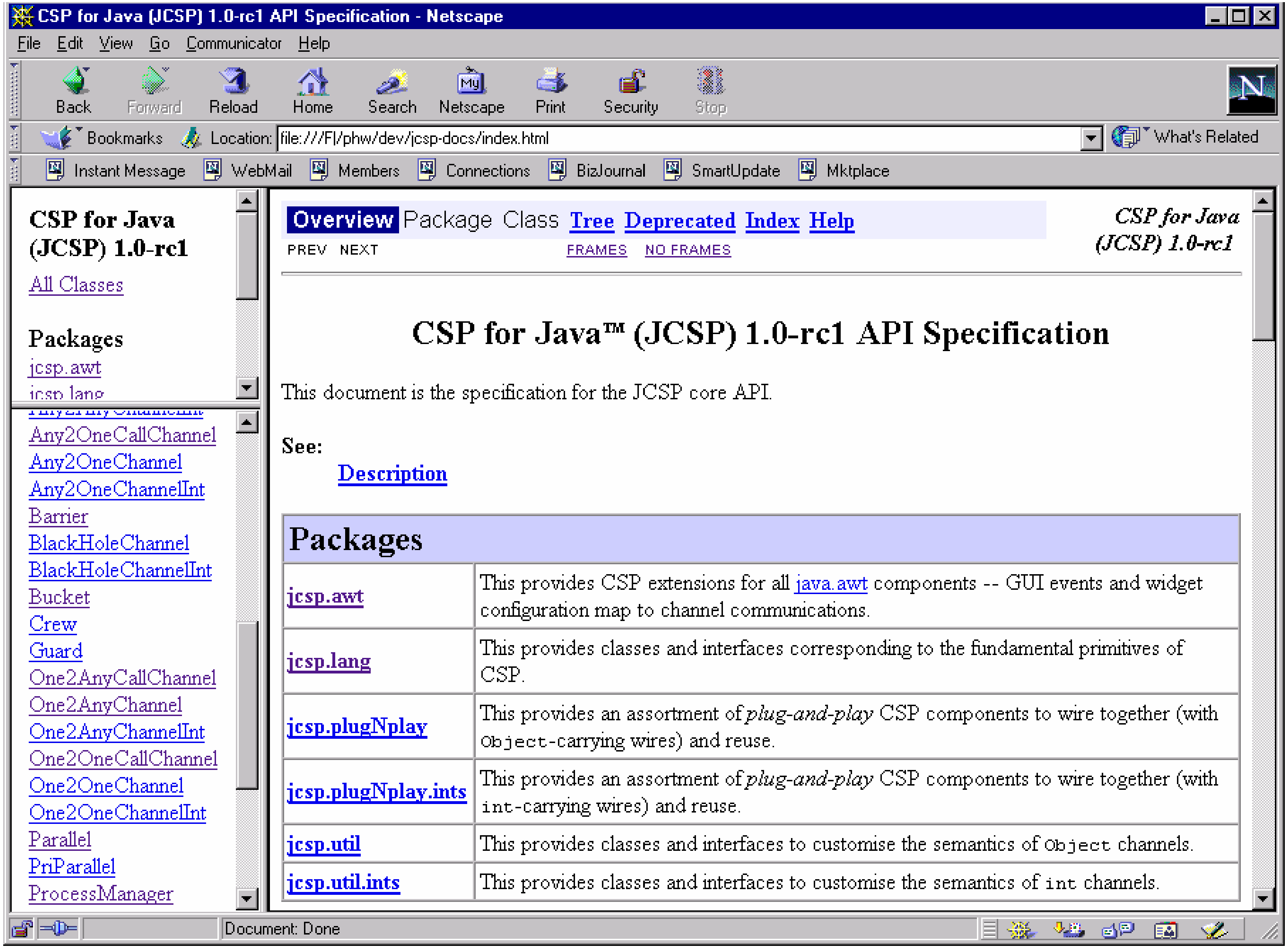
Putting CSP into practice …

JCSP

http://www.cs.ukc.ac.uk/projects/ofa/jcsp/


CSP for Java (JCSP) 1.0-rc1 API Specification - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: file:///F|/phw/dev/jcsp-docs/index.html
What's Related
Instant Message WebMail Members Connections BizJournal SmartUpdate Mktplace
CSP for Java (JCSP) 1.0-rc1
All Classes
Packages
jcsp.awt
jcsp.lang
Any2OneCallChannel
Any2OneChannel
Any2OneChannelInt
Barrier
BlackHoleChannel
BlackHoleChannelInt
Bucket
Crew
Guard
One2AnyCallChannel
One2AnyChannel
One2AnyChannelInt
One2OneCallChannel
One2OneChannel
One2OneChannelInt
Parallel
PriParallel
ProcessManager
Overview Package Class Tree Deprecated Index Help
PREV NEXT
FRAMES NO FRAMES
CSP for Java (JCSP) 1.0-rc1
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.
Document: Done

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 Sets of Channel Classes (and Interfaces)

`Object` channels

- carrying (references to) arbitrary Java objects

`int` channels

- carrying Java **`int`**s

Channel Interfaces and Classes

- Channel 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 will be the network builder's concern to choose the actual channel **classes** to use when connecting processes together.

`int` Channels

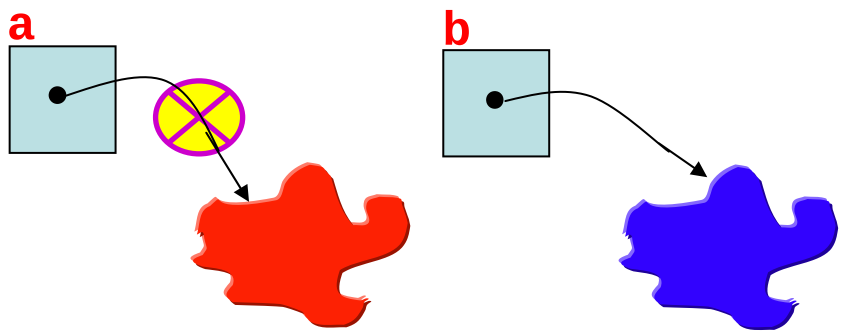
- The int channels are convenient and secure.

- 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 !!

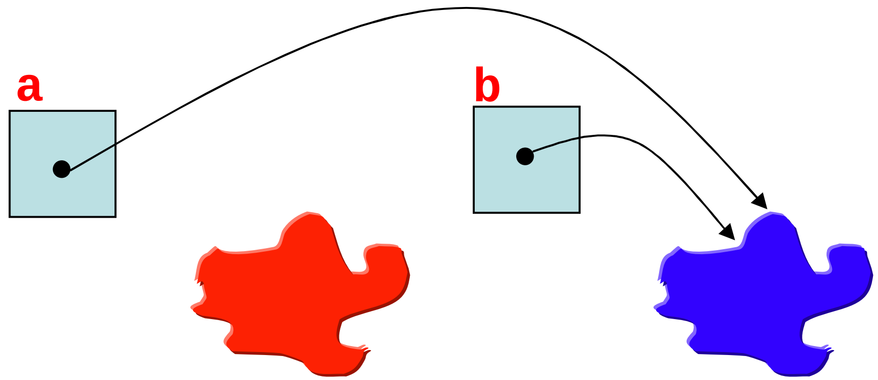
Java objects are referenced through variable names.

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



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

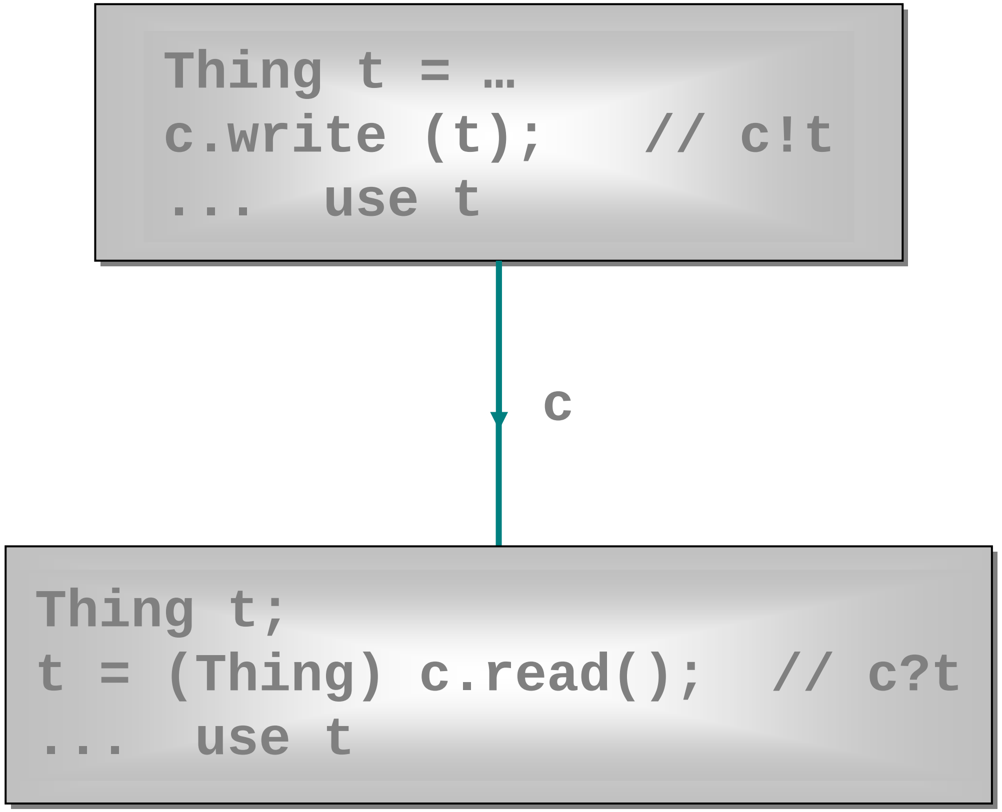
`a = b;`



`Object` Channels - Danger !!

- **`Object`** channels expose a danger

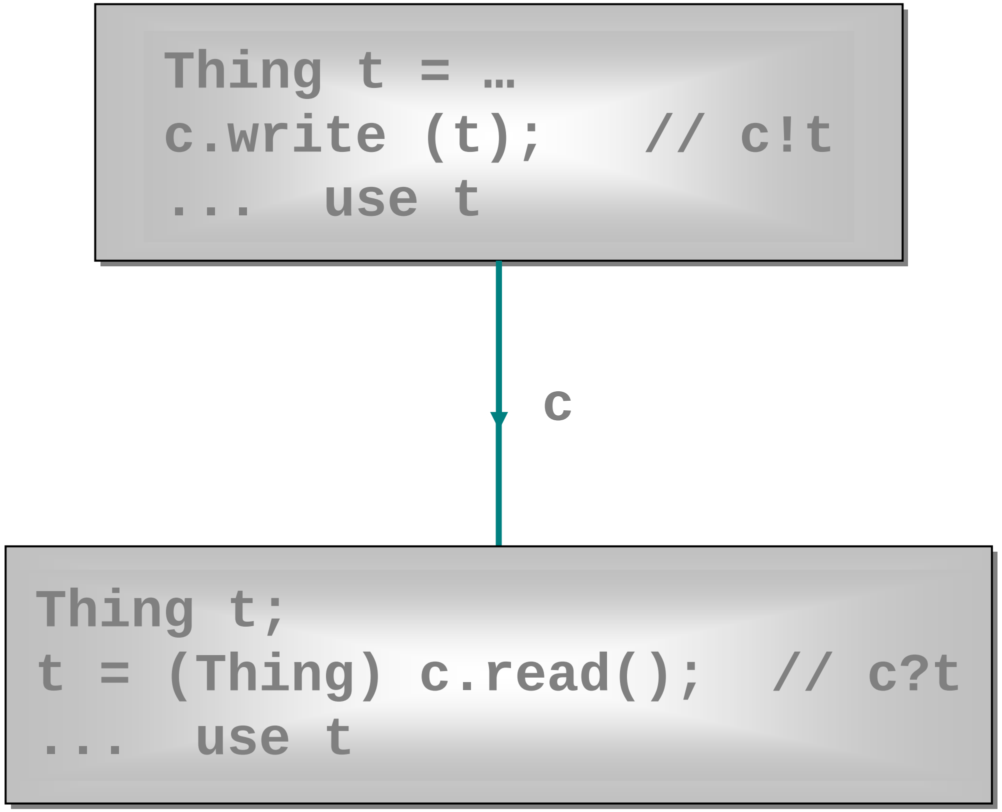
- Channel communication only communicates the **`Object`** reference.



`Object` Channels - Danger !!

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

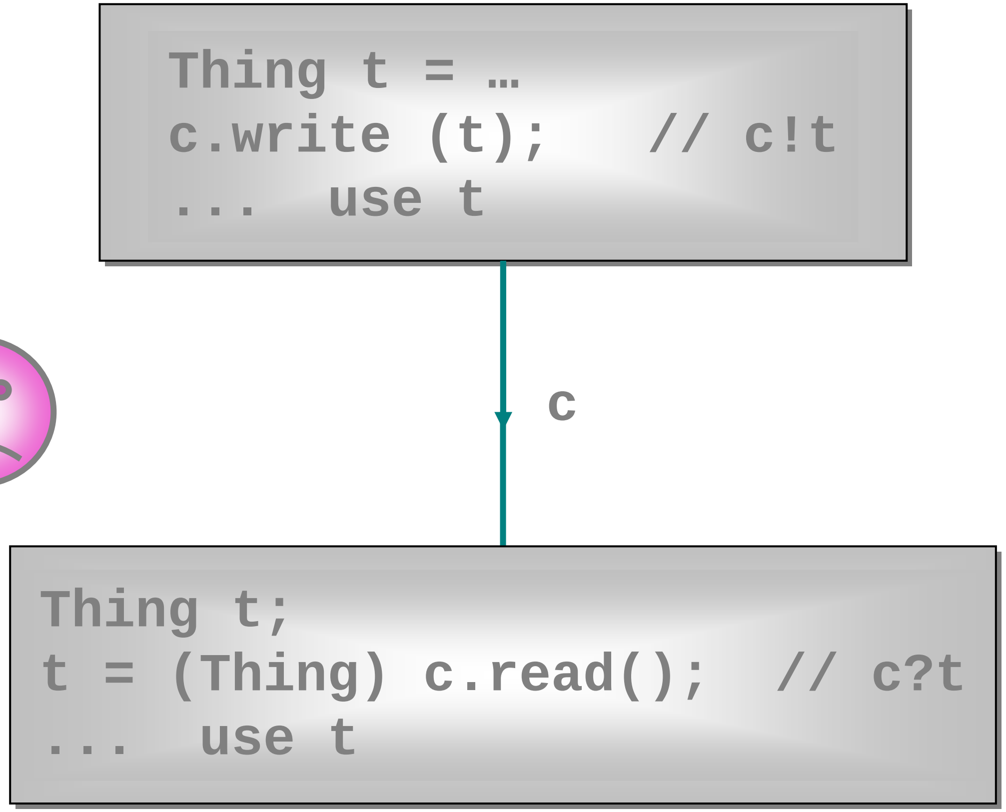
- If ***one*** of these processes modifies that object (in its … use `t`), the ***other*** one had better forget about it!



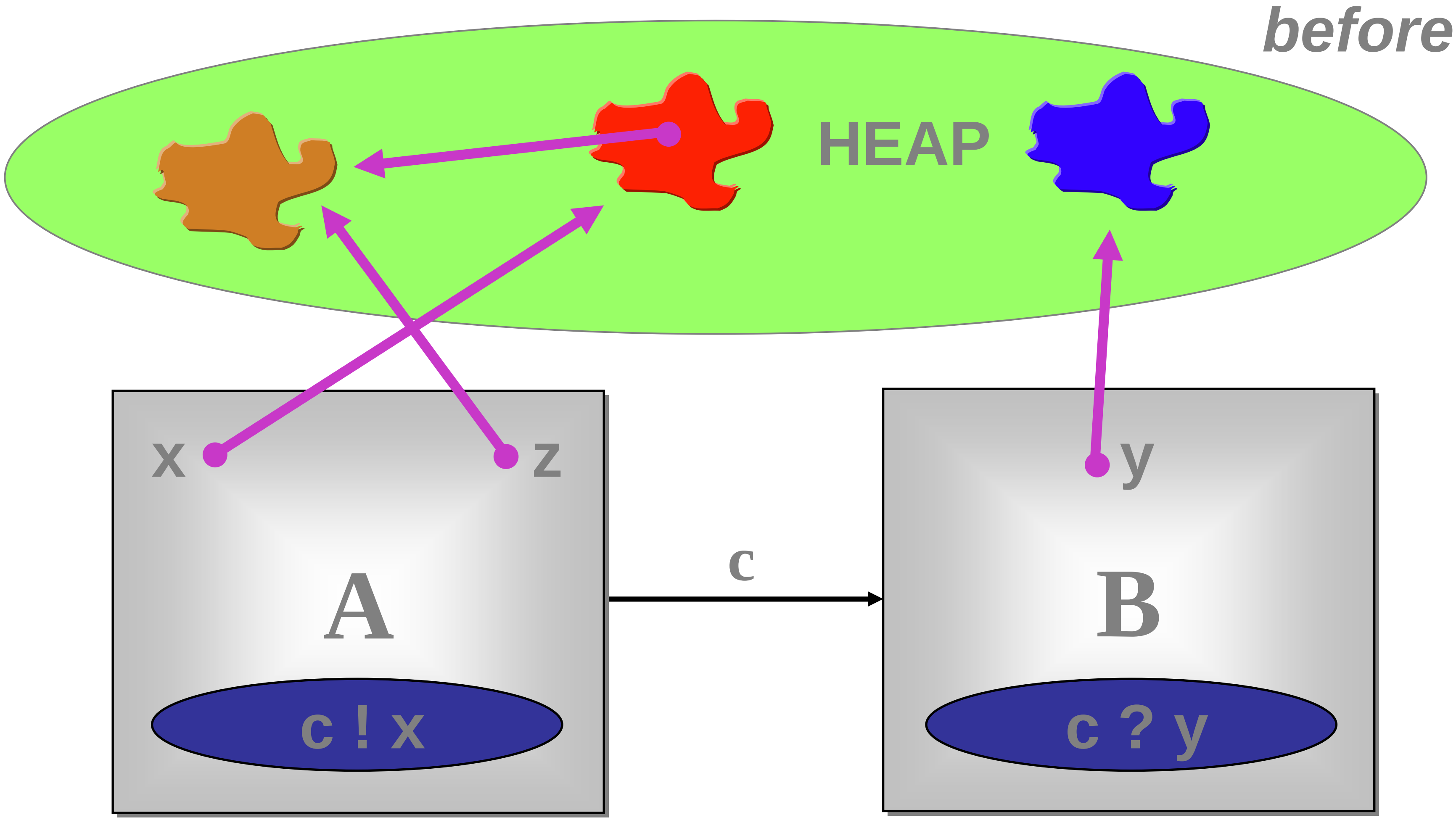
`Object` Channels - Danger !!

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

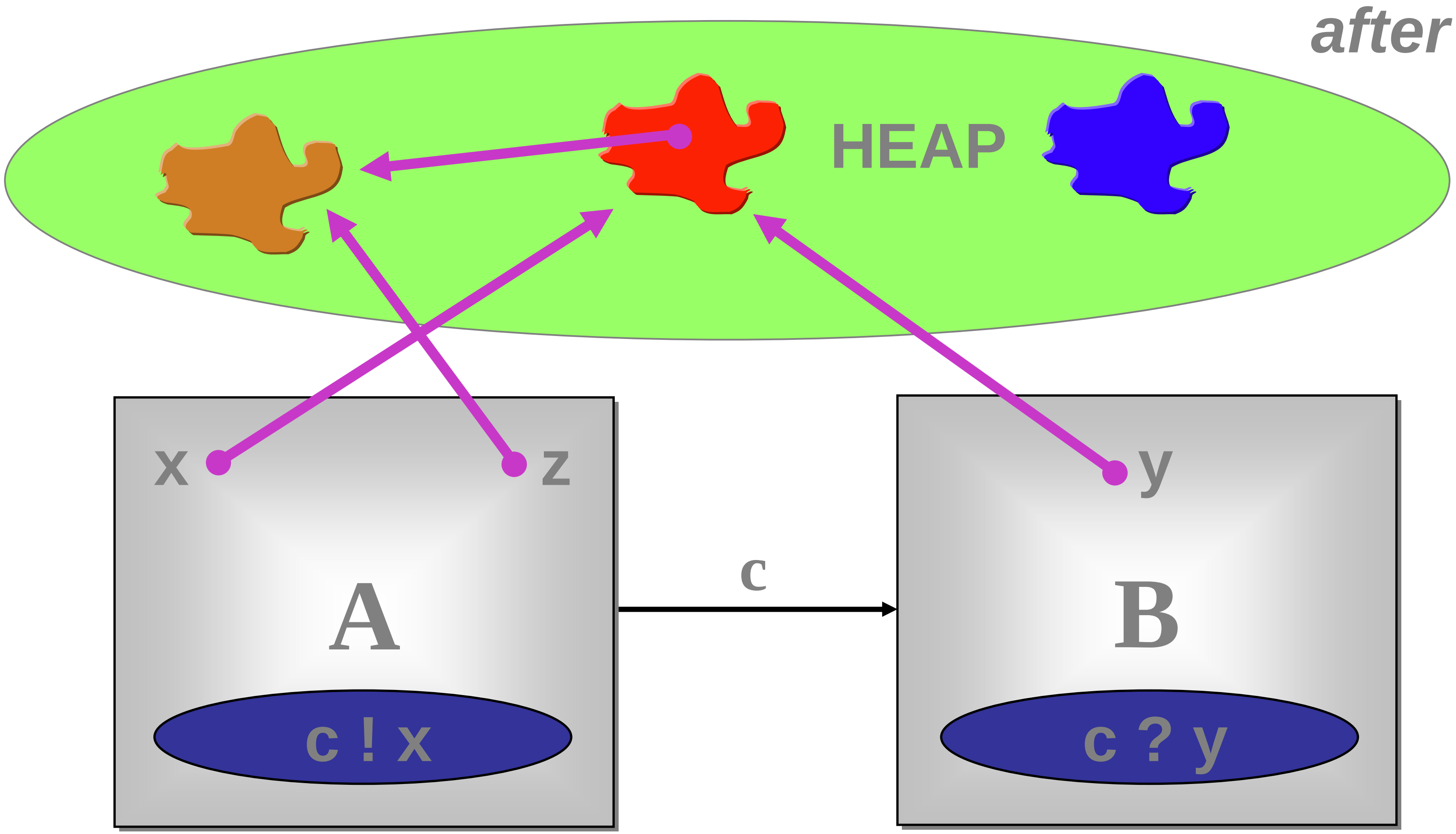
- We have design patterns to prevent this.



Reference Semantics

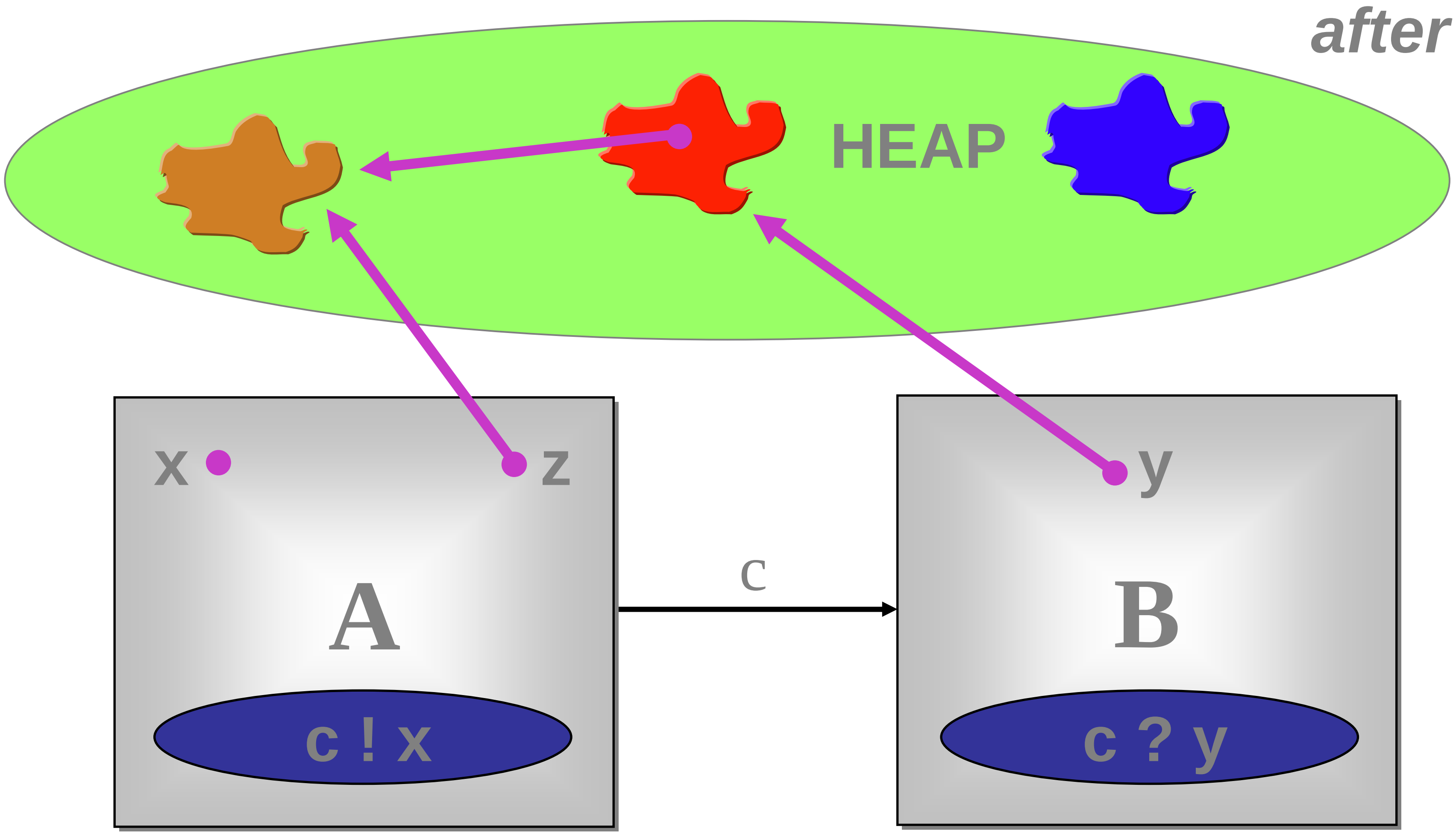


Reference Semantics



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 o);
}

interface ChannelInputInt {
  public int read ();
}

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

Channel Interfaces

- These 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 choose the actual channel **classes** to use when connecting processes together.

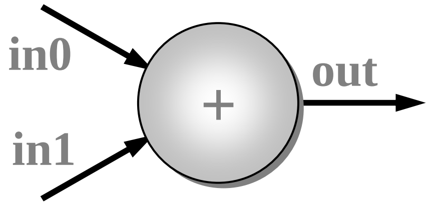
- Let's review some of the *Legoland* processes - this time in **JCSP**.

JCSP Process Structure

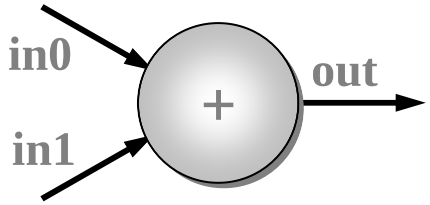
```
class Example implements CSProcess {

  ...  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);
    }
  }

}
```



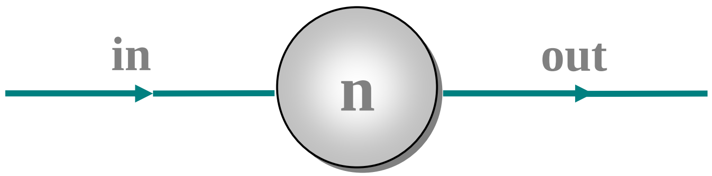

```
class PlusInt implements CSProcess {

  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!



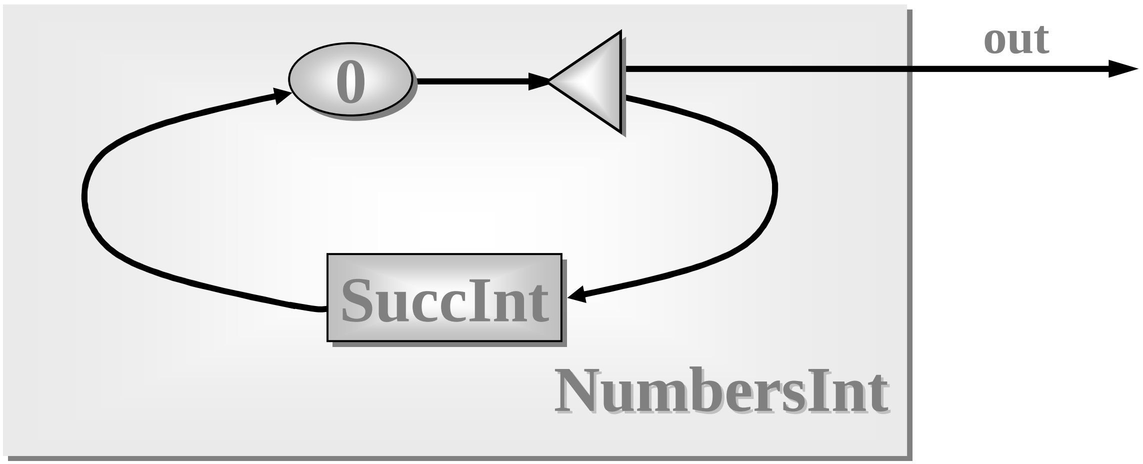

```
class PrefixInt implements CSProcess {

  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

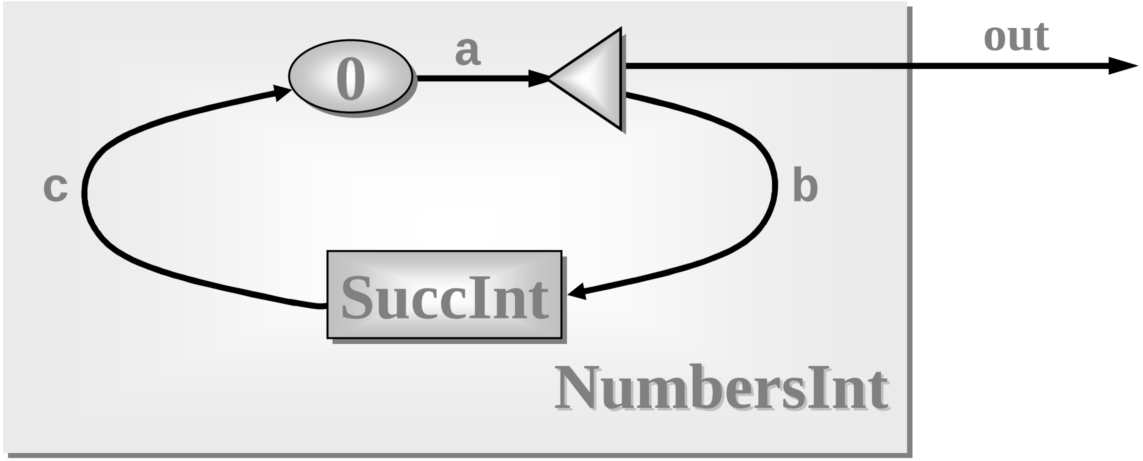
- We now want to be able to take instances of these ***processes*** (or components) and connect them together to form a network.
- The resulting network will itself be a ***process***.
- To do this, we need to construct some real wires - these are instances of the ***channel*** classes.
- We also need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

Parallel

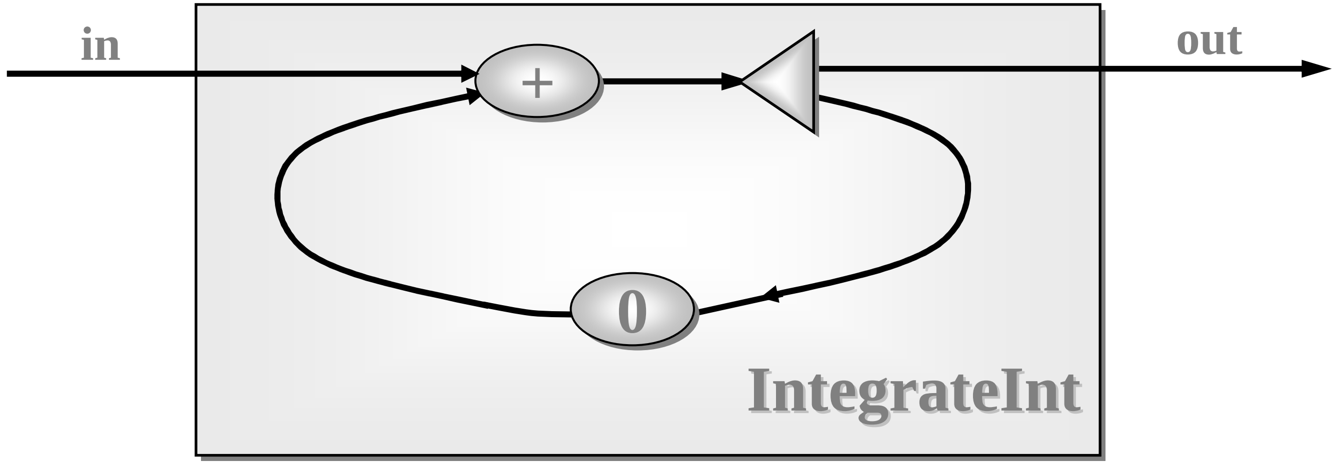
- ***Parallel*** is a **`CSProcess`** whose constructor takes an array of **`CSProcess`**es.
- Its ***run()*** method is the parallel composition of its given **`CSProcess`**es.
- The semantics is the same as for the CSP **||**.
- The ***run()*** terminates when and only when all of its component processes have terminated.



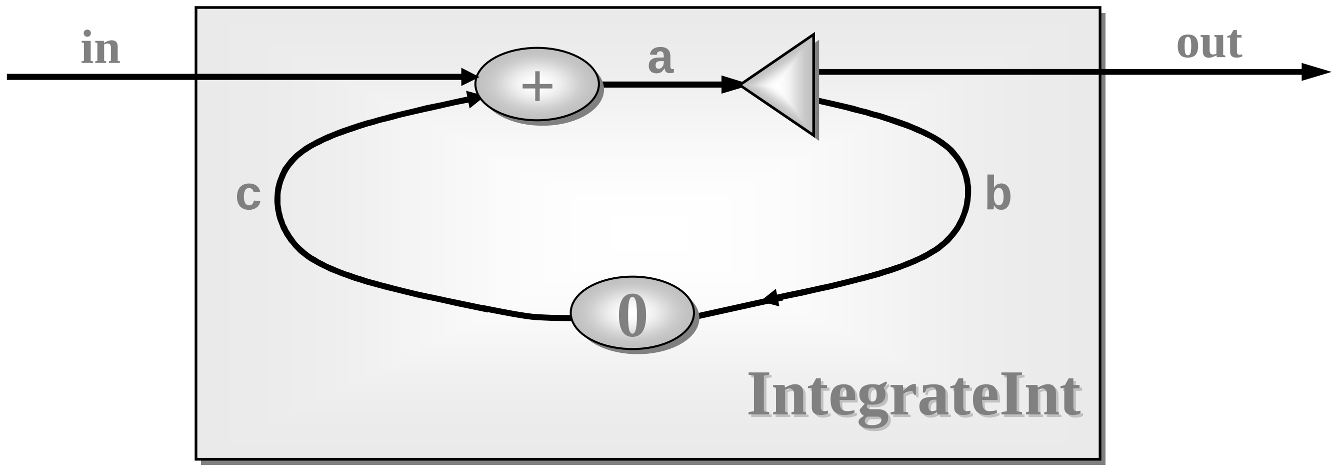

```
class NumbersInt implements CSProcess {

  private final ChannelOutputInt out;

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

  ...  public void run ()

}
```



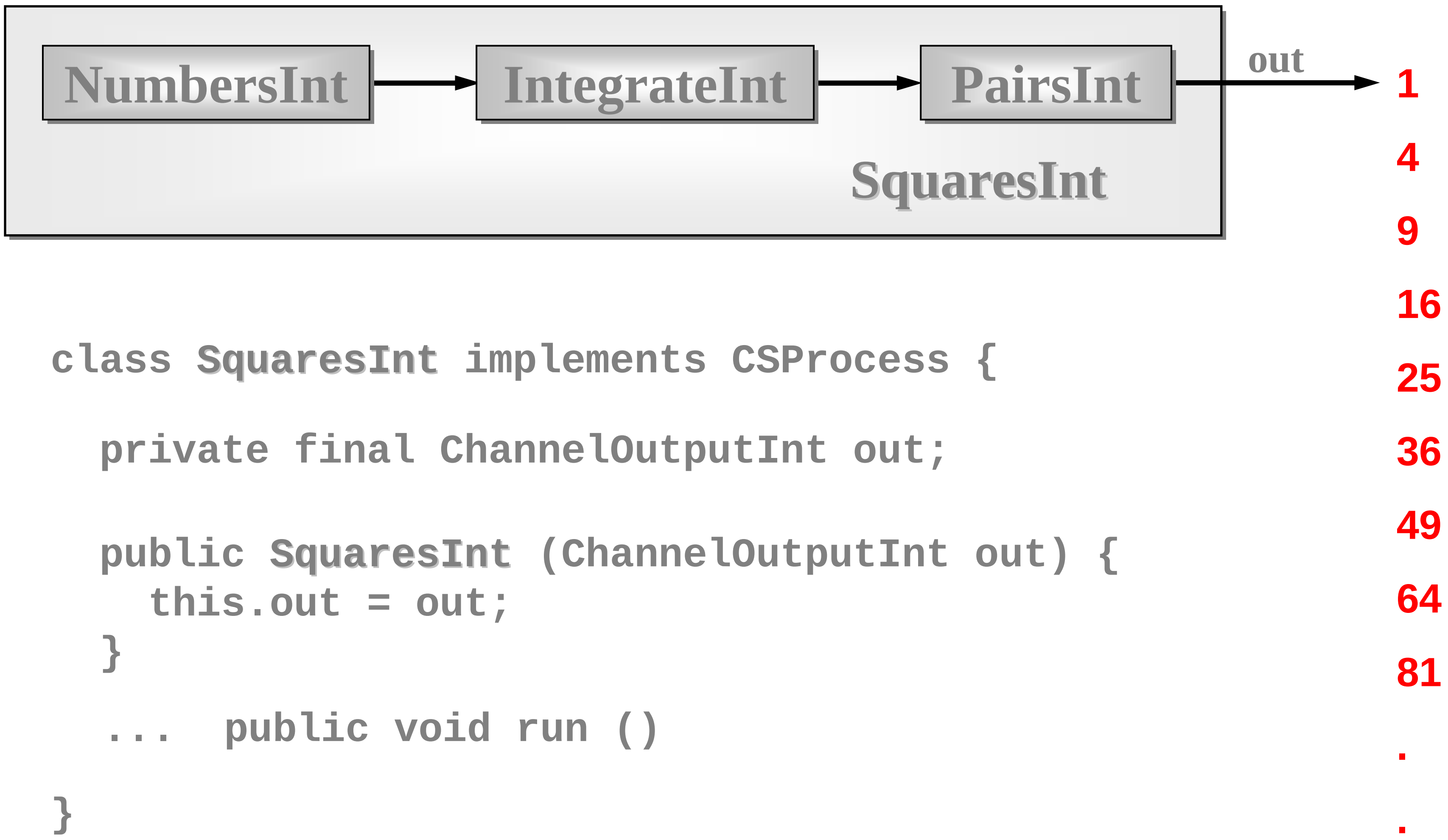

```
public void run () {

  One2OneChannelInt a = new One2OneChannelInt ();
  One2OneChannelInt b = new One2OneChannelInt ();
  One2OneChannelInt c = new One2OneChannelInt ();

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

}
```



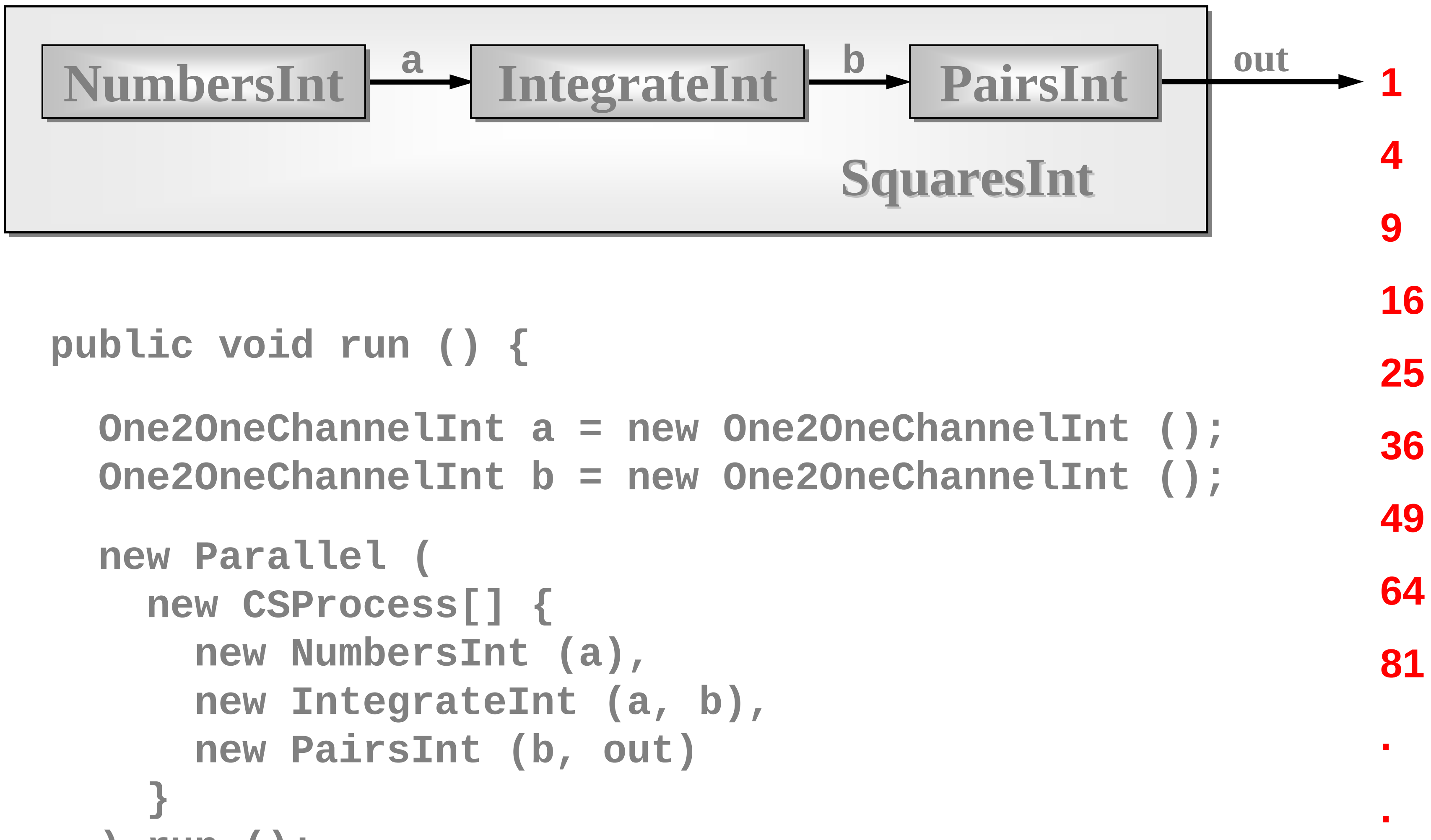

```
class IntegrateInt implements CSProcess {

  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 = new One2OneChannelInt ();
  One2OneChannelInt b = new One2OneChannelInt ();
  One2OneChannelInt c = new One2OneChannelInt ();

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

}
```



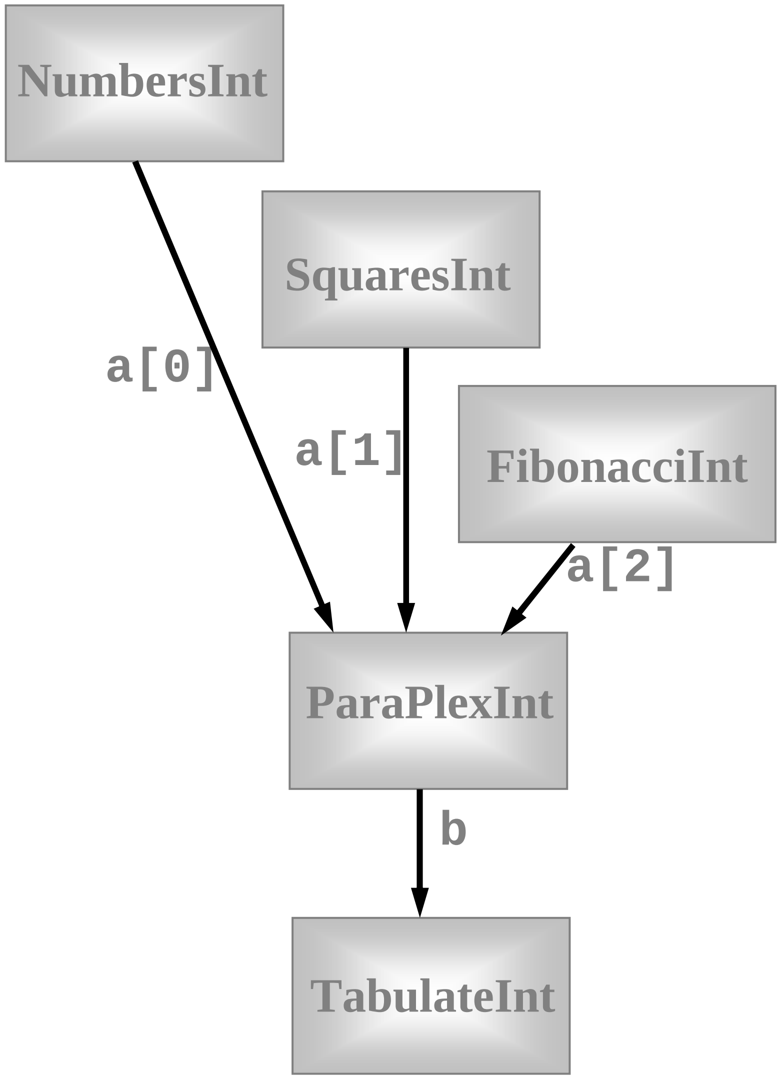

```
class SquaresInt implements CSProcess {

  private final ChannelOutputInt out;

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

  ...  public void run ()

}
```

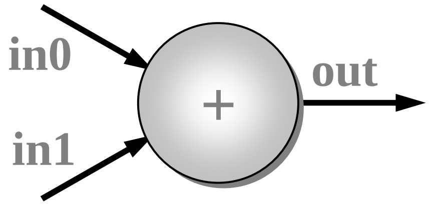



```
public void run () {

  One2OneChannelInt a = new One2OneChannelInt ();
  One2OneChannelInt b = new One2OneChannelInt ();

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

}
```

Quite a Lot of Processes

```
One2OneChannelInt[] a =
 One2OneChannelInt.create (3);
One2OneChannel b =
 new One2OneChannel ();

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

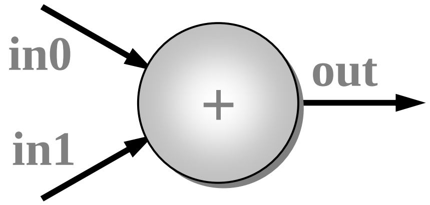





```
class PlusInt implements CSProcess {

  ...  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);
    }
  }

}
```

Change this!

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



This process does one input and terminates.

```
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

- 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.

Deterministic Processes

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

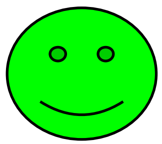
In the real world, it is sometimes the case that things happen as a result of:

- 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)
}
```

Five **JCSP** classes are (**`extend`**) **`Guard`**s:

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

Only the *1-1* and *any-1* channels extend the above (i.e. are *ALTable*).

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

Ready/Unready Guards

- A **channel** guard is ready **if** *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 **if** *its timeout has expired*.
- A **skip** guard is *always ready*.

Alternation

For ***ALT**ing*, 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***.

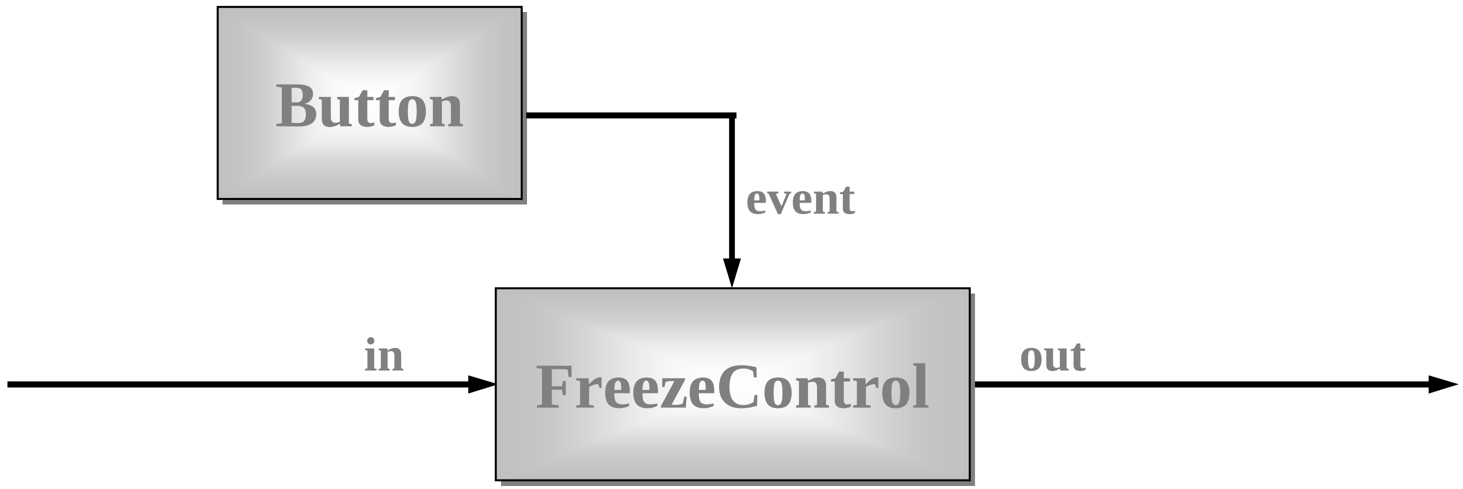
`alt.fairSelect()`

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.

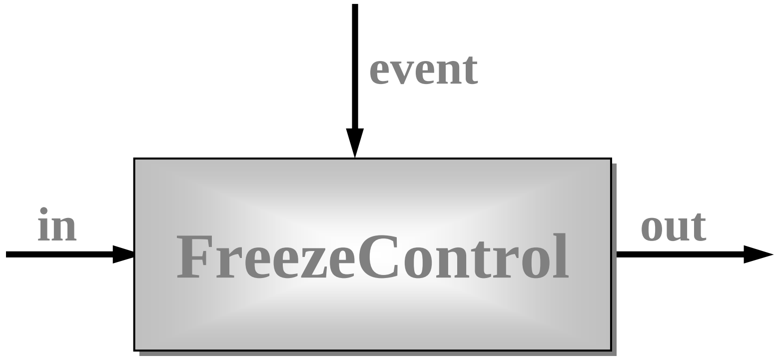
ALTing Between Events



- **`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

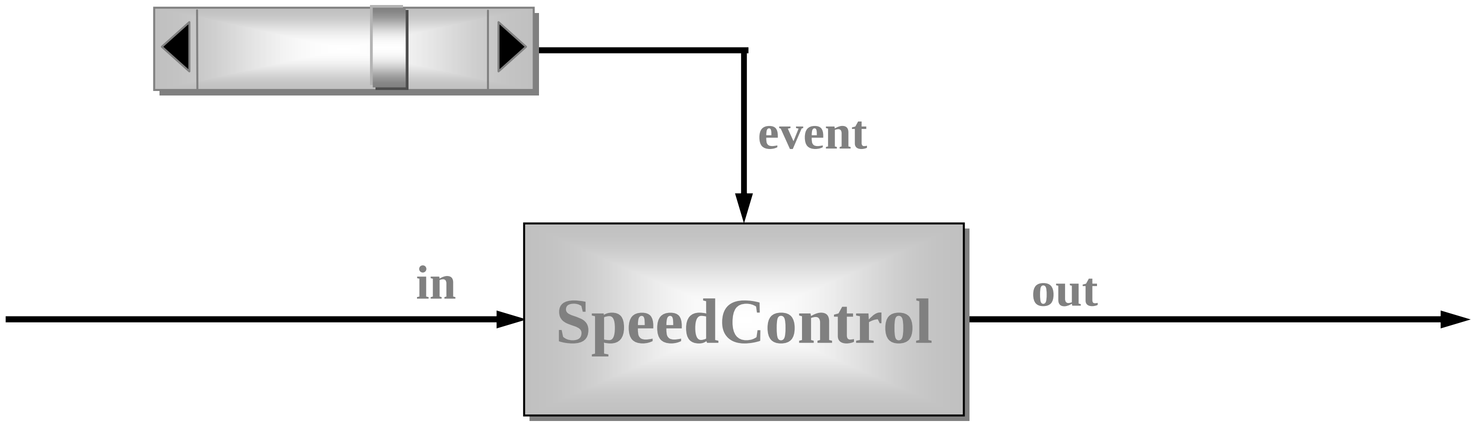



```
final Alternative alt =
  new Alternative (
    new Guard[] {event, in};
  );
final int EVENT = 0, IN = 1;
```

```
while (true) {
  switch (alt.priSelect ()) {
    case EVENT:
      event.read ();
      event.read ();
    break;
    case IN:
      out.write (in.read ());
    break;
  }
}
```

No *SPIN*

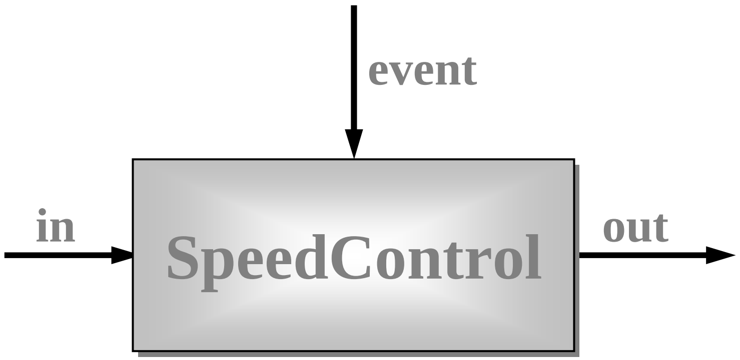
ALTing Between Events



- 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

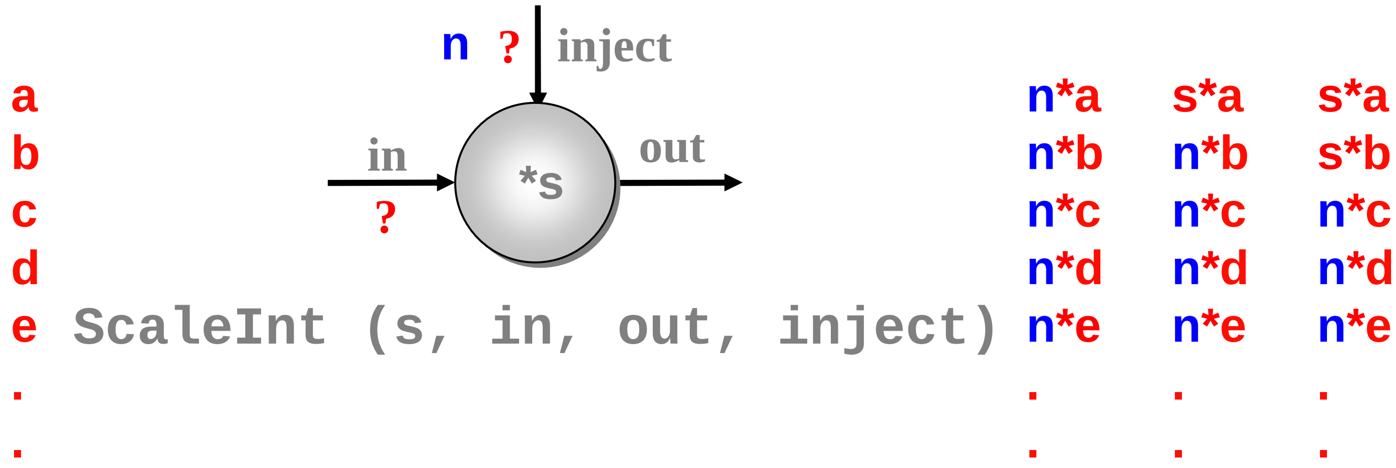



```
final CSTimer tim =
  new CSTimer ();
final Alternative alt =
  new Alternative (
    new Guard[] {event, tim};
  );
final int EVENT = 0, TIM = 1;
```

```
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;
  }
}
```

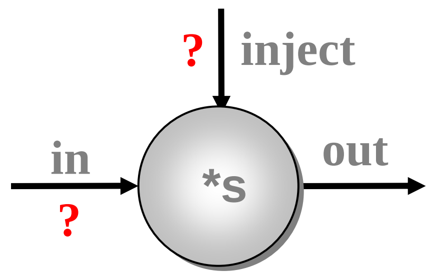
No *SPIN*

Another Control Process

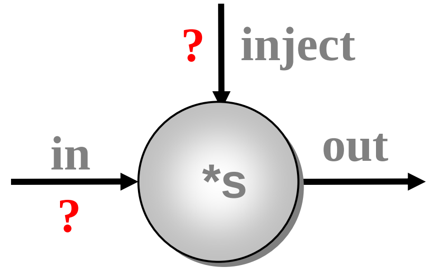



```
ScaleInt (s, in, out, inject) =
  (inject?s --> SKIP
   [PRI]
   in?a --> out!s*a --> SKIP
  );
 ScaleInt (s, in, out, inject)
```

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



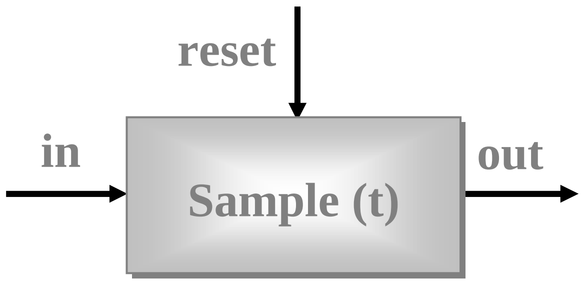

```
class ScaleInt implements CSProcess {

  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 ()

}
```



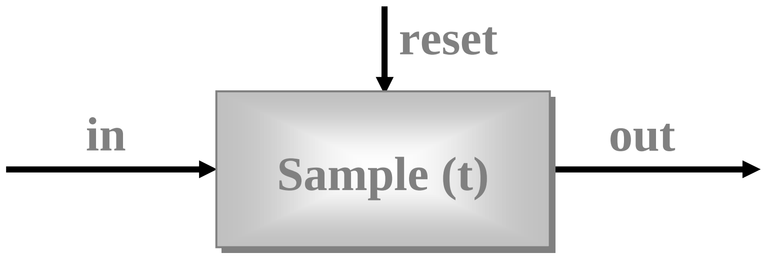

```
public void run () {

  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.

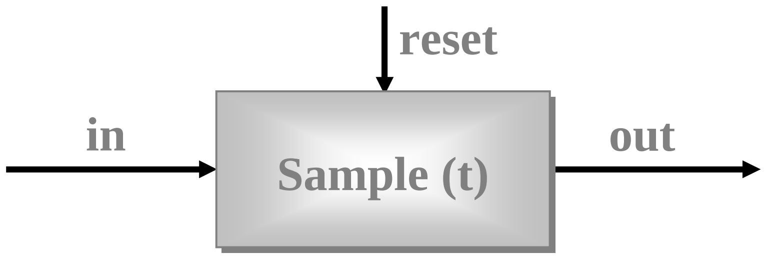
Real-Time Sampler

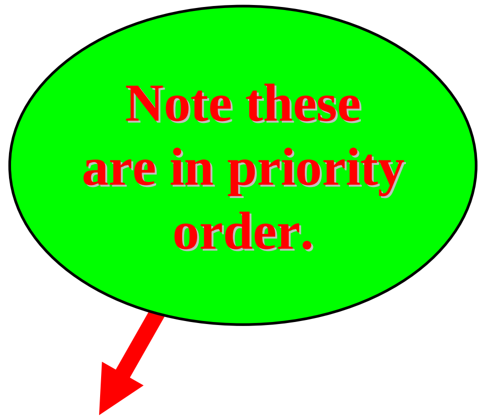


- This process services any of 3 events *(2 inputs and 1 timeout)* that may occur.
- Its **`t`** parameter represents a time interval. Every **`t`** time units, it must output the *last* object that arrived on its **`in`** channel during the previous time slice. If nothing arrived, it must output a **`null`**.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

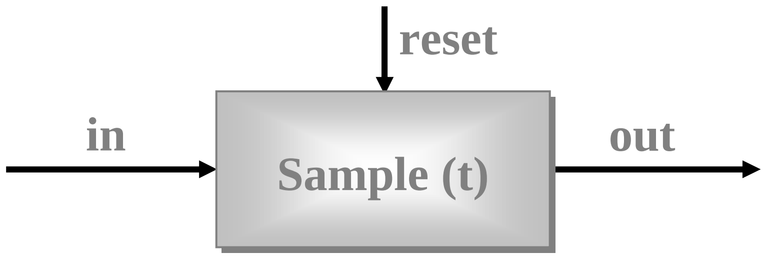



```
class Sample implements CSProcess {

  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 ()

}
```

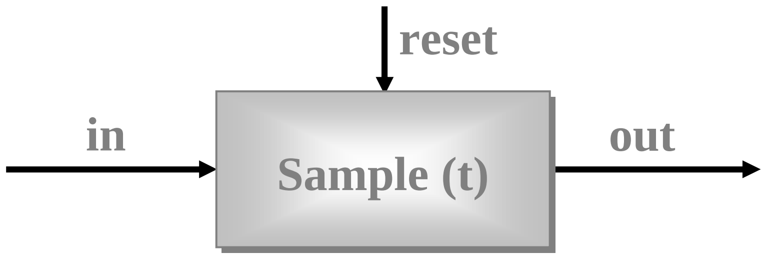





```
public void run () {

  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

}
```

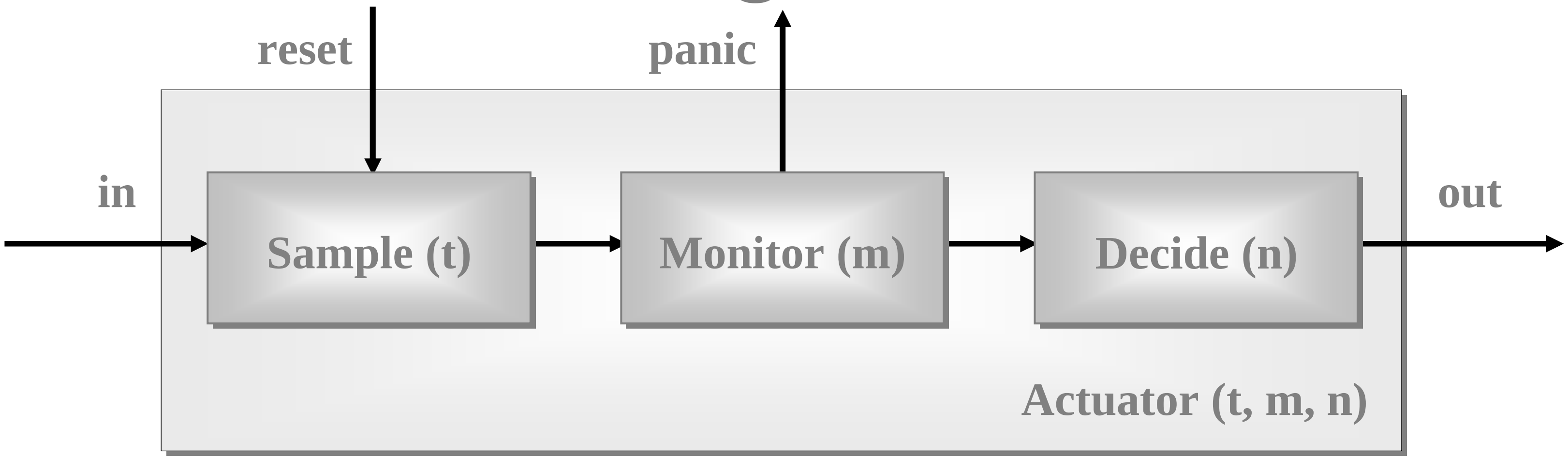



```
while (true) {
  switch (alt.priSelect ()) {
    case RESET:
      t = reset.read ();
    break;
    case TIM:
      out.write (sample);
      sample = null;
      timeout += t;
      tim.setAlarm (timeout);
    break;
    case IN:
      sample = in.read ();
    break;
  }
}
```

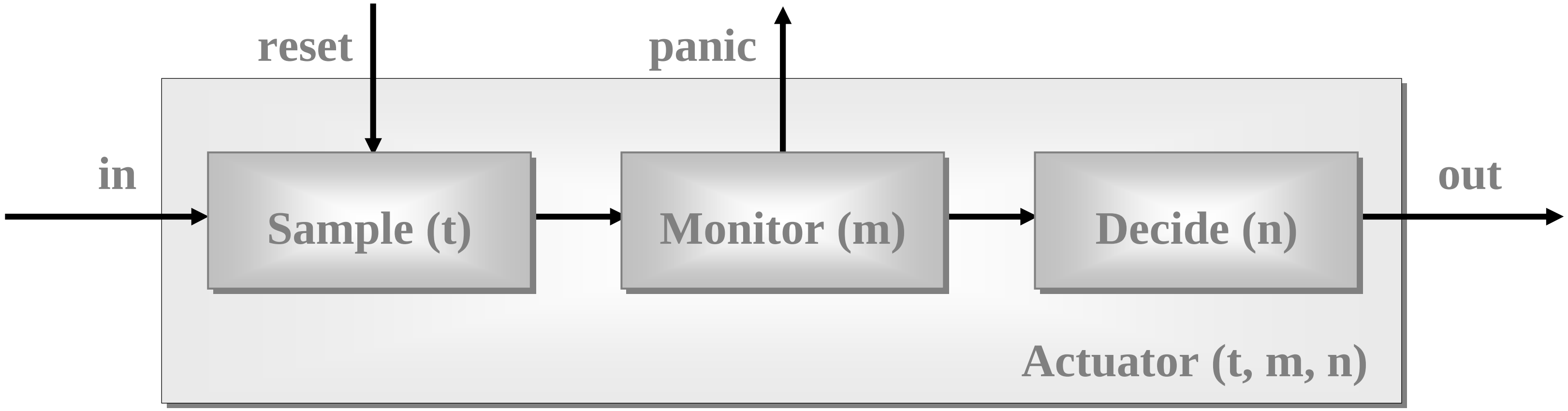



```
while (true) {
  switch (alt.priSelect ()) {
    case RESET:
      t = reset.read ();
      timeout = tim.read ();  // fall through
    case TIM:
      out.write (sample);
      sample = null;
      timeout += t;
      tim.setAlarm (timeout);
    break;
    case IN:
      sample = in.read ();
  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

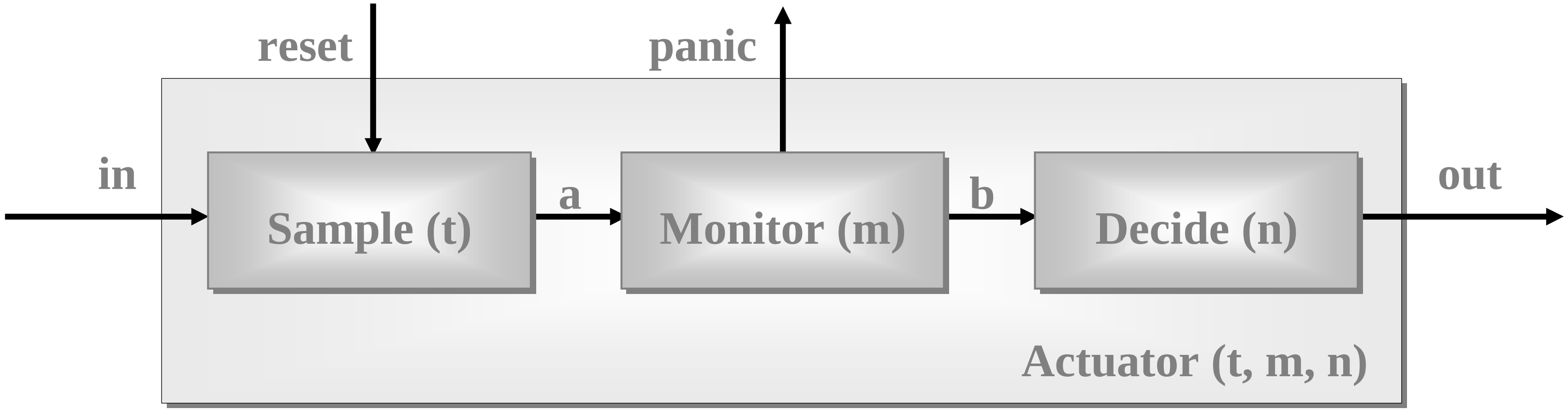
Final Stage Actuator



- **`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 = new One2OneChannel ();
  final One2OneChannel b = new One2OneChannel ();

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

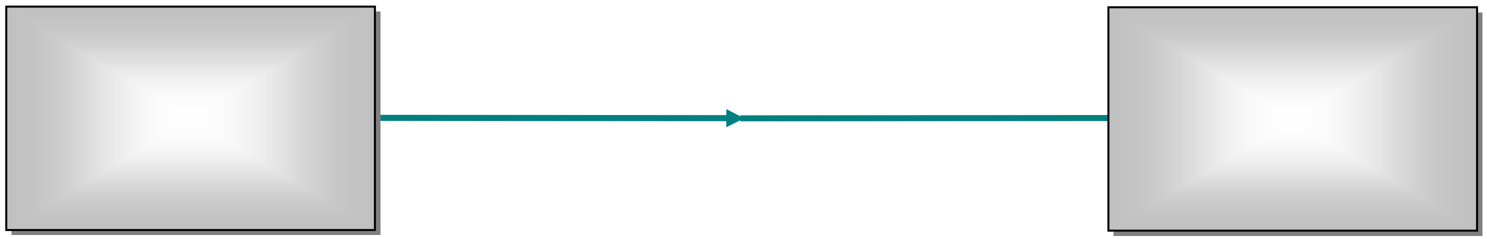
Shared Channels

- So far, all our channels have been point-to-point, zero-buffered and synchronised (i.e. standard **CSP** primitives);

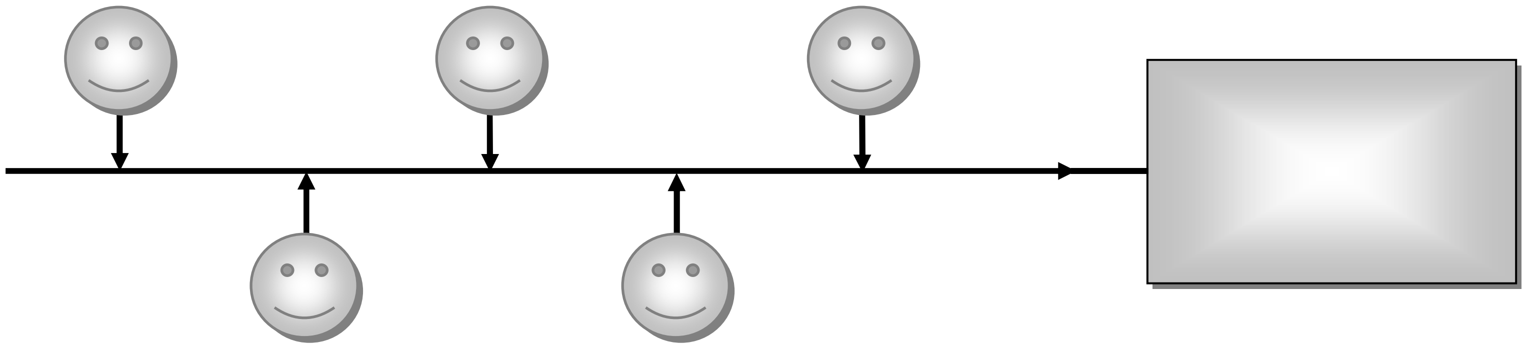
- **JCSP** also offers multi-way shared channels

- **JCSP** also offers buffered channels of various well-defined forms.

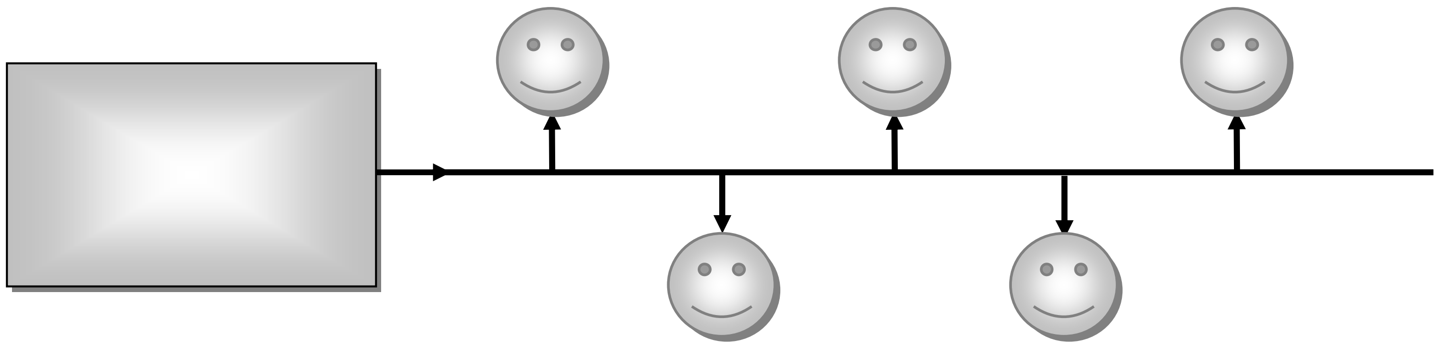
One2OneChannel



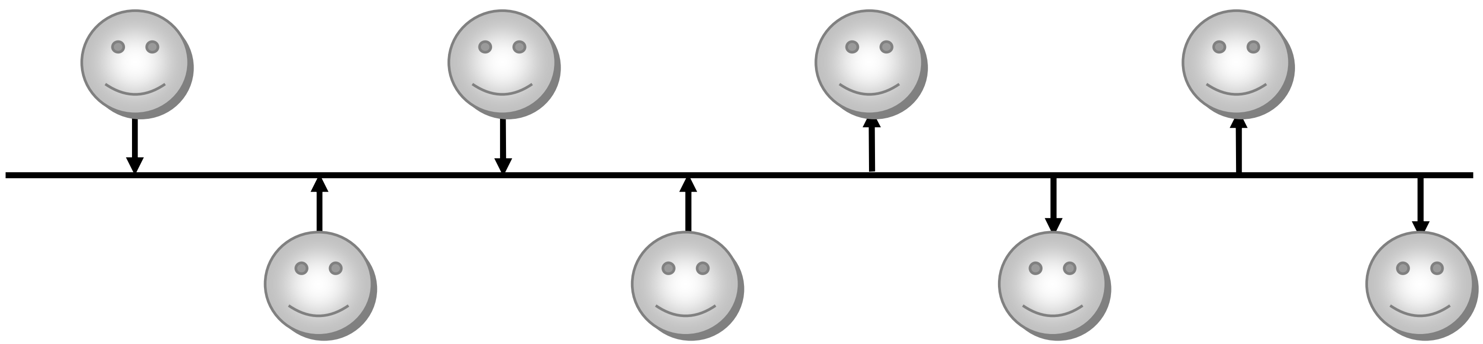
Any2OneChannel



One2AnyChannel



Any2AnyChannel



No ALTing!

`Object` Channel *classes*

```
class One2OneChannel
  extends AltingChannelInput
  implements ChannelOutput;

class One2AnyChannel
  implements ChannelInput,
             ChannelOutput;

class Any2OneChannel
  extends AltingChannelInput
  implements ChannelOutput;

class Any2AnyChannel
  implements ChannelInput,
             ChannelOutput;
```

- By default, channels are **zero-buffered** (*fully synchronised*).
- **`JCSP`** provides a set of channel *plugins* that provide a variety of buffering semantics (e.g. ***FIFO*** **blocking**, **overflowing**, **overwriting**, **infinite**).
- See **`jcsp.util`**.

`int` Channel *classes*

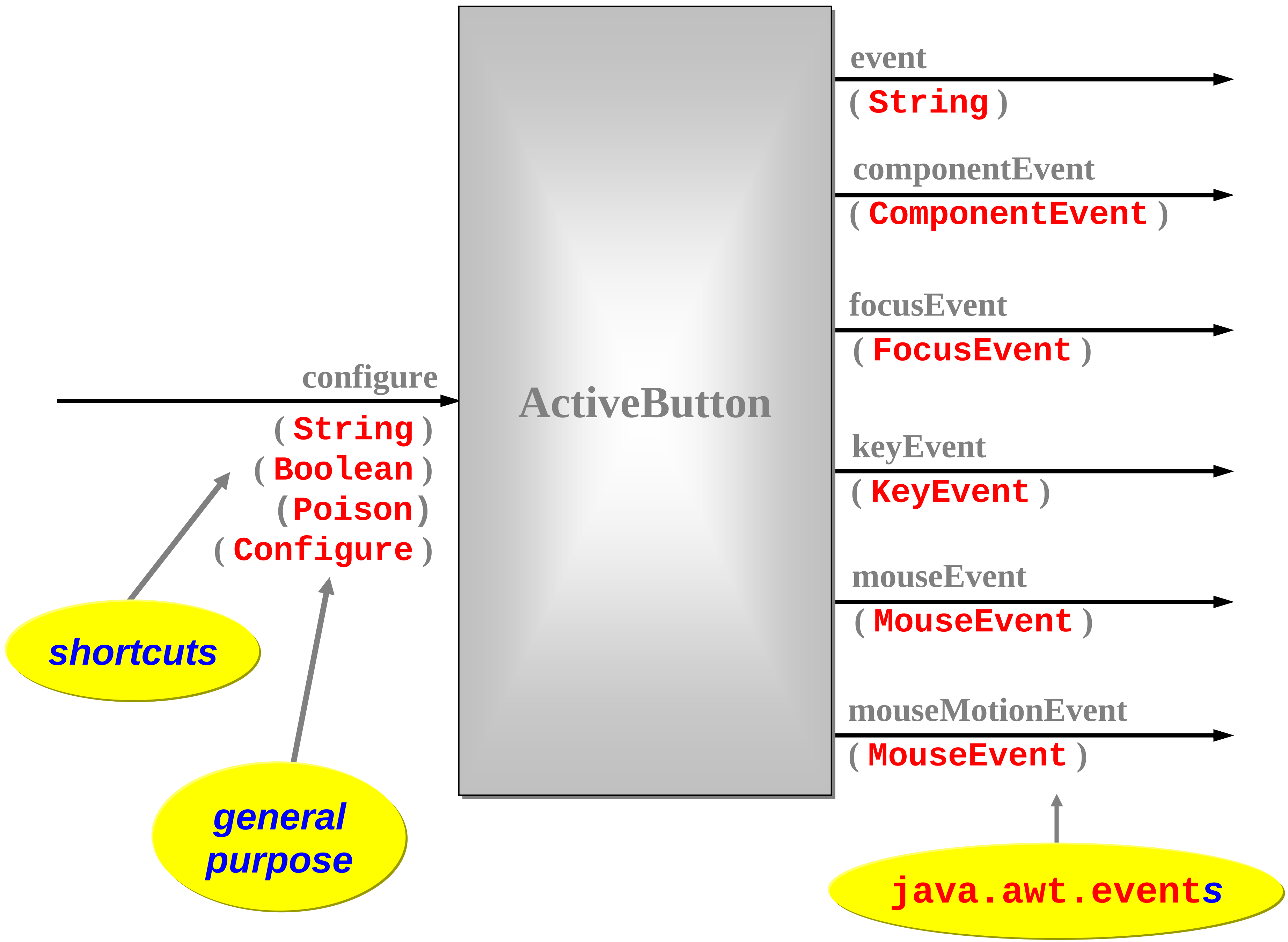
```
class One2OneChannelInt
  extends AltingChannelInputInt
  implements ChannelOutputInt;

class One2AnyChannelInt
  implements ChannelInputInt,
             ChannelOutputInt;

class Any2OneChannelInt
  extends AltingChannelInputInt
  implements ChannelOutputInt;

class Any2AnyChannelInt
  implements ChannelInputInt,
             ChannelOutputInt;
```

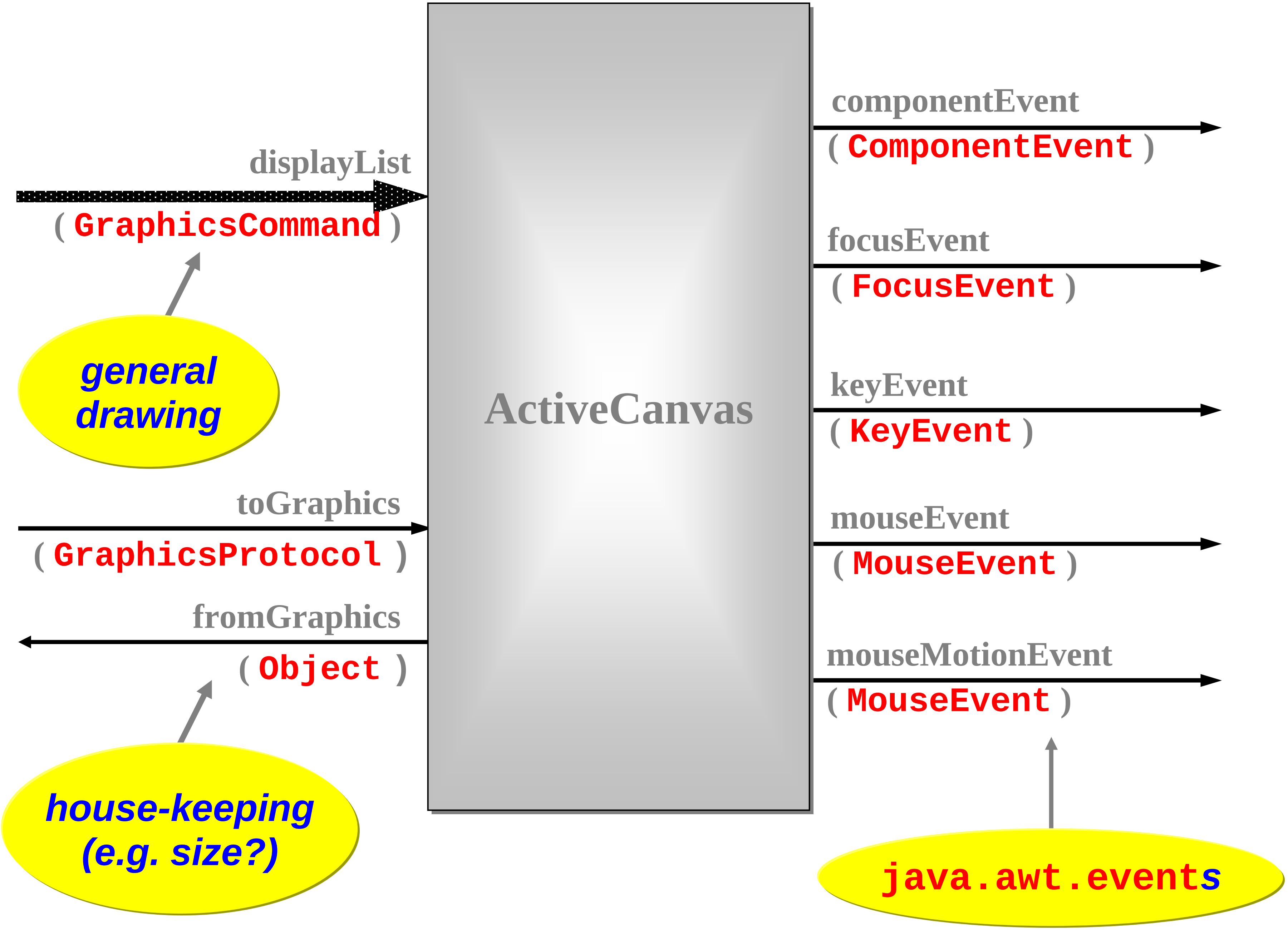
- By default, channels are **zero-buffered** (*fully synchronised*).
- **JCSP** provides a set of channel *plugins* that provide a variety of buffering semantics (e.g. ***FIFO* blocking**, **overflowing**, **overwriting**, **infinite**).
- See **`jcsp.util.ints`**.

Graphics and GUIs

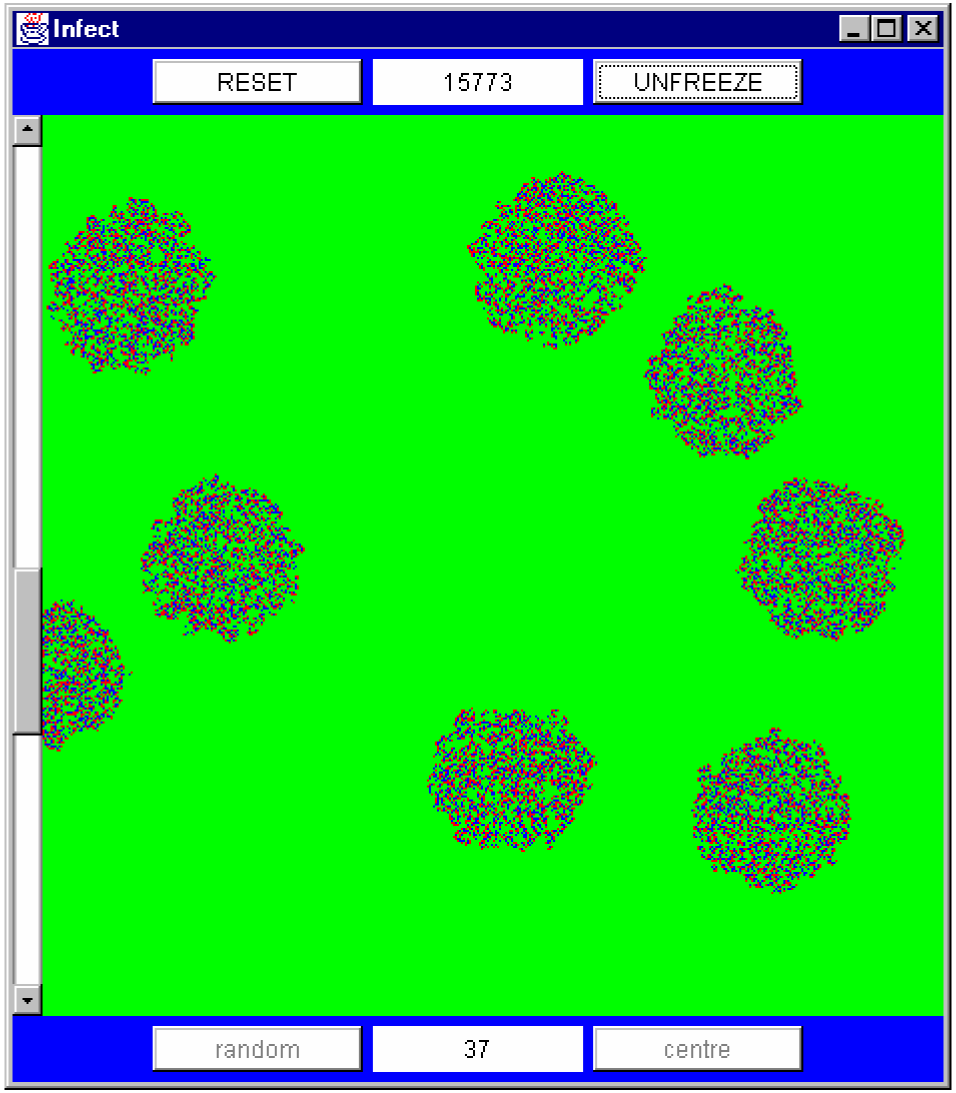
jcsp.awt = java.awt + channels

GUI events	→	channel communications
Widget configuration	→	channel communications
Graphics commands	→	channel communications

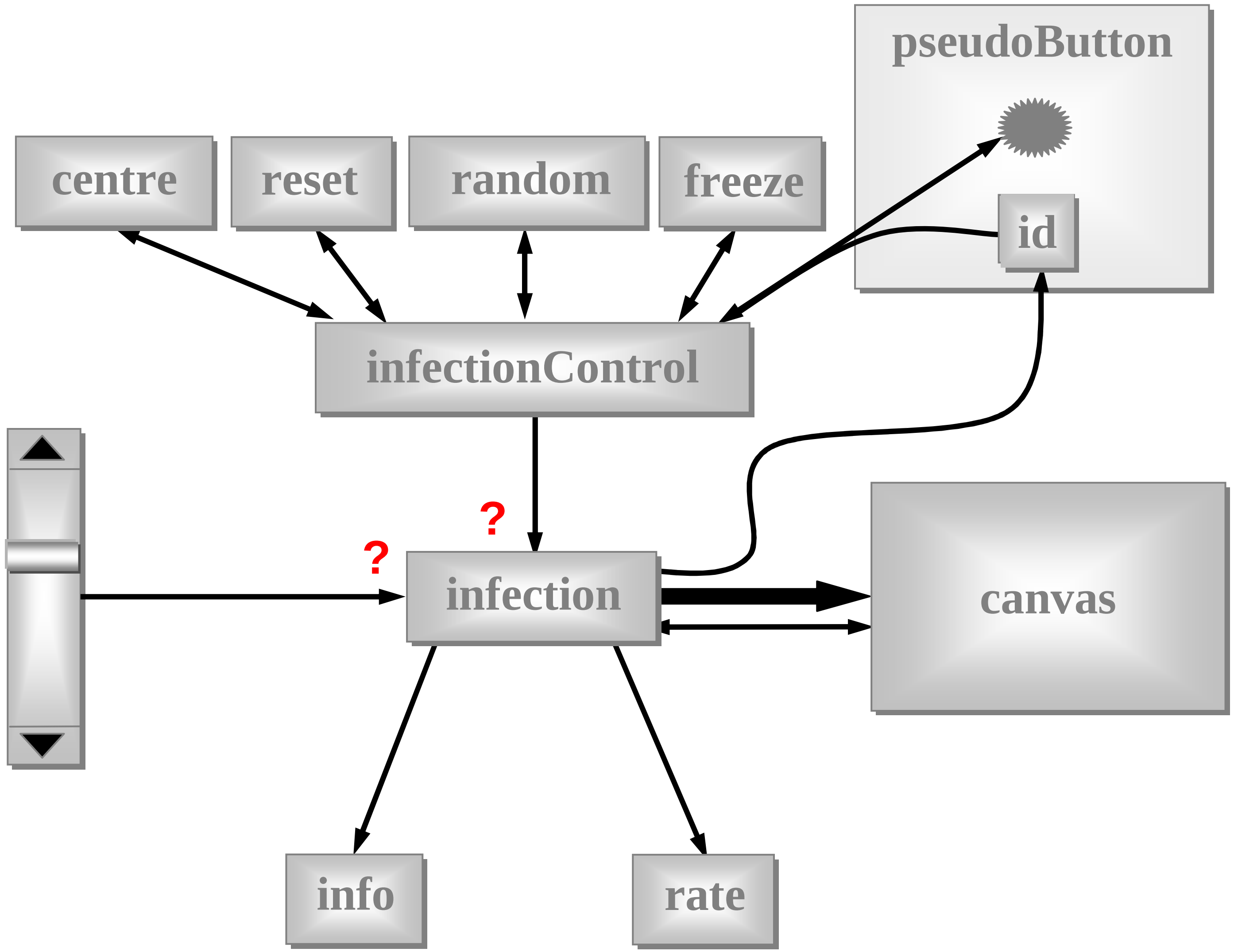

configure
(String)
(Boolean)
(Poison)
(Configure)
shortcuts
general purpose
ActiveButton
event
(String)
componentEvent
(ComponentEvent)
focusEvent
(FocusEvent)
keyEvent
(KeyEvent)
mouseEvent
(MouseEvent)
mouseMotionEvent
(MouseEvent)
java.awt.events


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

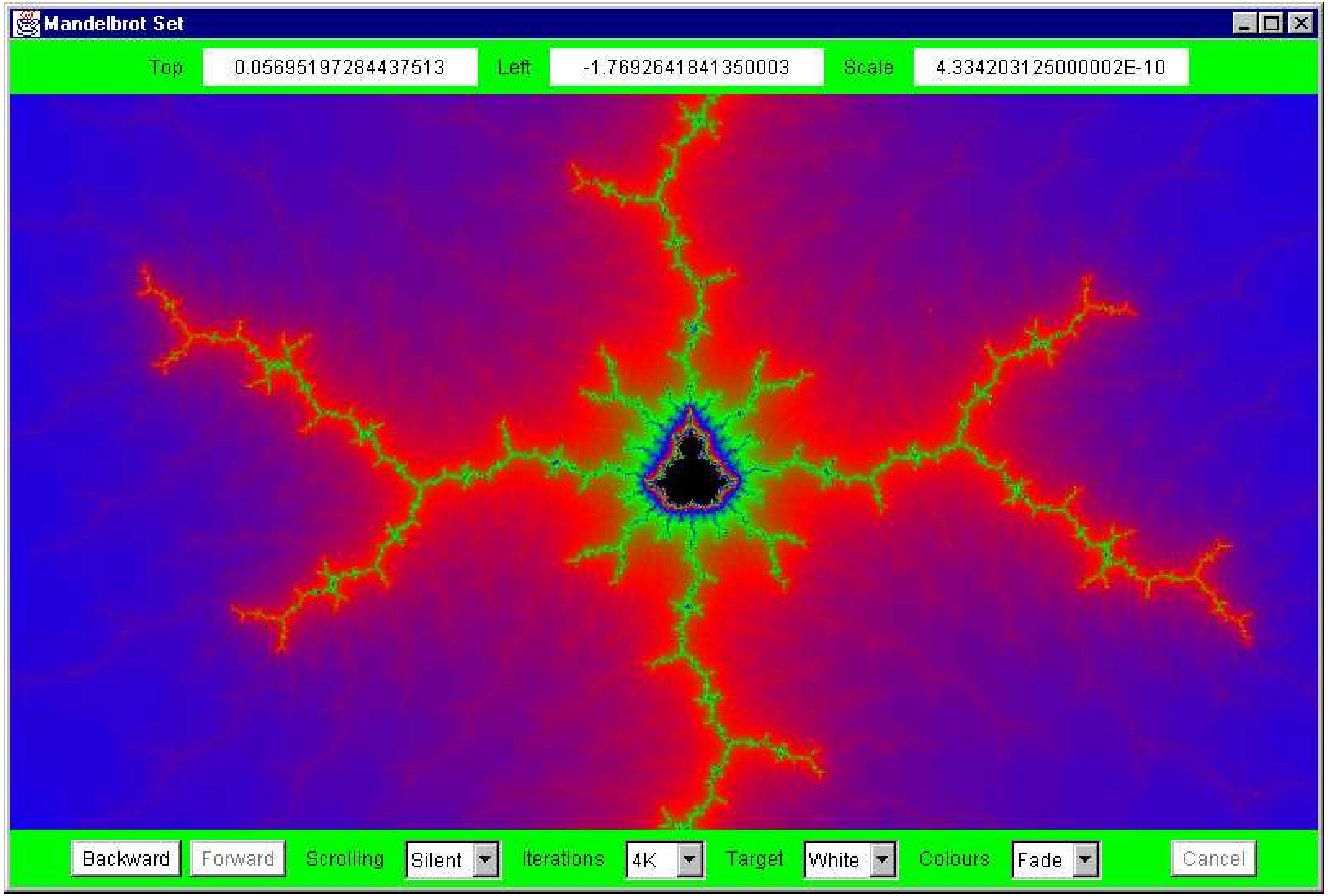
Infection



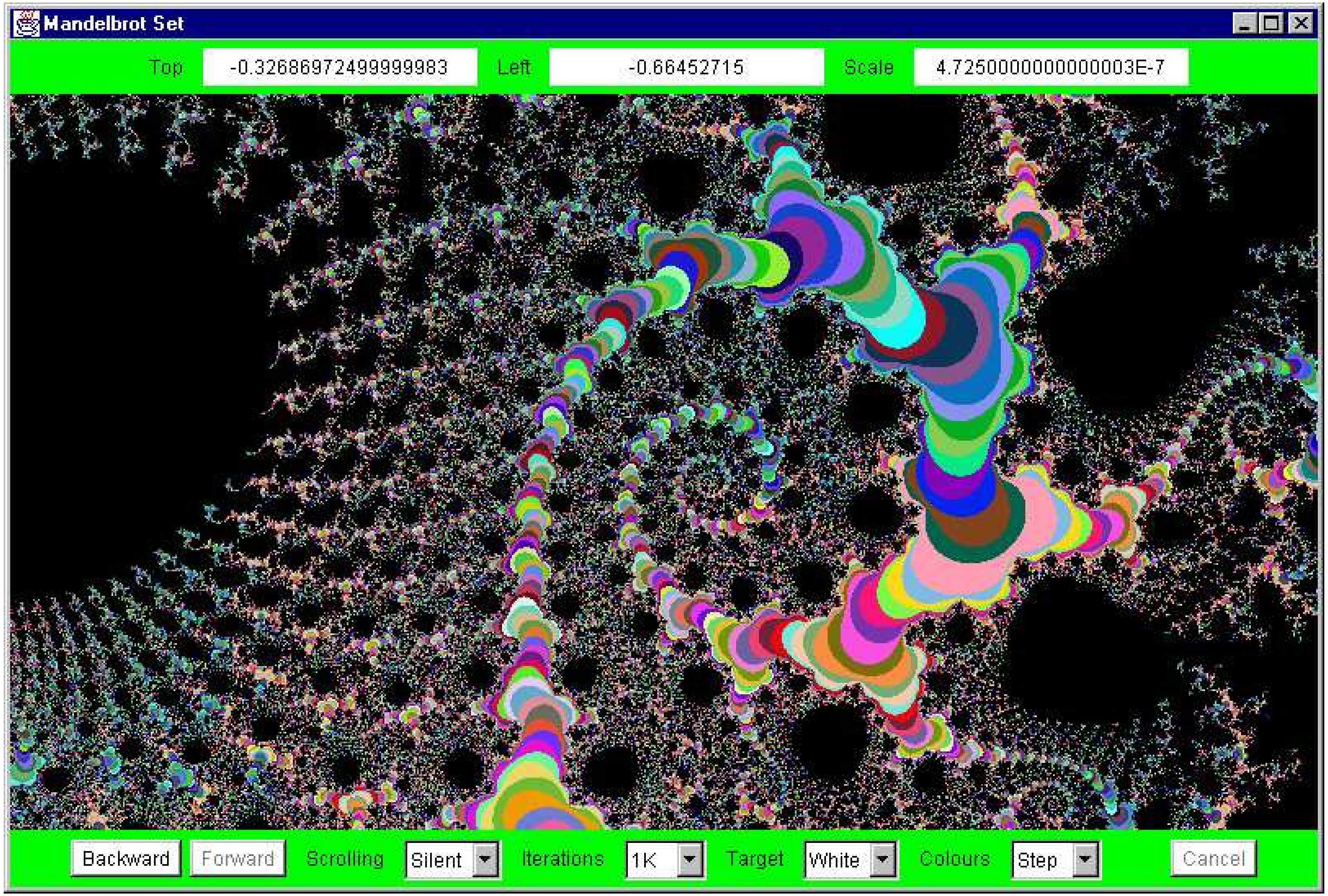
Infection


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

Mandelbrot

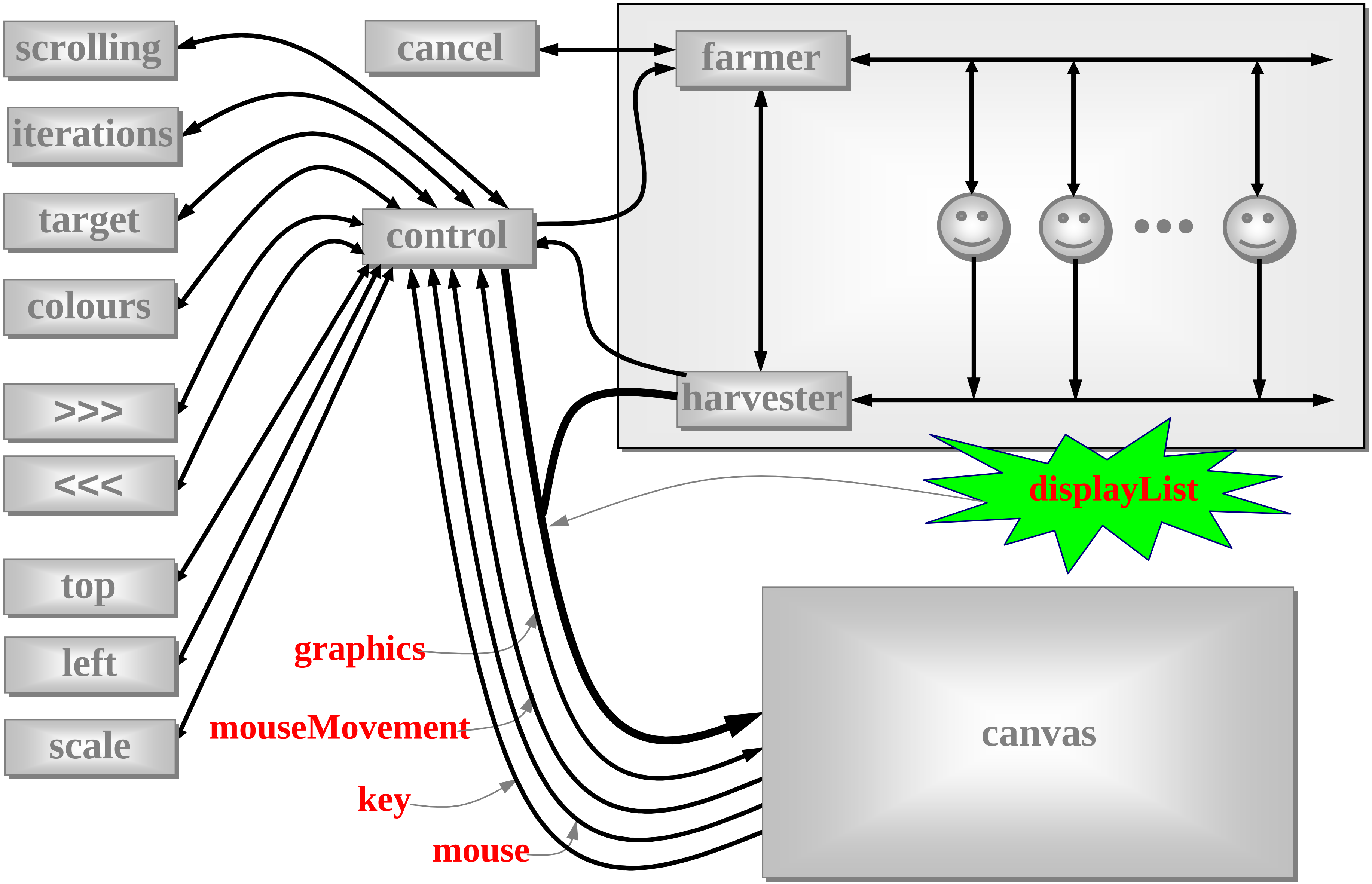

Mandelbrot Set
Top
0.05695197284437513
Left
-1.7692641841350003
Scale
4.334203125000002E-10
Backward
Forward
Scrolling
Silent
Iterations
4K
Target
White
Colours
Fade
Cancel

Mandelbrot



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

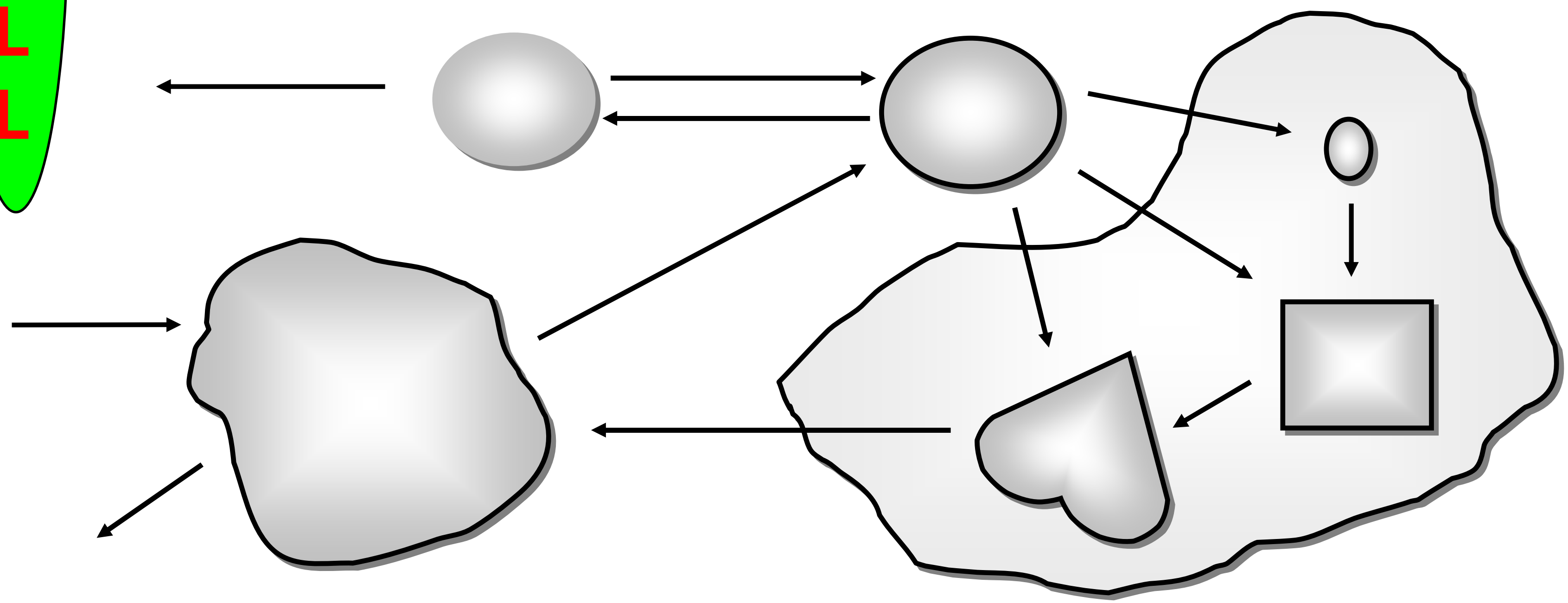
Mandelbrot


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

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

RECALL

… nuclear … human … astronomic …



Good News!

RECALL

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.**

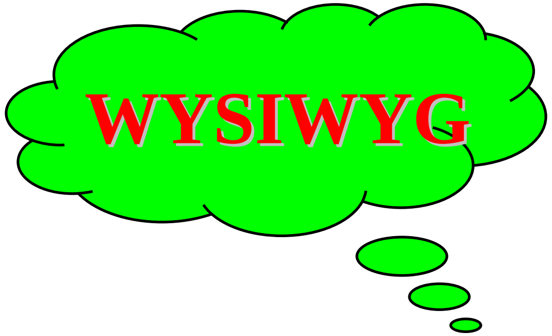
Other Work

- 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



- **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.