Process Oriented Design for Java - Concurrency for All

Peter Welch (p.h.welch@kent.ac.uk)
Computing Laboratory, University of Kent at Canterbury

Co631 (Concurrency Design and Practice)

Motivation and Applications

- **Thesis**
  - Natural systems are robust, efficient, long-lived and continuously evolving. ***We should take the hint!***
  - Look on concurrency as a ***core design mechanism*** – not as something difficult, used only to boost performance.
- **Some applications**
  - Hardware design and modelling.
  - Static embedded systems and parallel supercomputing.
  - Field-programmable embedded systems and dynamic supercomputing (e.g. **SETI-at-home**).
  - Dynamic distributed systems, eCommerce, operating systems and games.
  - Biological system and ***nanite*** modelling.

Nature is not organised as a single thread of control:

```
joe.eatBreakfast ();
sue.washUp ();
joe.driveToWork ();
sue.phone (sally);
US.government.sue (bill);
sun.zap (office);
```

???

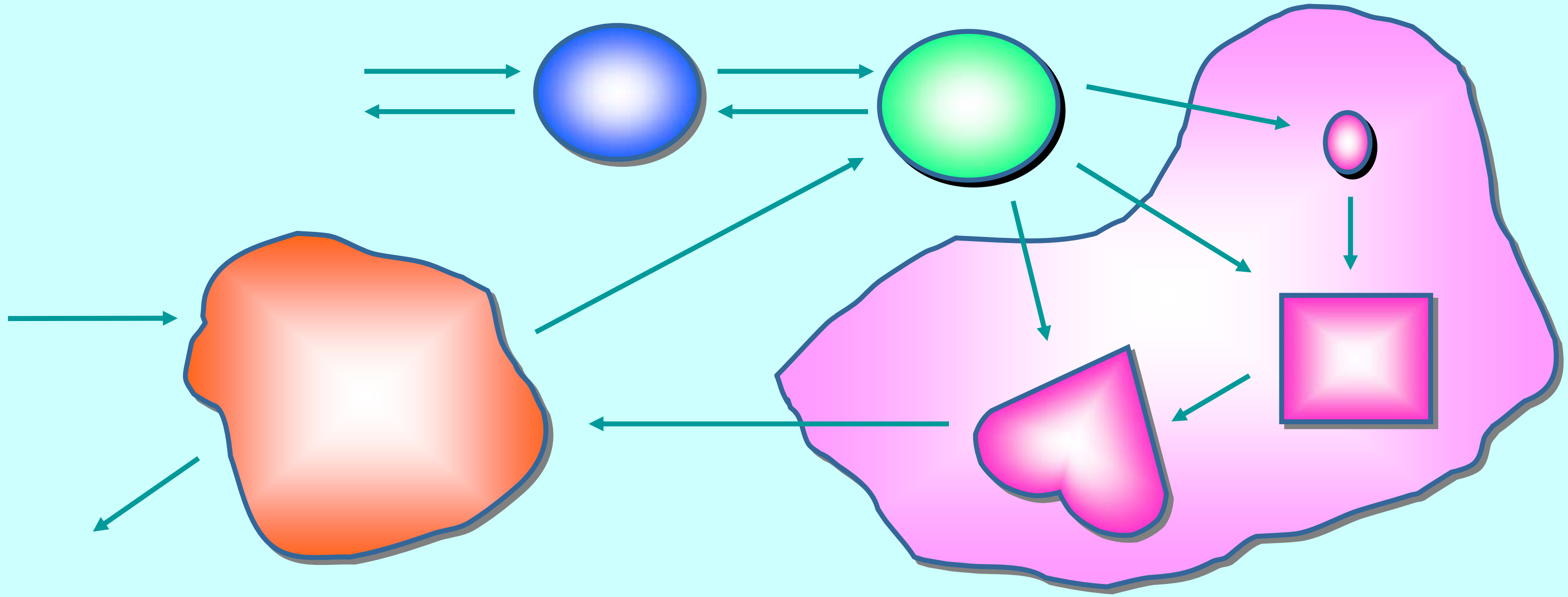
Nature is not bulk synchronous:

```
bill.acquire (everything);
bill.invent (everything);
bill.run (the.NET);
bill.anti (trust);
bill.invade (canada);
UNIVERSE.SYNC ();
```

???

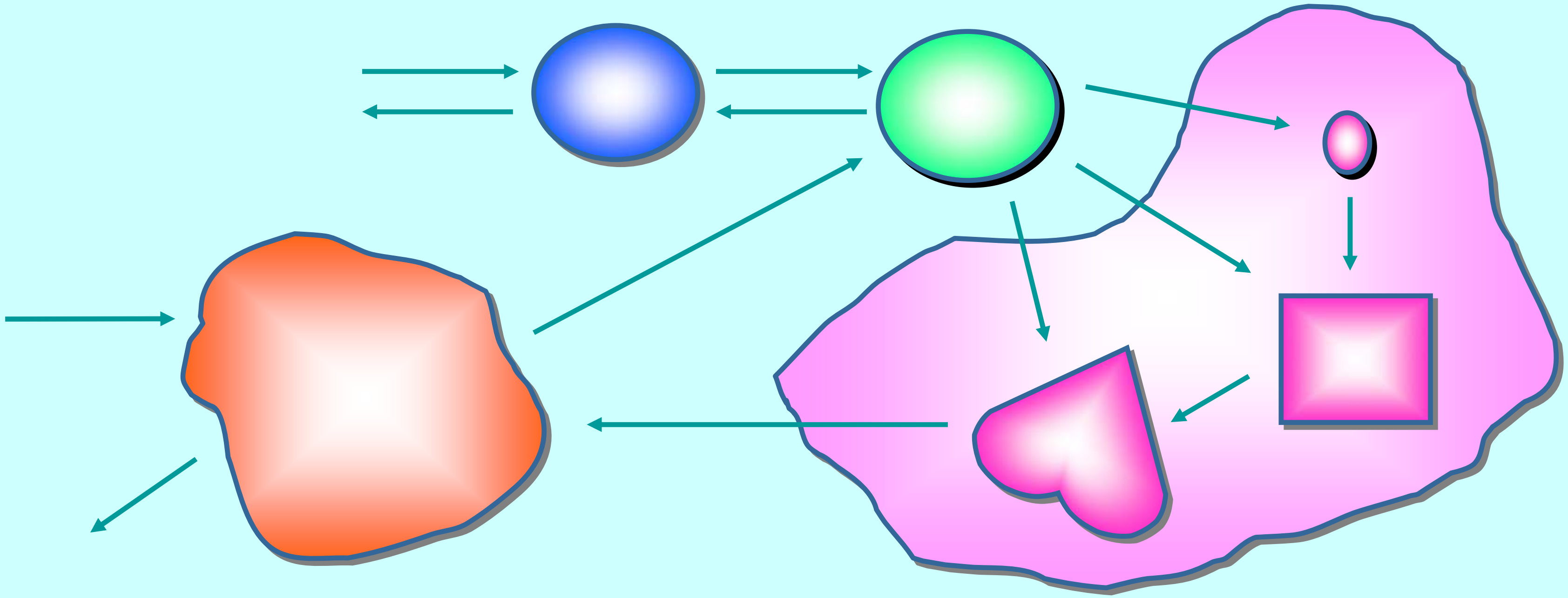
Nature has very large numbers of independent agents, interacting with each other in regular and chaotic patterns, at all levels of scale:

… nannite … human … astronomic ...



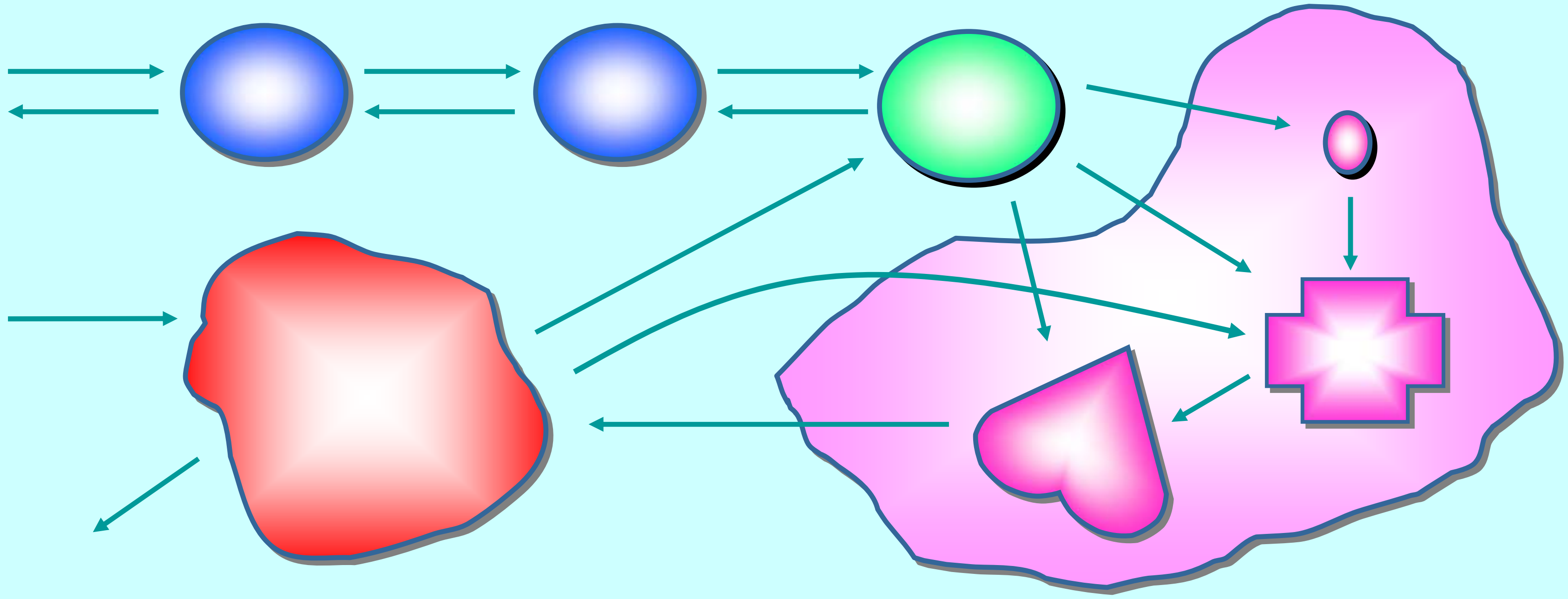
The networks are dynamic: growing, decaying and mutating internal topology (in response to environmental pressure and self-motivation):

… nannite … human … astronomic ...



The networks are dynamic: growing, decaying and mutating internal topology (in response to environmental pressure and self-motivation):

… nanite … human … astronomic ...



The Real World and Concurrency

Computer systems - to be of use in this world - need to model that part of the world for which it is to be used.

If that modeling can reflect the natural concurrency in the system … it should be ***simpler***.

Yet concurrency is thought to be an ***advanced*** topic, ***harder*** than serial computing (which therefore needs to be mastered first).

This tradition is WRONG!

… which has (radical) implications on how we should educate people for computer science …

… and on how we apply what we have learnt …

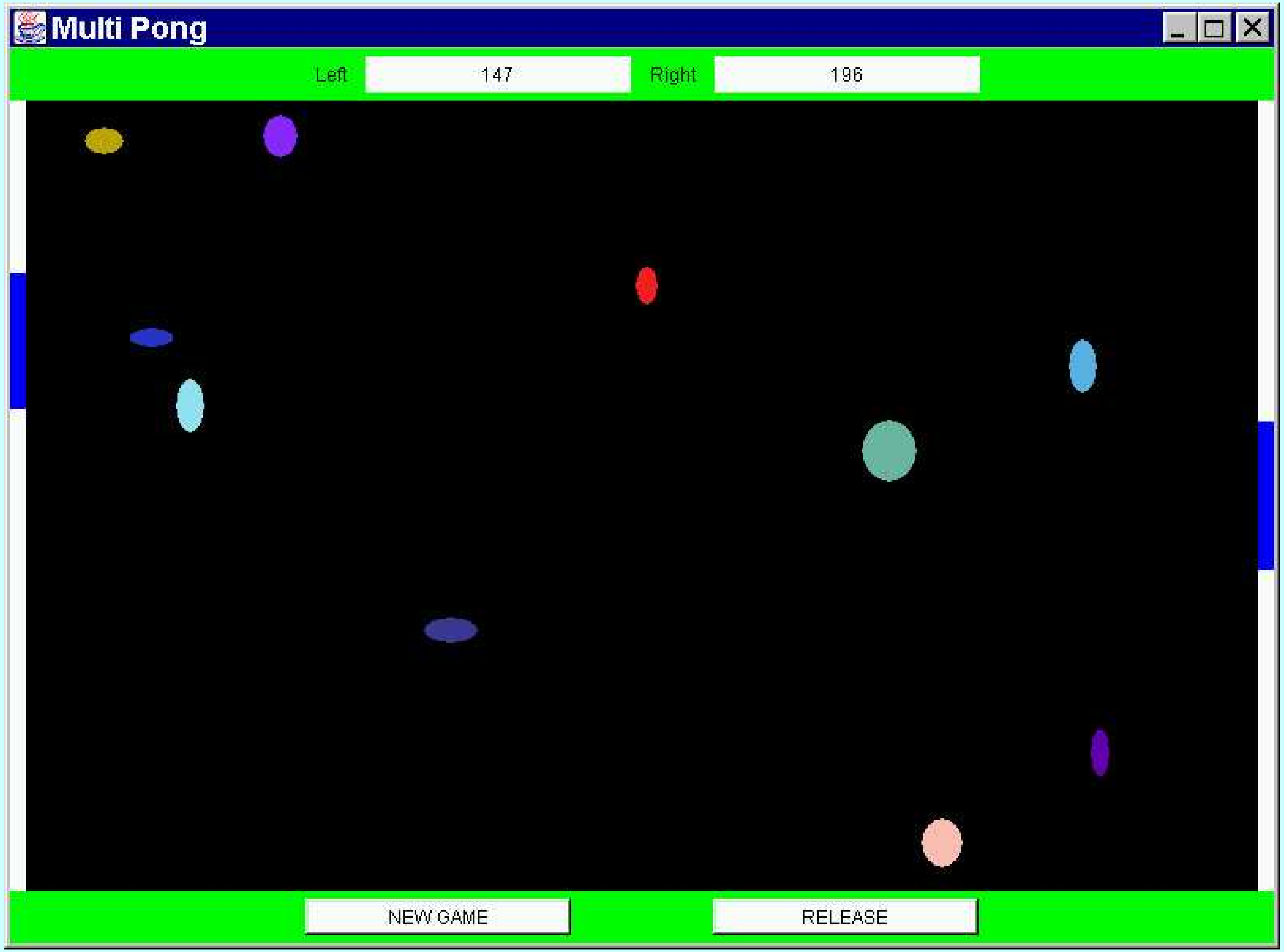
What we want from Parallelism

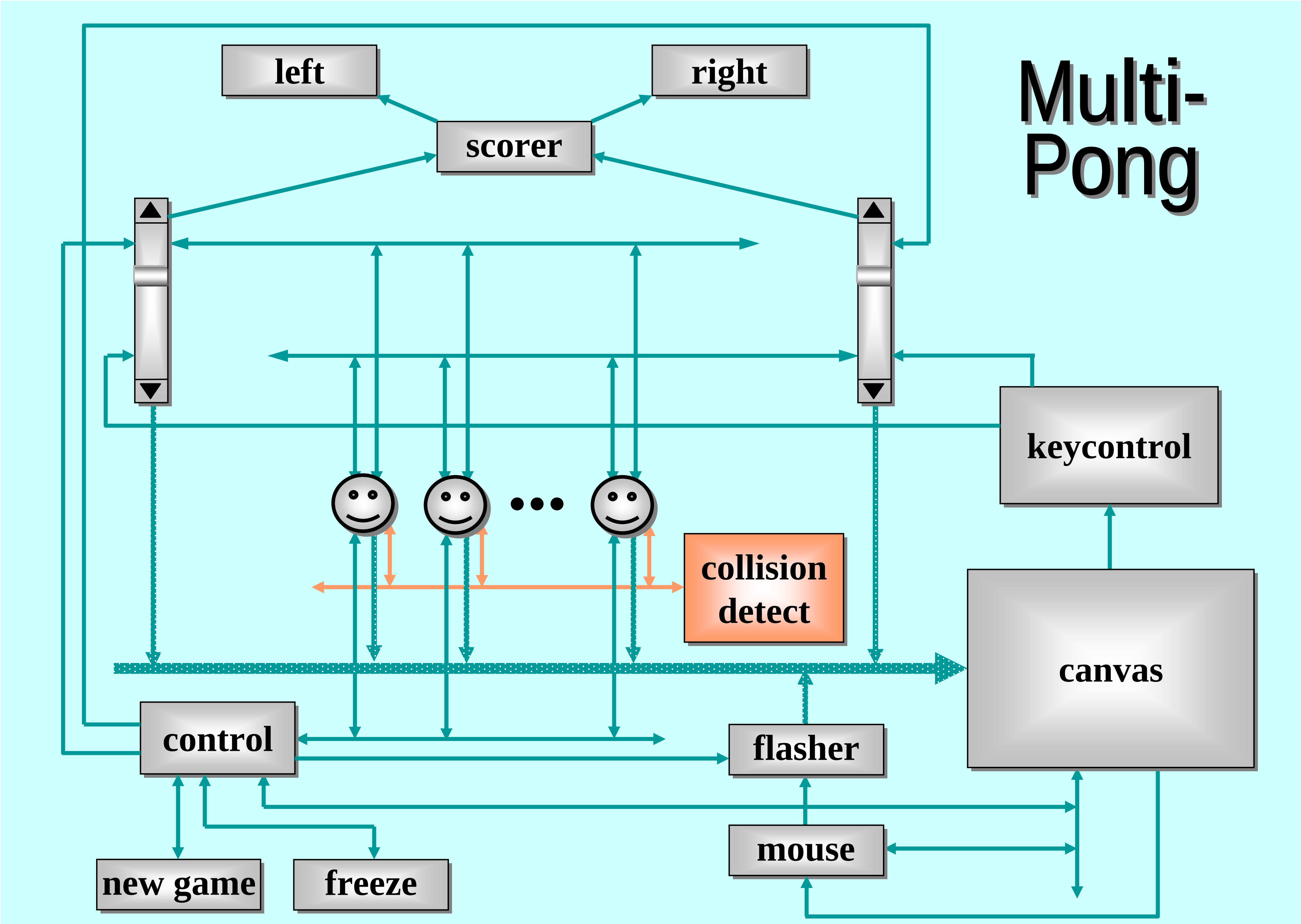
- A powerful tool for ***simplifying*** the description of systems.

- ***Performance*** that spins out from the above, but is ***not*** the primary focus.

- A model of concurrency that is ***mathematically clean***, yields no engineering surprises and scales well with system complexity.

Multi-Pong




Multi-Pong
left
right
scorer
keycontrol
collision detect
canvas
control
flasher
mouse
new game
freeze

Good News!

The good news is that we can worry about each process on its own. ***A process interacts with its environment through its channels*. *It does not interact directly with other processes*.**

Some processes have *serial* implementations - ***these are just like traditional serial programs***.

Some processes have *parallel* implementations - ***networks of sub-processes (think hardware)***.

Our skills for serial logic sit happily alongside our new skills for concurrency - there is no conflict. *This will scale!*

Java Monitors - CONCERNS

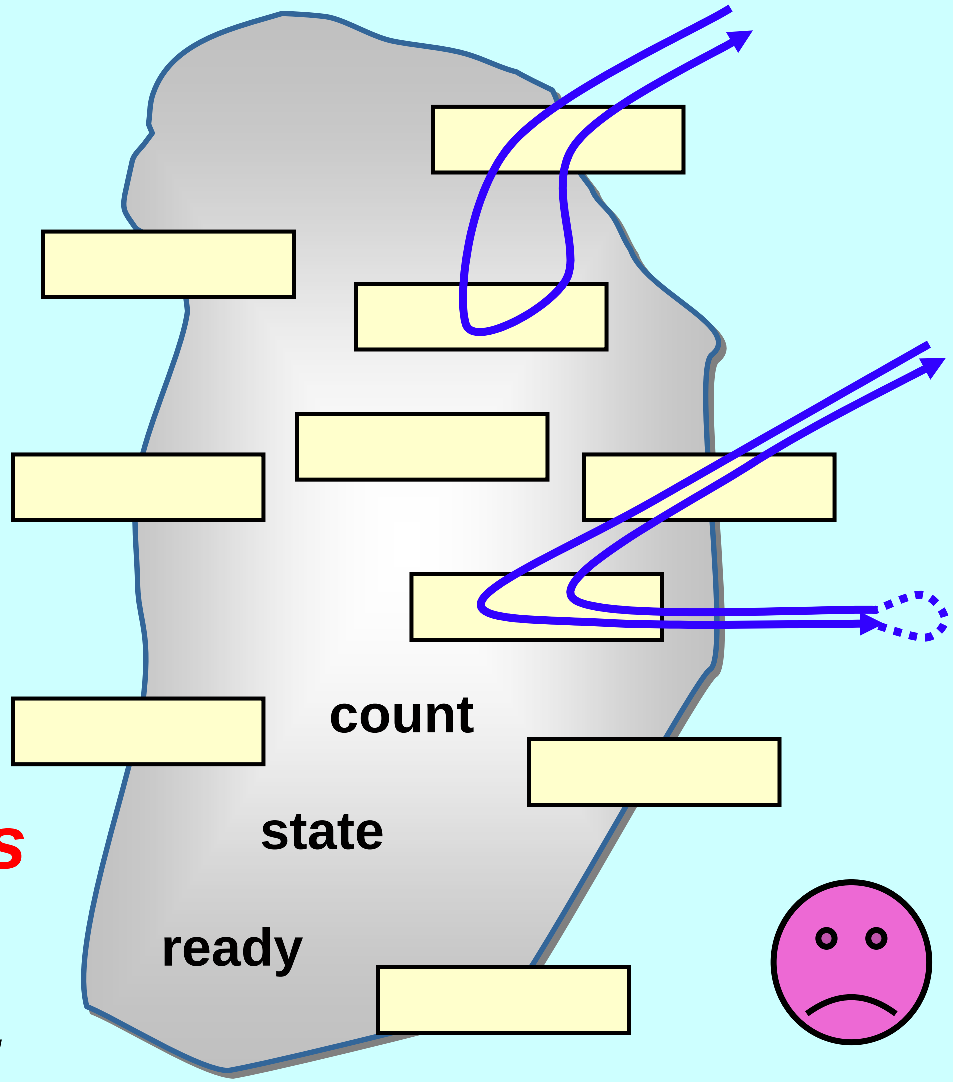
- ***Easy to learn*** - but ***very difficult to apply*** … ***safely*** …
- Monitor methods are ***tightly interdependent*** - their semantics compose in ***complex ways*** … the whole skill lies in setting up and staying in control of these complex interactions …
- Threads have no structure … there are no ***threads within threads*** …
- Big problems when it comes to ***scaling up complexity*** …

Objects Considered Harmful

Most objects are dead - they have no life of their own.

All methods have to be invoked by an external thread of control - they have to be ***caller oriented*** …

*… **a somewhat curious property** of so-called **object oriented** design.*

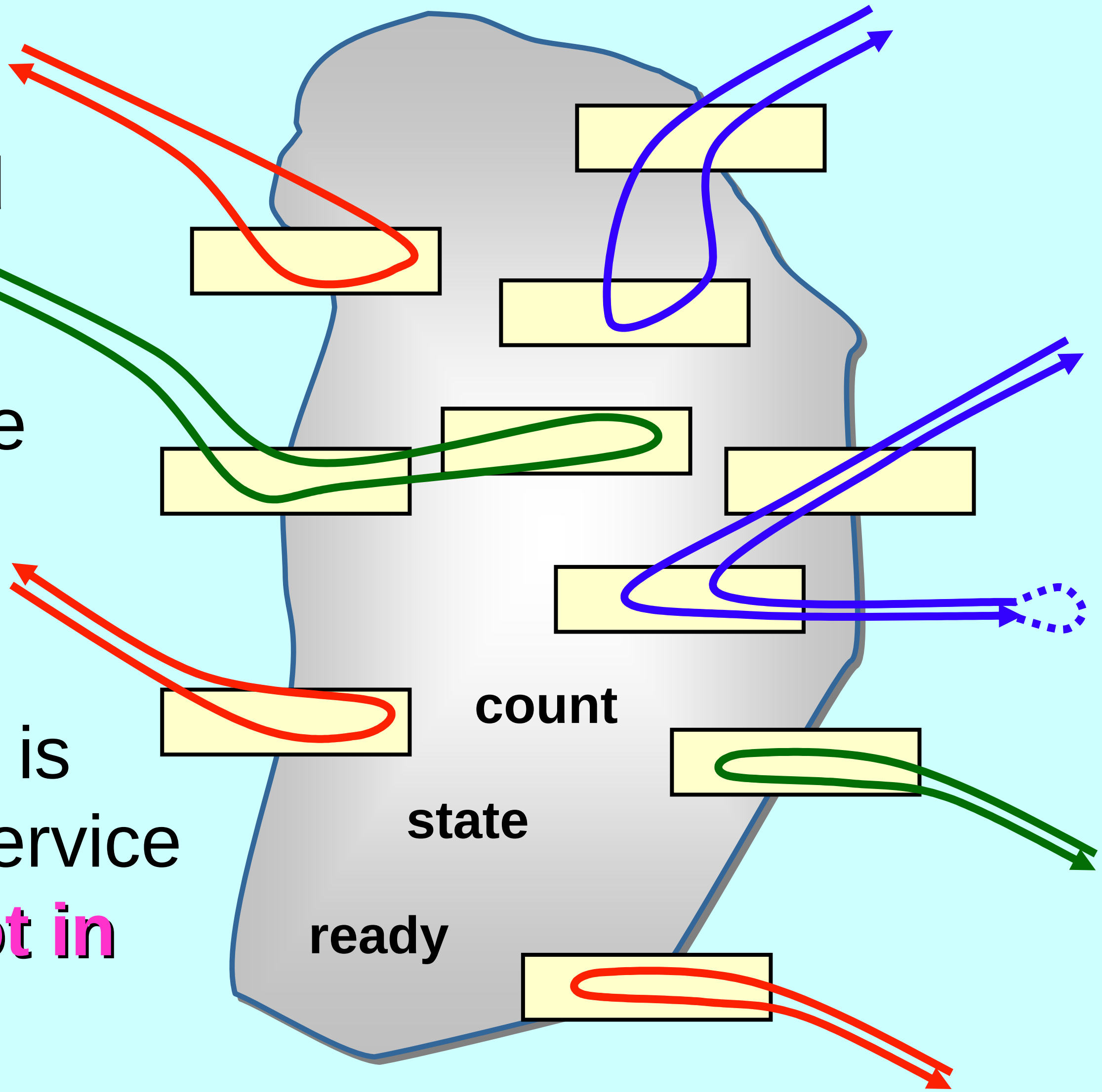


Objects Considered Harmful

The object is at the mercy of ***any*** thread that sees it.

Nothing can be done to prevent method invocation ...

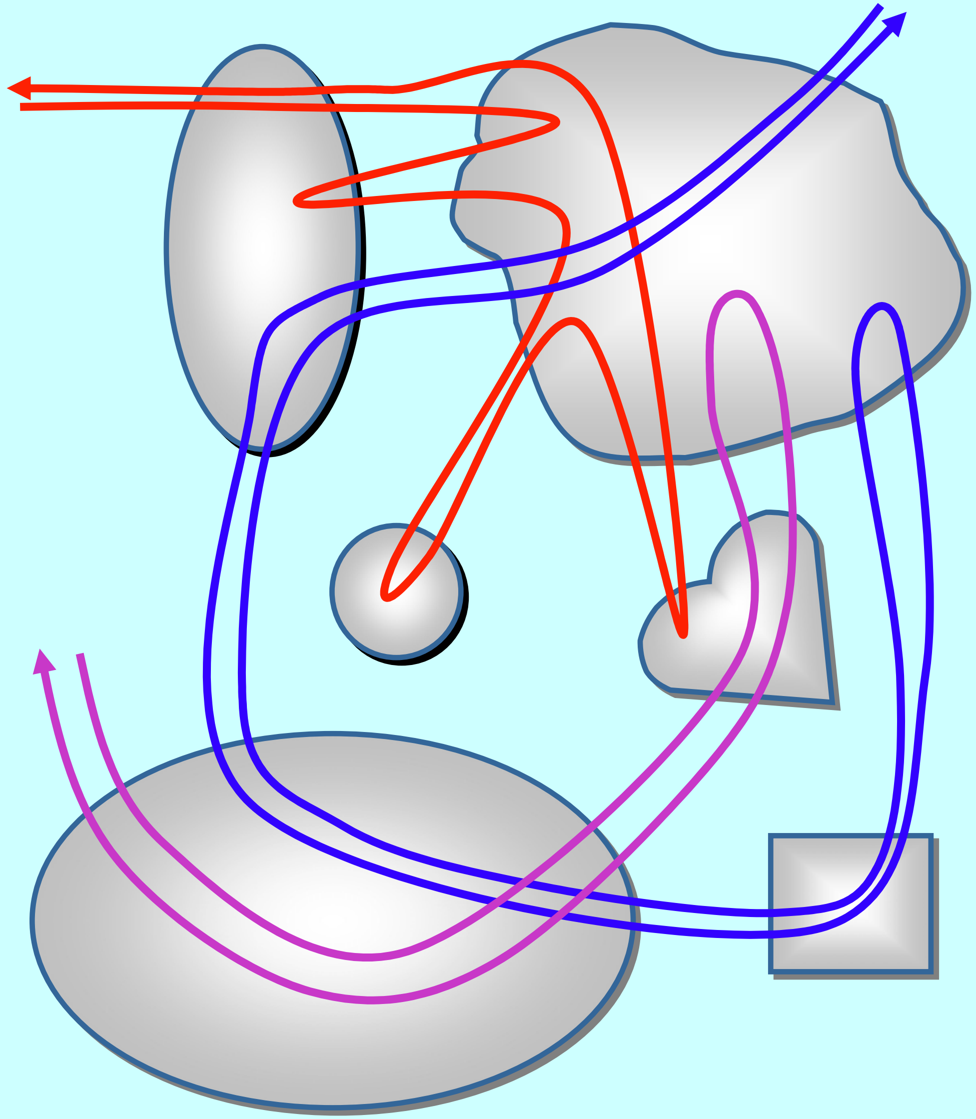
… even if the object is not in a fit state to service it. **The object is not in control of its life.**



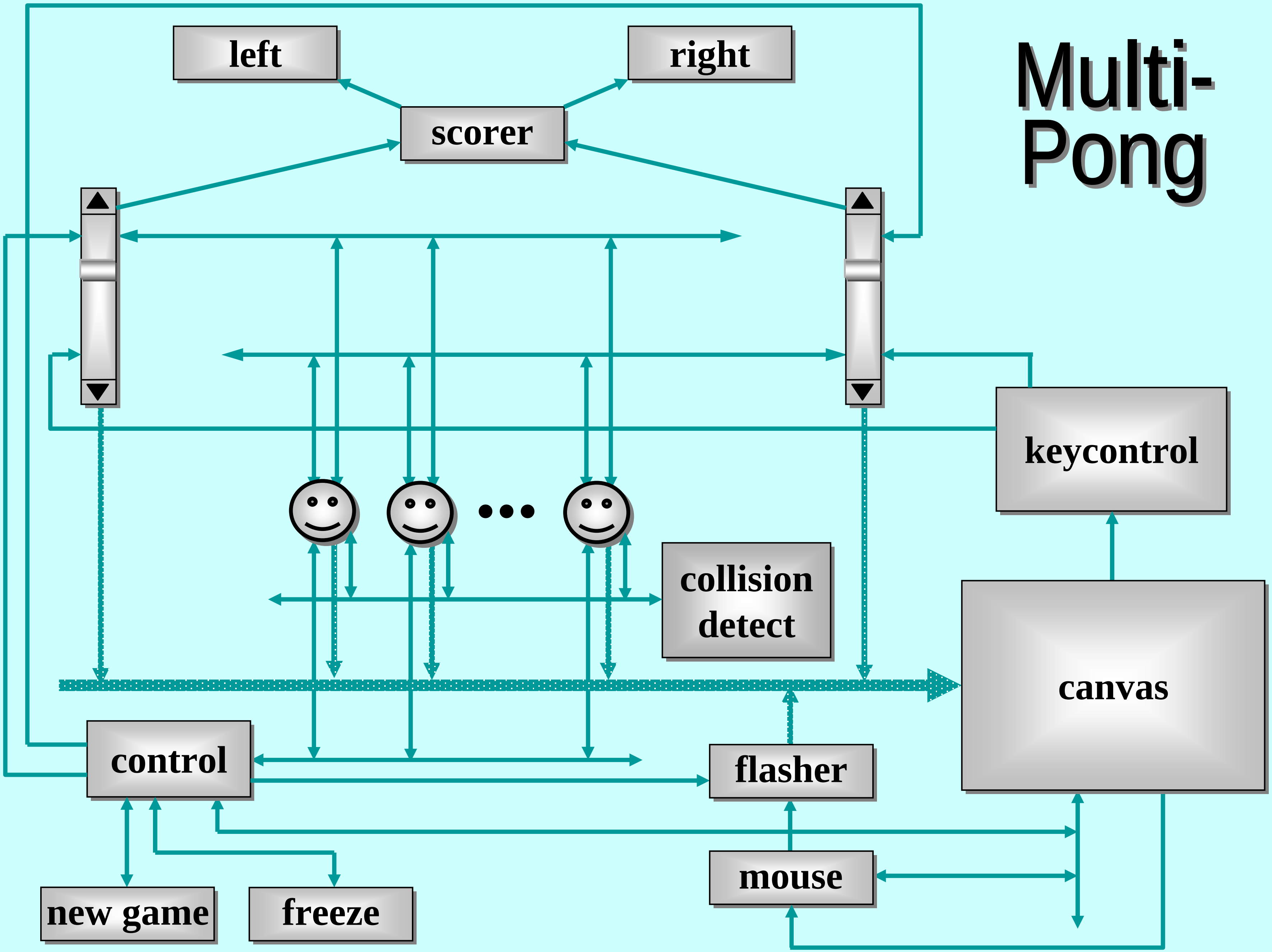
Objects Considered Harmful

Each single thread of control snakes around objects in the system, bringing them to life ***transiently*** as their methods are executed.

Threads cut across object boundaries leaving spaghetti-like trails, ***paying no regard to the underlying structure***.



Multi-Pong


left
right
scorer
keycontrol
collision detect
canvas
control
flasher
mouse
new game
freeze

Java Monitors - CONCERNS

- Almost all multi-threaded codes making direct use of the Java monitor primitives that we have seen ***(including our own)*** contained race or deadlock hazards.
- Sun’s Swing classes are not thread-safe … **why not?**
- **One of our codes contained a race hazard that did not trip for two years. This had been in daily use, its sources published on the web and its algorithms presented without demur to several Java literate audiences.**

Java Monitors - CONCERNS

<java.sun.com/products/jfc/tsc/articles/threads/threads1.html>

- ***"If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug."***

- ***"Component developers do not have to have an in-depth understanding of threads programming: toolkits in which all components must fully support multithreaded access, can be difficult to extend, particularly for developers who are not expert at threads programming."***

Java Monitors - CONCERNS

<java.sun.com/products/jfc/tsc/articles/threads/threads1.html>

- ***"It is our basic belief that extreme caution is warranted when designing and building multi-threaded applications … use of threads can be very deceptive … in almost all cases they make debugging, testing, and maintenance vastly more difficult and sometimes impossible. Neither the training, experience, or actual practices of most programmers, nor the tools we have to help us, are designed to cope with the non-determinism … this is particularly true in Java … we urge you to think twice about using threads in cases where they are not absolutely necessary …"***

Java Monitors - CONCERNS

- No guarantee that any **`synchronized`** method will ever be executed … (e.g. ***stacking*** JVMs)

- Even if we had above promise (e.g. ***queueing*** JVMs), standard design patterns for **`wait()`**/**`notify()`** fail to guarantee liveness (***"Wot, no chickens?"***)

See:

http://www.hensa.ac.uk/parallel/groups/wotug/java/discussion/3.html

http://www.nist.gov/itl/div896/emaildir/rt-j/msg00385.html

http://www. nist.gov/itl/div896/emaildir/rt-j/msg00363.html

Java Monitors - CONCERNS

- Threads yield non-determinacy (and, therefore, scheduling sensitivity) straight away ...
- No help provided to guard against race hazards ...
- Overheads too high (> 30 times ***???****)*
- Tyranny of Magic Names (e.g for *listener* callbacks)
- Learning curve is long …
- Scalability (both in logic and performance) ***???***
- Theoretical foundations ***???***
  - (deadlock / livelock / starvation analysis ***???***)
  - (rules / tools ***???***)

Java Monitors - CONCERNS

- So, Java monitors are not something with which we want to think - certainly not on a daily basis.
- But concurrency should be a powerful tool for **simplifying** the description of systems …
- **So it needs to be something I want to use - and am comfortable with - on a daily basis!**

Communicating Sequential Processes (CSP)

A mathematical theory for specifying and verifying complex patterns of behaviour arising from interactions between concurrent objects.

CSP has a formal, and ***compositional***, semantics that is in line with our informal intuition about the way things work.

Claim

Why CSP?

- Encapsulates fundamental principles of communication.
- Semantically defined in terms of structured mathematical model.
- Sufficiently expressive to enable reasoning about deadlock and livelock.
- Abstraction and refinement central to underlying theory.
- Robust ***and commercially supported*** software engineering tools exist for formal verification.

Why CSP?

- **CSP** libraries available for Java (**JCSP, CTJ**).

- Ultra-lightweight kernels* have been developed yielding ***sub-microsecond*** overheads for context switching, process startup/shutdown, synchronized channel communication and high-level shared-memory locks.

- Easy to learn and easy to apply …

* not yet available for JVMs (or Core JVMs!)

Why CSP?

- After 5 hours teaching:
  - exercises with 20-30 threads of control
  - regular and irregular interactions
  - appreciating and eliminating race hazards, deadlock, etc.

- **CSP** is (parallel) architecture neutral:
  - message-passing
  - shared-memory

So, what is CSP?

CSP deals with ***processes***, ***networks*** of processes and various forms of ***synchronisation*** / ***communication*** between processes.

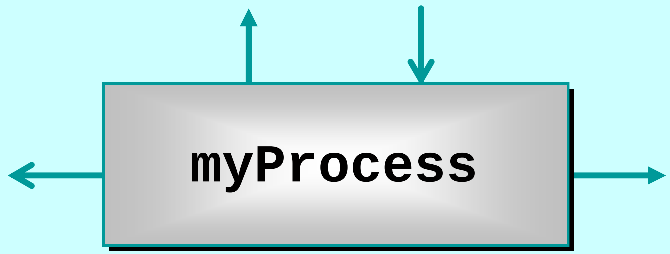
A network of processes is also a process - so **CSP** naturally accommodates layered network structures (***networks of networks***).

We do not need to be mathematically sophisticated to work with **CSP**. That sophistication is pre-engineered into the model. We benefit from this simply by using it.

Processes

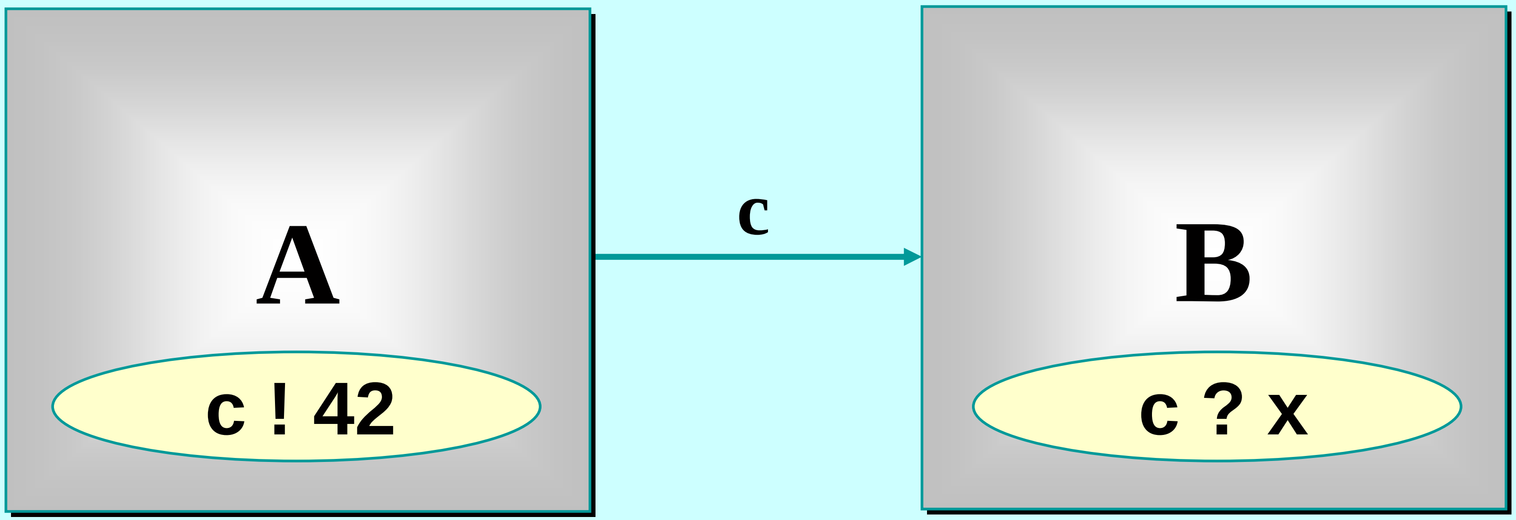
- A **process** is a component that encapsulates some data structures and algorithms for manipulating that data.
- Both its data and algorithms are **private**. The outside world can neither see that data nor execute those algorithms! [They are not *objects*.]
- The algorithms are executed by the process in its own thread (or threads) of control.
- So, how does one process interact with another?

Processes



- The simplest form of interaction is *synchronised* message-passing along ***channels***.
- The simplest forms of channel are ***zero-buffered*** and ***point-to-point*** (i.e. *wires*).
- But, we can have ***buffered*** channels (*blocking*/*overwriting*).
- And ***any-1***, ***1-any*** and ***any-any*** channels.
- And structured multi-way synchronisation (e.g. ***barriers***) …
- And high-level (e.g. **CREW**) *shared-memory* locks …

Synchronised Communication

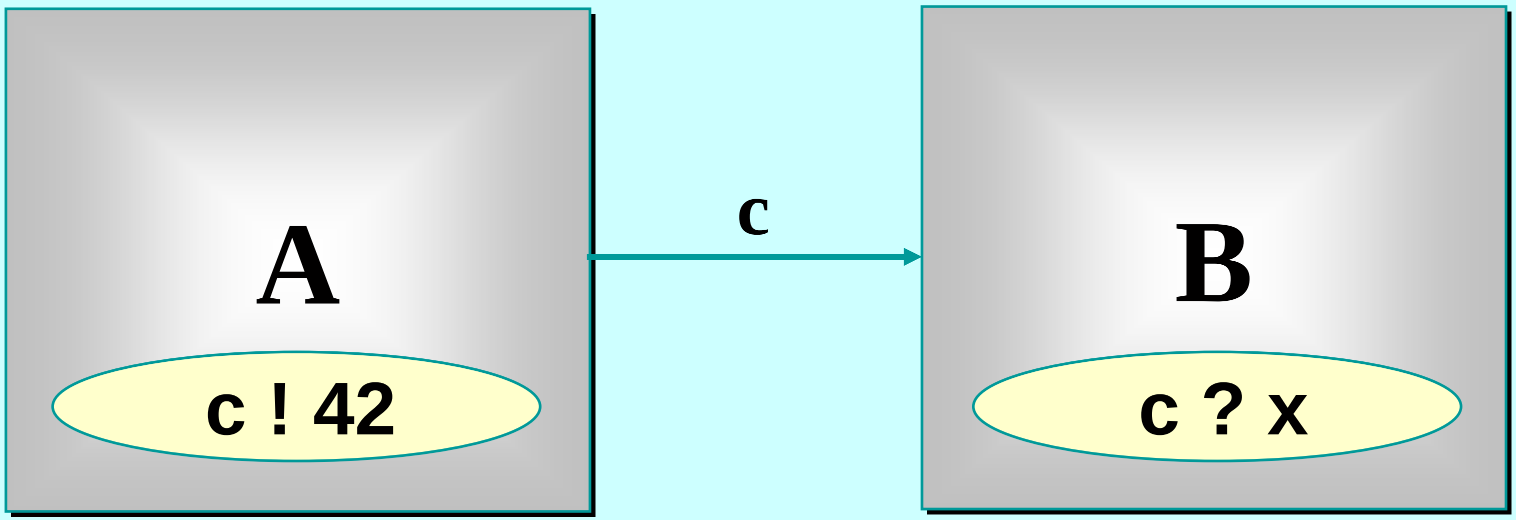


A may ***write*** on ***c*** at any time, but has to wait for a ***read***.

B may ***read*** from ***c*** at any time, but has to wait for a ***write***.

(A (c) || B (c)) \ {c}

Synchronised Communication

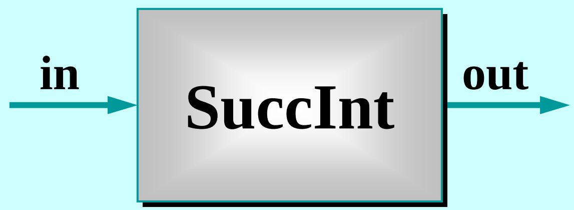


Only when both ***A*** and ***B*** are ready can the communication proceed over the channel ***c***.

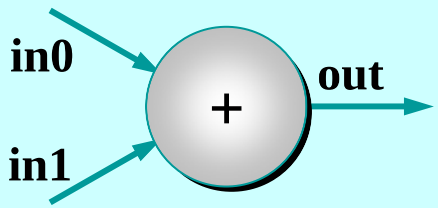
(A (c) || B (c)) \ {c}

'Legoland' Catalog

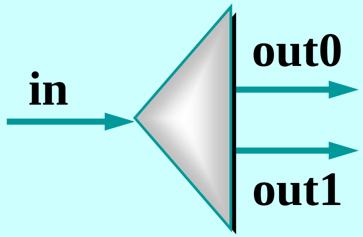
`IdInt (in, out)`



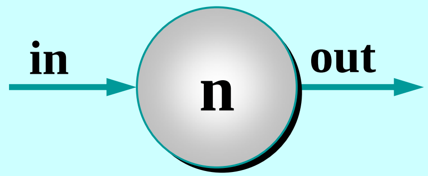
`SuccInt (in, out)`



`PlusInt (in0, in1, out)`



`Delta2Int (in, out0, out1)`



`PrefixInt (n, in, out)`

`TailInt (in, out)`

'Legoland' Catalog

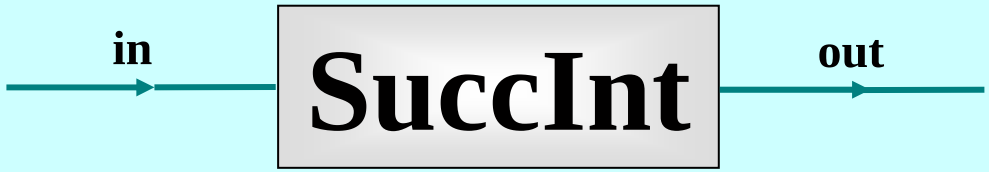
- This is a catalog of fine-grained processes - think of them as pieces of hardware (e.g. chips). They process data (`int`s) flowing through them.
- They are presented not because we suggest working at such fine levels of granularity …
- They are presented in order to build up fluency in working with parallel logic.

'Legoland' Catalog

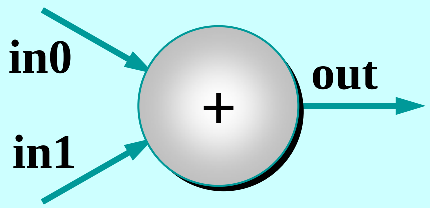
- Parallel logic should become just as easy to manage as serial logic.
- This is not the traditionally held view …
- But that tradition is **wrong**.
- **CSP/occam** people have always known this.

Let's look at some **CSP** *pseudo-code* for these processes …


```
IdInt (in, out) = in?x --> out!x --> IdInt (in, out)
```

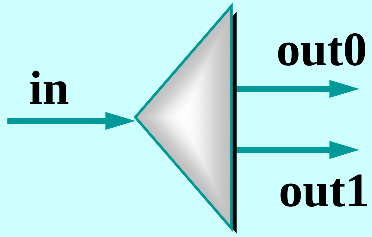



```
SuccInt (in, out) = in?x --> out!(x + 1) --> SuccInt (in, out)
```



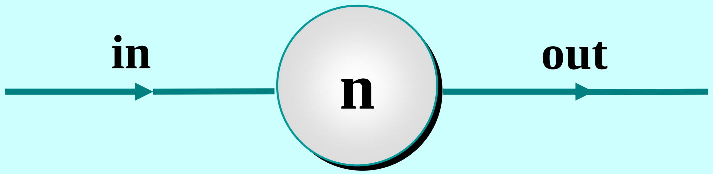
Note the parallel input

```
PlusInt (in0, in1, out) =
  (in0?x0 --> SKIP || inl?x1 --> SKIP);
  out!(x0 + x1) --> PlusInt (in0, in1, out)
```



Note the parallel output

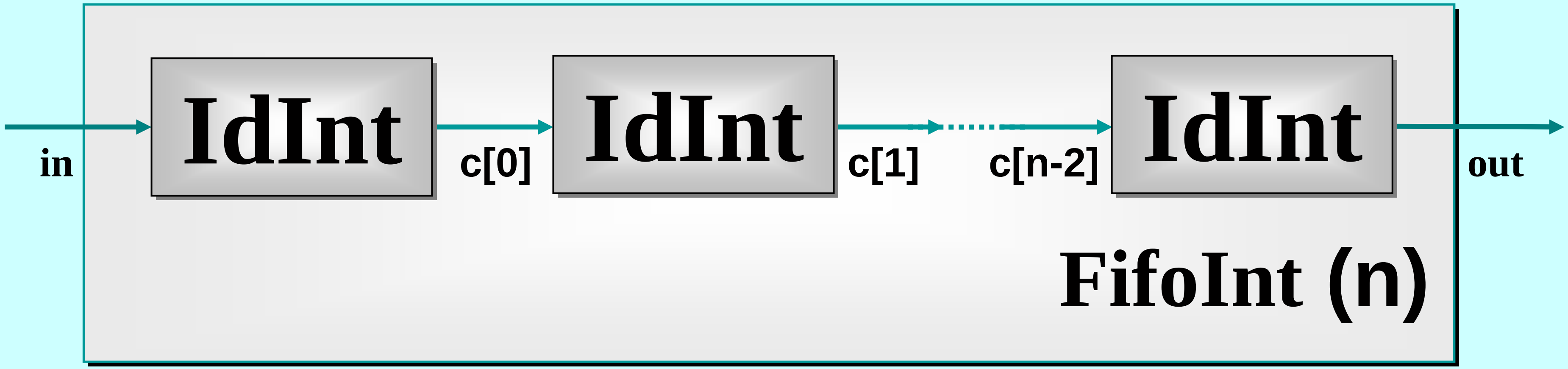
```
Delta2Int (in, out0, out1) =
  in?x --> (out0!x --> SKIP || out1!x --> SKIP);
  Delta2Int (in, out0, out1)
```




```
PrefixInt (n, in, out) = out!n --> IdInt (in, out)
```


```
TailInt (in, out) = in?x --> IdInt (in, out)
```

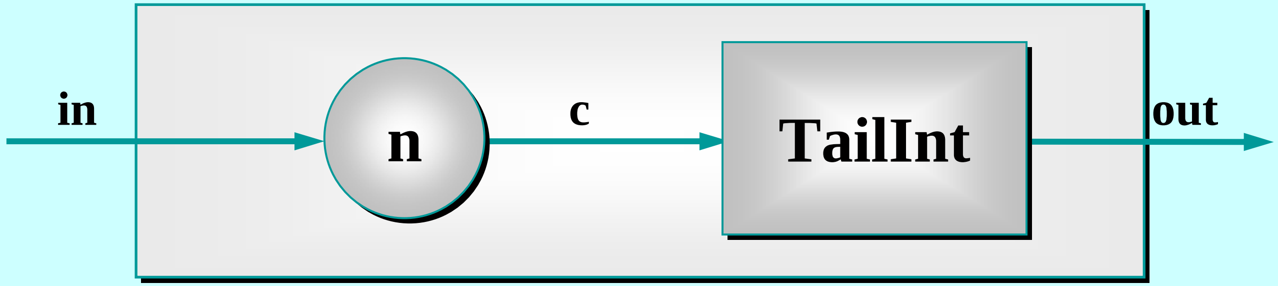
A Blocking FIFO Buffer



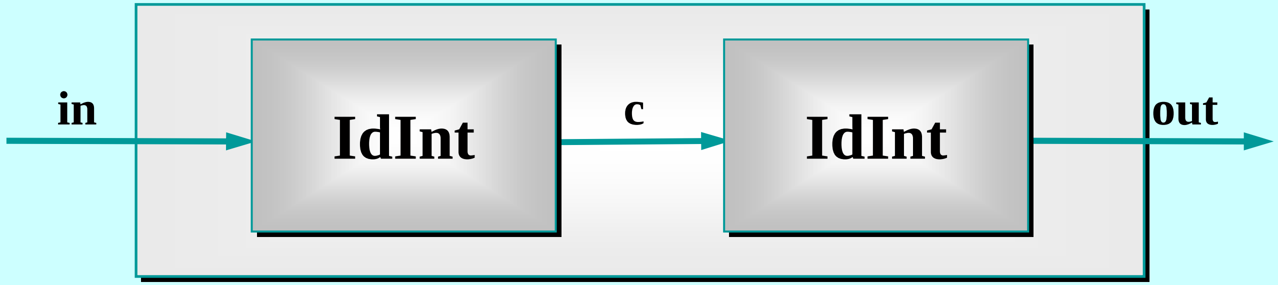

```
FifoInt (n, in, out) =
  IdInt (in, c[0]) ||
  ([||i = 0 FOR n-2] IdInt (c[i], c[i+1])) ||
  IdInt (c[n-2], out)
```

Note: this is such a common idiom that it is provided as a (channel) primitive in JCSP.

A Simple Equivalence



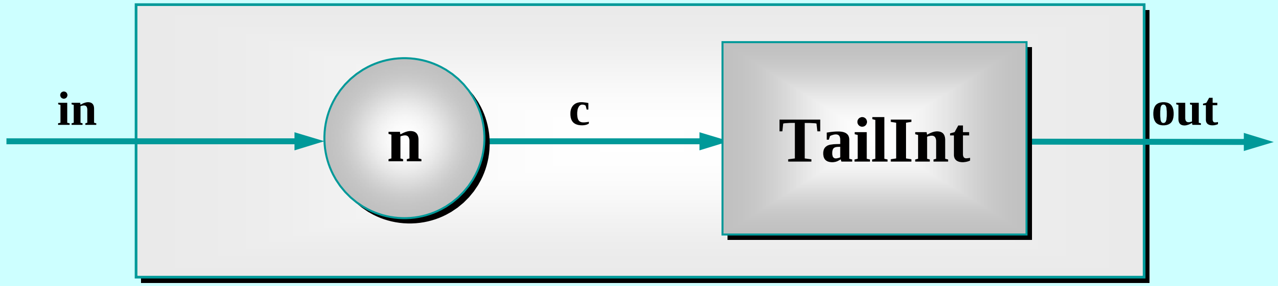
(PrefixInt (n, in, c) || TailInt (c, out)) \ {c}



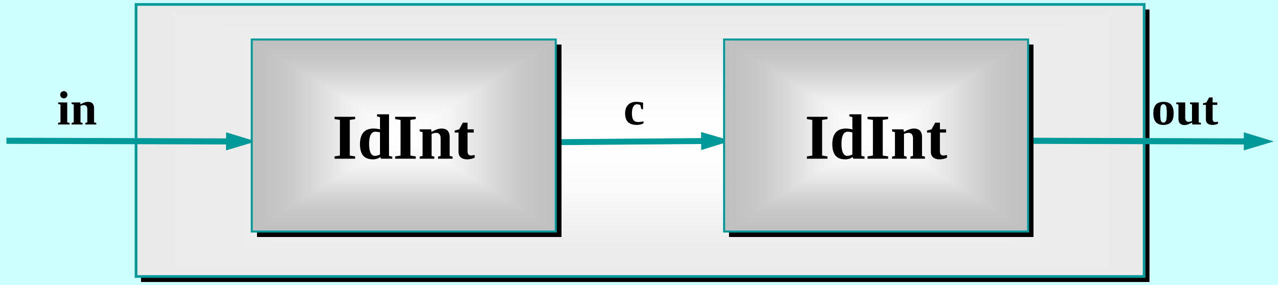
(IdInt (in, c) || IdInt (c, out)) \ {c}

The outside world can see no difference between these two 2-place FIFOs …

A Simple Equivalence




```
(PrefixInt (n, in, c) || TailInt (c, out)) \ {c}
```




```
(IdInt (in, c) || IdInt (c, out) ) \ {c}
```

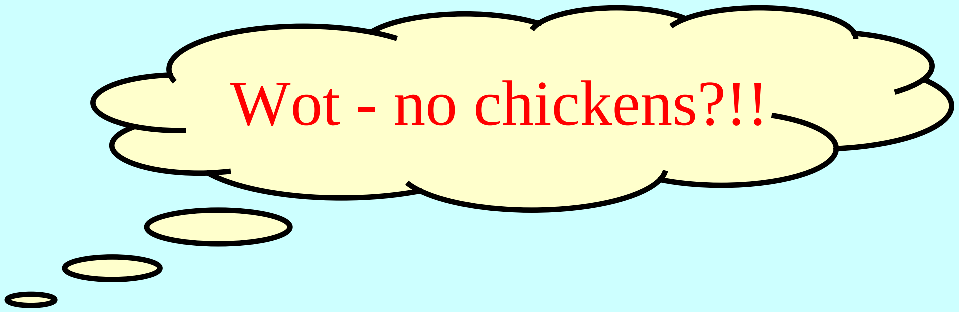
The proof that they are equivalent is a two-liner from the definitions of **!**, **?**, **-->**, **** and **||**.

Good News!

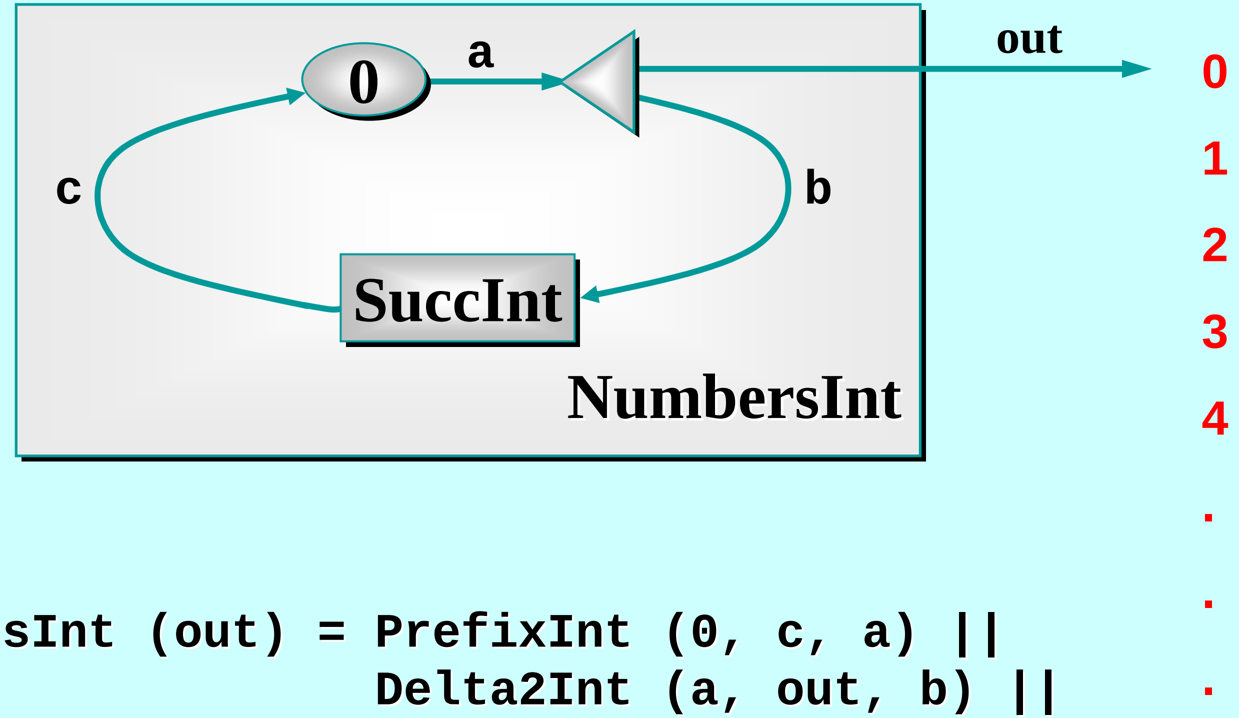
The good news is that we can ‘see’ this semantic equivalence with just one glance.

[CLAIM] **CSP** semantics cleanly reflects our intuitive feel for interacting systems.

This quickly builds up confidence …



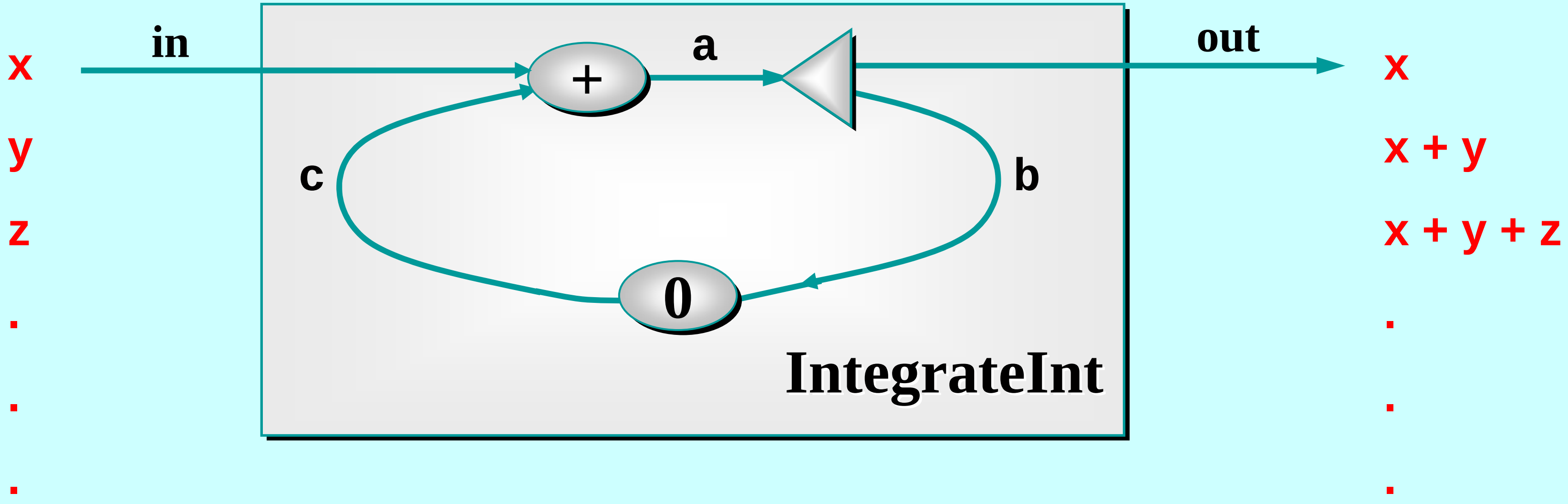
Some Simple Networks




```
NumbersInt (out) = PrefixInt (0, c, a) ||
                   Delta2Int (a, out, b) ||
                   SuccInt (b, c)
```

Note: this pushes numbers out so long as the receiver is willing to take it.

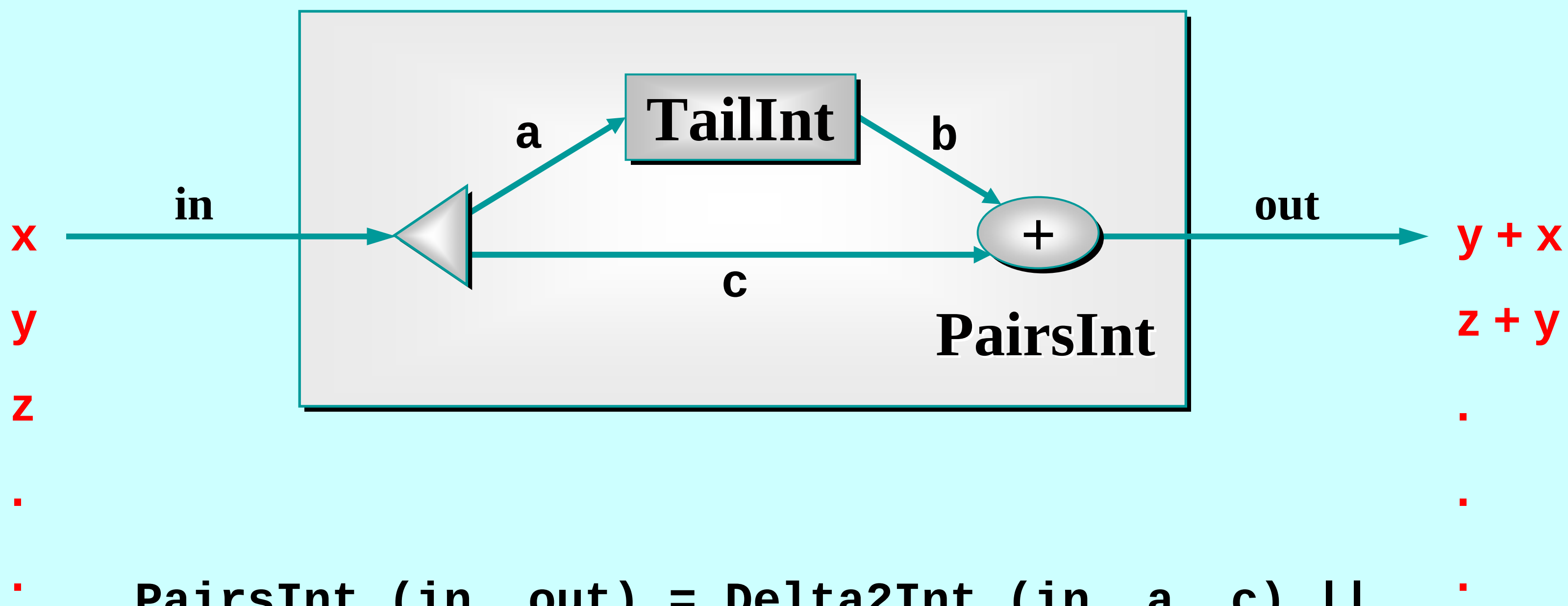
Some Simple Networks




```
IntegrateInt (out) = PlusInt (in, c, a) ||
                     Delta2Int (a, out, b) ||
                     PrefixInt (0, b, c)
```

Note: this outputs one number for every input it gets.

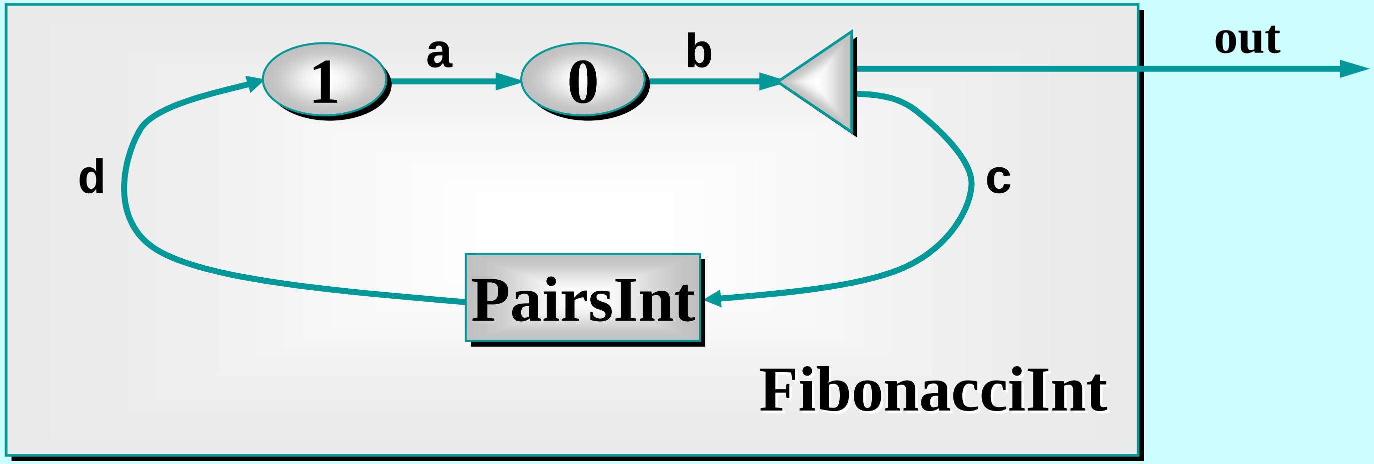
Some Simple Networks




```
PairsInt (in, out) = Delta2Int (in, a, c) ||
                     TailInt (a, b) ||
                     PlusInt (b, c, out)
```

Note: this needs two inputs before producing one output. Thereafter, it produces one number for every input it gets.

Some Layered Networks

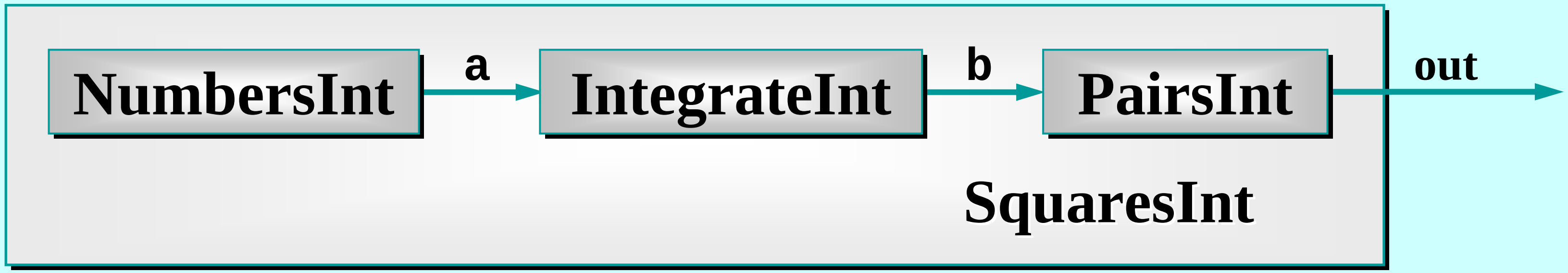



```
FibonacciInt (out) = PrefixInt (1, d, a) ||
                     PrefixInt (0, a, b) ||
                     Delta2Int (b, out, c) ||
                     PairsInt (b, c)
```

0
1
1
2
3
5
8
13
21
34
.
.

Note: the two numbers needed by **`PairsInt`** to get started are provided by the two **`PrefixInt`**s. Thereafter, only one number circulates on the feedback loop. If only one **`PrefixInt`** had been in the circuit, deadlock would have happened (with each process waiting trying to input).

Some Layered Networks



1
4
9
16
25
36
49
64
81
.
.

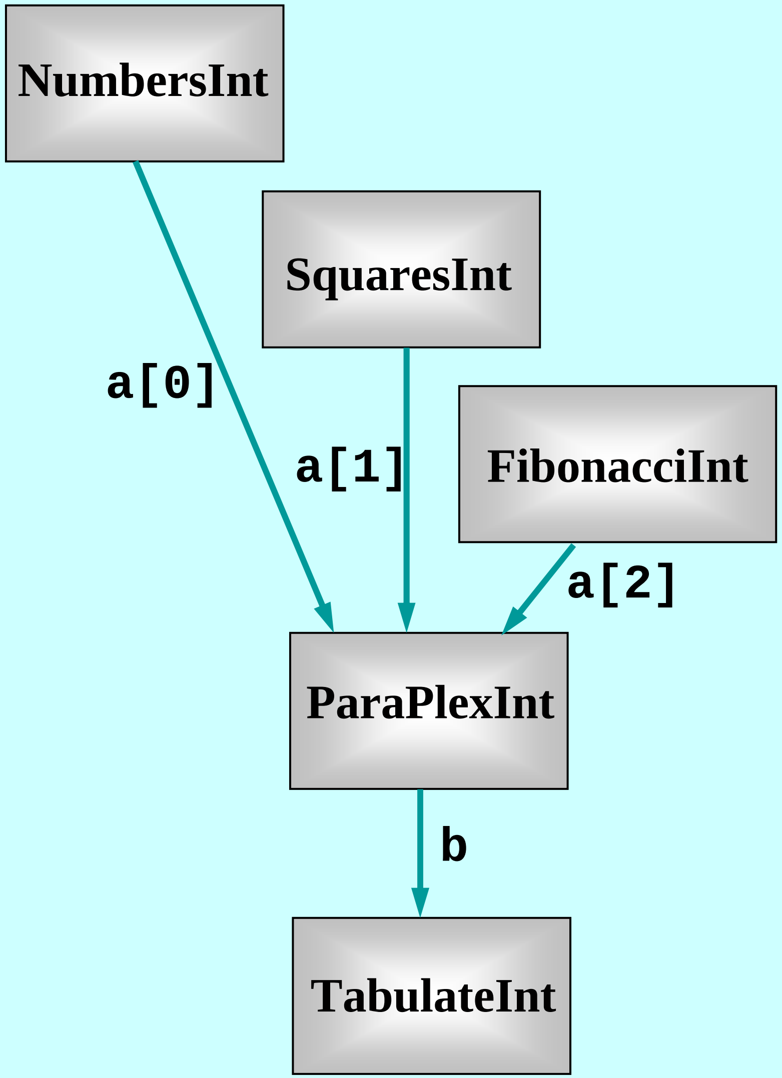
```
SquaresInt (out) = NumbersInt (a) ||
                   IntegrateInt (a, b) ||
                   PairsInt (b, out)
```

Note: the traffic on individual channels:

```
<a>   = [0,  1,  2,  3,  4,  5,  6,  7,  8, ...]
<b>   = [0,  1,  3,  6, 10, 15, 21, 28, 36, ...]
<out> = [1,  4,  9, 16, 25, 36, 49, 64, 81, ...]
```

Quite a Lot of Processes

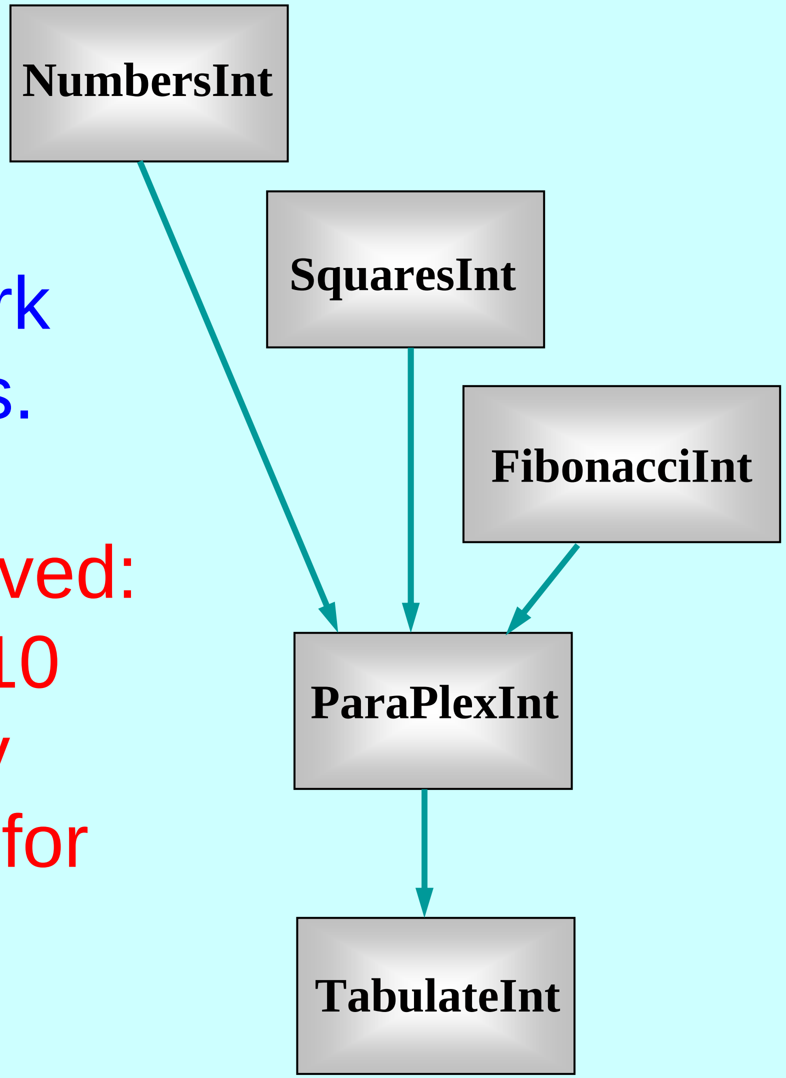
```
NumbersInt (a[0]) ||
SquaresInt (a[1]) ||
FibonacciInt (a[2]) ||
ParaPlexInt (a, b) ||
TabulateInt (b)
```



Quite a Lot of Processes

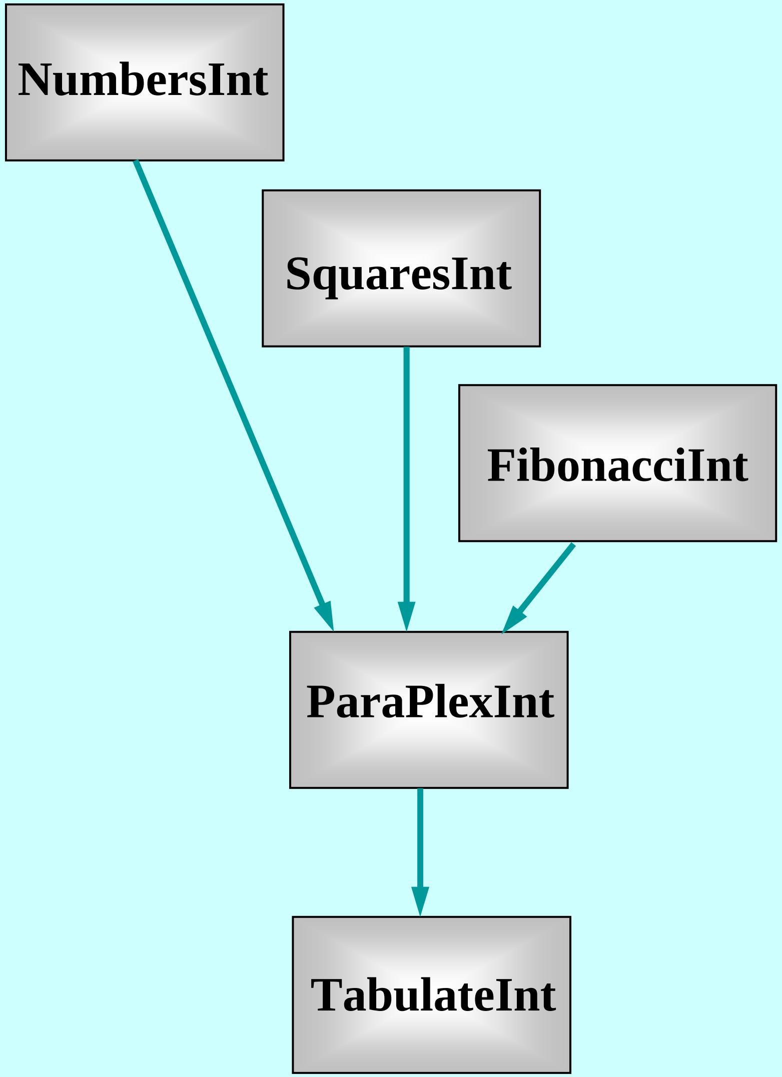
At this level, we have a network of 5 communicating processes.

In fact, 28 processes are involved: 18 non-terminating ones and 10 low-level transients repeatedly starting up and shutting down for parallel input and output.



Quite a Lot of Processes

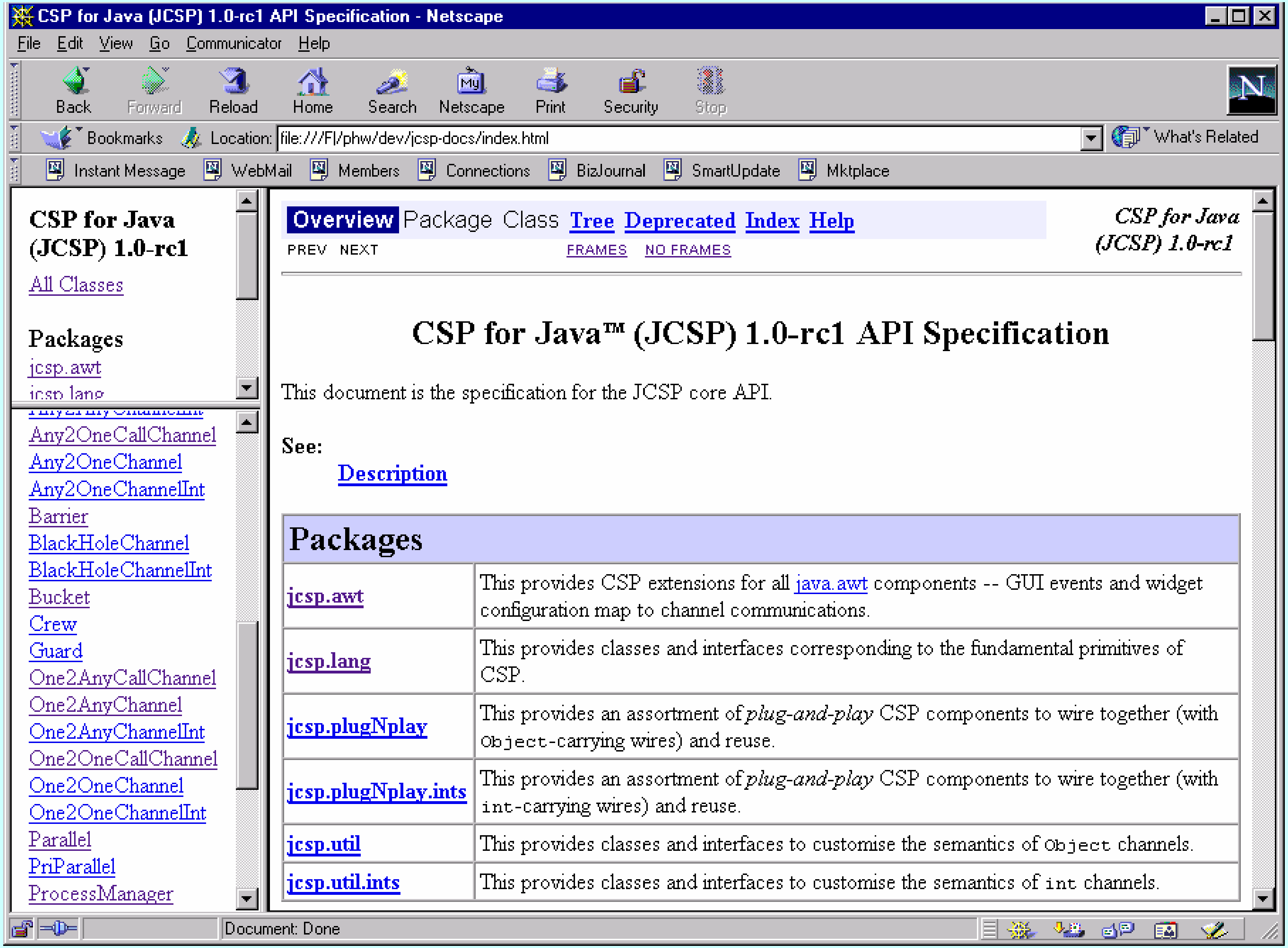
Fortunately, CSP semantics are ***compositional*** - which means that we only have to reason at each layer of the network in order to design, understand, code, and maintain it.



Putting CSP into practice ...

JCSP

Google: JCSP



CSP for Java™ (JCSP) 1.0-rc1 API Specification

This document is the specification for the JCSP core API.

See:

Description

Packages	
jcsp.awt	This provides CSP extensions for all java.awt components -- GUI events and widget configuration map to channel communications.
jcsp.lang	This provides classes and interfaces corresponding to the fundamental primitives of CSP.
jcsp.plugNplay	This provides an assortment of *plug-and-play* CSP components to wire together (with `Object`-carrying wires) and reuse.
jcsp.plugNplay.ints	This provides an assortment of *plug-and-play* CSP components to wire together (with `int`-carrying wires) and reuse.
jcsp.util	This provides classes and interfaces to customise the semantics of `Object` channels.
jcsp.util.ints	This provides classes and interfaces to customise the semantics of `int` channels.

CSP for Java (JCSP)

- A ***process*** is an object of a class implementing the ***CSProcess*** interface:

```
interface CSProcess {
  public void run();
}
```

- The *behaviour* of the process is determined by the body given to the **`run()`** method in the implementing class.

JCSP Process Structure

```
class Example implements CSProcess {

  ...  private shared synchronisation objects
       (channels etc.)
  ...  private state information

  ...  public constructors
  ...  public accessors(gets)/mutators(sets)
       (only to be used when not running)

  ...  private support methods (part of a run)
  ...  public void run() (process starts here)

}
```

Two Channel Interfaces
(classes are hidden)

`Object` channels
- carrying (references to) arbitrary Java objects

`int` channels
- carrying Java **`int`**s

Channel-End Interfaces

- Channel-end interfaces are what the processes see. Processes only need to care what kind of data they carry (**`int`**s or **`Object`**s) and whether the channels are for output, input or *ALTing* (i.e. *choice*) input.

- It is the network builder's concern to choose the variety of channel (e.g. *synchronous*, *buffered*, *shared*) to use when connecting processes together.

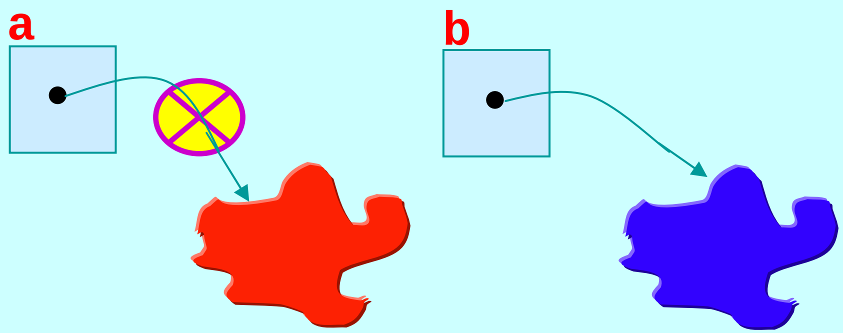
int Channels

- The **int** channels are convenient and secure.
- As with ***occam-π***, it's difficult to introduce race hazards.
- For completeness, ***JCSP*** should provide channels for carrying all of the Java primitive data-types. These would be trivial to add. So far, there has been no pressing need.

`Object` Aliasing – Danger !!!

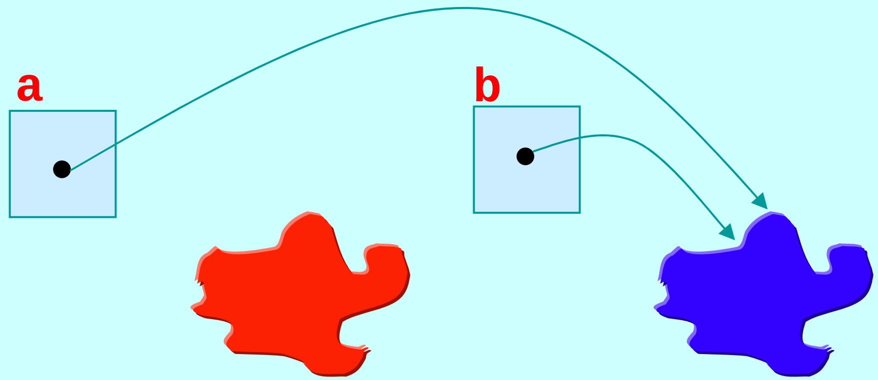
Java objects are referenced through variable names.

`Thing a = ..., b = ...;`



a and **b** are now ***aliases*** for the same object!

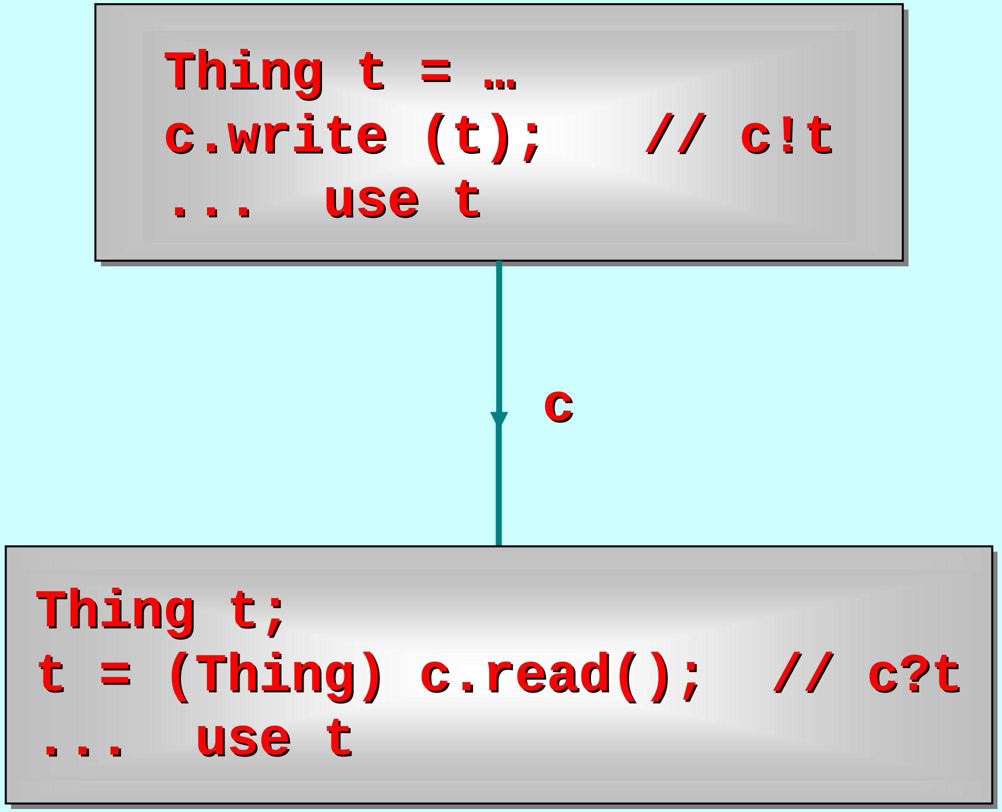
`a = b;`



Object Channels – Danger !!

- **Object** channels expose a danger not present in **occam-π**.

- Channel communication only communicates the **Object** reference.



`Object` Channels – Danger !!

- After the communication, each process has a reference (in its variable **`t`**) to the *same* object.

- If *one* of these processes modifies that object (its **`t`**), the *other* one had better forget about it!

```
Thing t = …
c.write (t);    // c!t
...   use t
```

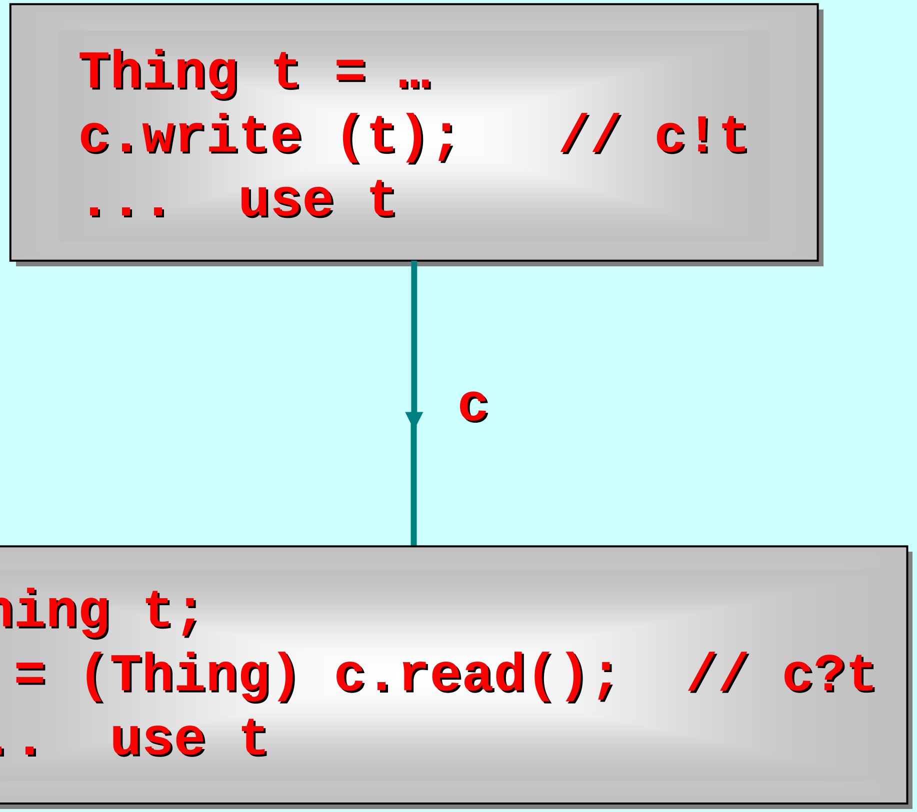
c

```
Thing t;
t = (Thing) c.read();  // c?t
...   use t
```

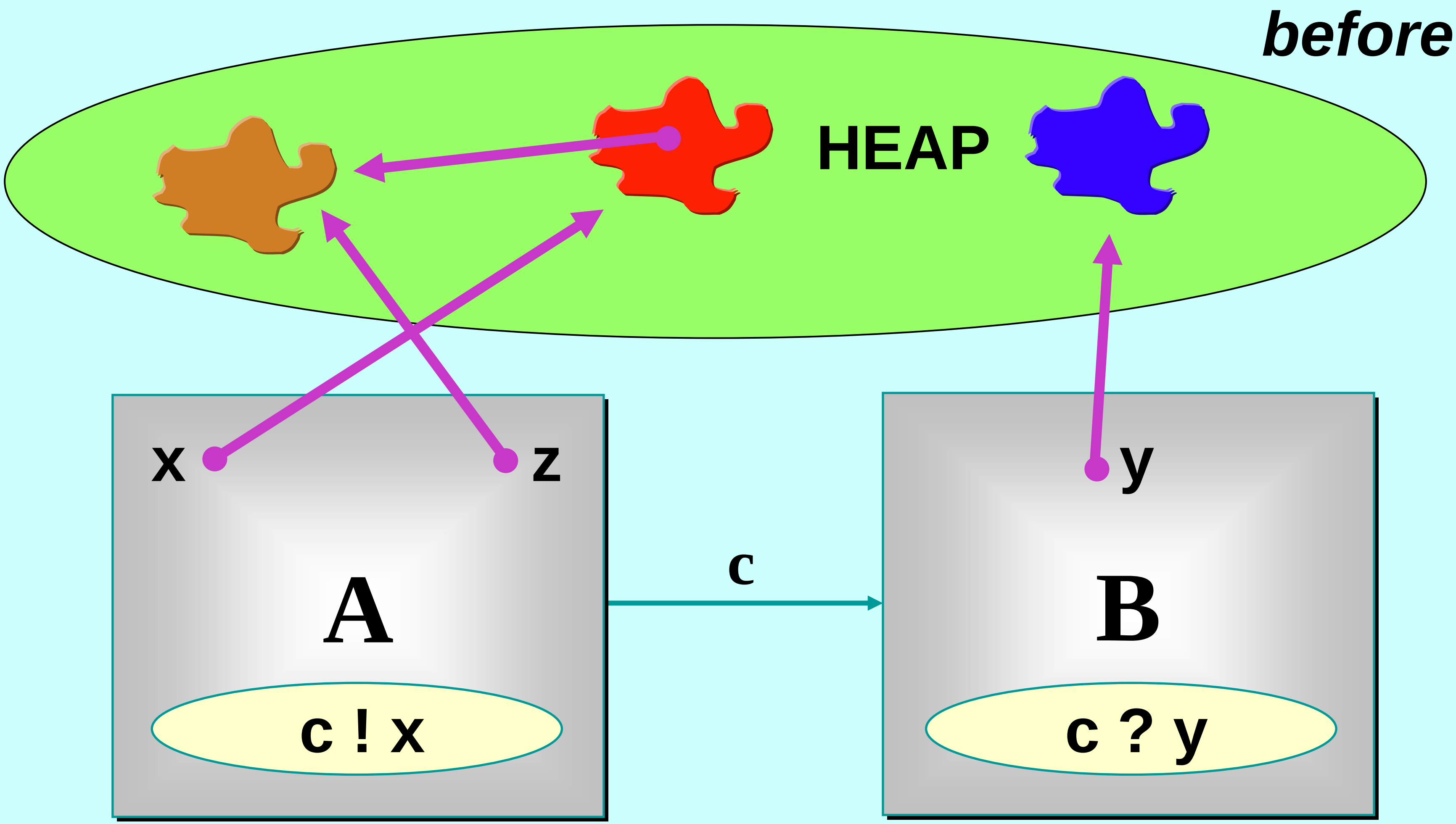
`Object` Channels – Danger !!

- Otherwise, **occam-π**'s parallel usage rule is violated and we will be at the mercy of ***when*** the processes get scheduled for execution - a ***RACE HAZARD***! 

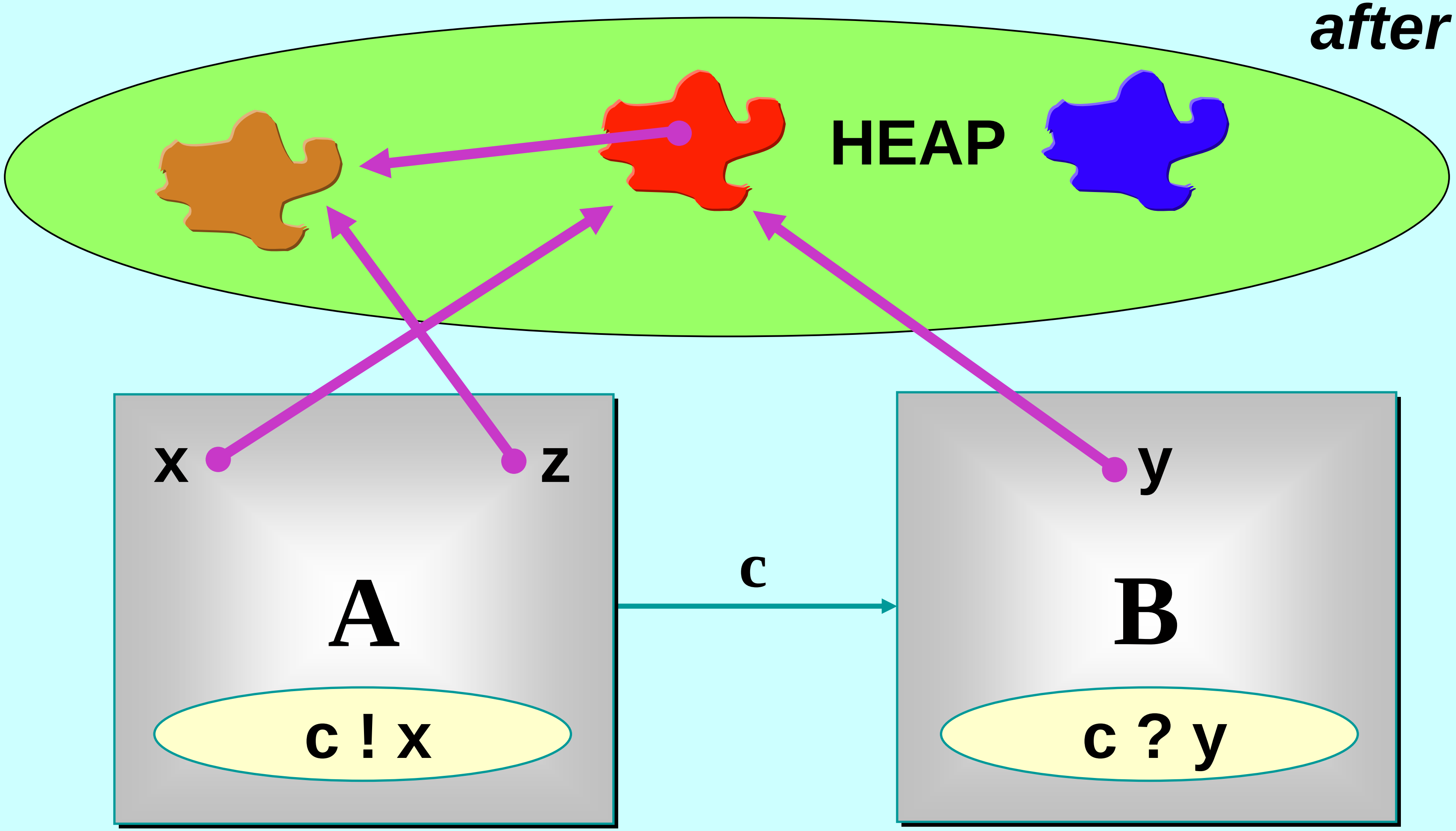
- We have design patterns to prevent this. 



Reference Semantics

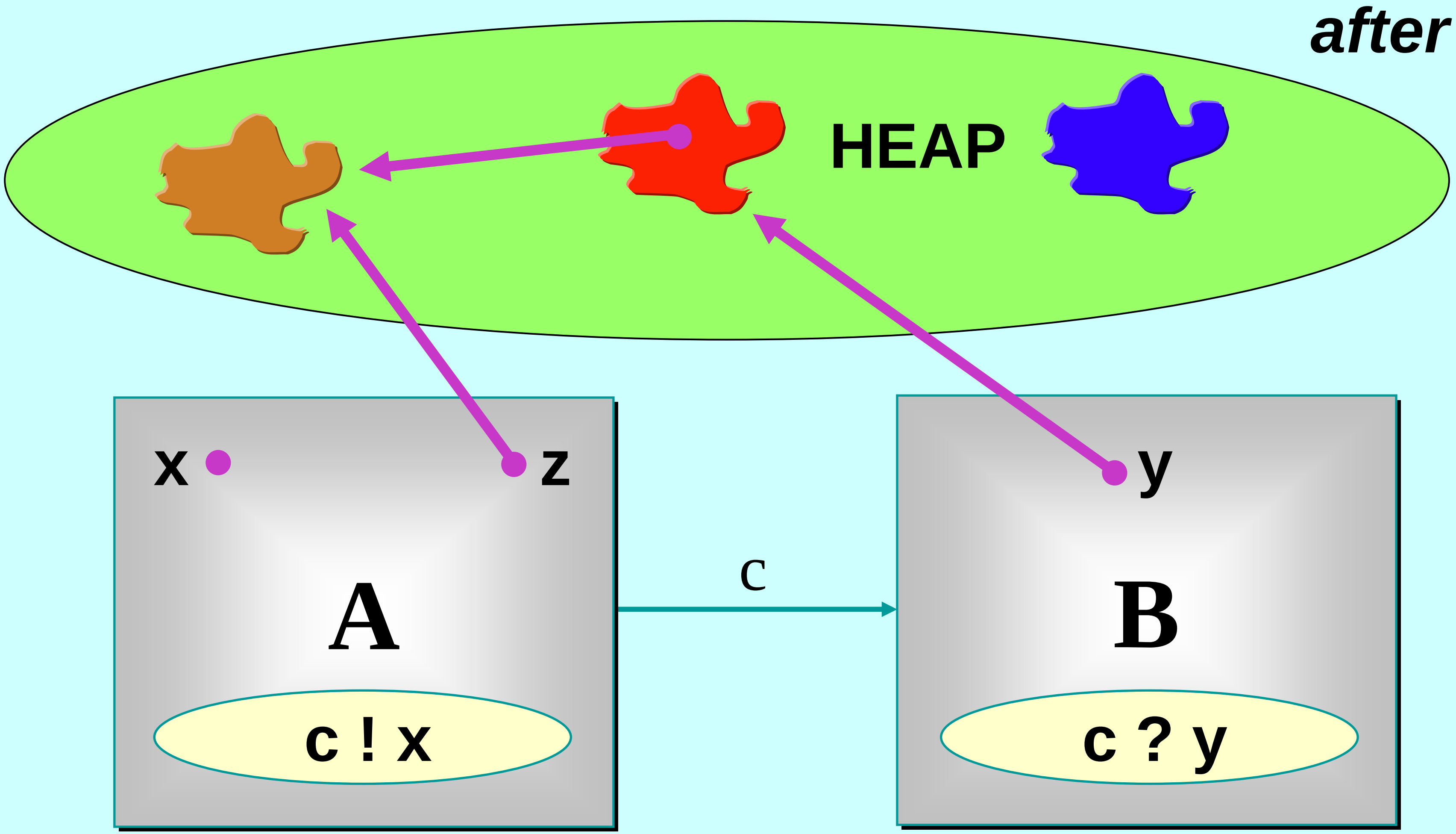


Reference Semantics



Red and brown objects are parallel compromised!

Reference Semantics



Even if the source variable is nulled, brown is done for!!

Classical occam

Different in-scope variables ***implies*** different pieces of data (*zero aliasing*).

Automatic guarantees against ***parallel race hazards*** on data access … and against ***serial aliasing accidents***.

Overheads for ***large*** data communications:

- space (needed at both ends for both copies);
- time (for copying).

Java / JCSP

Hey … it's Java … so ***aliasing*** is endemic.

No guarantees against ***parallel race hazards*** on data access … or against ***serial aliasing accidents***. We must look after ourselves.

Overheads for ***large*** data communications:

- space (***shared*** by both ends);
- time is O(1).

`Object` and `Int` Channels (*interfaces*)

```
interface ChannelOutput {
  public void write (Object o);
}

interface ChannelInput {
  public Object read ();
}

abstract class
 AltingChannelInput
  extends Guard
  implements ChannelInput {
}
```

```
interface ChannelOutputInt {
  public void write (int i);
}

interface ChannelInputInt {
  public int read ();
}

abstract class
 AltingChannelInputInt
  extends Guard
  implements ChannelInputInt {
}
```

Channel-End Interfaces

- *Channel-ends* are what the processes see – they only care what kind of data they carry (`int`s or `Object`s) and whether the channels are for output, input or ***ALTing*** (i.e. *choice*) input.

- It will be the network builder's concern to decide the kinds of *channels* to be used and construct them for connecting processes.

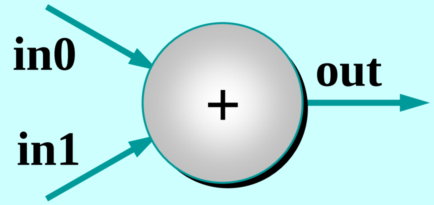
- Let's review some of the *Legoland* processes - this time in **JCSP**.

JCSP Process Structure

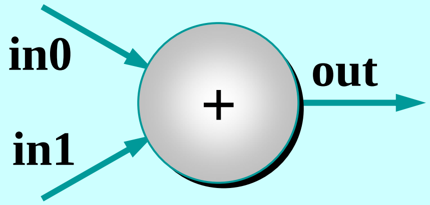
```
class Example implements CSProcess {

  ...   private shared synchronisation objects
        (channels etc.)
  ...   private state information

  ...   public constructors
  ...   public accessors(gets)/mutators(sets)
        (only to be used when not running)

  ...   private support methods (part of a run)
  ...   public void run() (process starts here)

}
```

reminder


```
class SuccInt implements CSProcess {

  private final ChannelInputInt in;
  private final ChannelOutputInt out;

  public SuccInt (ChannelInputInt in,
                  ChannelOutputInt out) {
    this.in = in;
    this.out = out;
  }

  public void run () {
    while (true) {
      int n = in.read ();
      out.write (n + 1);
    }
  }

}
```



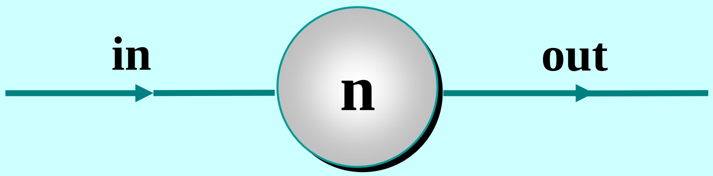

```
class PlusInt implements CSProcess {

  private final ChannelInputInt in0;
  private final ChannelInputInt in1;
  private final ChannelOutputInt out;


  public PlusInt (ChannelInputInt in0,
                  ChannelInputInt in1,
                  ChannelOutputInt out) {
    this.in0 = in0;
    this.in1 = in1;
    this.out = out;
  }

  ...  public void run ()

}
```




```
class PlusInt implements CSProcess {

  ...  private final channels (in0, in1, out)

  ...  public PlusInt (ChannelInputInt in0, ...)

  public void run () {
    while (true) {
      int n0 = in0.read ();
      int n1 = in1.read ();
      out.write (n0 + n1);
    }
  }

}
```

serial ordering

Note: the inputs really need to be done in parallel - later!




```
class PrefixInt implements CSProcess {

  private final int n;
  private final ChannelInputInt in;
  private final ChannelOutputInt out;

  public PrefixInt (int n, ChannelInputInt in,
                    ChannelOutputInt out) {
    this.n = n;
    this.in = in;
    this.out = out;
  }

  public void run () {
    out.write (n);
    new IdInt (in, out).run ();
  }

}
```

Process Networks

- We now want to be able to take instances of these *processes* (or components) and connect them together to form a network.
- The resulting network will itself be a *process*.
- To do this, we need to construct some real wires - these are instances of (JCSP internal) *channel* classes – we only get (Java) *interfaces* to them.
- We also need a way to compose everything together – the **`Parallel`** constructor.

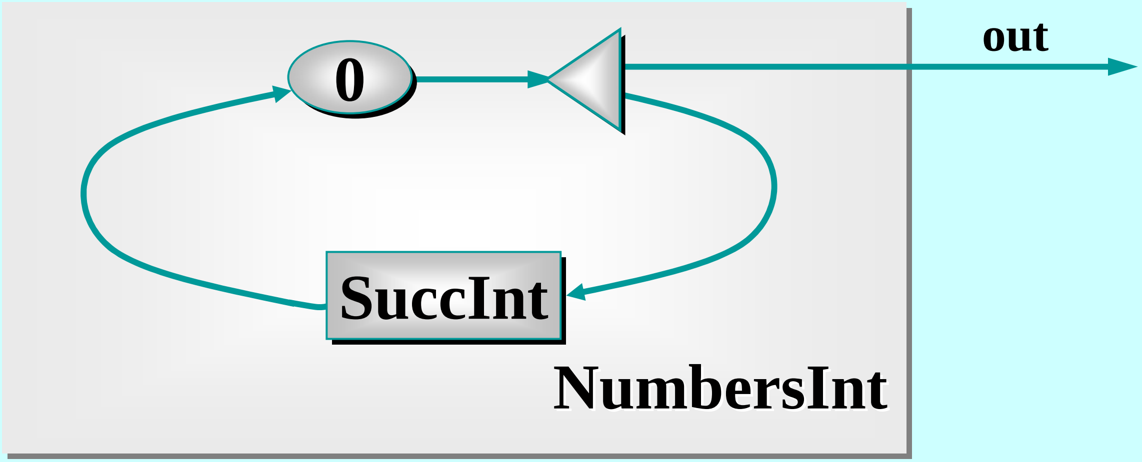
Parallel

- ***Parallel*** is a **`CSProcess`** whose constructor takes an array of **`CSProcess`**es.

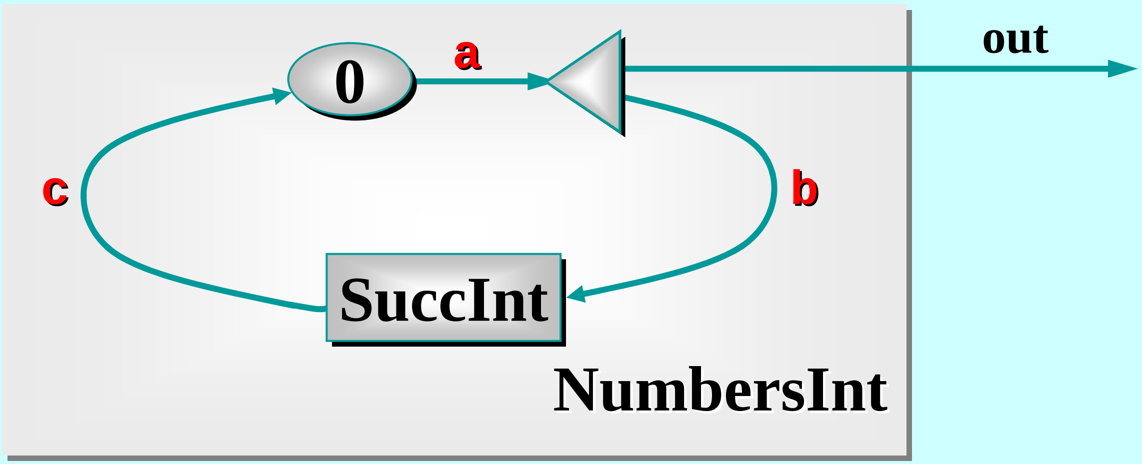
- Its ***run()*** method is the parallel composition of its given **`CSProcess`**es.

- The semantics is the same as for the **occam-π** **PAR** (or CSP **||**).

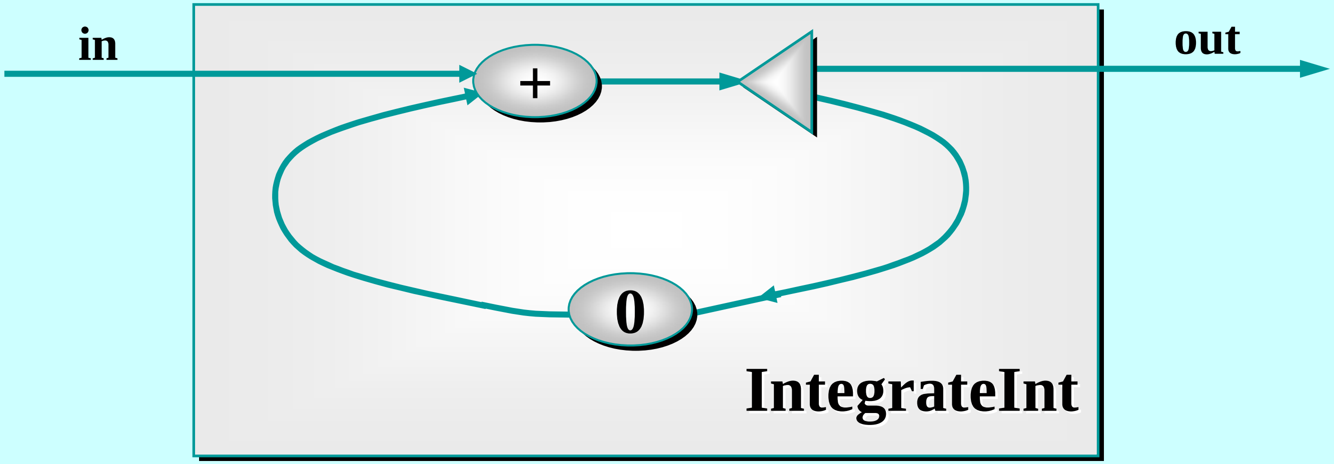
- The ***run()*** terminates when and only when all of its component processes have terminated.



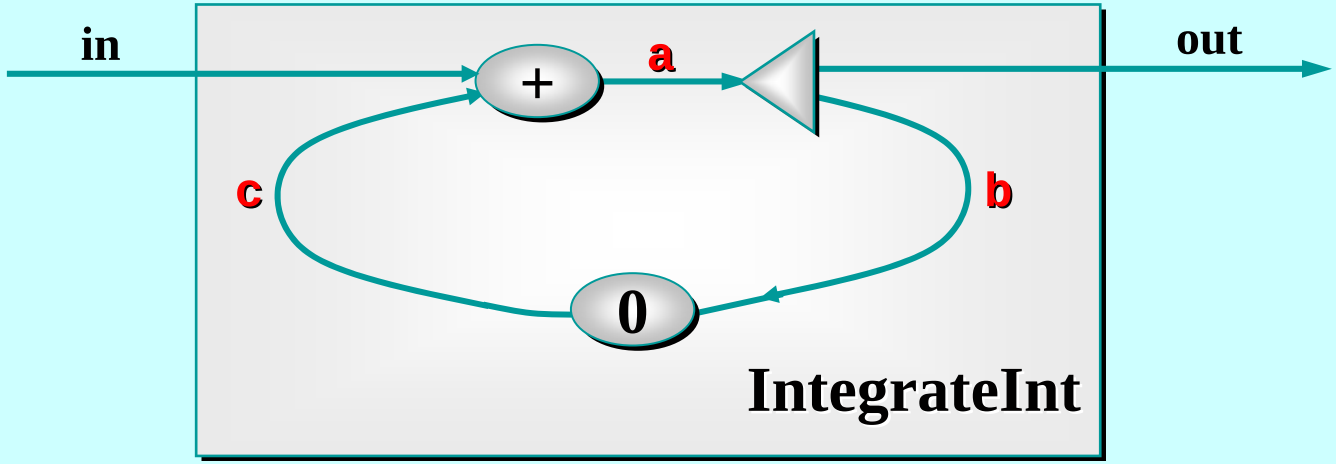

```
class NumbersInt implements CSProcess {

  private final ChannelOutputInt out;

  public NumbersInt (ChannelOutputInt out) {
    this.out = out;
  }

  ...  public void run ()

}
```



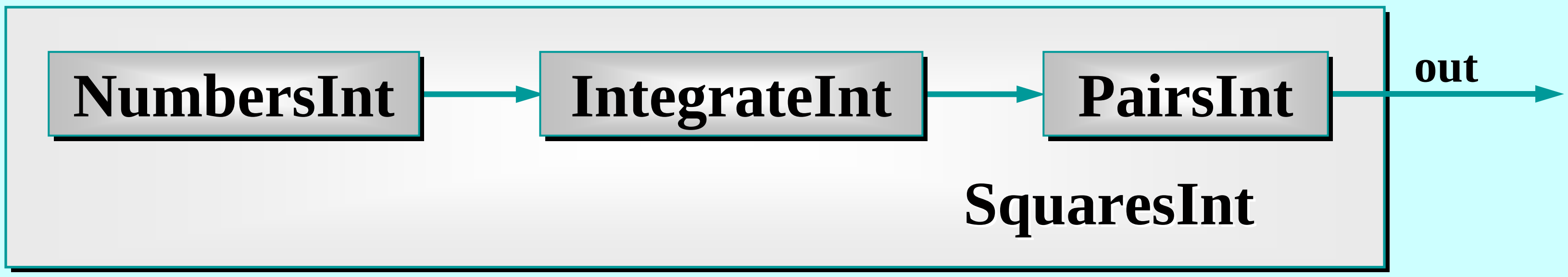

```
public void run () {

  One2OneChannelInt a = Channel.one2oneInt ();
  One2OneChannelInt b = Channel.one2oneInt ();
  One2OneChannelInt c = Channel.one2oneInt ();

  new Parallel (
    new CSProcess[] {
      new PrefixInt (0, c.in(), a.out()),
      new Delta2Int (a.in(), out, b.out()),
      new SuccInt (b.in(), c.out())
    }
  ).run ();

}
```




```
class IntegrateInt implements CSProcess {

  private final ChannelInputInt in;
  private final ChannelOutputInt out;

  public IntegrateInt (ChannelInputInt in,
                       ChannelOutputInt out) {
    this.in = in;
    this.out = out;
  }

  ...  public void run ()

}
```

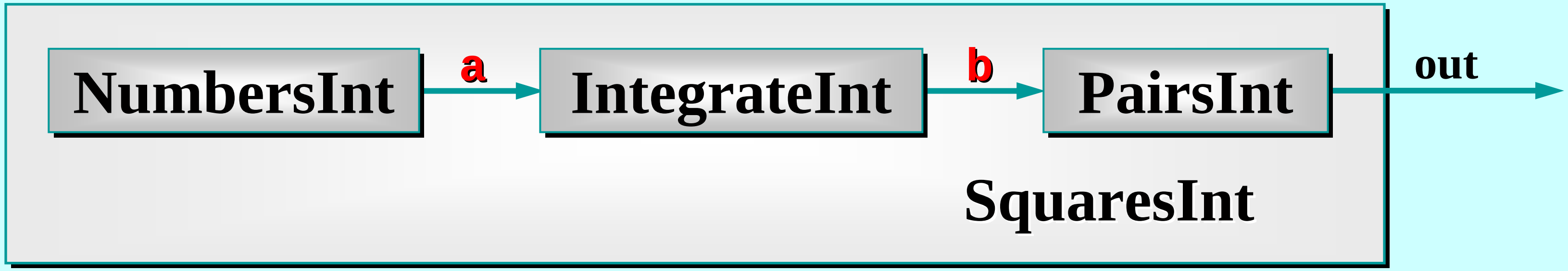



```
public void run () {

  One2OneChannelInt a = Channel.one2oneInt ();
  One2OneChannelInt b = Channel.one2oneInt ();
  One2OneChannelInt c = Channel.one2oneInt ();

  new Parallel (
    new CSProcess[] {
      new PlusInt (in, c.in(), a.out()),
      new Delta2Int (a.in(), out, b.out()),
      new PrefixInt (0, b.in(), c.out())
    }
  ).run ();

}
```



1
4
9
16
25
36
49
64
81
.
.

```
class SquaresInt implements CSProcess {

  private final ChannelOutputInt out;

  public SquaresInt (ChannelOutputInt out) {
    this.out = out;
  }

  ...  public void run ()

}
```

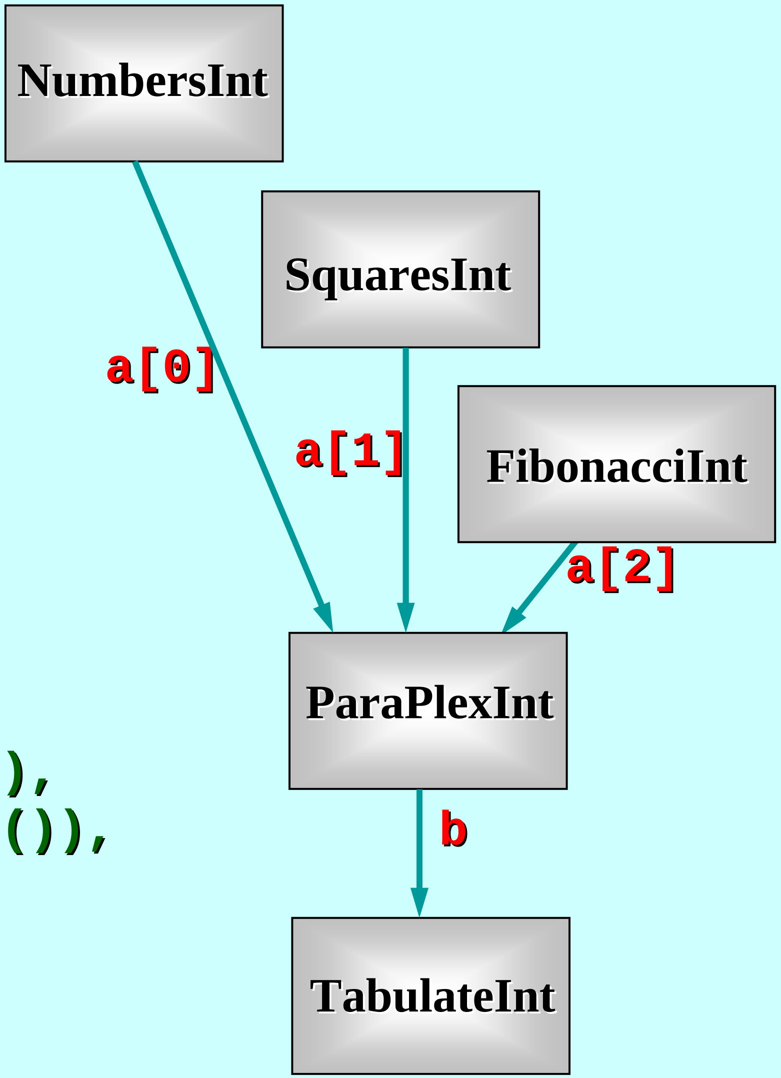


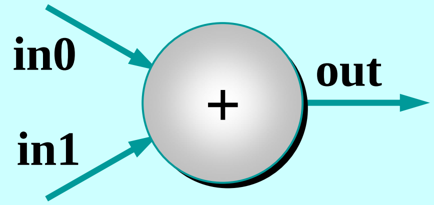

```
public void run () {

  One2OneChannelInt a = Channel.one2oneInt ();
  One2OneChannelInt b = Channel.one2oneInt ();

  new Parallel (
    new CSProcess[] {
      new NumbersInt (a.out()),
      new IntegrateInt (a.in(), b.out()),
      new PairsInt (b.in(), out)
    }
  ).run ();

}
```

1
4
9
16
25
36
49
64
81
.
.

Quite a Lot of Processes

```
One2OneChannelInt[] a =
 Channel.one2oneIntArray(3);
One2OneChannel b =
 Channel.one2one();

ChannelInputInt[] a_in =
 Channel.getInputIntArray(a);

new Parallel (
  new CSProcess[] {
    new NumbersInt (a[0].out()),
    new SquaresInt (a[1].out()),
    new FibonacciInt (a[2].out()),
    new ParaPlexInt (a_in, b.out()),
    new TabulateInt (b.in())
  }
).run ();
```

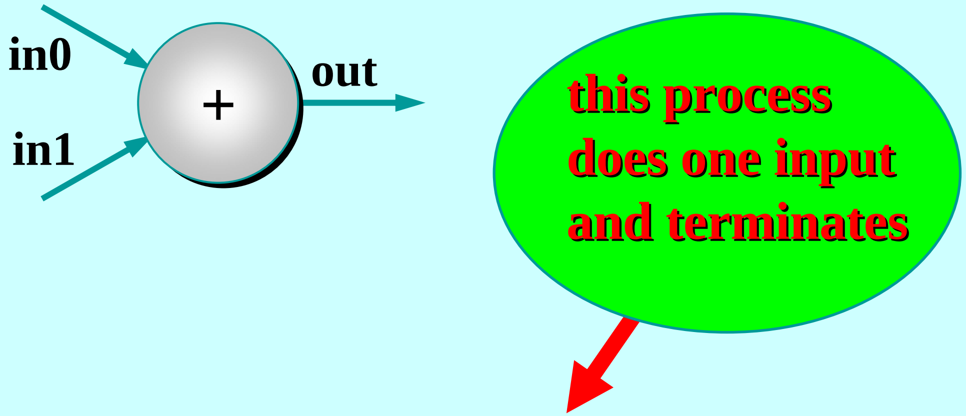





```
class PlusInt implements CSProcess {

  ...  private final channels (in0, in1, out)

  ...  public PlusInt (ChannelInputInt in0, ...)

  public void run () {
    while (true) {
      int n0 = in0.read ();
      int n1 = in1.read ();
      out.write (n0 + n1);
    }
  }

}
```

serial ordering

Note: the inputs really need to be done in parallel - now!




```
public void run () {

  ProcessReadInt readIn0 = new ProcessReadInt (in0);
  ProcessReadInt readIn1 = new ProcessReadInt (in1);

  CSProcess parRead =
    new Parallel (new CSProcess[] {readIn0, readIn1});

  while (true) {
    parRead.run ();
    out.write (readIn0.value + readIn1.value);
  }

}
```

Note: the inputs are now done in parallel.

Implementation Note

- As in the transputer (and KRoC **occam-π** etc.), a **JCSP** **Parallel** object runs its first (n-1) components in *separate* Java threads and its last component in *its own* thread of control.
- When a **Parallel.run()** terminates, the **Parallel** object parks all its threads for reuse in case the **Parallel** is run again.
- So processes like **PlusInt** incur the overhead of Java thread creation *only during its first cycle*.
- That's why we named the **parRead** process before loop entry, rather than constructing it anonymously each time within the loop.

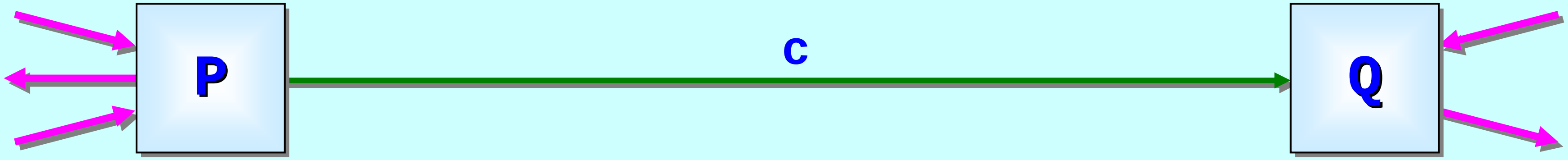
Channel *"Ends"* in occam-π

Each process gets its own "ends" of its external channels

```
PROC P (CHAN STUFF out!, ...)
  ...  local state
  SEQ
    ...  initialise state
    WHILE running
      SEQ
        ...  do stuff
        out ! value
        ...  more stuff
:
```

```
PROC Q (CHAN STUFF in?, ...)
  ...  local state
  SEQ
    ...  initialise state
    WHILE running
      SEQ
        ...  do stuff
        in ? x
        ...  more stuff
:
```

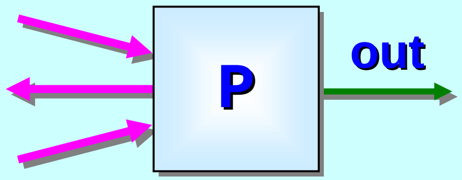
Channel *"Ends"* in occam-π



Each process gets its own "ends" of its external channels

```
CHAN STUFF c:
...  other channels
PAR
  P (c!, ...)
  Q (c?, ...)
  ...  other processes
```

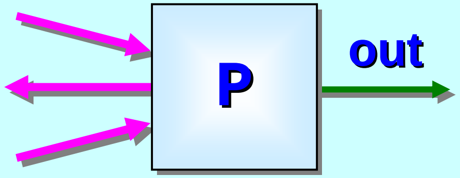
Channel *"Ends"* in JCSP




```
class P implements CSProcess {

  private final ChannelOutput out;
  ...  other channels and local state

  public P (ChannelOutput out, ...) {
    this.out = out;
    ...
  }

  public void run () {...}

}
```

Each process gets its own "ends" of its external channels

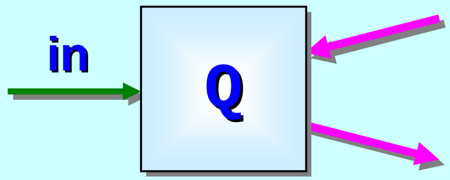
Channel *"Ends"* in JCSP




```
class P implements CSProcess {

  ...  external channels and local state

  public P (ChannelOutput out, ...) {...}


  public void run () {
    ...  initialise local state
    while (running) {
      ...  do stuff
      out.write (value);
      ...  more stuff
  }

}
```

Each process gets its own "ends" of its external channels

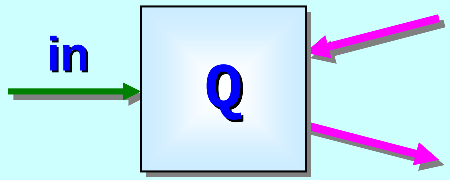
Channel *“Ends”* in JCSP



*Each process gets its own **“ends”** of its external channels*

```
class Q implements CSProcess {

  private final ChannelInput in;
  ...   other channels and local state

  public Q (ChannelInput in, ...) {
    this.in = in;
    ...
  }

  public void run () {...}

:
```

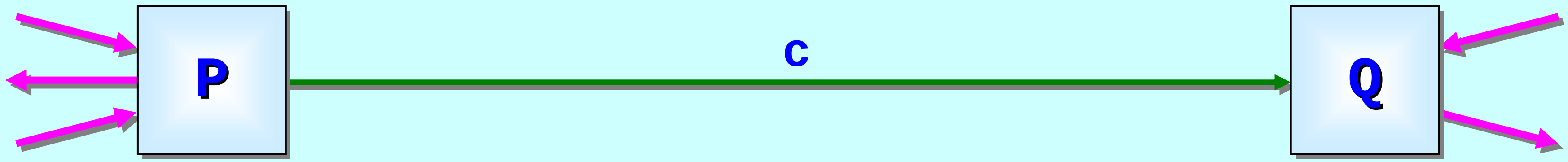
Channel *"Ends"* in JCSP



*Each process gets its own **"ends"** of its external channels*

```
class Q implements CSProcess {

  ...  external channels and local state

  public Q (ChannelInput in, ...) {...}

  public void run () {
    ...  initialise local state
    while (running) {
      ...  do stuff
      x = (Stuff) in.read ();
      ...  more stuff
  }

}
```

Channel *"Ends"* in JCSP 1.1



Each process gets its own "ends" of its external channels

```
final One2OneChannel c = Channel.one2one ();
...  other channels

new Parallel (
  new CSProcess[] {
    new P (c.out (), ...),
    new Q (c.in (), ...),
    ...  other processes
  }
).run ();
```

Channel Interfaces* in JCSP 1.1

ChannelOutput

```
public void write (Object o)
```

ChannelInput

```
public Object read ()
```

One2OneChannel

```
  public ChannelOutput out ()
* public ChannelInput in ()
```

NO DANGER: users see only Java interfaces. The classes behind them are invisible, unrelated by class hierarchy and cannot be cast into each other. Processes must be given correct channel "ends".

**** Ignoring Alting***

Channel Interfaces* in JCSP 1.1

ChannelOutputInt

```
public void write (int i)
```

ChannelInputInt

```
public int read ()
```

One2OneChannelInt

```
  public ChannelOutputInt out ()
* public ChannelInputInt in ()
```

NO DANGER: users see only Java interfaces. The classes behind them are invisible, unrelated by class hierarchy and cannot be cast into each other. Processes must be given correct channel "ends".

**** Ignoring Alting***

Channel Interfaces* in JCSP 1.1

ChannelOutput

```
public void write (Object o)
```

ChannelInput

```
public Object read ()
```

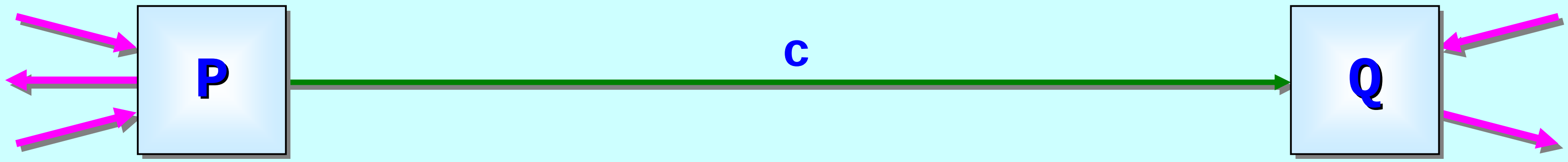
One2OneChannel

```
  public ChannelOutput out ()
* public ChannelInput in ()
```

NO DANGER: users see only Java interfaces. The classes behind them are invisible, unrelated by class hierarchy and cannot be cast into each other. Processes must be given correct channel "ends".

**** Ignoring Alting***

Channel "*Ends*" in JCSP 1.1



Each process gets its own "ends" of its external channels

```
final One2OneChannel c = Channel.one2one ();
...  other channels

new Parallel (
  new CSProcess[] {
    new P (c.out (), ...),
    new Q (c.in (), ...),
    ...  other processes
  }
).run ();
```

channel manufacture

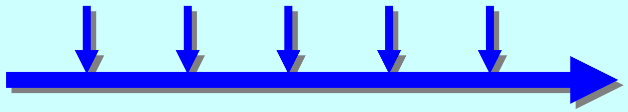
Channel Manufacture

All channels are made using **`static`** methods of the **`Channel`** class.

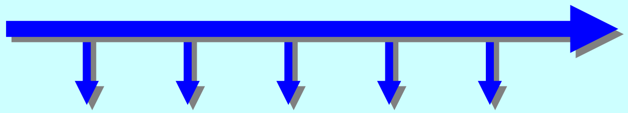
Decide whether the "ends" are to be shared:

`Channel.one2one ()`

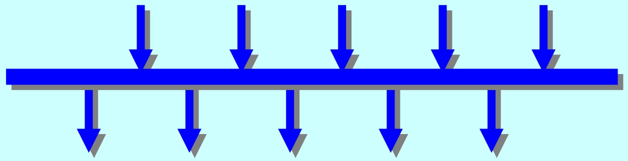
`Channel.any2one ()`



`Channel.one2any ()`



`Channel.any2any ()`



Channel Manufacture

All channels are made using `static` methods of the `Channel` class.

Decide whether the channels are to be buffered and, if so, how:

```
Channel.one2one (new Buffer (42))

Channel.any2one (new OverWriteOldestBuffer (8))

Channel.one2any (new OverFlowingBuffer (100))

Channel.any2any (new InfiniteBuffer ())
```

Channel Manufacture

All channels are made using **`static`** methods of the **`Channel`** class.

Decide whether the channels are poisonable and, if so, their immunity:

```
Channel.one2one (10)

Channel.any2one (5)

Channel.one2any (1000)

Channel.any2any (0)
```

Immunity Level: the channel is immune to poisons up to this strength ...

Channel Manufacture

All channels are made using **static** methods of the **Channel** class.

The channels may be buffered and poisonable:

```
Channel.one2one (new Buffer (42), 10)
```

buffer type and capacity ...

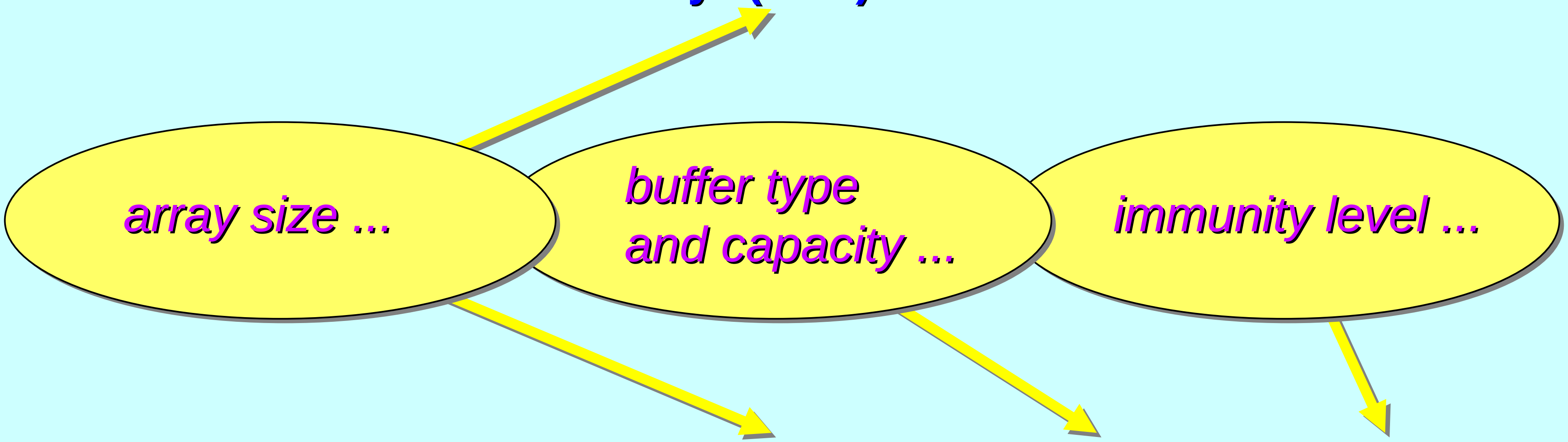
immunity level ...

Channel Manufacture

All channels are made using **`static`** methods of the **`Channel`** class.

Arrays of channels – all kinds – may be built in one go:

```
Channel.one2oneArray (100)
```




```
Channel.any2oneArray (200, new Buffer (42), 10)
```

Channel Manufacture

All channels are made using **static** methods of the **Channel** class.

Channels may be specialised to carry **int**s:

```
Channel.one2oneInt ()

Channel.any2oneIntArray (200, new Buffer (42), 10)
```

In future, channels will be specialised using Java generics ...

Channel Summary

The JCSP process view and use of its external channels:

Sees: **`ChannelInput`**, **`AltingChannelInput`**,
`ChannelOutput`, **`ChannelInputInt`**, *etc.*

Increased safety – cannot violate "endianness" ...

A process does not (usually) care about the kind of channel – whether it is shared, buffered, poisonable, ...*

* If a process needs to share an external channel-end between many sub-processes, it must be given one that is shareable – i.e. an **Any** end. JCSP 1.1 does cater for this.

Channel Summary

The JCSP network view of channels:

The correct channel "ends" must be extracted from channels and plugged into the processes using them ...

Increased safety – cannot violate "endianness" ...

A wide range of channel kinds (fully synchronised, buffered, poisonable, typed) are built from the **`Channel`** *class...*

JCSP processes work only with *interfaces* both for channels (whatever their kind) and for channel-ends. We think this will prove safer than providing *classes*.

Deterministic Processes (CSP)

So far, our parallel systems have been ***deterministic***:

- the values in the output streams depend only on the values in the input streams;
- the semantics is scheduling independent;
- no race hazards are possible.

CSP parallelism, on its own, ***does not introduce non-determinism***.

This gives a firm foundation for exploring real-world models which cannot always behave so simply.

Non-Deterministic Processes (CSP)

In the real world, it is sometimes the case that things happen as a result of:

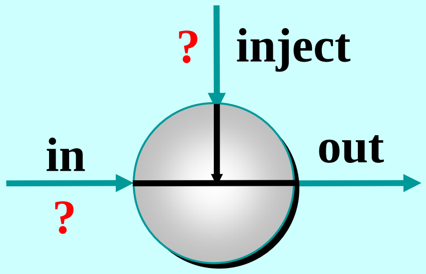
- what happened in the past;
- when (or, at least, in what order) things happened.

In this world, things are scheduling dependent.

CSP (and **JCSP**) addresses these issues ***explicitly***.

Non-determinism does not arise by default.

A Control Process



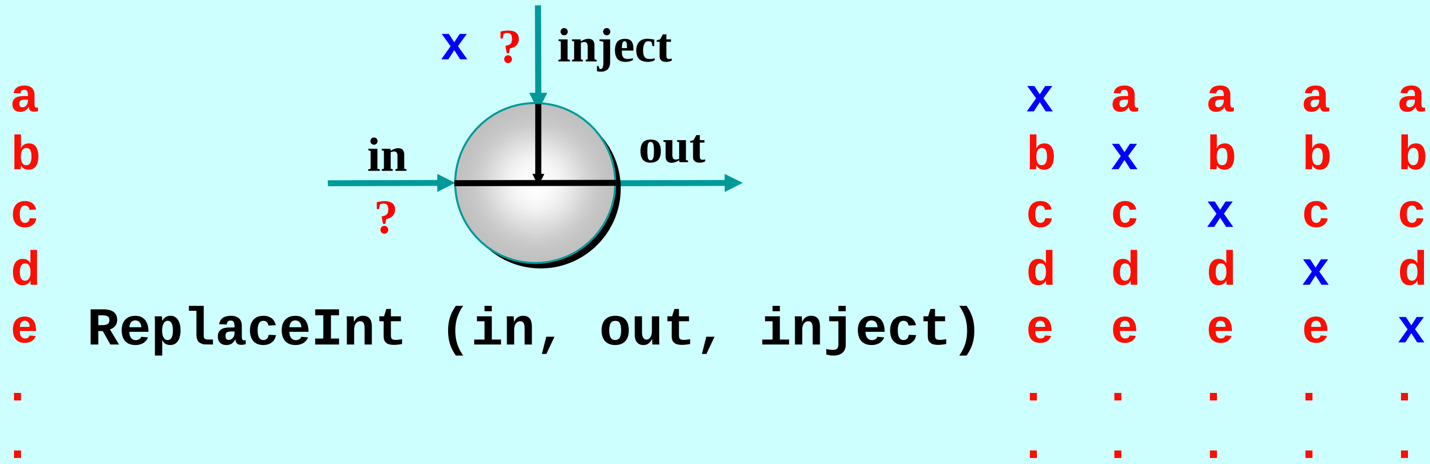
`ReplaceInt (in, out, inject)`

Coping with the real world - making choices …

In **`ReplaceInt`**, data normally flows from **`in`** to **`out`** unchanged.

However, if something arrives on **`inject`**, it is output on **`out`** - ***instead of*** the next input from **`in`**.

A Control Process

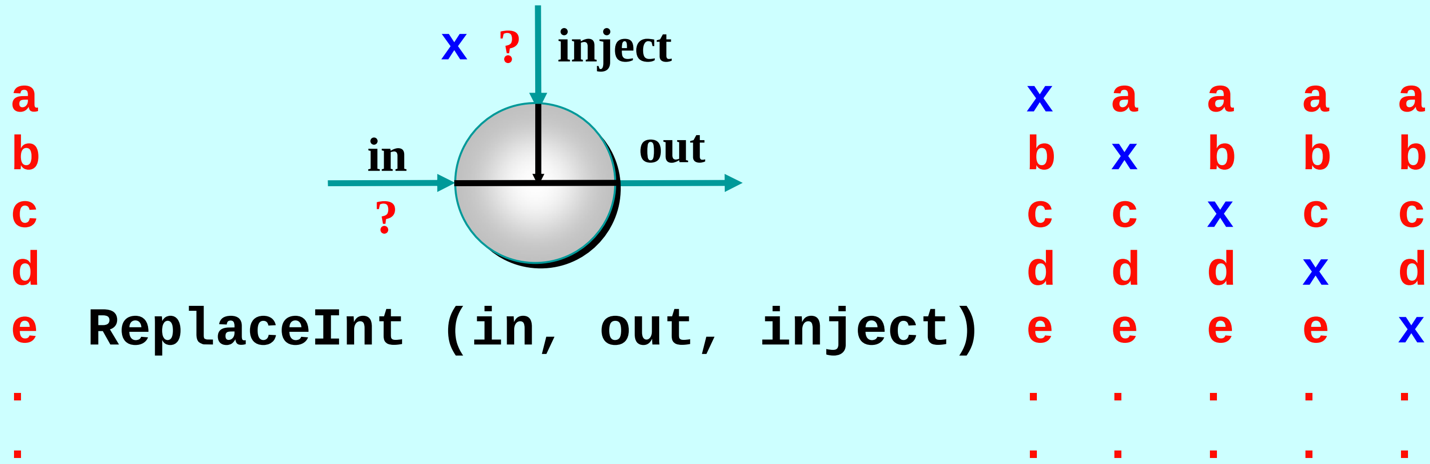


The **out** stream depends upon:

- The values contained in the **in** and **inject** streams;
- the ***order*** in which those values arrive.

The **out** stream is ***not*** determined just by the **in** and **inject** streams - it is ***non-deterministic***.

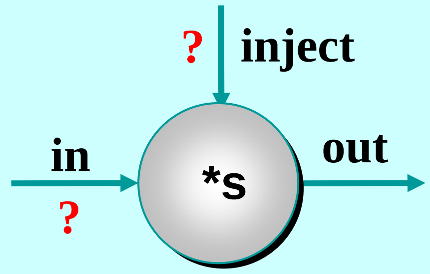
A Control Process




```
ReplaceInt (in, out, inject) =
  (inject?x --> ((in?a --> SKIP) || (out!x --> SKIP))
   [PRI]
   in?a --> out!a --> SKIP
  );
  ReplaceInt (in, out, inject)
```

Note: **`[]`** is the (external) choice operator of CSP.
`[PRI]` is a prioritised version - giving priority to the event on its left.

Another Control Process



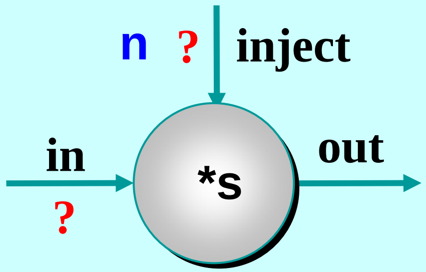
`ScaleInt (s, in, out, inject)`

Coping with the real world - making choices …

In **`ScaleInt`**, data flows from **`in`** to **`out`**, getting scaled by a factor of **`s`** as it passes.

Values arriving on **`inject`**, reset that **`s`** factor.

Another Control Process



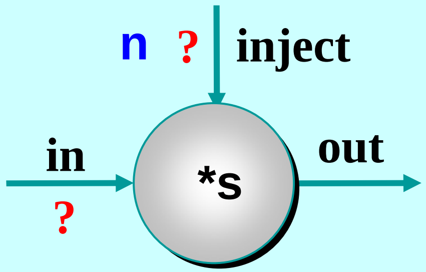
a		n*a	s*a	s*a
b		n*b	n*b	s*b
c		n*c	n*c	n*c
d		n*d	n*d	n*d
e	`ScaleInt (s, in, out, inject)`	n*e	n*e	n*e
.		.	.	.
.		.	.	.

The **`out`** stream depends upon:

- The values contained in the **`in`** and **`inject`** streams;
- the ***order*** in which those values arrive.

The **`out`** stream is ***not*** determined just by the **`in`** and **`inject`** streams - it is ***non-deterministic***.

Another Control Process



a   n*a   s*a   s*a

b   n*b   n*b   s*b

c   n*c   n*c   n*c

d   n*d   n*d   n*d

e **ScaleInt (s, in, out, inject)** n*e   n*e   n*e

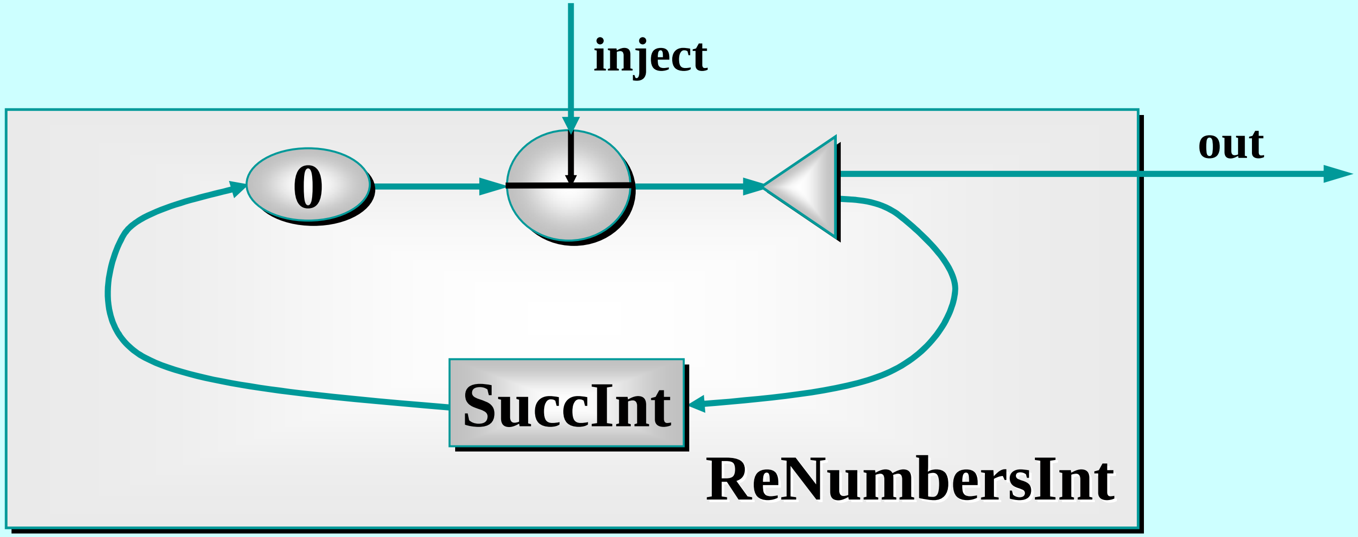
.   .   .   .

.   .   .   .

```
ScaleInt (s, in, out, inject) =
  (inject?s --> SKIP
   [PRI]
   in?a --> out!s*a --> SKIP
  );
 ScaleInt (s, in, out, inject)
```

Note: **[]** is the (external) choice operator of CSP.
[PRI] is a prioritised version - giving priority to the event on its left.

Some Resettable Networks

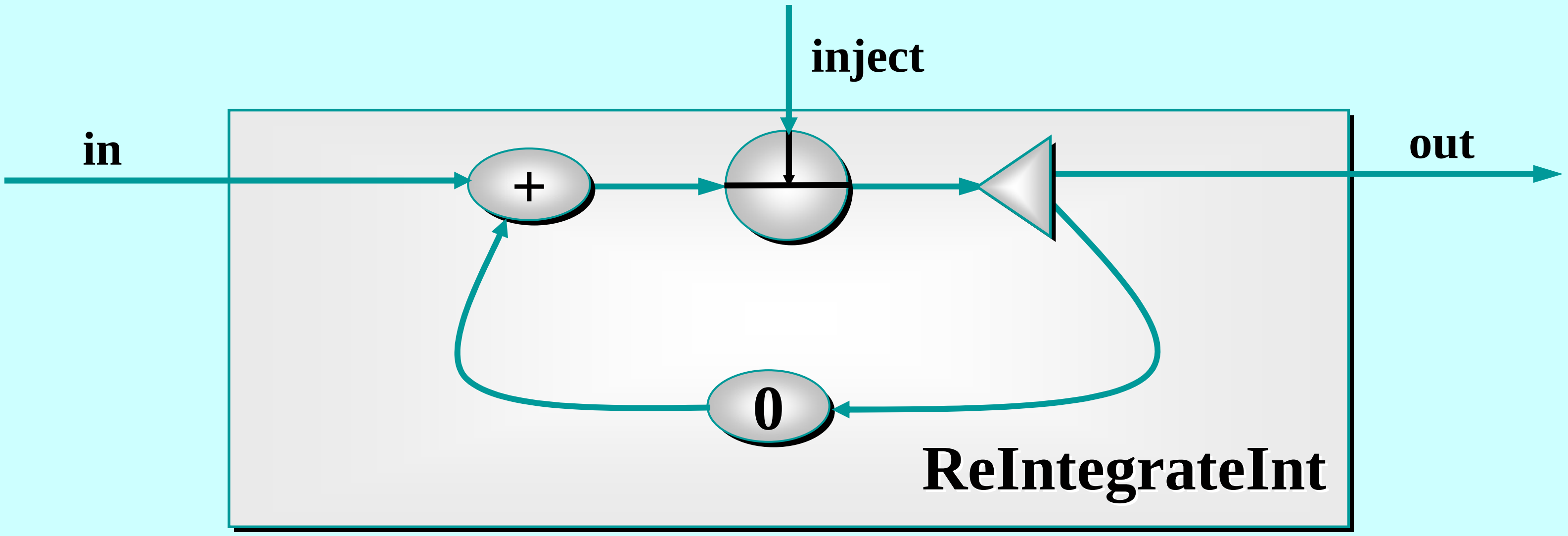


This is a *resettable* version of the **NumbersInt** process.

If nothing is sent down **inject**, it behaves as before.

But it may be reset to count from *any* number at *any* time.

Some Resettable Networks

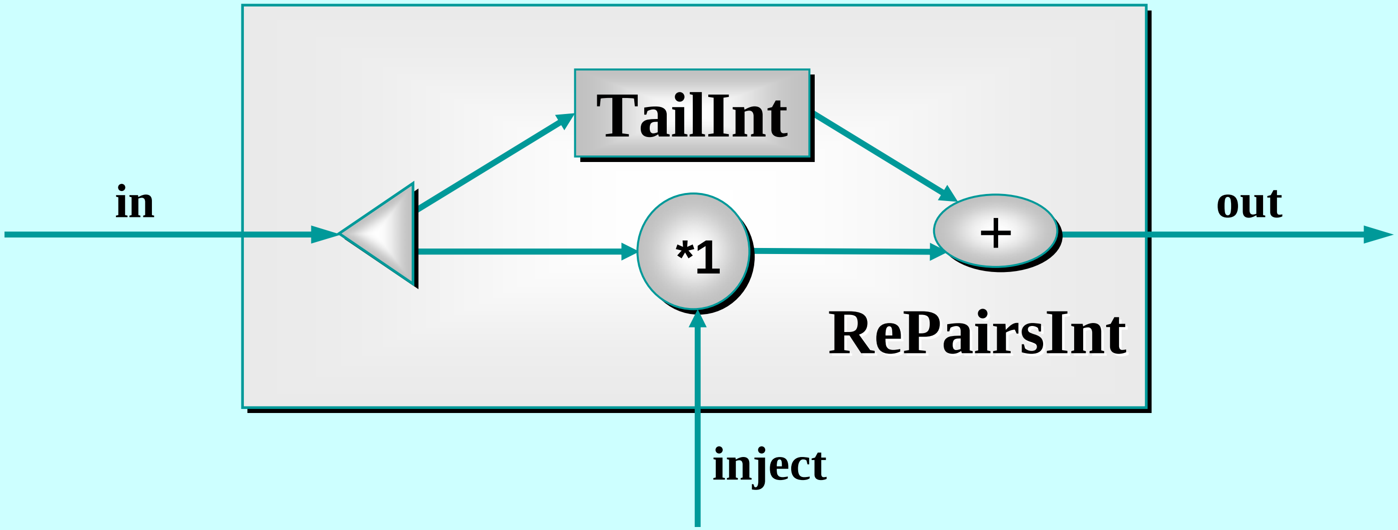


This is a *resettable* version of the **`IntegrateInt`** process.

If nothing is sent down **`inject`**, it behaves as before.

But its running sum may be reset to *any* number at *any* time.

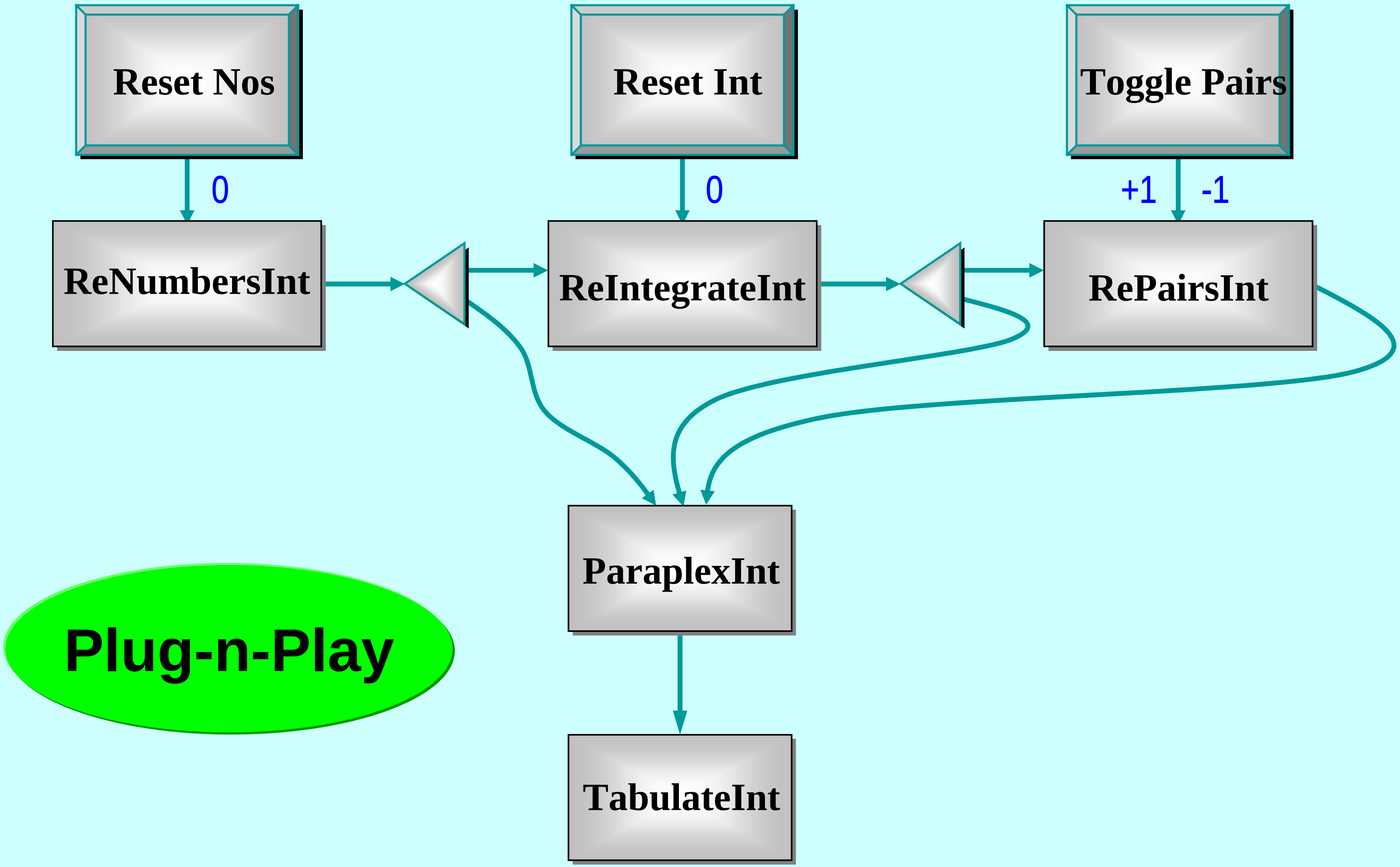
Some Resettable Networks



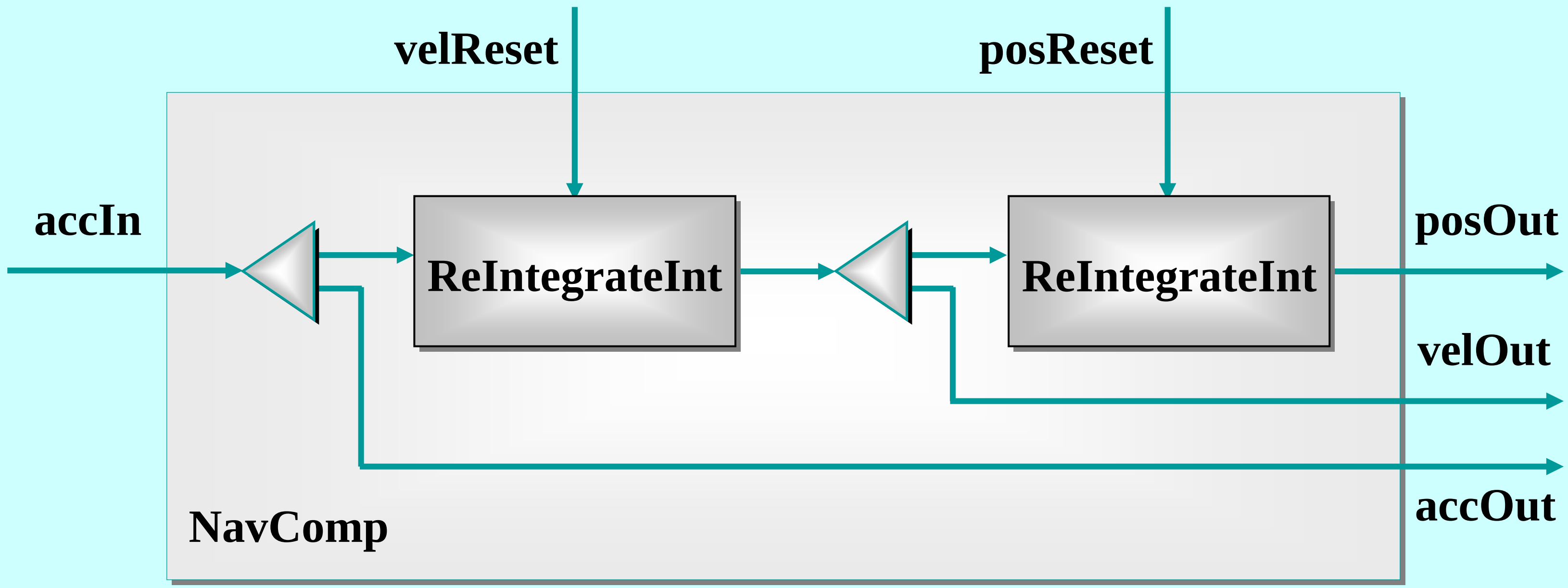
This is a *resettable* version of the **PairsInt** process.

By sending **-1** or **+1** down **inject**, we can toggle its behaviour between **PairsInt** and **DiffentiateInt** (a device that cancels the effect of **IntegrateInt** if pipelined on to its output).

A Controllable Machine



An Inertial Navigation Component



- **`accIn`**: carries *regular* accelerometer samples;
- **`velReset`**: velocity *initialisation* and *corrections*;
- **`posReset`**: position *initialisation* and *corrections*;
- **`posOut`/`velOut`/`accOut`**: *regular* outputs.

Deterministic Processes (JCSP)

So far, our JCSP systems have been ***determistic***:

- the values in the output streams depend only on the values in the input streams;
- the semantics is scheduling independent;
- no race hazards are possible.

CSP parallelism, on its own, *does not introduce non-determinism*.

This gives a firm foundation for exploring real-world models which cannot always behave so simply.

Non-Deterministic Processes (JCSP)

In the real world, it is sometimes the case that things happen as a result of:

- what happened in the past;
- when (or, at least, in what order) things happened.

In this world, things are scheduling dependent.

CSP (JCSP) addresses these issues ***explicitly***.

Non-determinism does not arise by default.

Alternation - the CSP Choice

```
public abstract class Guard {
  ...  package-only abstract methods (enable/disable)
}
```

Six **JCSP** classes are (i.e. **`extend`**) **`Guard`**s:

`AltingChannelInput`	**(Objects)**
`AltingChannelInputInt`	**(ints)**
`AltingChannelAccept`	**(CALLs)**
`AltingBarrier`	**(Barriers)**
`CSTimer`	**(timeouts)**
`Skip`	**(polling)**

The **`in()`** methods of *One2One* and *Any2One* channels return *Alting* (rather than ordinary) channel-ends.

The **`in()`** methods of *One2Any* and *Any2Any* channels return ordinary channel-ends – no *Alting* on them.

Alternation* - the CSP Choice

```
public abstract class Guard {
  ...  package-only abstract methods (enable/disable)
}
```

Six **JCSP** classes are (i.e. **`extend`**) **`Guard`**s:

`AltingChannelInput`	**(Objects)**
`AltingChannelInputInt`	**(ints)**
`AltingChannelAccept`	**(CALLs)**
`AltingBarrier`	**(Barriers)**
`CSTimer`	**(timeouts)**
`Skip`	**(polling)**

The **`in()`** methods of *One2One* and *Any2One* channels return *Alting* (rather than ordinary) channel-ends.

***Alternation is named after the occam ALT …**

Ready/Unready Guards

- A **channel** guard is ready **iff** *data is pending* - i.e. a process at the other end has output to (or called) the channel and this has not yet been input (or accepted).
- A **timer** guard is ready **iff** *its timeout has expired*.
- A **skip** guard is *always ready*.

Alternation

For ***ALTing***, a **JCSP** process must have a **`Guard[]`** array - this can be any mix of channel inputs, call channel accepts, timeouts or skips:

```
final Guard[] guards = {...};
```

It must construct an ***Alternative*** object for each such guard array:

```
final Alternative alt =
  new Alternative (guards);
```

The ***ALT*** is carried out by invoking one of the three varieties of select methods on the alternative.

`alt.select()`

This blocks passively until one or more of the guards are ready. Then, it makes an ***ARBITRARY*** choice of one of these ready guards and returns the index of that chosen one. If that guard is a ***channel***, the *ALTing* process must then ***read*** from (or ***accept)*** it.

`alt.priSelect()`

Same as above - except that if there is more than one ready guard, it chooses the one with the ***lowest index***.

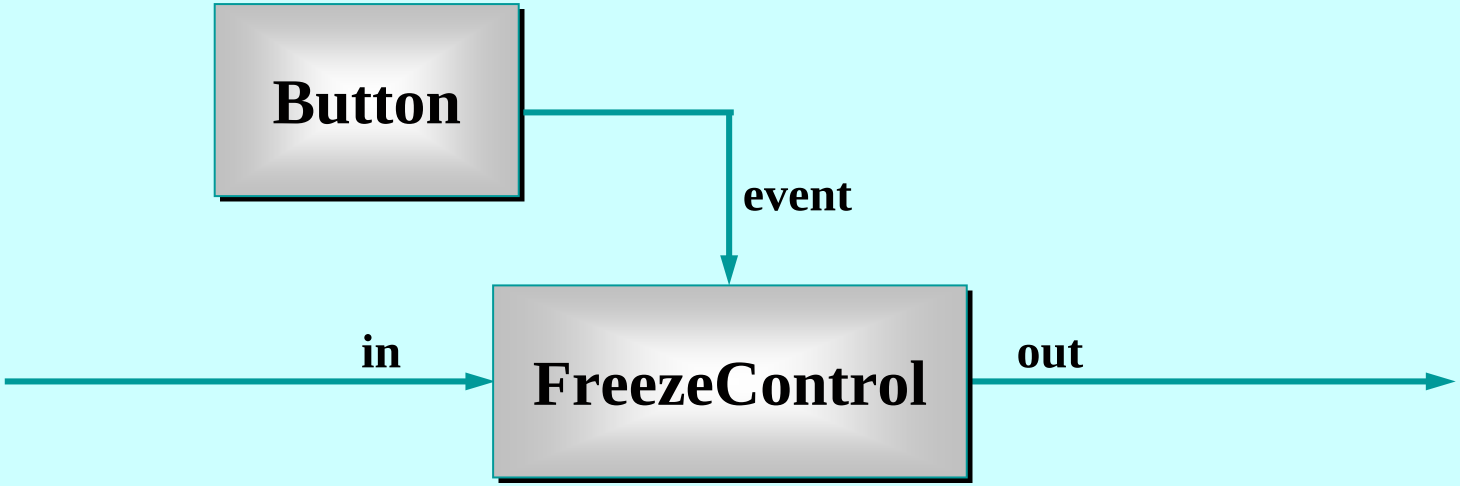
alt.fairSelect()

Same as above - except that if there are more than one ready guards, it makes a ***FAIR*** choice.

This means that, in successive invocations of ***alt.fairSelect ()***, no ready guard will be chosen twice if another ready guard is available. At worst, no ready guard will miss out on ***n*** successive selections (where ***n*** is the number of guards).

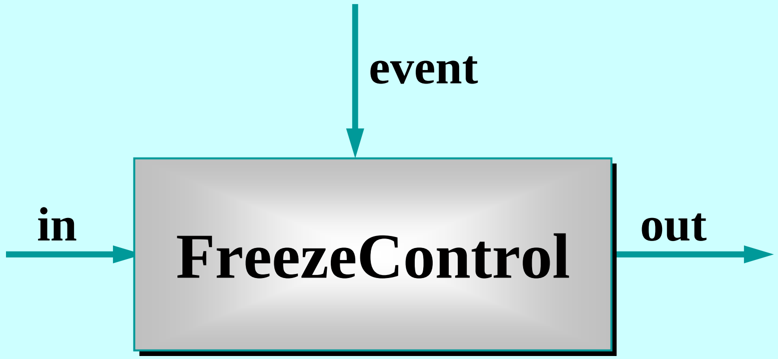
Fair alternation is possible because an ***Alternative*** object is tied to ***one*** set of guards.

ALTing Between Events



- **`Button`** is a (GUI widget) process that outputs a *ping* whenever it's clicked.
- **`FreezeControl`** controls a data-stream flowing from its **`in`** to **`out`** channels. Clicking the **`Button`** freezes the data-stream - clicking again resumes it.

ALTing Between Events



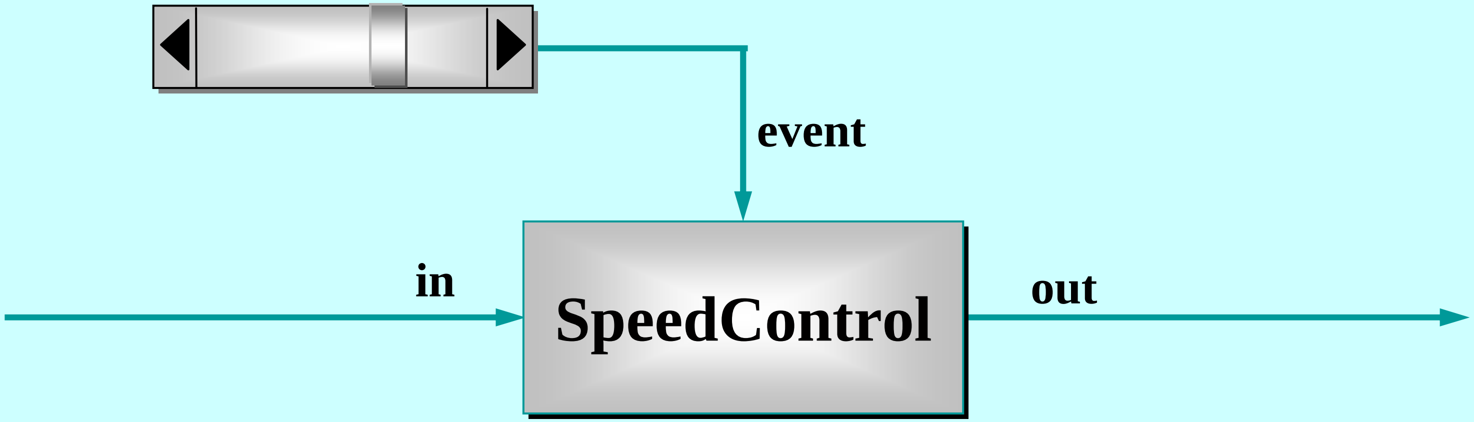

```
final Alternative alt =
  new Alternative (
    new Guard[] {event, in};
  );

final int EVENT = 0, IN = 1;
```

Indices to the Guard array

```
while (true) {

  switch (alt.priSelect ()) {

    case EVENT:
      event.read ();
      event.read ();
    break;

    case IN:
      out.write (in.read ());
    break;

  }

}
```

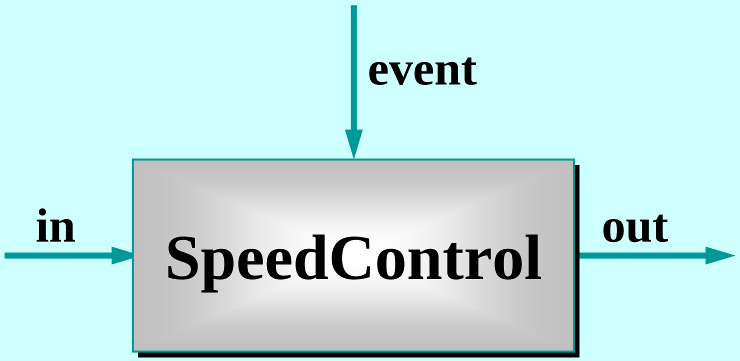
No *SPIN* when frozen

ALTing Between Events



- The *slider* (GUI widget) process outputs an integer (**`0`**..**`100`**) whenever its *slider-key* is moved.
- **`SpeedControl`** controls the speed of a data-stream flowing from its **`in`** to **`out`** channels. Moving the *slider-key* changes that speed – from *frozen* (**`0`**) to some defined *maximum* (**`100`**).

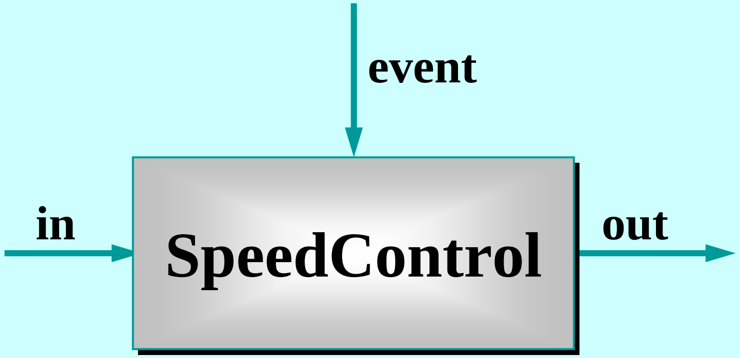
ALTing Between Events




```
final CSTimer tim =
  new CSTimer ();

final Alternative alt =
  new Alternative (
    new Guard[] {event, tim};
  );

final int EVENT = 0, TIM = 1;
```

```
long timeout = tim.read () + interval;
tim.setAlarm (timeout);

while (true) {

  switch (alt.priSelect ()) {

    case EVENT:
      ...  handle the slider event

    case TIM:
      ...  handle the timeout event

  }

}
```

ALTing Between Events

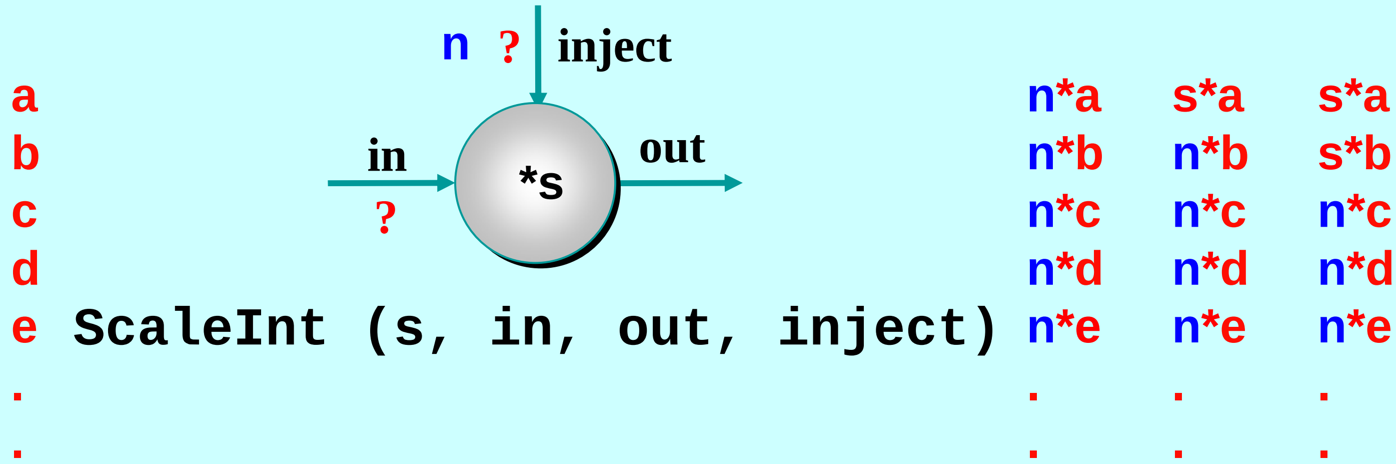



```
final CSTimer tim =
  new CSTimer ();

final Alternative alt =
  new Alternative (
    new Guard[] {event, tim};
  );

final int EVENT = 0, TIM = 1;
```

```
long timeout = tim.read () + interval;
tim.setAlarm (timeout);

while (true) {

  switch (alt.priSelect ()) {

    case EVENT:
      int position = event.read ();
      while (position == 0) {
        position = event.read ();
      }
      speed = (position*maxSpd)/maxPos
      interval = 1000/speed;  // ms
      timeout = tim.read ();
      // fall through

    case TIM:
      timeout += interval;
      tim.setAlarm (timeout);
      out.write (in.read ());
    break;

  }
}
```

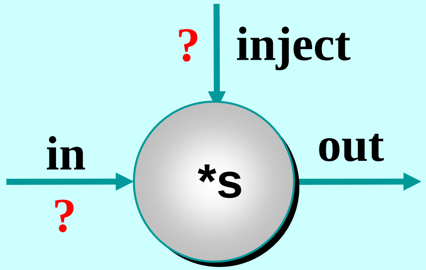
No *SPIN*
when frozen

Another Control Process



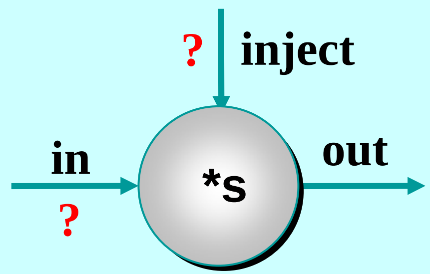

```
ScaleInt (s, in, out, inject) =
  (inject?s --> SKIP
   [PRI]
   in?a --> out!s*a --> SKIP
  );
 ScaleInt (s, in, out, inject)
```

Note: **[]** is the (external) choice operator of CSP.
[PRI] is a prioritised version - giving priority to the event on its left.



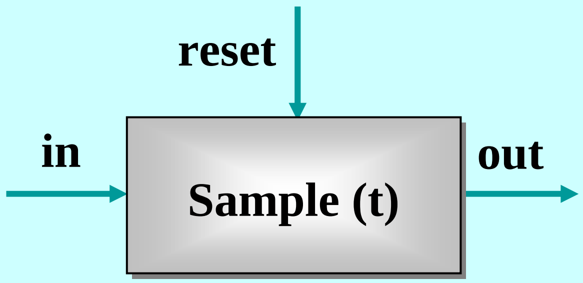

```
class ScaleInt implements CSProcess {

  private int s;
  private final AltingChannelInputInt in, inject;
  private final ChannelOutputInt out;

  public ScaleInt (int s, AltingChannelInputInt in,
                   AltingChannelInputInt inject,
                   ChannelOutputInt out) {
    this.s = s;
    this.in = in;
    this.inject = inject;
    this.out = out;
  }

  ...  public void run ()

}
```

```
public void run () {

  final Alternative alt =
    new Alternative (new Guard[] {inject, in});

  final int INJECT = 0, IN = 1;  // guard indices

  while (true) {
    switch (alt.priSelect ()) {
      case INJECT:
        s = inject.read ();
      break;
      case IN:
        final int a = in.read ();
        out.write (s*a);
      break;
    }
  }

}
```

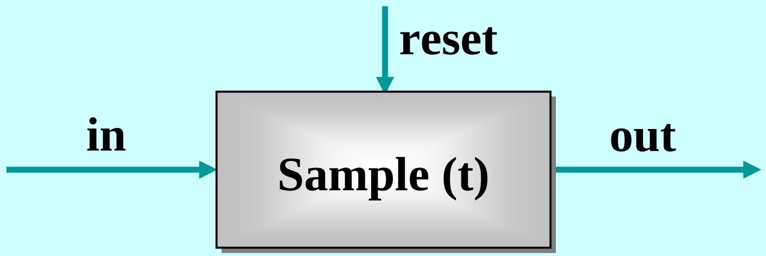


Note these are in priority order.

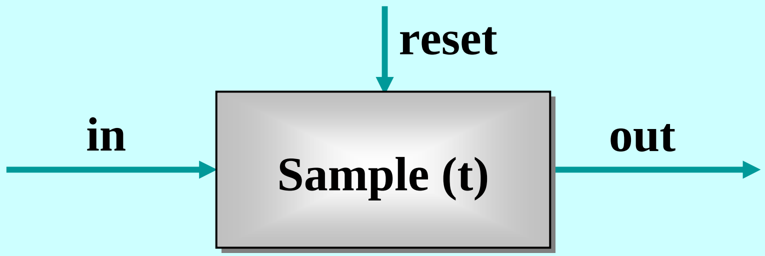
Real-Time Sampler



- This process services any of 3 events *(2 inputs and 1 timeout)* that may occur.
- Its **`t`** parameter represents a time interval. Every **`t`** time units, it must output the *last* object that arrived on its **`in`** channel during the previous time slice. If nothing arrived, it must output a **`null`**.
- The length of the timeslice, **`t`**, may be reset at any time by a new value arriving on its **`reset`** channel.

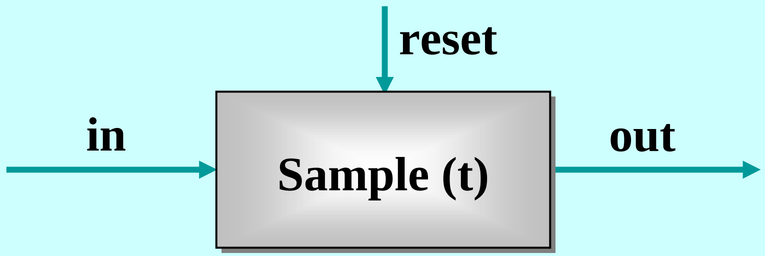



```
class Sample implements CSProcess {

  private final long t;
  private final AltingChannelInput in;
  private final AltingChannelInputInt reset;
  private final ChannelOutput out;

  public Sample (long t,
                 AltingChannelInput in,
                 AltingChannelInputInt reset,
                 ChannelOutput out) {
    this.t = t;
    this.in = in;
    this.reset = reset;
    this.out = out;
  }

  ...  public void run ()

}
```

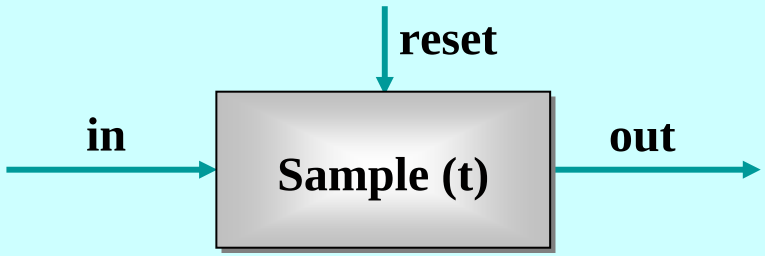


Note these are in priority order.

```
public void run () {

  final CSTimer tim = new CSTimer ();

  final Alternative alt =
    new Alternative (new Guard[] {reset, tim, in});

  final int RESET = 0, TIM = 1, IN = 2;   // indices

  Object sample = null;
  long timeout = tim.read () + t;
  tim.setAlarm (timeout);

  ...  main loop

}
```

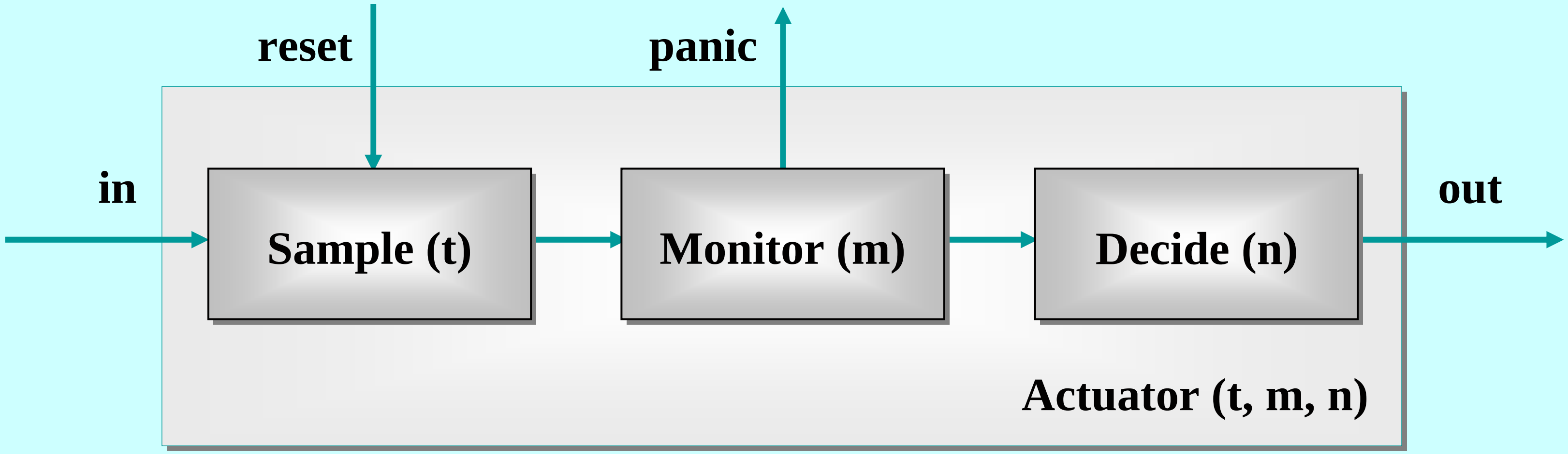



```
while (true) {
  switch (alt.priSelect ()) {
    case RESET:
      t = reset.read ();
    break;
    case TIM:
      out.write (sample);
      sample = null;
      timeout += t;
      tim.setAlarm (timeout);
    break;
    case IN:
      sample = in.read ();
    break;
  }
}
```

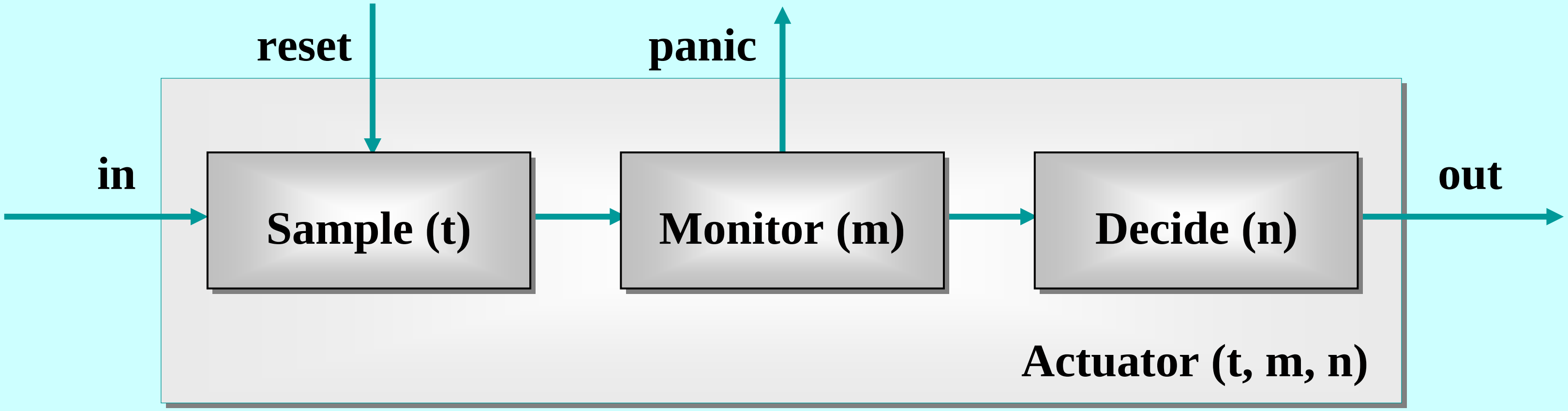



```
while (true) {
  switch (alt.priSelect ()) {
    case RESET:
      t = reset.read ();
      timeout = tim.read ();  // fall through
    case TIM:
      out.write (sample);
      sample = null;
      timeout += t;
      tim.setAlarm (timeout);
    break;
    case IN:
      sample = in.read ();
    break;
  }
}
```

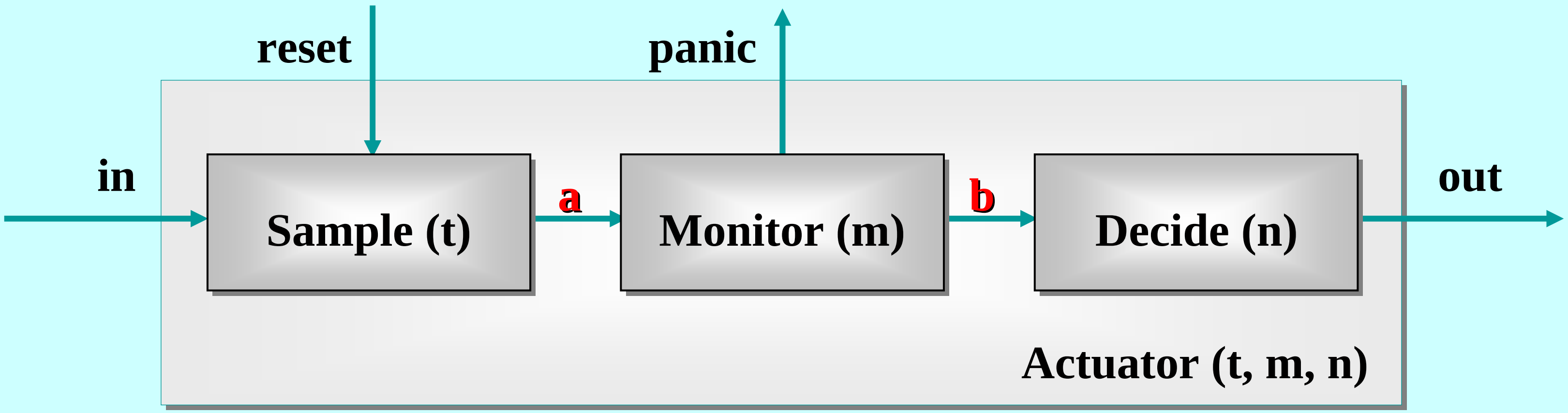
Final Stage Actuator



- **`Sample(t)`**: *every* **`t`** time units, output *latest* **`in`**put (or **`null`** *if none*); the value of **`t`** may be **`reset`**;
- **`Monitor(m)`**: copy input to output counting **`null`**s - if **`m`** *in a row*, send panic message and terminate;
- **`Decide(n)`**: copy non-**`null`** input to output and *remember* last n outputs - convert **`null`**s to a *best guess* depending on those last n outputs.




```
class Actuator implements CSProcess {

  ...   private state (t, m and n)

  ...   private interface channels
        (in, reset, panic and out)

  ...   public constructor
        (assign parameters t, m, n, in, reset,
         panic and out to the above fields)

  ...   public void run ()

}
```




```
public void run ()

  final One2OneChannel a = Channel.One2One ();
  final One2OneChannel b = Channel.One2One ();

  new Parallel (
    new CSProcess[] {
      new Sample (t, in, reset, a.out()),
      new Monitor (m, a.in(), panic, b.out()),
      new Decide (n, b.in(), out)
    }
  ).run ();

}
```

Pre-conditioned Alternation

We may set an array of **`boolean`** *pre-conditions* on any of the **`select`** operations of an **`Alternative`**:

```
switch (alt.fairSelect (depends)) {...}
```

The **`depends`** array must have the same length as the **`Guard`** array to which the **`alt`** is bound.

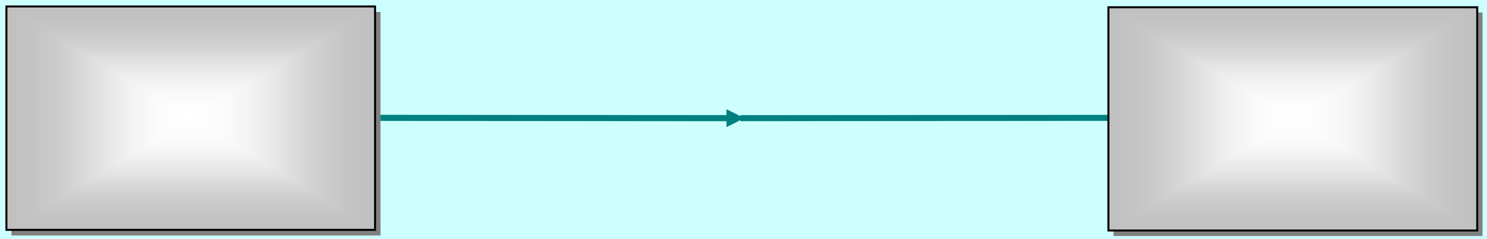
The **`depends`** array, set at run-time, *enables/disables* the guards at corresponding indices. If **`depends[i]`** is **`false`**, that guard will be ignored - even if *ready*.

This gives considerable flexibility to how we program the willingness of a process to service events.

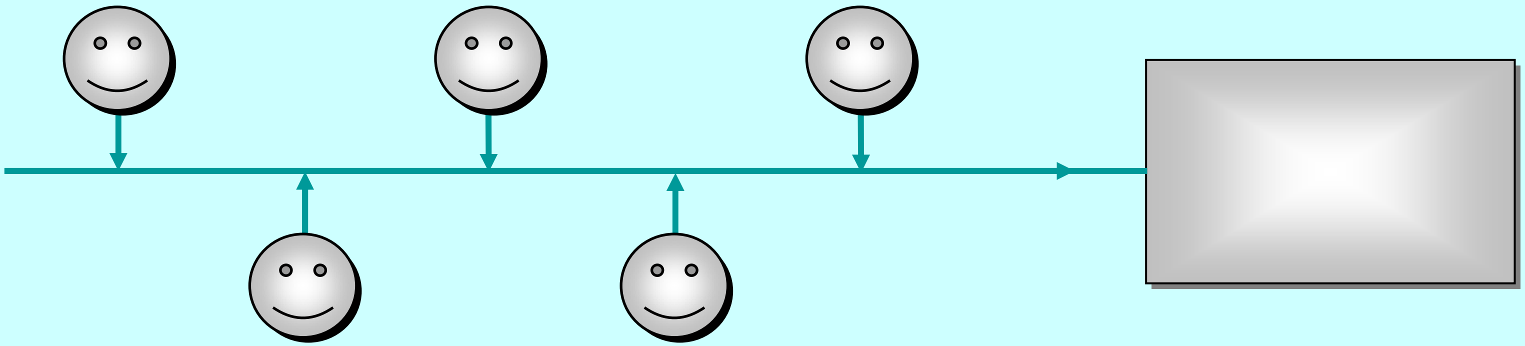
Shared Channels

- So far, all our channels have been point-to-point, zero-buffered and synchronised (i.e. standard **CSP** primitives);
- **JCSP** also offers multi-way shared channels (in the style of **occam-π**);
- **JCSP** also offers buffered channels of various well-defined forms.

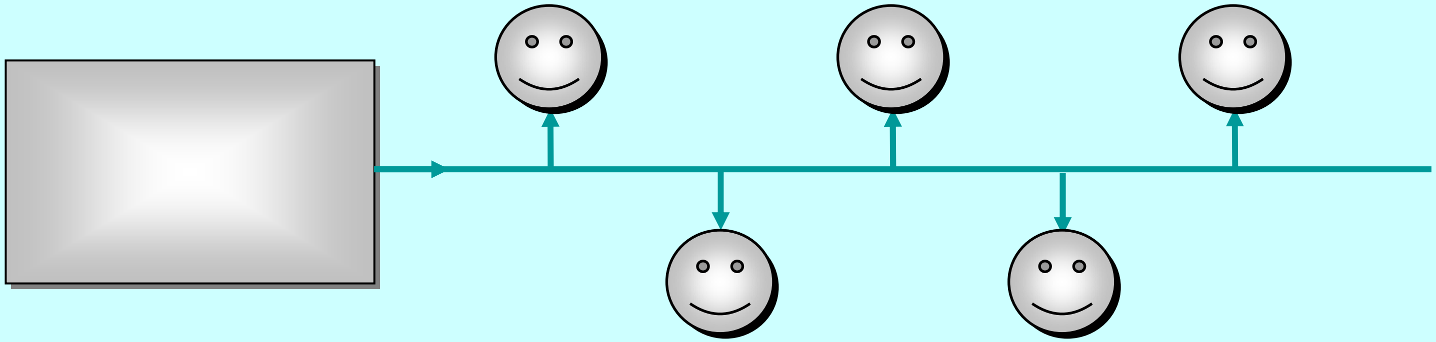
One2OneChannel



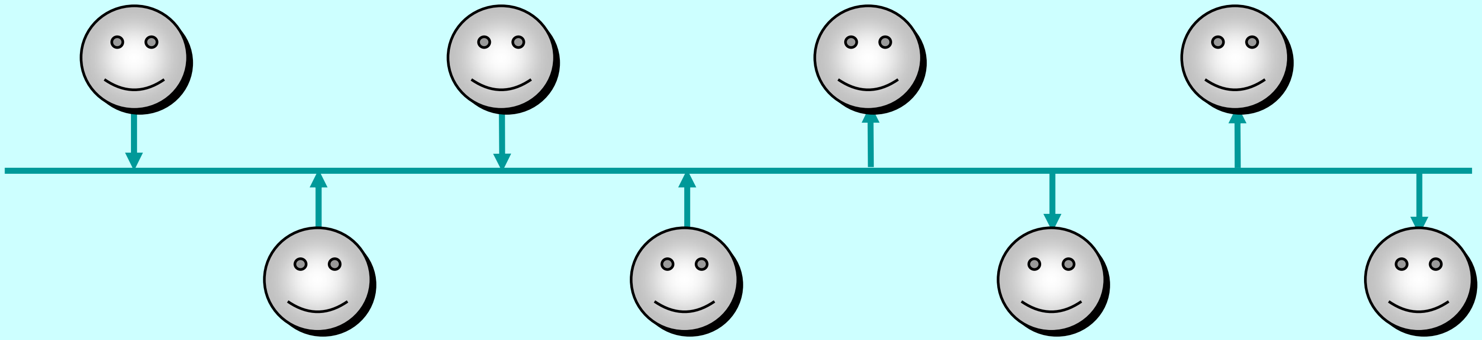
Any2OneChannel



One2AnyChannel



Any2AnyChannel



No ALTing!

Channel Interfaces in *JCSP 1.1*

ChannelOutput
`public void write (Object o)`

ChannelInput
`public Object read ()`

One2OneChannel
`public ChannelOutput out ()` `public AltingChannelInput in ()`

The *abstract class* **`AltingChannelInput`** *extends* the *abstract class* **`Guard`** and *implements* the *interface* **`ChannelInput`**.

Channel Interfaces in *JCSP 1.1*

ChannelOutput
`public void write (Object o)`

ChannelInput
`public Object read ()`

Any2OneChannel
`public SharedChannelOutput out ()` `public AltingChannelInput in ()`

The *interface* **`SharedChannelOutput`** *extends* the *interface* **`ChannelOutput`**. It may be safely shared by internal processes.

Channel Interfaces in *JCSP 1.1*

`ChannelOutput`
`public void write (Object o)`

`ChannelInput`
`public Object read ()`

`One2AnyChannel`
`public ChannelOutput out ()` `public SharedChannelInput in ()`

The *interface* **`SharedChannelInput`** *extends* the *interface* **`ChannelInput`**. It may be safely shared by internal processes.

Channel Interfaces in *JCSP 1.1*

ChannelOutput
`public void write (Object o)`

ChannelInput
`public Object read ()`

Any2AnyChannel
`public SharedChannelOutput out ()` `public SharedChannelInput in ()`

Neither *interface* **`SharedChannelInput`** nor **`SharedChannelOutput`** may be used for *ALTing*.

Channel Interfaces in *JCSP 1.1*

ChannelOutputInt
`public void write (int i)`

ChannelInputInt
`public int read ()`

One2OneChannelInt
`public ChannelOutputInt out ()` `public AltingChannelInputInt in ()`

The *abstract class* **`AltingChannelInputInt`** *extends* the *abstract class* **`Guard`** and *implements* the *interface* **`ChannelInputInt`**.

Channel Interfaces in JCSP 1.1

ChannelOutputInt
`public void write (int i)`

ChannelInputInt
`public int read ()`

Any2OneChannelInt
`public SharedChannelOutputInt out ()` `public AltingChannelInputInt in ()`

The *interface* **`SharedChannelOutputInt`** *extends* the *interface* **`ChannelOutputInt`**. It may be safely shared by internal processes.

Channel Interfaces in *JCSP* 1.1

ChannelOutputInt
`public void write (int i)`

ChannelInputInt
`public int read ()`

One2AnyChannelInt
`public ChannelOutputInt out ()` `public SharedChannelInputInt in ()`

The *interface* **`SharedChannelInputInt`** *extends* the *interface* **`ChannelInputInt`**. It may be safely shared by internal processes.

Channel Interfaces in *JCSP 1.1*

ChannelOutputInt
`public void write (int i)`

ChannelInputInt
`public int read ()`

Any2AnyChannelInt
`public SharedChannelOutputInt out ()` `public SharedChannelInputInt in ()`

Neither *interface* **`SharedChannelInputInt`** nor **`SharedChannelOutputInt`** may be used for *ALTing*.

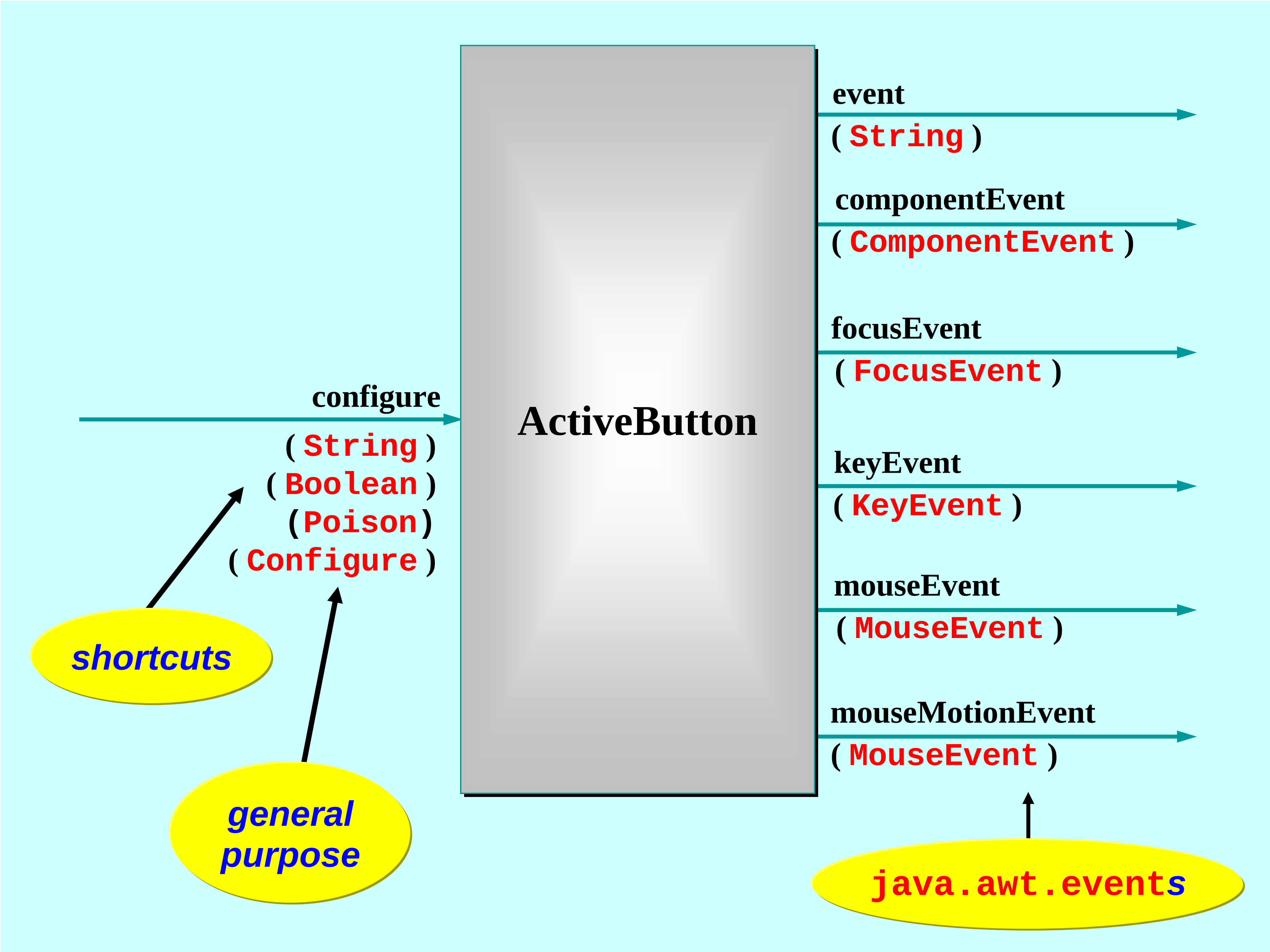
Graphics and GUIs

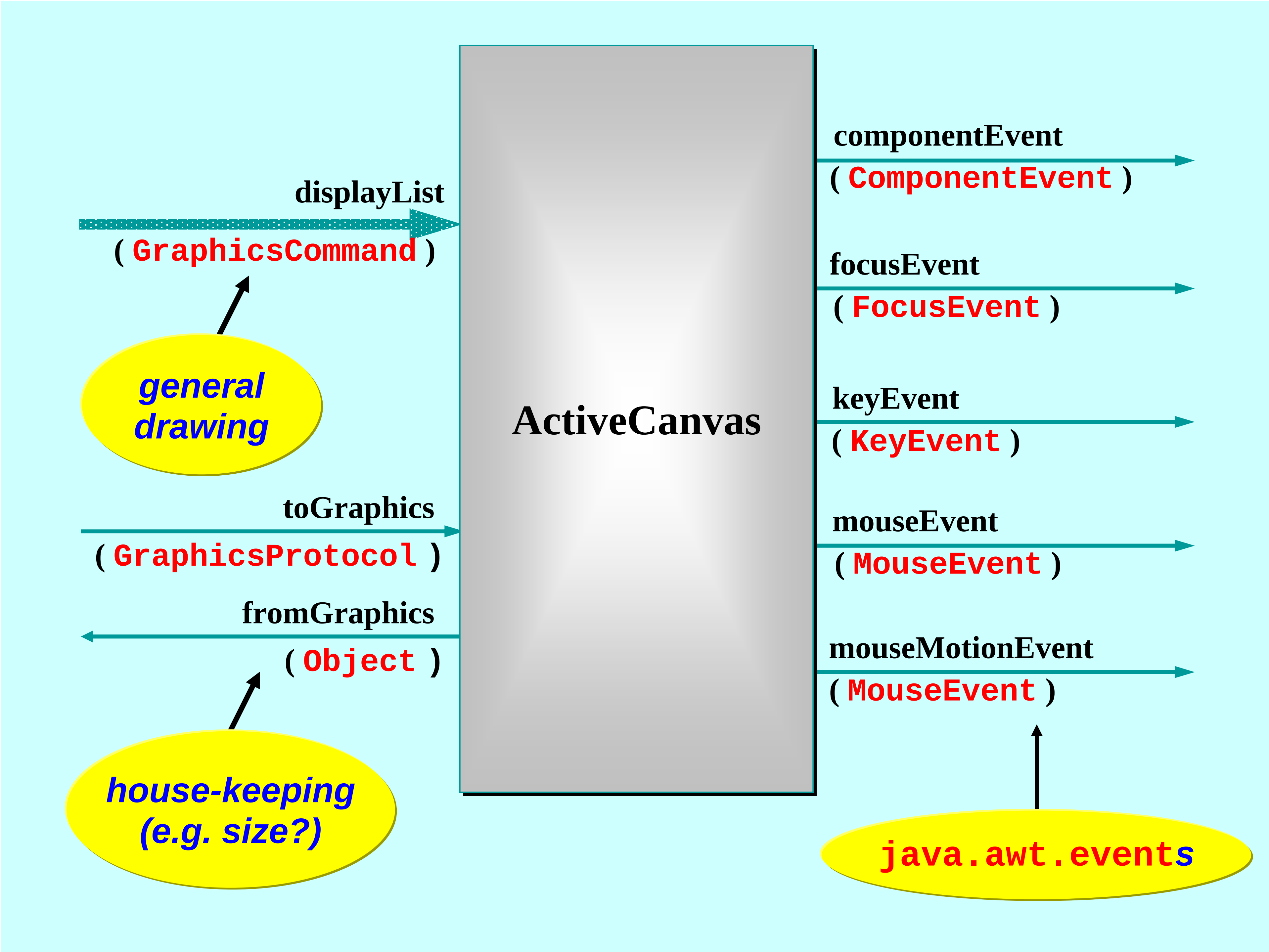
jcsp.awt = java.awt + channels

GUI events → channel communications

Widget configuration → channel communications

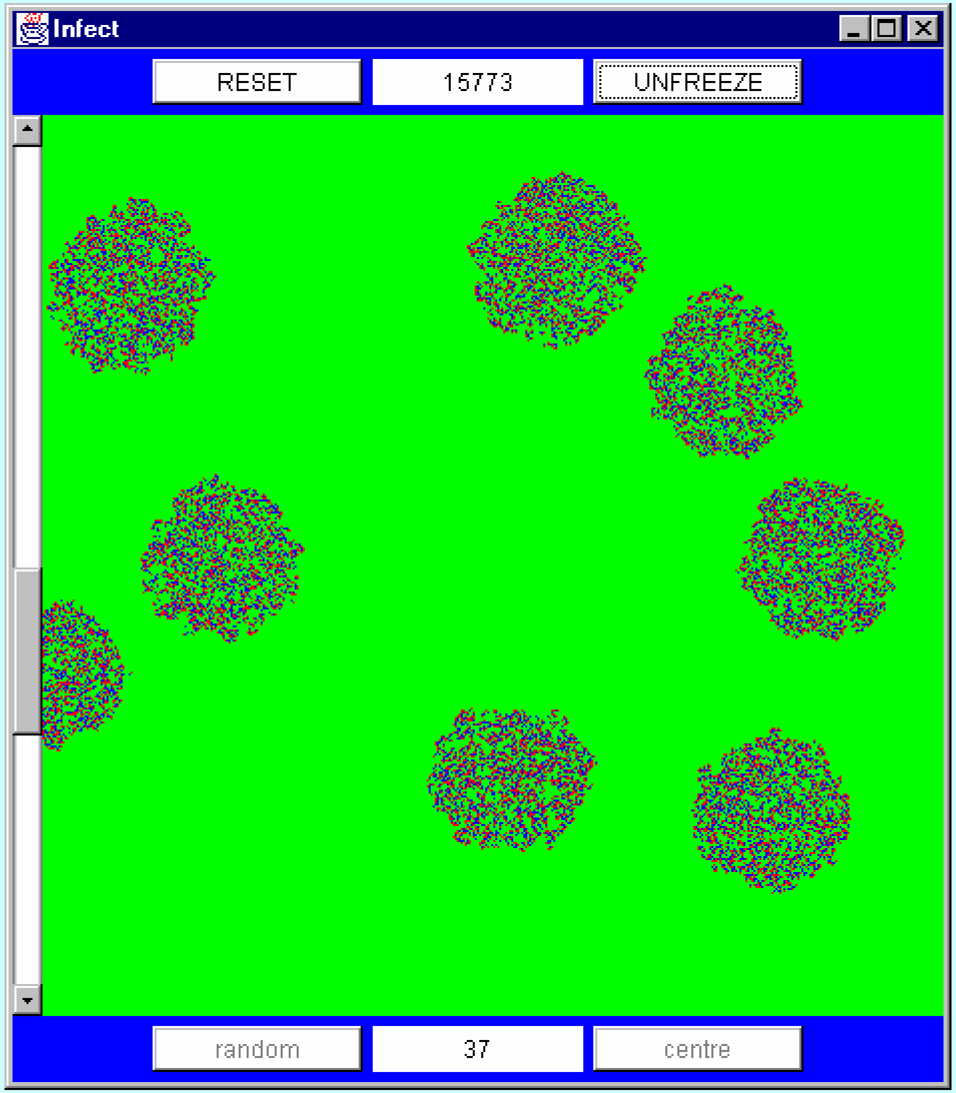
Graphics commands → channel communications


ActiveButton
configure
(String)
(Boolean)
(Poison)
(Configure)
shortcuts
general purpose
event
(String)
componentEvent
(ComponentEvent)
focusEvent
(FocusEvent)
keyEvent
(KeyEvent)
mouseEvent
(MouseEvent)
mouseMotionEvent
(MouseEvent)
java.awt.events

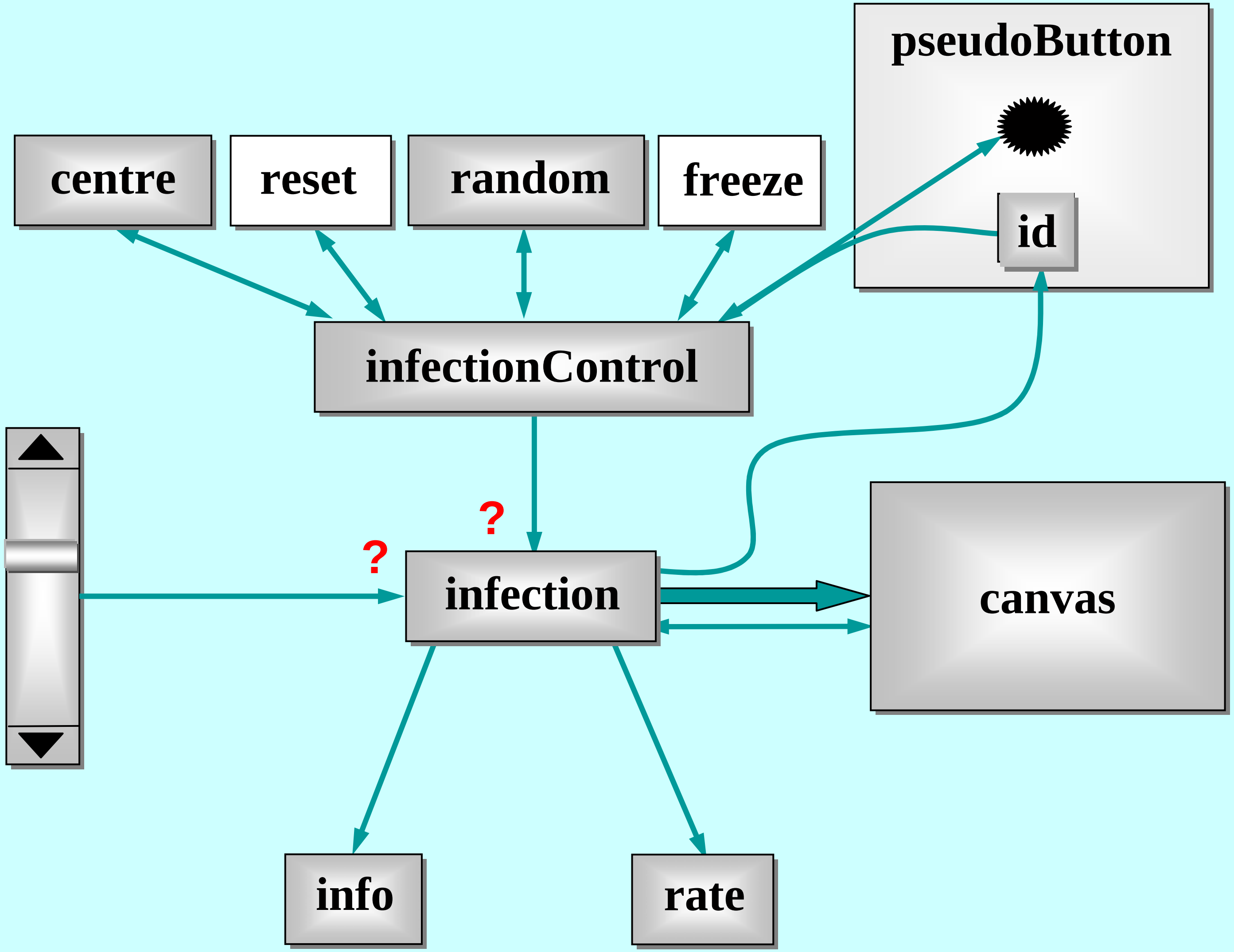


displayList
(GraphicsCommand)
general drawing
toGraphics
(GraphicsProtocol)
fromGraphics
(Object)
house-keeping (e.g. size?)
ActiveCanvas
componentEvent
(ComponentEvent)
focusEvent
(FocusEvent)
keyEvent
(KeyEvent)
mouseEvent
(MouseEvent)
mouseMotionEvent
(MouseEvent)
java.awt.events

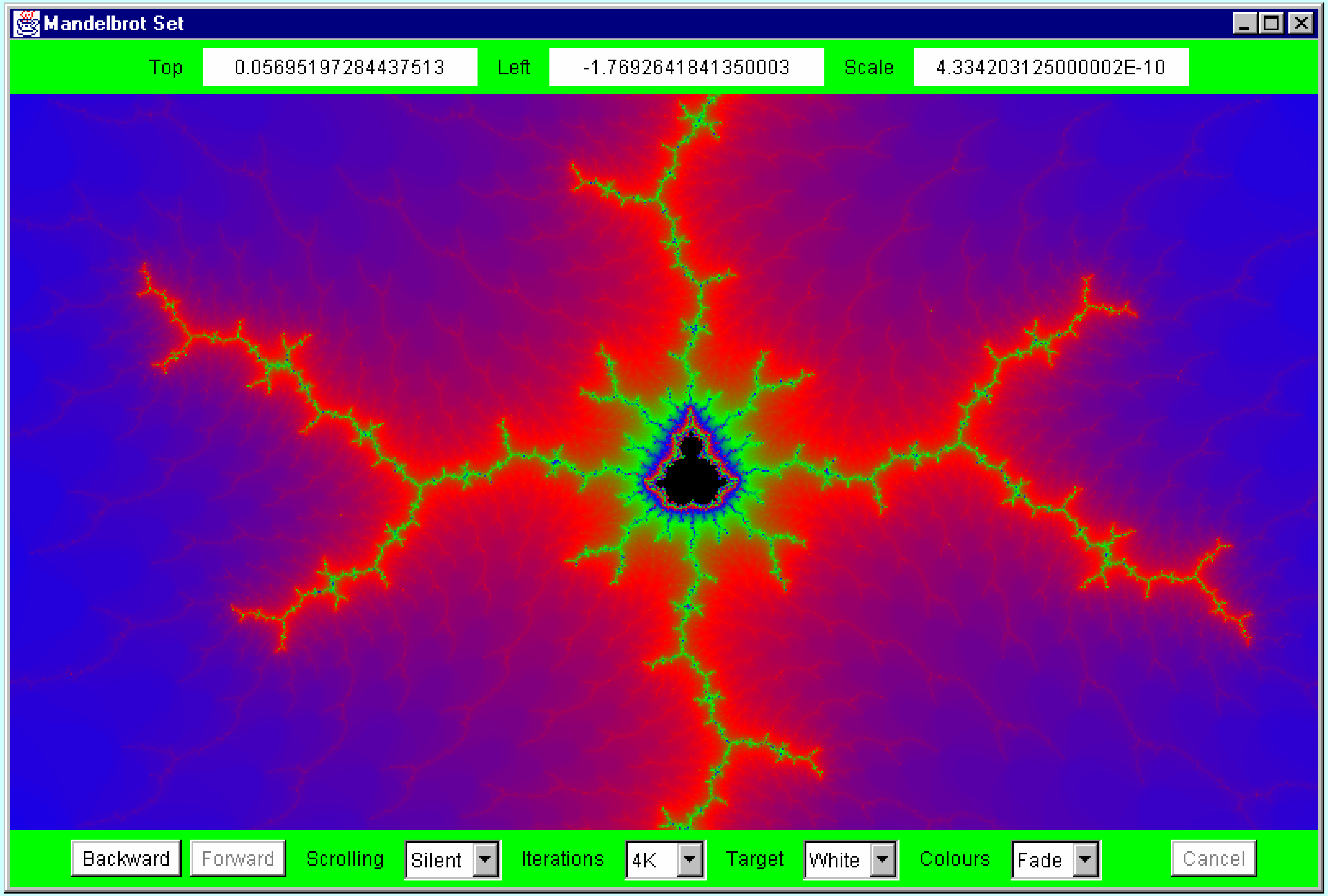
Infection



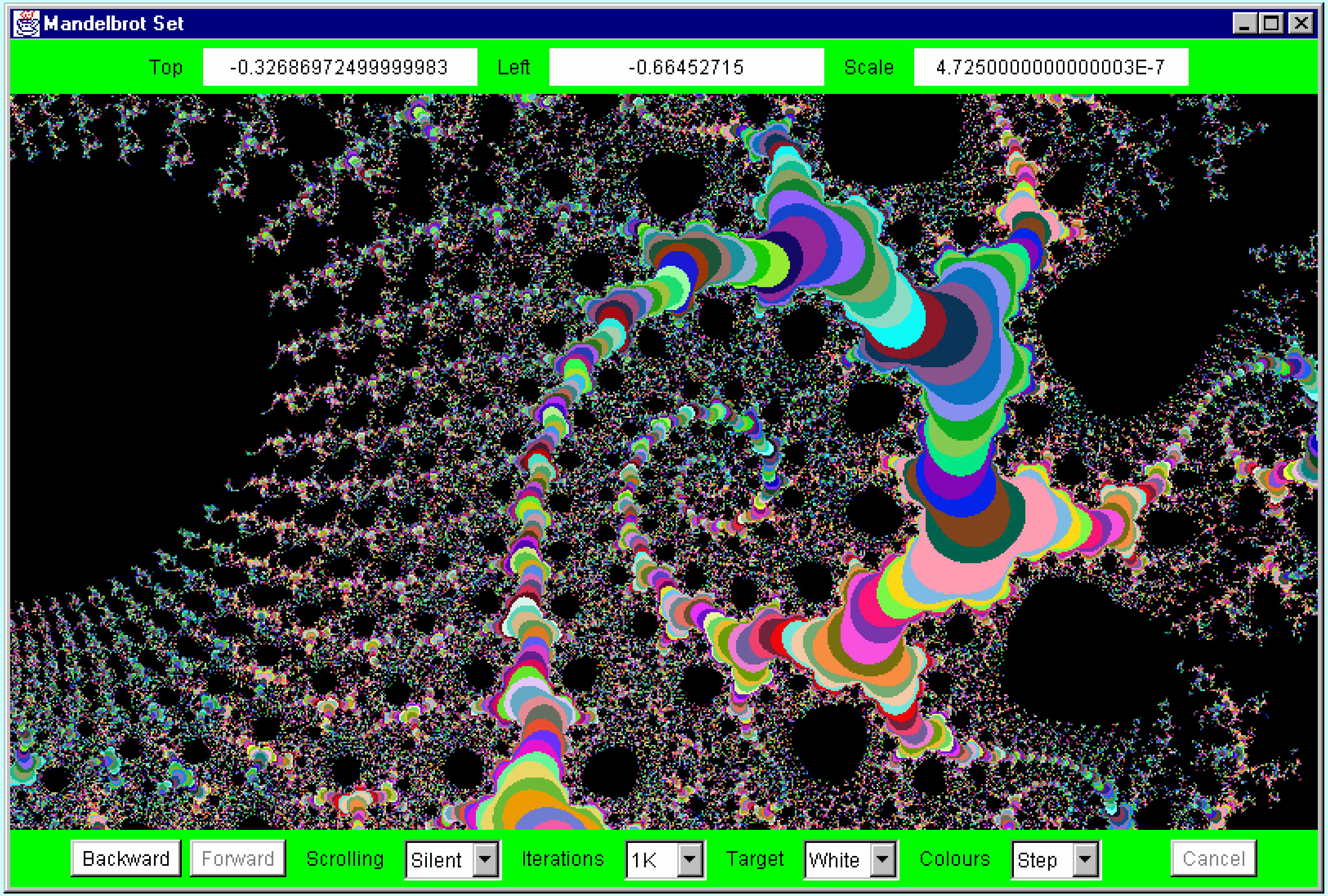
Infection


pseudoButton
centre
reset
random
freeze
id
infectionControl
?
?
infection
canvas
info
rate

Mandelbrot

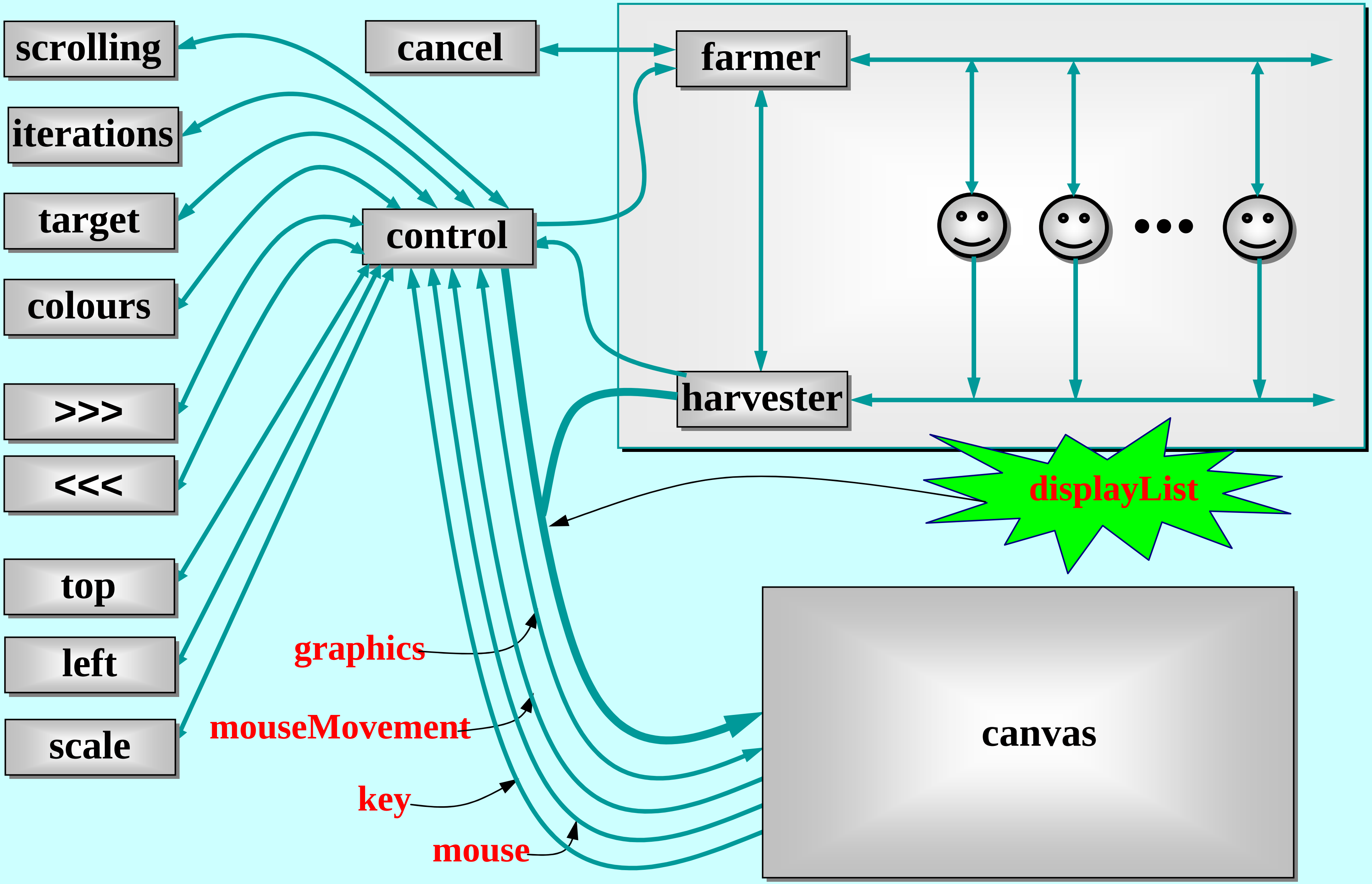

Mandelbrot Set
Top
0.05695197284437513
Left
-1.7692641841350003
Scale
4.334203125000002E-10
Backward
Forward
Scrolling
Silent
Iterations
4K
Target
White
Colours
Fade
Cancel

Mandelbrot



Mandelbrot Set
Top
-0.32686972499999983
Left
-0.66452715
Scale
4.7250000000000003E-7
Backward
Forward
Scrolling
Silent
Iterations
1K
Target
White
Colours
Step
Cancel

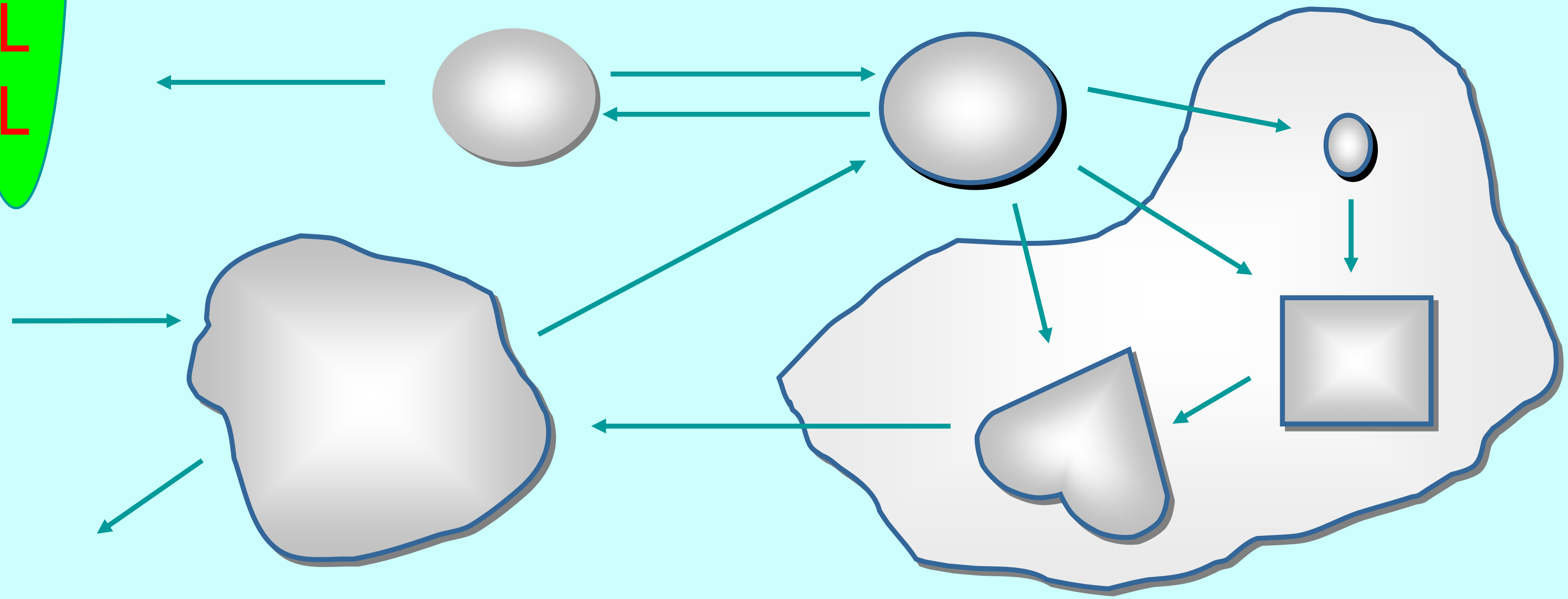
Mandelbrot


scrolling
iterations
target
colours
>>>
<<<
top
left
scale
cancel
control
farmer
harvester
displayList
graphics
mouseMovement
key
mouse
canvas

RECALL

Nature has very large numbers of independent agents, interacting with each other in regular and chaotic patterns, at all levels of scale:

… nuclear … human … astronomic ...



Good News!

R
E
C
A
L
L

The good news is that we can worry about each process on its own. *A process interacts with its environment through its channels*. It does not interact directly with other processes.

Some processes have *serial* implementations - ***these are just like traditional serial programs***.

Some processes have *parallel* implementations - ***networks of sub-processes***.

Our skills for serial logic sit happily alongside our new skills for concurrency - there is no conflict. ***This will scale!***

Other Work

- A **CSP** model for the Java monitor mechanisms (**`synchronized`**, **`wait`**, **`notify`**, **`notifyAll`**) has been built.
- This enables *any* Java threaded system to be analysed in **CSP** terms - e.g. for formal verification of freedom from deadlock/livelock.
- Confidence gained through the formal proof of correctness of the **JCSP** channel implementation:
  - a JCSP channel is a non-trivial monitor - the CSP model for monitors transforms this into an even more complex system of CSP processes and channels;
  - using FDR, that system has been proven to be a refinement of a single CSP channel and *vice versa* - **Q.E.D.**

Other Work

- Higher level synchronisation primitives (e.g. **JCSP** ***CALL channels***, ***barriers***, ***buckets***, …) that capture good patterns of working with low level CSP events.
- Proof rules and design tool support for the above.
- **CSP** *kernels* and their binding into JVMs to support **JCSP**.
- ***Communicating Threads for Java (CTJ)***:
  - this is another Java class library based on CSP principles;
  - developed at the University of Twente (Netherlands) with special emphasis on real-time applications - it's excellent;
  - **CTJ** and **JCSP** share a common heritage and reinforce each other's on-going development - we do talk to each other!

Distributed JCSP.net

- Network channels + plus simple brokerage service for letting JCSP systems find and connect to each other transparently (from anywhere on the *Internet*).
- Virtual channel infrastructure to support this. All application channels auto-multiplexed over *single* (auto-generated) TCP/IP link between any two JVMs.
- Channel Name Server (CNS) provided. Participating JCSP systems just need to know where this is. More sophisticated brokers are easily bootstrapped on top of the CNS (using JCSP).
- ***Killer Application Challenge:***
  - second generation Napster (*no central control or database*) …

Summary

WYSIWYG

Plug-n-Play

- **CSP** has a ***compositional*** semantics.
- **CSP** concurrency can ***simplify*** design:
  - data encapsulation within processes does not break down (unlike the case for objects);
  - channel interfaces impose clean decoupling between processes (unlike method interfaces between objects).
- **JCSP** enables direct Java implementation of **CSP** design.

Summary

- CSP kernel overheads are sub-100-nanosecond (KRoC/CCSP). *Currently,* JCSP depends on the underlying Java threads/monitor implementation.
- ***Rich mathematical foundation:***
  - 20 years mature - recent extensions include simple priority semantics;
  - higher level design rules (e.g. *client-server*, *resource allocation priority*, *IO-par*) with formally proven guarantees (e.g. freedom from deadlock, livelock, process starvation);
  - commercially supported tools (e.g. FDR).
- We don't need to be mathematically sophisticated to take advantage of CSP. It's built-in. Just use it!

Summary

- ***Process Oriented Design*** (processes, syncs, alts, parallel, layered networks).
- ***WYSIWYG:***
  - each process considered individually (own data, own control threads, external synchronisation);
  - leaf processes in network hierarchy are ordinary *serial* programs - all our past skills and intuition still apply;
  - *concurrency* skills sit happily alongside the old serial ones.
- Race hazards, deadlock, livelock, starvation problems: we have a rich set of design patterns, theory, intuition and tools to apply.

Conclusions

- We are ***not*** saying that Java's threading mechanisms need changing.
- Java is sufficiently flexible to allow ***many*** concurrency paradigms to be captured.
- **JCSP** is just a ***library*** - Java needs no language change to support **CSP**.
- **CSP** rates serious consideration as a basis for any real-time specialisation of Java:
  - ***quality*** (robustness, ease of use, scalability, management of complexity, formalism);
  - ***lightness*** (overheads do not invalidate the above benefits - they encourage them).

Acknowledgements

- Paul Austin - the original developer of JCSP (**p_d_austin@hotmail.com**).
- Andy Bakkers and Gerald Hilderink - the CTJ library (**bks@el.utwente.nl**, **G.H.Hilderink@el.utwente.nl**).
- Jeremy Martin - for the formal proof of correctness of the JCSP channel (**Jeremy.Martin@comlab.ox.ac.uk**)
- Nan Schaller (**ncs@cs.rit.edu**), Chris Nevison (**chris@cs.colgate.edu**) and Dyke Stiles (**dyke.stiles@ece.usu.edu**) - for pioneering the teaching.
- The **WoTUG** community - its workshops, conferences and people.

URLs

CSP www.comlab.ox.ac.uk/archive/csp.html

JCSP www.cs.ukc.ac.uk/projects/ofa/jcsp/

CTJ www.rt.el.utwente.nl/javapp/

KRoC www.cs.ukc.ac.uk/projects/ofa/kroc/

java-threads@ukc.ac.uk

www.cs.ukc.ac.uk/projects/ofa/java-threads/

WoTUG

wotug.ukc.ac.uk/

Stop Press

JCSP Networking Edition

JCSP.net

www.quickstone.com