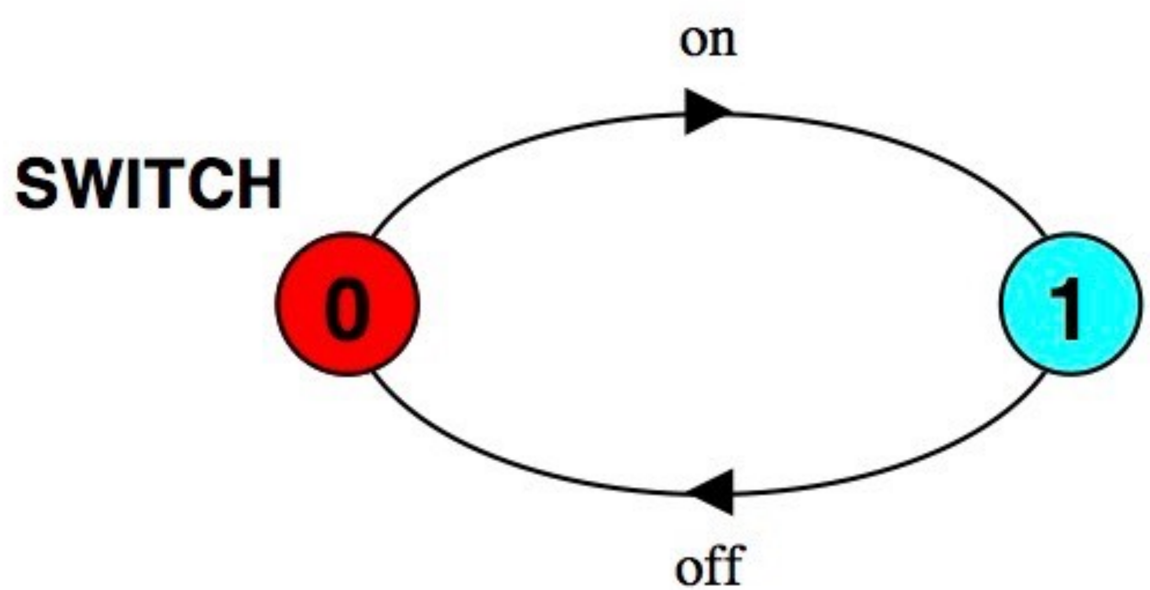
Model concurrent systems as *sets of sequential finite state processes*

Finite State Processes

FSP is a *textual notation* for specifying a finite state process:

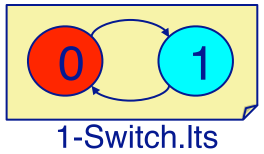
```
SWITCH = (on -> off -> SWITCH).
```

LTS (labeled transition system) is a *graphical notation* for interpreting a processes as a labelled transition system:



The *meaning* of a process is a set of possible *traces* :

`on → off → on → off → on → off → on ...`



FSP and LTS are two complementary notations for representing finite state processes.

FSP is textual language for specifying processes as sequences of actions with choices, while LTS is a graphical notation that resembles finite state automata. In both cases, the semantics of the notations is a set of possible traces, i.e., sequences of actions.

`SWITCH` is a process that can participate in the action `on`, then in `off`, and then returns to its initial state. Note how process names start with an upper case letter and actions are in lower case. The examples can all be run with the LTSA tool. A snapshot of the tool and all examples from this course can be found in the examples repo:

```
git clone git://scg.unibe.ch/lectures-cp-examples
```

For the latest version of the tool, visit the LTSA web site:

https://www.doc.ic.ac.uk/ltsa/

FSP Summary

Action prefix	`(x->P)`	***Parallel composition***	`(P||Q)`
Choice	`(x->P|y->Q)`	***Replicator***	`forall [I:1..N] P(I)`
Guarded Action	`(when B x->P|y->Q)`	***Process labelling***	`a:P`
Alphabet extension	`P + S`	***Process sharing***	$\{a_1,...,a_n\}$`::P`
Conditional	`x-> If B then P else Q`	***Priority High***	`||C=(P||Q)<<`$\{a_1,…,a_n\}$
Relabelling	`/`$\{new_1/old_1,…\}$	***Priority Low***	`||C=(P||Q)>>`$\{a_1,…,a_n\}$
Hiding	`\`$\{a_1,…,a_n\}$	***Safety property***	`property P`
Interface	`@`$\{a_1,…,a_n\}$	***Progress property***	`progress P = `$\{a_1,…,a_n\}$

We will encounter and use these features in the lectures to come …

This slide is provided both as a quick reference and as an overview to give a flavour of what the FSP language looks like. Each of these operators can be used to compose processes or modify processes, yielding a new process each time.

We will encounter many of these operators in later lectures, and explain them in detail when necessary.

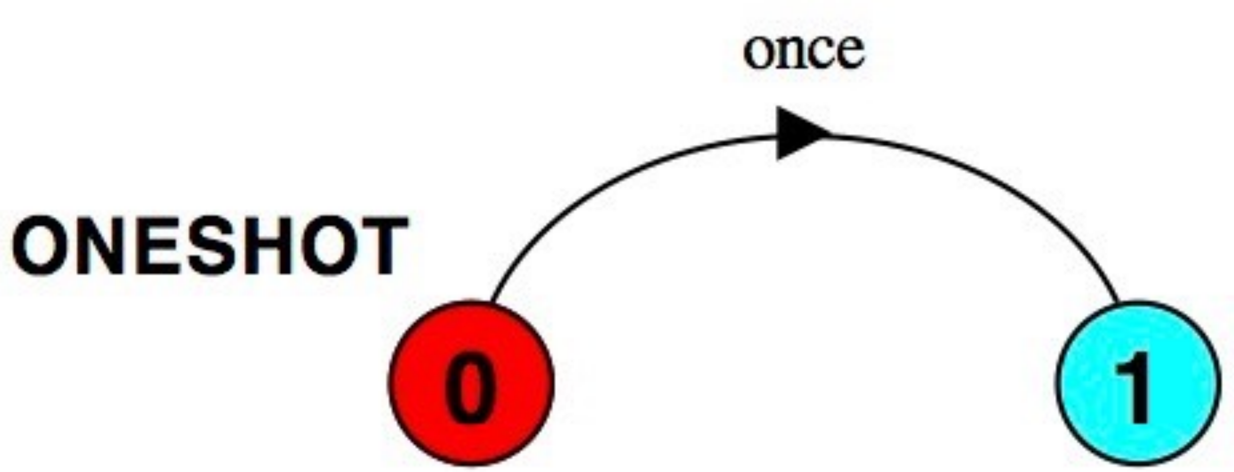
For a brief explanation, see the on-line appendix:

https://www.doc.ic.ac.uk/~jnm/book/ltsa/Appendix-A-2e.html

FSP — Action Prefix

If `x` is an action and `P` a process then `(x->P)` is a process that initially engages in the action `x` and then behaves like `P`.

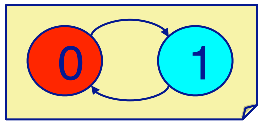
```
ONESHOT = (once -> STOP).
```



A terminating process

Convention:

Processes start with `UPPERCASE`, actions start with `lowercase`.



2-OneShot.lts

The most basic operator in FSP is *action prefix*. Here the process `ONESHOT` may participate in the action `once`, turning into the process `STOP`.

`STOP` is a predefined process that does nothing.

Note that a process expression in FSP also represents the *state* of a process. In the LTS graph of the process `ONESHOT`, state (0) is the same as `ONESHOT` while state (1) is the same as `STOP`.

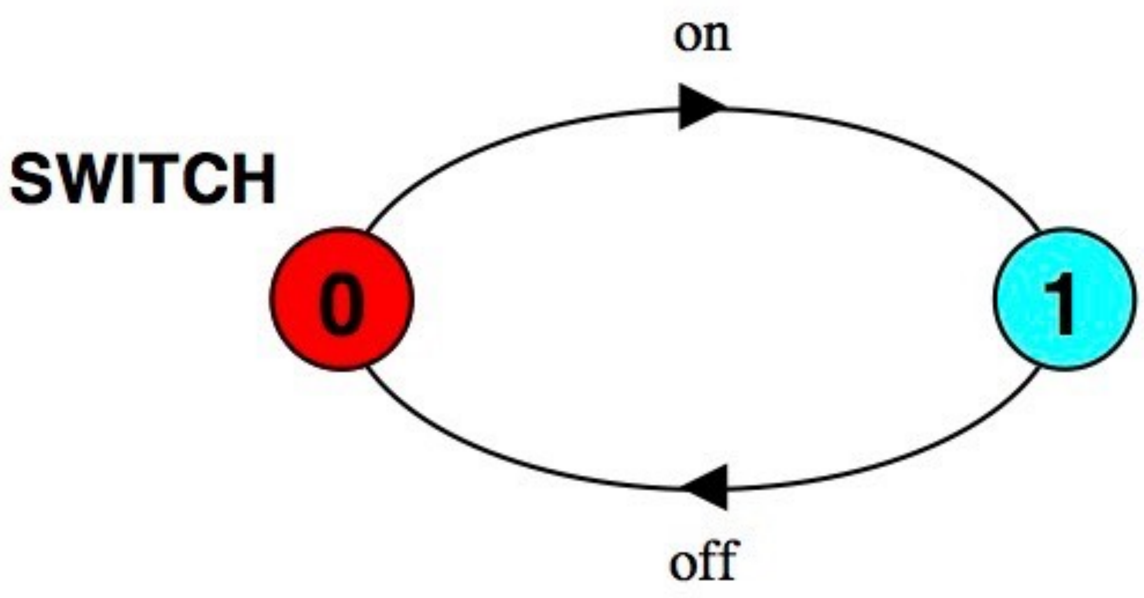
The *meaning* of the process `ONESHOT` is the trivial set of traces:

{ `once` }.

FSP — Recursion

Repetitive behaviour uses recursion:

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



A recursively-defined process is one that, following a particular sequence of actions, may return to its initial state.

The process `SWITCH` starts in the state (process) `OFF`, then transitions to `ON`, and back again. In the LTS diagram, `OFF` is the same as state (0) and `ON` is state (1).

Note the syntax: `SWITCH` is defined as a comma-separated sequence of process definitions, terminated with a period.

The meaning of `SWITCH` is the singleton set of traces

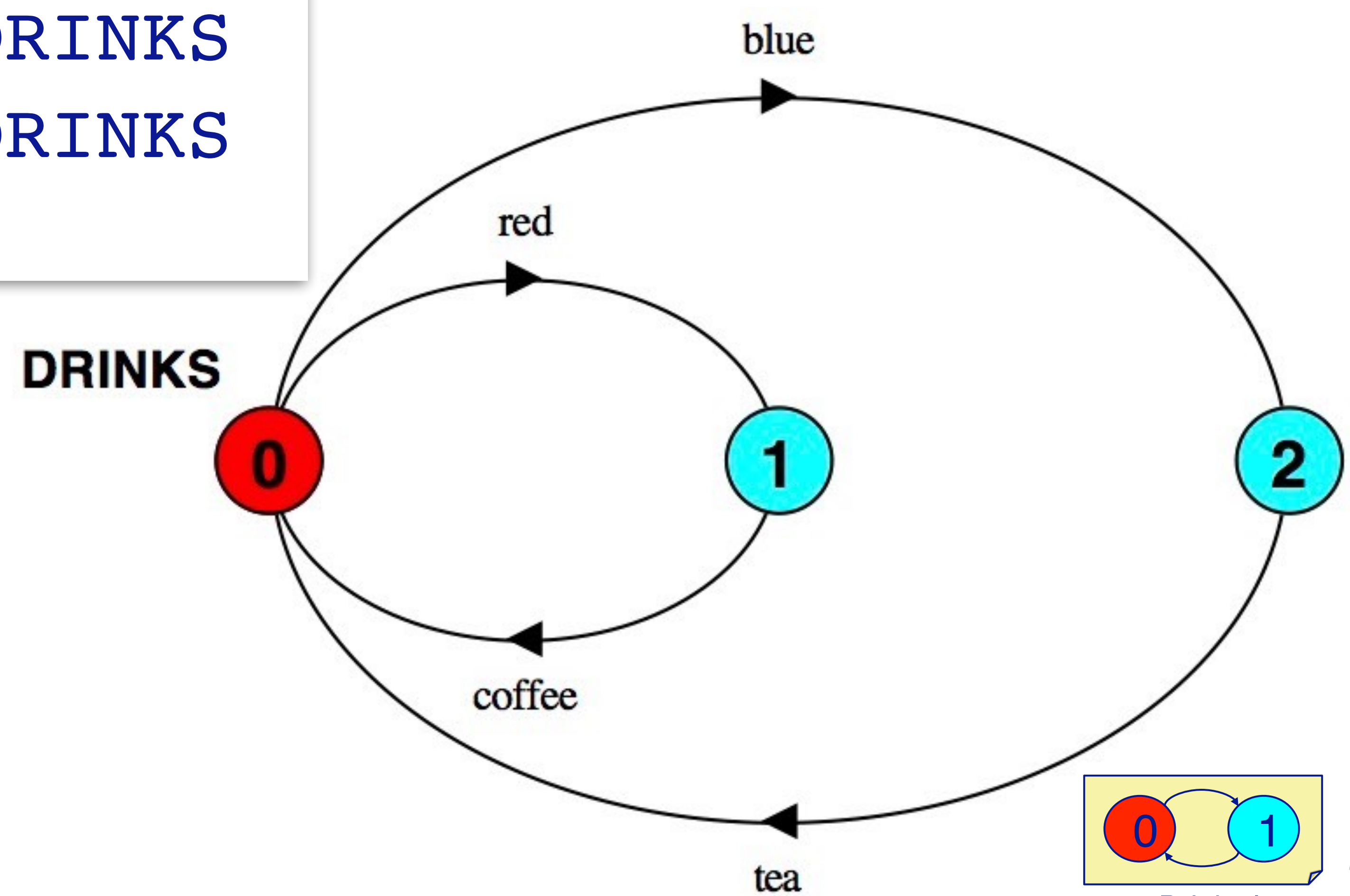
{ on off on off ... }

FSP — 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`.

If `x` occurs, the process then behaves like `P`; otherwise, if `y` occurs, it behaves like `Q`.

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



What are the possible traces of `DRINKS`?

3-Drinks.lts

A process that can only follow a fixed sequence of actions is not very interesting. The *choice operator* allows a process to follow two possible different paths.

Note that the actions that `DRINKS` may participate in are *mutually exclusive*. If the red button is pushed (action “`red`”), then the drinks machine deterministically produces a cup of `coffee`.

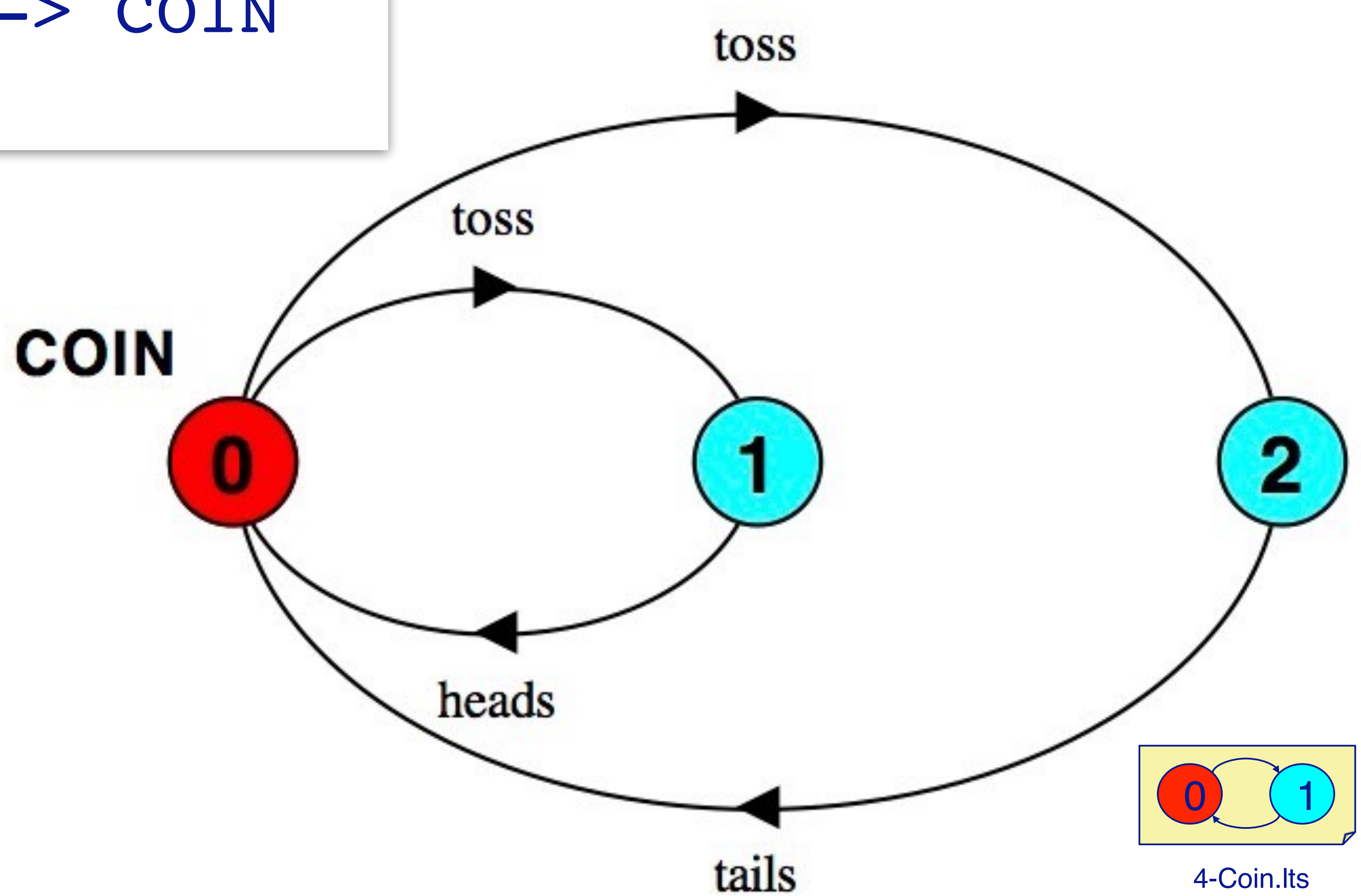
Note how the notion of an “action” in FSP (“transition” in LTS) can be used to model different kinds of events (pushing a button, producing a cup of coffee or tea). Within FSP/LTS, there is no notion of a direction of an interaction between processes, but there may be in the world that is being modeled.

What are the possible traces of the `DRINKS` machine?

FSP — Non-determinism

`(x->P | x->Q)` performs `x` and then behaves as either `P` or `Q`.

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



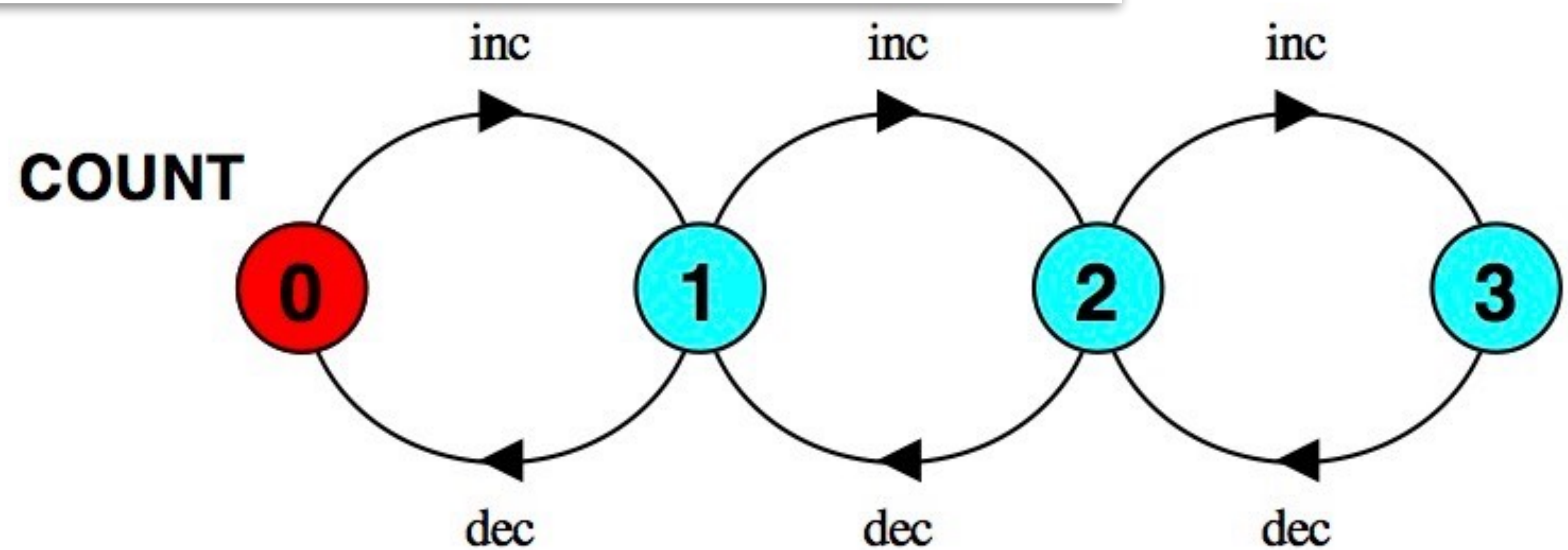
4-Coin.lts

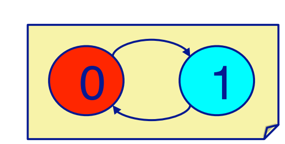
This example is very similar to the `DRINKS` machine, except that the choices here are *not* mutually exclusive. A `COIN` can be tossed, at which point it non-deterministically moves either to the state `heads->COIN` or `tails->COIN`.

FSP — Guarded actions

`(when B x->P | y->Q)` means that *when the guard* `B` *is true* then *either* `x` or `y` may be chosen; otherwise if `B` is false then only `y` may be chosen. (default case is optional)

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





5-Count.lts

Guards can be used as a high-level way to define more complex processes. It is important to realize that these guards are actually static, not dynamic. This means that the “variables” and guards can be macro-expanded.

This example is equivalent to:

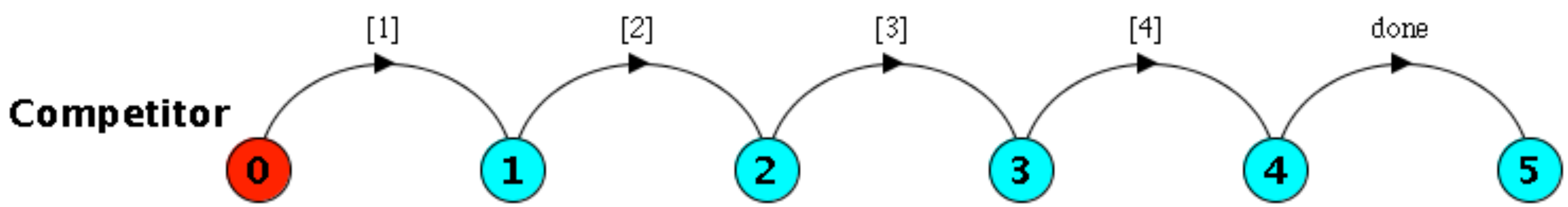
```
COUNT = COUNT0,
COUNT0 = ( inc -> COUNT1 ),
COUNT1 = ( inc -> COUNT2 | dec -> COUNT0 ),
COUNT2 = ( inc -> COUNT3 | dec -> COUNT1 ),
COUNT3 = ( dec -> COUNT2 ).
```

Clearly the version with macro variables and guards is easier to read and write.

Competitor FSP

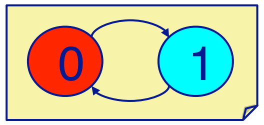
Competitor can be modelled as a single, sequential, finite state process:

```
Competitor =([1]->[2]->[3]->[4]-> done-> STOP).
```



Or, more generically:

```
const N       = 4
COMPETITOR = KM[0],
KM[n:0..N] = ( when (n<N) [n+1] -> KM[n+1]
             | when(n==N) done -> STOP).
```



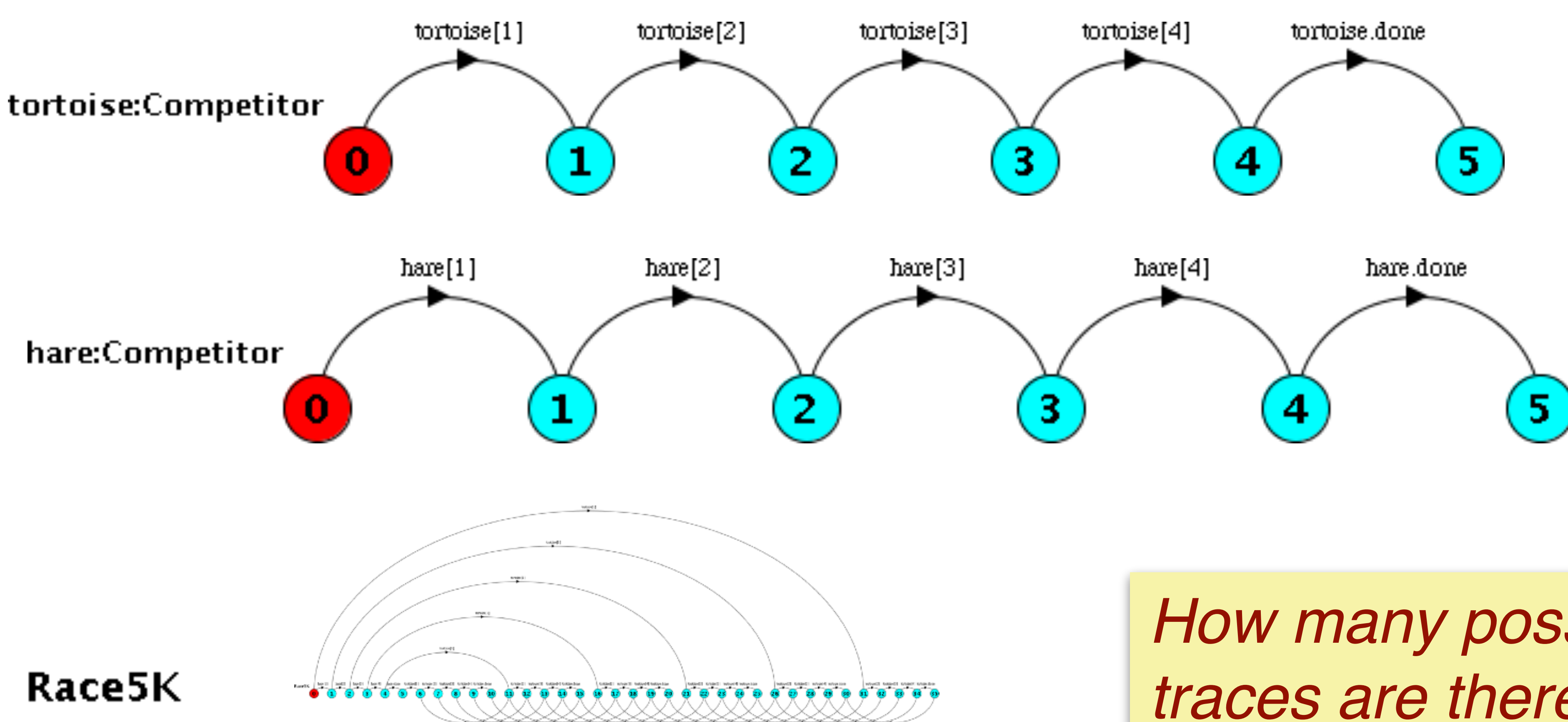
6-Competitor.lts

Here we model the `Competitor` Java class we saw earlier in the lecture. From the process modeling perspective, the only interesting point is that every kilometer, the `Competitor` announces its progress.

FSP — Concurrency

We can *relabel* the transitions of `Simple` and concurrently *compose* two copies of it:

```
||Race5K = (tortoise:Competitor || hare:Competitor).
```



How many possible traces are there?

There are two things going on in this example.

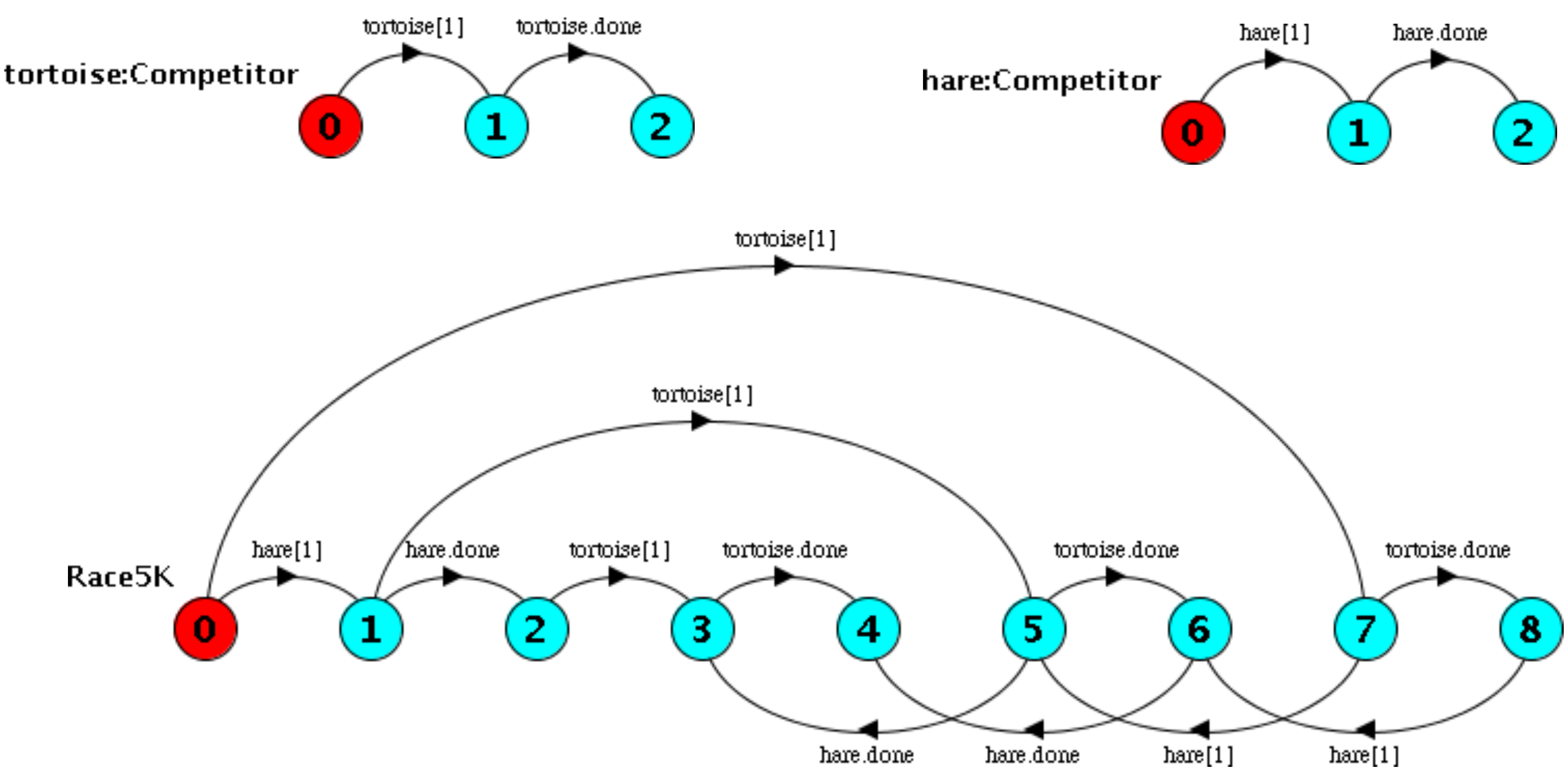
First we *prefix* two instances of `Competitor`, one with "`tortoise`" and the other with "`hare`". This has the effect of *relabeling all the actions* of each process, so `[1]` becomes `tortoise[1]`, and so on. These relabeled events are *independent* of each other, as their names are different.

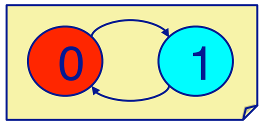
Second, we *concurrently compose* these two processes with the `||` operator. (Note that this is different from the choice operator `|` we saw earlier.)

These two processes may now proceed independently, *arbitrarily interleaving* their actions. (In a later lecture we will see how two processes may synchronize with each other if they share common actions.)

FSP — Composition

If we restrict ourselves to two steps, the composition will have nine states:





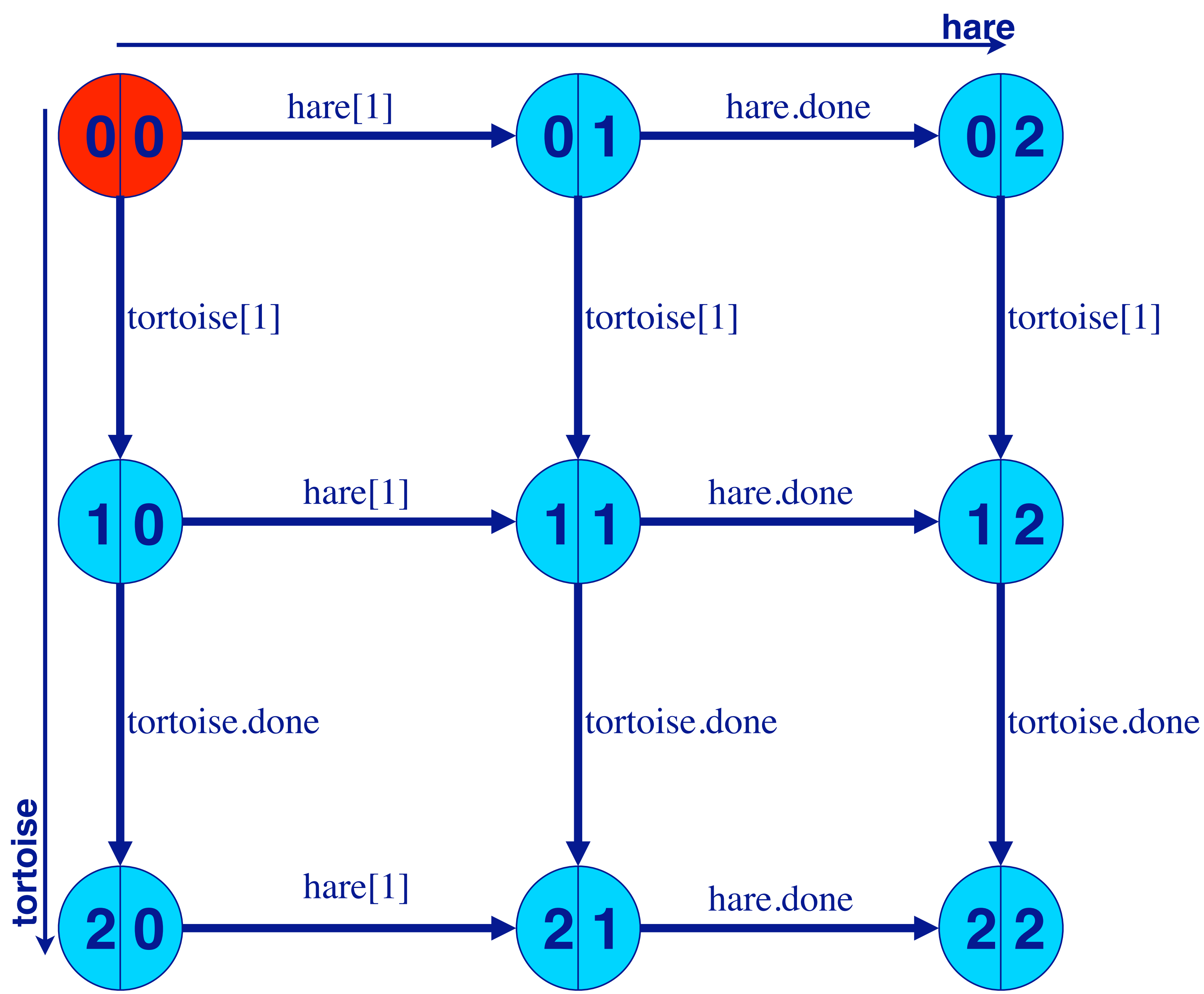
6-Competitor.lts

Try this example with LTSA. Change the number of iterations from 5 to 2 to get this result.

How does LTSA decide which are the shared states of the composed processes?

Composition state space

The state space of two composed processes is (at most) the Cartesian product of the individual state spaces

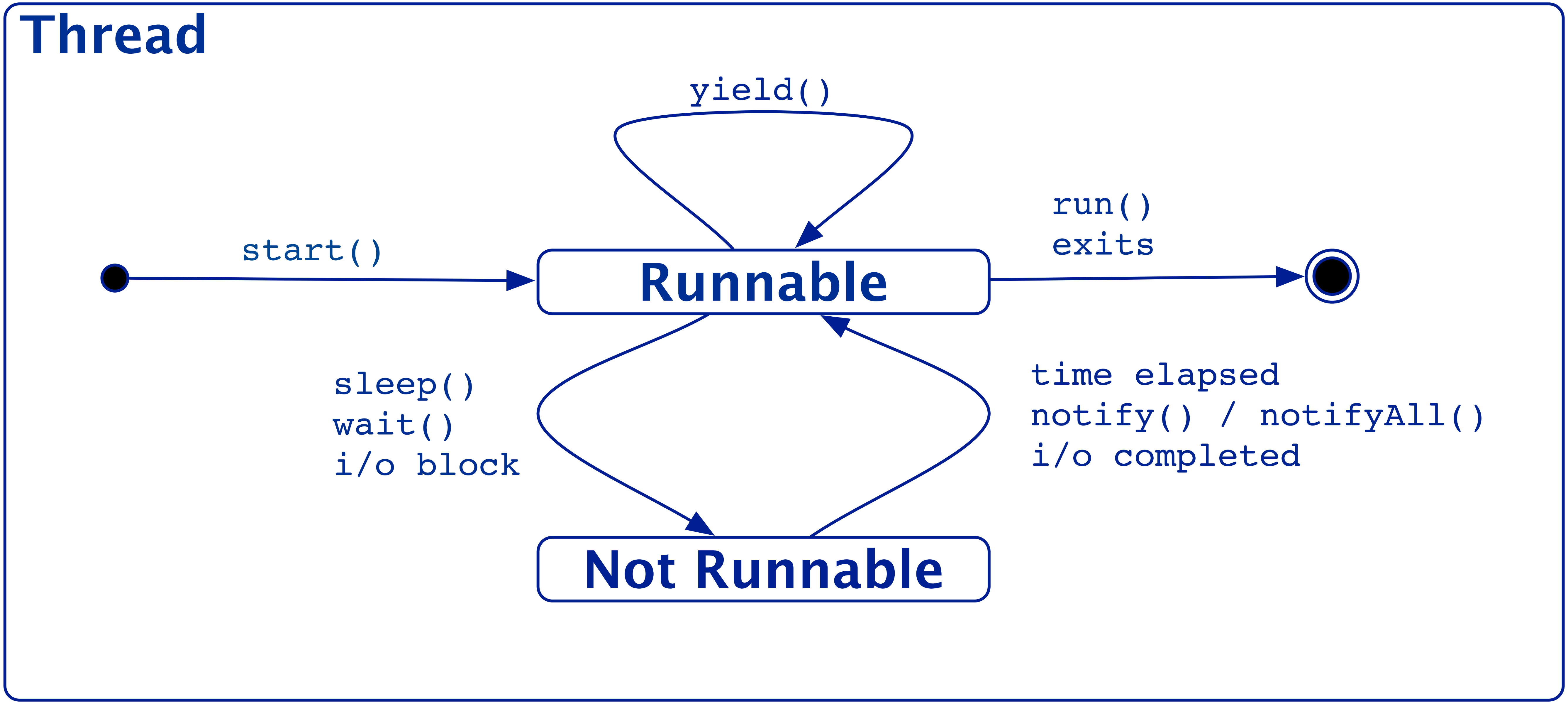


If two independent process are composed, the resulting state space is potentially the Cartesian product of the two state spaces. Since the tortoise and the hare do not interact, we obtain 9 states.

(Later we will see that if composed processes need to synchronize, the resulting state space may be drastically reduced.)

Compare this diagram with the one in the previous slide produced by the LTSA tool.

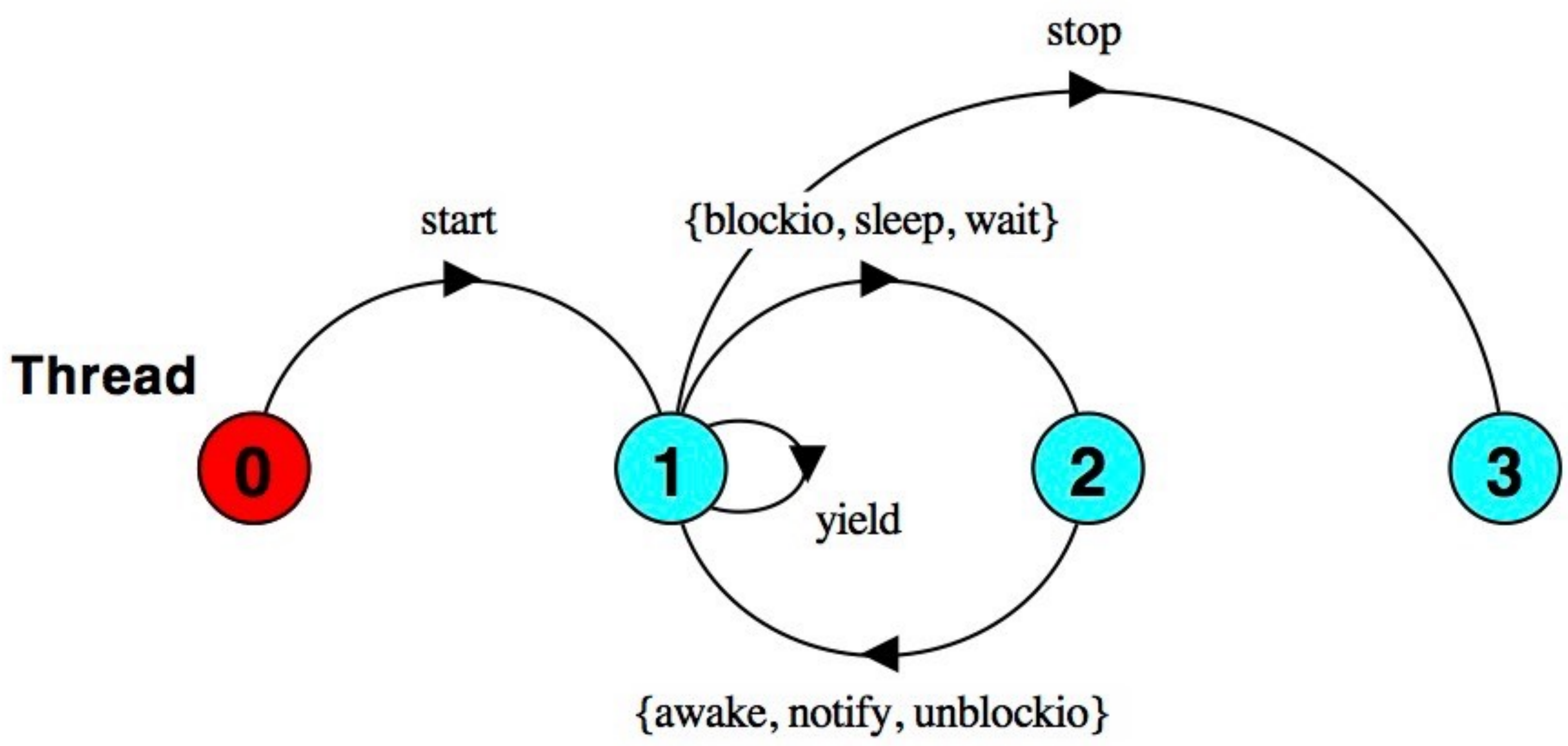
Transitions between Thread States

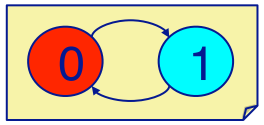


Recall the UML statechart describing the possible state transitions of a Java thread. We can easily model this with FSP.

LTS for Thread States

```
Thread = ( start -> Runnable ),
Runnable =
     ( yield -> Runnable
     | {sleep, wait, blockio} -> NotRunnable
     | stop -> STOP ),
NotRunnable =
     ( {awake, notify, unblockio} -> Runnable ).
```





7-Thread.lts

The only new feature of FSP here is the ability to specify a set of actions in a transition to a new process state.

What you should know!

- *What are finite state processes?*
- *How are they used to model concurrency?*
- *What are traces, and what do they model?*
- *How can the same FSP have multiple traces?*
- *How do you create a new thread in Java?*
- *What states can a Java thread be in?*
- *How can it change state?*
- *What is the Runnable interface good for?*
- *What is a critical section?*
- *When should you declare a method to be synchronized?*

Can you answer these questions?

> How do you specify an FSP that repeatedly performs hello, but may stop at any time?
> How many states and how many possible traces does the full Race5K FSP have?
> When should your class inherit from Thread?
> How can concurrency invalidate a class invariant?
> What happens if you call wait or notify outside a synchronized method or block?
> When is it better to use synchronized blocks rather than methods?