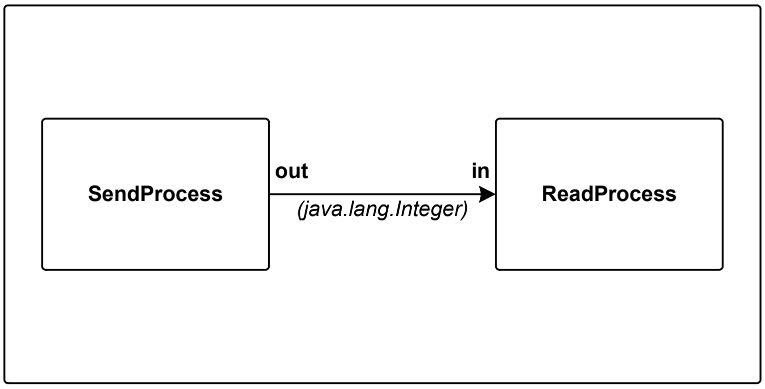
An Introduction to JCSP Base Edition

In JCSP, a *process* is a piece of code that can be executed concurrently. Any process class must implement the `CSProcess` interface, which is the basic building block of all JCSP programs. The interface has only one method - `public void run()` - which simply consists of the code that will be executed.

A process should encapsulate all its data structures and algorithms. Other processes should not be able to see that data or execute those algorithms. Running processes interact solely via JCSP channels, events or carefully synchronised access to shared objects - not by calling each other's methods as in regular Java programming.

Using channels

Our first example demonstrates the very simple process network illustrated below:



`SendProcess` and `ReadProcess` are connected by a `One2OneChannel`.

`SendProcess` repeatedly sends the numbers 0 -199 down this channel whilst `ReadProcess` receives them and prints them to the screen.

If we were implementing this program using JCSP on a single computer we might well approach it something like this:

1. Define a class called `SendProcess` which implements `CSProcess`. The run method of a `CSProcess` determines what that process will do when once started - and in this case it repeatedly sends the Integers 0 to 199 down its channel output, `out`. The `ChannelOutput` object (which is obtained from the `One2OneChannel` as we shall see later) is passed to the process in its constructor and then stored locally in a private field.

```
import com.quickstone.jcsp.lang.*;

public class SendProcess implements CSProcess {
  private ChannelOutput out;

  public SendProcess(ChannelOutput out) {
    this.out = out;
  }

  public void run() {
    int i = 0;
    while (true) {
      i = (i + 1) % 200;
      out.write(new Integer(i));
    }
  }
}
```

We use the ChannelOutput interface to prevent attempts to input on this channel by mistake.

Our process is passed the ChannelOutput it is to use in the constructor and stores it in its own private field.

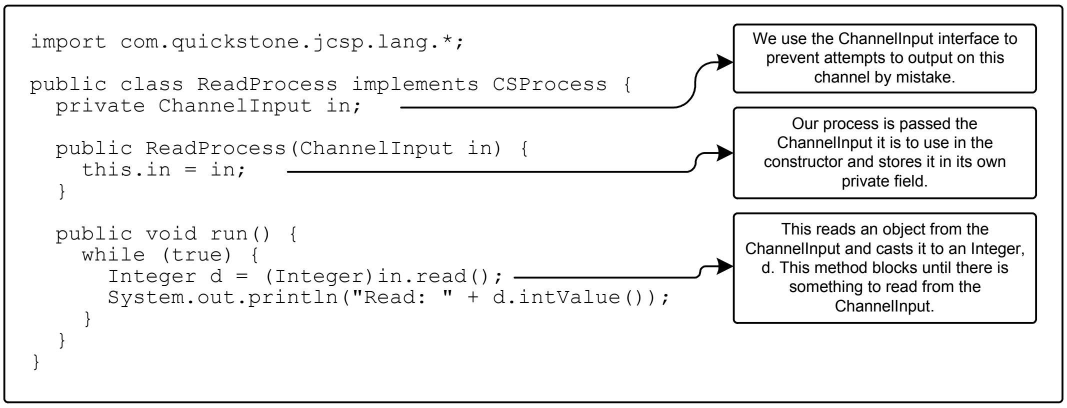
The run method contains all the code we want this process to execute. More complex processes can call other internal private methods from the run method.

Writes an Integer to the ChannelOutput. This method blocks until the data has been accepted by the reader process calling read on the ChannelInput

The process waits at the `out.write(new Integer(i));` line until the data has been accepted by the receiving process at the other end of the channel. This is because all channels in JCSP are zero-buffered by default - the `write` method will 'block' (i.e. suspend execution of its thread) until the receiving process has called `read` and the data has been transferred.

Note: JCSP also supports buffered channels which can store data within the channel. This allows the writing process to complete the write method and get on with something else without having to block and wait for the reading process to call read. See the API for `com.quickstone.jcsp.lang.ChannelDataStore` for more on buffered channels.

2. Define a class called `ReadProcess` which also implements `CSProcess`. The `run` method waits at the channel input, `in`, until it receives an object (in this case an `Integer`). Then it prints it to the screen and loops back to wait for the next one.



```
import com.quickstone.jcsp.lang.*;

public class ReadProcess implements CSProcess {
  private ChannelInput in;

  public ReadProcess(ChannelInput in) {
    this.in = in;
  }

  public void run() {
    while (true) {
      Integer d = (Integer)in.read();
      System.out.println("Read: " + d.intValue());
    }
  }
}
```

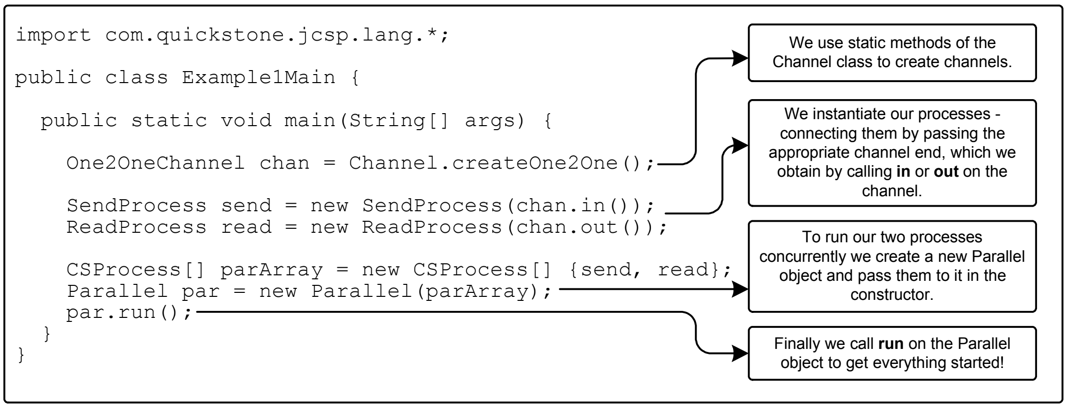
We use the ChannelInput interface to prevent attempts to output on this channel by mistake.

Our process is passed the ChannelInput it is to use in the constructor and stores it in its own private field.

This reads an object from the ChannelInput and casts it to an Integer, d. This method blocks until there is something to read from the ChannelInput.

The channel input mechanism is the reverse of the output mechanism described above - in this case the `read` method blocks until some data has been sent down the channel for the method to read. Only then can the `run` method continue.

3. Define a main class which creates an instance of `SendProcess` and an instance of `ReadProcess`, connecting them with a `One2OneChannel` - the correct end of which is passed to each process in its constructor.



```
import com.quickstone.jcsp.lang.*;

public class Example1Main {

  public static void main(String[] args) {

    One2OneChannel chan = Channel.createOne2One();

    SendProcess send = new SendProcess(chan.in());
    ReadProcess read = new ReadProcess(chan.out());

    CSProcess[] parArray = new CSProcess[] {send, read};
    Parallel par = new Parallel(parArray);
    par.run();
  }
}
```

We use static methods of the Channel class to create channels.

We instantiate our processes - connecting them by passing the appropriate channel end, which we obtain by calling **in** or **out** on the channel.

To run our two processes concurrently we create a new Parallel object and pass them to it in the constructor.

Finally we call **run** on the Parallel object to get everything started!

We obtain the `ChannelInput` and `ChannelOutput` objects from the channel by calling `in` and `out` on it respectively. A `One2OneChannel`, as you might imagine, is designed to link two processes exclusively, so each `ChannelInput` and `ChannelOutput` can only be plugged into a single process.

We put both the processes that we want to run concurrently into an array of `CSProcess`es and pass this array to a new `Parallel` object. We then invoke the `Parallel` object's `run` method which in turn invokes the `run` method of all its constituent processes and launches each one in a separate thread. The `Parallel` object's run method will only terminate when all of its constituent processes have terminated. In this example that will never happen as both processes contain an infinite loop.

The example above is a rather long-winded way of writing the code for the main class - we can make it more concise by taking advantage of Java's anonymous inner classes to create most of the objects on the fly:

```
import com.quickstone.jcsp.lang.*;

public class Example1Main {

  public static void main(String[] args) {

    One2OneChannel chan = Channel.createOne2One();

    new Parallel(new CSProcess[] {
      new SendProcess(chan.in()),
      new ReadProcess(chan.out())
    }).run();
  }
}
```

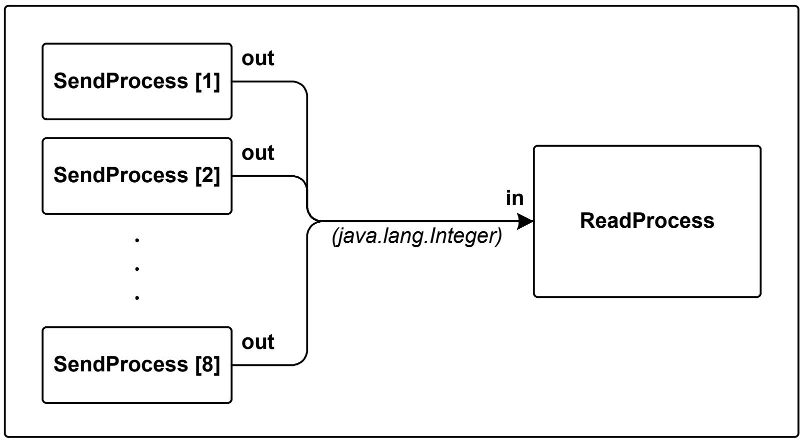
It's worth noting at this point that the constituent processes of a `Parallel` object cannot be modified whilst the `Parallel` is running. `Parallel` does have two methods, `addProcess` and `removeProcess`, which can be used to modify the `CSProcess` array but if they are called while the `Parallel` is running the changes will not take effect until the `Parallel` object's `run` method has terminated and been re-invoked.

Note: If you do need to be able to spawn processes on the fly, you can use the `ProcessManager` class which enables a `CSProcess` to be spawned concurrently with the process doing the spawning. This class also provides methods to manage the spawned process: `start`, `suspend`, `resume`, `join` and `stop`. In some circumstances the ability to dynamically spawn processes in response to events can be very useful. However, spawning processes is not the normal way of creating a network in JCSP and requires the programmer to be more wary of creating concurrency problems such as deadlock, livelock and race-hazards. If you do need to use the `ProcessManager` class, then be sure and read the accompanying documentation in the API.

Other Basic Channel Types

As mentioned above, a `One2OneChannel` can only have one writer process and one reader process. JCSP also includes Any2One, One2Any and Any2Any channels, for use by multiple writers and readers.

As an example, we could modify the process network above so that multiple `SendProcess`es send numbers to a single `ReadProcess` via an `Any2OneChannel`:

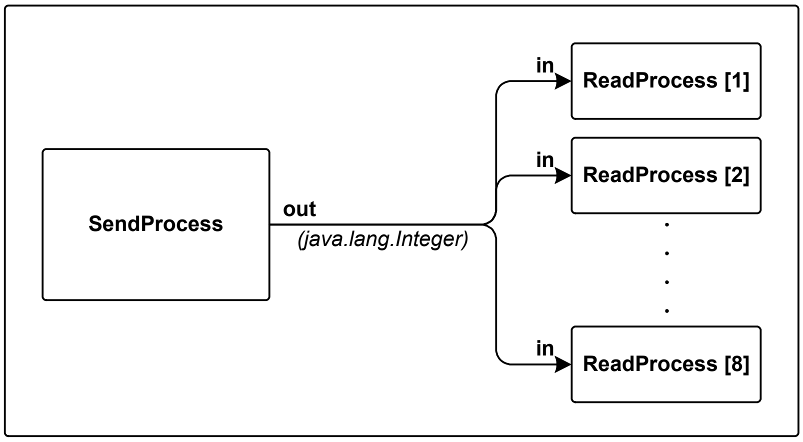


To do this, we need only change the main class:

```
import com.quickstone.jcsp.lang.*;

public class Example2Main {

   public static void main(String[] args) {

      Any2OneChannel chan = Channel.createAny2One();

      CSProcess[] procArray = new CSProcess[9];
      procArray[0] = new ReadProcess(chan.in());
      for (int i = 1; i < procArray.length; i++) {
        procArray[i] = new SendProcess(chan.out());
      }

      new Parallel(procArray).run();
   }
}
```

This will result in each number from 0 - 199 being received and printed out 8 times by the `ReadProcess` (it gets each number once from each `SendProcess`). Note that the order in which the numbers are printed may not be consistent. This is because each `SendProcess` is competing for processor time, and this is determined by the underlying thread scheduling performed by the JVM.

For our next example we will reverse the situation like so:



```
import com.quickstone.jcsp.lang.*;

public class Example3Main {

   public static void main(String[] args) {
      One2AnyChannel chan = Channel.createOne2Any();

      CSProcess[] procArray = new CSProcess[9];
      procArray[0] = new SendProcess(chan.out());
      for (int i = 1; i < procArray.length; i++) {
        procArray[i] = new ReadProcess(chan.in());
      }

      new Parallel(procArray).run();
   }
}
```

This would result in each number being printed out only once. This is because One2Any and Any2Any channels are not broadcasters - the reader processes compete to receive the message, and only one of them will actually receive and print it. Broadcasting can easily be achieved by writing active processes. (`com.quickstone.jcsp.plugNplay.DynamicDelta` is a good example and also a handy general-purpose broadcasting process). One2Any and Any2Any channels are mainly used for things like farming tasks out to multiple worker processes, where it doesn't matter which process gets the task.

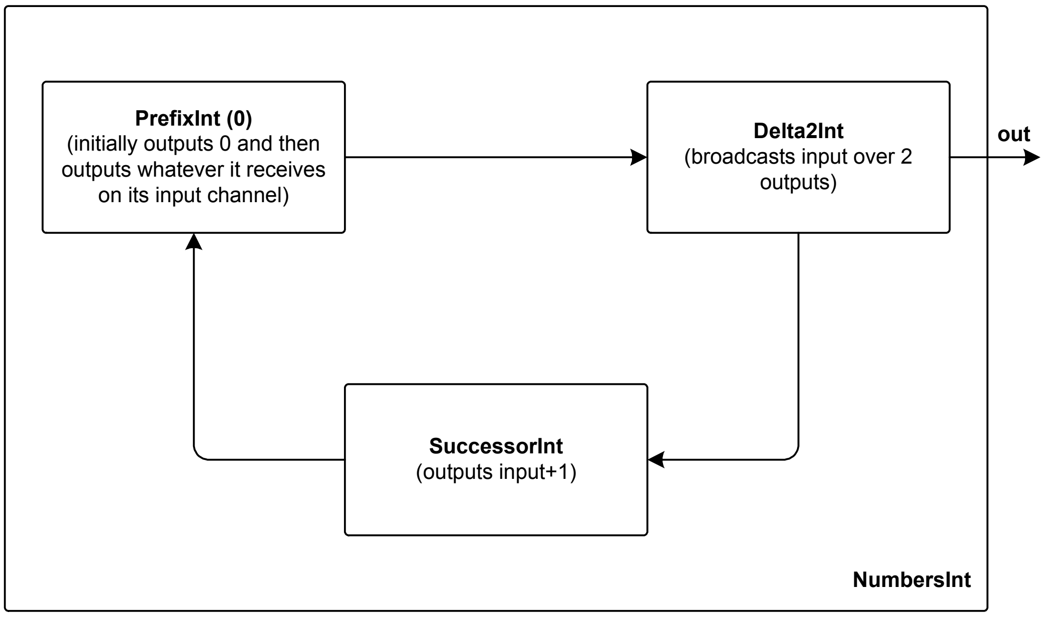
Any serializable object can be sent down a channel, but primitive types such as `int`s and `boolean`s cannot. However, as it is fairly common to use channels just for integer numbers, each channel also has an `int` only implementation (for example `One2OneChannelInt`) which has a slightly reduced overhead.

Layered Process Networks

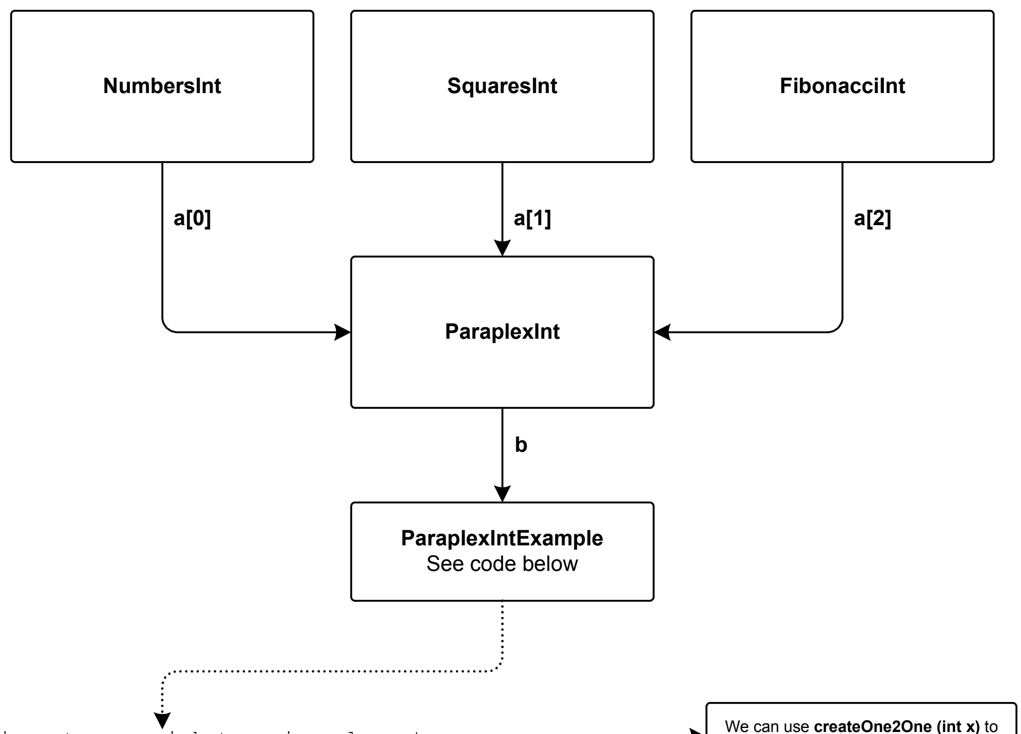
One of the nicest features of JCSP is the way process networks can be 'layered' to make them easier to understand and maintain. The diagram on the next page shows a process network which prints out tabulated columns of natural numbers, perfect squares and the Fibonacci sequence.

At this level, we are only aware of five communicating processes: three that generate the respective sequences of integers, one that multiplexes a single item from each sequence into a single packet and the in-lined process that receives this packet and tabulates its contents. And, at this level, that is all we need to think about.

However, each of the processes in the diagram above contains its own sub-network of processes and in the case `SquaresInt` and `FibonacciInt`, sub-sub-networks. A network diagram for `NumbersInt` is shown below to demonstrate. More details and diagrams for each process can be found in the API documentation for the `com.quickstone.jcsp.plugNplay.ints` package.


PrefixInt (0)
(initially outputs 0 and then outputs whatever it receives on its input channel)
Delta2Int
(broadcasts input over 2 outputs)
out
SuccessorInt
(outputs input+1)
NumbersInt

Altogether, the example on the next page contains 28 parallel processes. One of the key benefits of JCSP is that we do not have to reason about all those 28 processes at the same time to reason about how they behave in this application. We can build up the complexity in layers.



```
import com.quickstone.jcsp.lang.*;
import com.quickstone.jcsp.plugNplay.ints.*;

class ParaplexIntExample {

  public static void main (String[] args) {

    final One2OneChannelInt[] a =
      IntChannel.createOne2One(3);

    final One2OneChannel b =
      IntChannel.createOne2One();

    ChannelInputInt[] inArray =
      new ChannelInputInt[3];

    for (int i = 0; i < inArray.length; i++) {
      inArray[i] = a[i].in();
    }

    final ChannelInput bin = b.in();
    final ChannelOutput bout = b.out();

    new Parallel (new CSProcess[] {

      new NumbersInt  (a[0].out()),
      new SquaresInt  (a[1].out()),
      new FibonacciInt  (a[2].out()),

      new ParaplexInt (inArray, bout),

      new CSProcess () {
        public void run () {

          System.out.println
            ("\n\t\tNumbers\t\tSquares\t\tFibonacci\n");

          while (true) {
            int[] data = (int[]) bin.read();
            for (int i = 0; i < data.length; i++) {
              System.out.print ("\t\t" + data[i]);
            }
            System.out.println();
          }
        }
      }
    }).run();
  }
}
```

We can use **createOne2One (int x)** to create an array of x One2One channels. Notice we're using IntChannel instead of Channel to create int only channels.

ParaplexInt takes an array of ChannelInputInts in its constructor, so here we create one from the channel array, **a**, we made earlier.

Here we grab the channel ends from our One2OneChannel **b** for later use.

Here we create a new Parallel object and pass it a CSProcess array which we create on the fly. We instantiate each of the generator processes and the ParaplexInt, which merges the output from each generator into a single array and sends it down a single channel, b. The processes are connected by the channels we created earlier.

Note that we are creating the last process, which prints out the numbers, as an anonymous inner class of this main class. We already have a handle on the ChannelInput we need to use to read the data (**bin**), so we don't require a constructor to get it.

The main loop of our in-lined process waits on its ChannelInput for data to become available, then prints it out and loops round to wait for the next bit.

Finally we call run on the Parallel object to get the program started!

The Alternative Class

There are many situations where we want a process to be able to 'listen' on several input channels. However if we just plug them into the process and then do a series of `read`s we run into the problem of the process waiting (perhaps indefinitely) for data on one channel whilst other channels are desperate to be serviced. This is where the `Alternative` class comes in.

The `Alternative` class is used when we want a process to be able to passively listen on several incoming channels and select between them when more than one is sending simultaneously. This can be done either arbitrarily, fairly or giving certain channels priority over others, and is known as `ALT`ing.

The `Alternative` constructor takes an array of '`Guards`' to be `ALT`ed over. As described above, a `Guard` can be a channel input, but JCSP also provides several other kinds to allow more complex behaviour to be implemented. In the example below we use a `CSTimer` as a `Guard` to implement a timeout. For more information on the various types of `Guard` see the API for `com.quickstone.jcsp.lang.Alternative`.

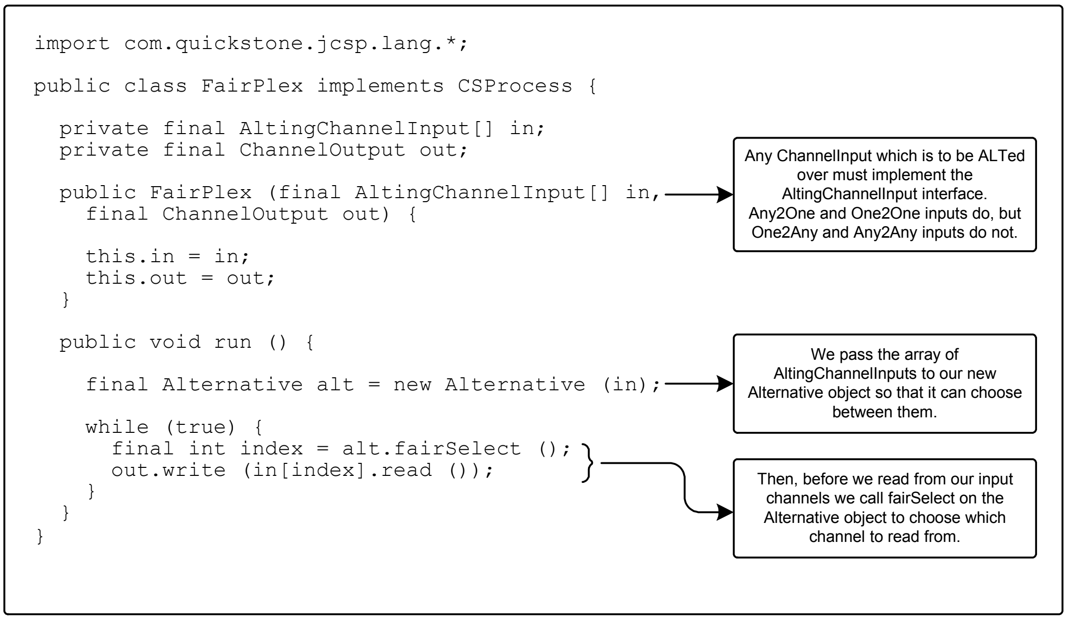
The `Alternative` object is instructed to select a `Guard` to service by invoking one of the following methods. The methods differ in the way that they choose which `Guard` to select in the case when two or more `Guard`s are ready. If no `Guard`s are ready the `Alternative` will simply wait passively until one is. Each method returns an `int` - the index of the `Guard` that was selected. This can then be used to read from that `Guard`, or if more complex behaviour is required a switch statement can be used to carry out the appropriate actions.

`select`: waits for one or more of the `Guard`s to become ready. If more than one becomes ready, it makes an arbitrary choice between them.

`priSelect`: also waits for one or more of the `Guard`s to become ready. However, if more than one becomes ready, it chooses the first one listed.

`fairSelect`: also waits for one or more of the `Guard`s to become ready. If more than one becomes ready, it prioritises its choice so that the `Guard` it chose the last time it was invoked has lowest priority this time. This enables an upper bound on service times to be calculated and ensures that no ready `Guard` can be indefinitely starved of service.

A Fair Multiplexor



```
import com.quickstone.jcsp.lang.*;

public class FairPlex implements CSProcess {

  private final AltingChannelInput[] in;
  private final ChannelOutput out;

  public FairPlex (final AltingChannelInput[] in,
    final ChannelOutput out) {

    this.in = in;
    this.out = out;
  }

  public void run () {

    final Alternative alt = new Alternative (in);

    while (true) {
      final int index = alt.fairSelect ();
      out.write (in[index].read ());
    }
  }
}
```

Any ChannelInput which is to be ALTed over must implement the AltingChannelInput interface. Any2One and One2One inputs do, but One2Any and Any2Any inputs do not.

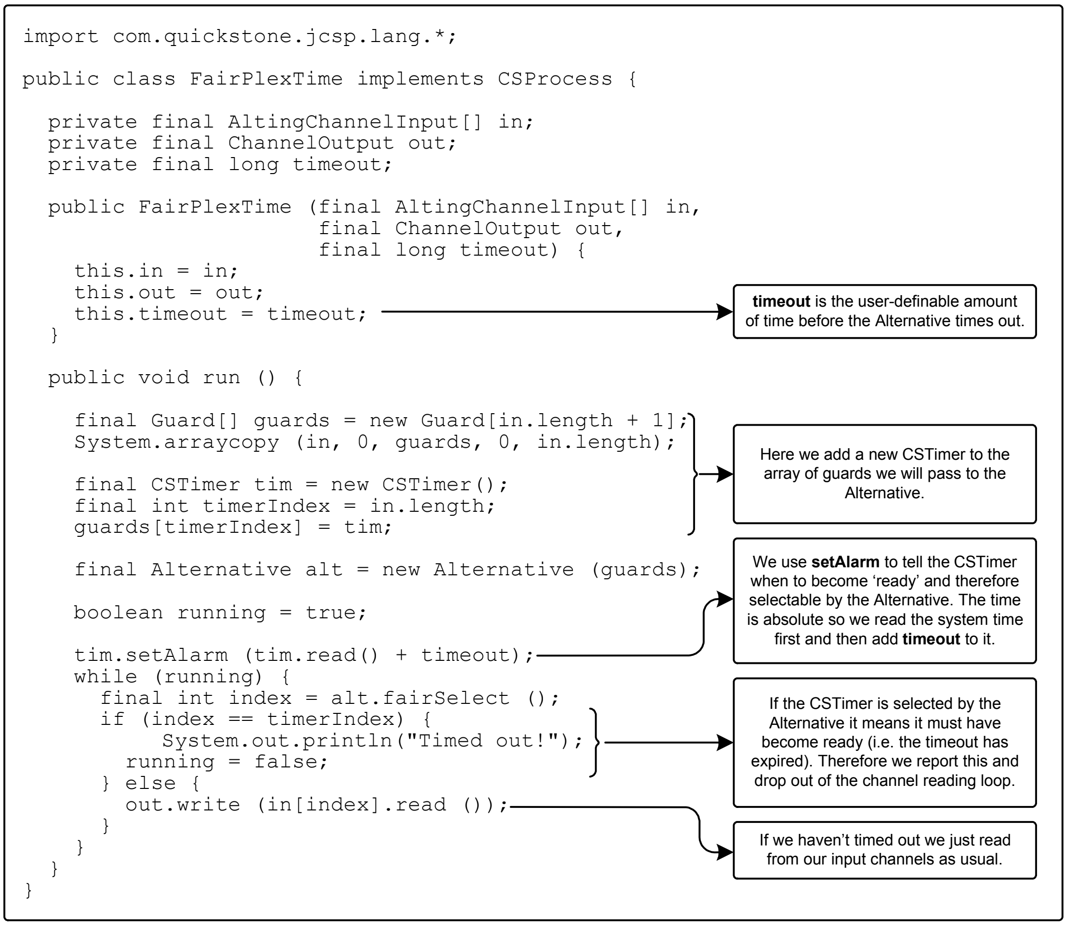
We pass the array of AltingChannelInputs to our new Alternative object so that it can choose between them.

Then, before we read from our input channels we call fairSelect on the Alternative object to choose which channel to read from.

This example demonstrates a process that fairly multiplexes traffic from its array of input channels to its single output channel. No input channel will be starved, regardless of the eagerness of its competitors.

Note that if `priSelect` were used above, higher-indexed channels would be starved if lower-indexed channels were continually demanding service. If `select` were used, no starvation analysis would be possible. The `select` mechanism should therefore only be used when starvation is not an issue.

A Fair Multiplexor with a Timeout



```
import com.quickstone.jcsp.lang.*;

public class FairPlexTime implements CSProcess {

  private final AltingChannelInput[] in;
  private final ChannelOutput out;
  private final long timeout;

  public FairPlexTime (final AltingChannelInput[] in,
                       final ChannelOutput out,
                       final long timeout) {
    this.in = in;
    this.out = out;
    this.timeout = timeout;
  }

  public void run () {

    final Guard[] guards = new Guard[in.length + 1];
    System.arraycopy (in, 0, guards, 0, in.length);

    final CSTimer tim = new CSTimer();
    final int timerIndex = in.length;
    guards[timerIndex] = tim;

    final Alternative alt = new Alternative (guards);

    boolean running = true;

    tim.setAlarm (tim.read() + timeout);
    while (running) {
      final int index = alt.fairSelect ();
      if (index == timerIndex) {
           System.out.println("Timed out!");
        running = false;
      } else {
        out.write (in[index].read ());
      }
    }
  }
}
```

timeout is the user-definable amount of time before the Alternative times out.

Here we add a new CSTimer to the array of guards we will pass to the Alternative.

We use **setAlarm** to tell the CSTimer when to become ‘ready’ and therefore selectable by the Alternative. The time is absolute so we read the system time first and then add **timeout** to it.

If the CSTimer is selected by the Alternative it means it must have become ready (i.e. the timeout has expired). Therefore we report this and drop out of the channel reading loop.

If we haven’t timed out we just read from our input channels as usual.

This example modifies `FairPlex` to produce a process that fairly multiplexes traffic from its input channels to its single output channel, but which times out after a user-settable time. This is achieved by using a `CSTimer` as one of the `Guard`s, which will only become ready after the amount of time specified by the user in the constructor has expired.

Note that if `priSelect` were used above, higher-indexed `Guard`s would be starved if lower-indexed `Guard`s were continually demanding service - and the timeout would never be noticed. However, `fairSelect` ensures that the `CSTimer` will be selected shortly after it expires and becomes ready.

If we wanted the channel read loop to only timeout if *nothing* had come in on the input channels for the specified time, we could simply reset the timer after every `read` by adding another `tim.setAlarm (tim.read() + timeout);` to the `else` block at the end of the program.

Pre-conditions

Each `Guard` in an `Alternative` may be pre-conditioned with a run-time test to decide if it should be considered at all in the current choice. This allows considerable flexibility - for example, we can decide whether timeouts should be set or channels refused depending on the run-time state of the Alternative process. To do this, we set up an array of `booleans` which corresponds to the `Guard` array and then call ...`Select(boolean[] preCondition)`. Now a `Guard` will only be selected if it is ready and its corresponding `boolean` is true.

The example below modifies the `FairPlexTime` class so that the timeout is only allowed to happen once a user-defined number of objects have been sent down the output channel.

```
import com.quickstone.jcsp.lang.*;
import java.util.*;

public class FairPlexPreTime implements CSProcess {

  private final AltingChannelInput[] in;
  private final ChannelOutput out;
  private final long timeout;
  private final int objBeforeTimeout;

  public FairPlexPreTime (
         final AltingChannelInput[] in,
         final ChannelOutput out,
         final long timeout,
         final int objBeforeTimeout) {

    this.in = in;
    this.out = out;
    this.timeout = timeout;
    this.objBeforeTimeout = objBeforeTimeout;
  }

  public void run () {

    final Guard[] guards = new Guard[in.length + 1];
    System.arraycopy (in, 0, guards, 0, in.length);

    final CSTimer tim = new CSTimer ();
    final int timerIndex = in.length;
    guards[timerIndex] = tim;

    final boolean[] preConds = new
      boolean[guards.length];
    Arrays.fill(preConds, true);
    preConds[timerIndex] = false;

    final Alternative alt = new Alternative (guards);

    boolean running = true;

    tim.setAlarm (tim.read () + timeout);
    int numberSent = 0;
    while (running) {
      preConds[timerIndex] = (numberSent >=
                              objBeforeTimeout);
      final int index = alt.fairSelect(preConds);
      if (index == timerIndex) {
        System.out.println("Timed Out!");
        running = false;
      } else {
        out.write (in[index].read ());
        numberSent++;
      }
    }
  }
}
```

Here we set up our precondition array of booleans which corresponds directly to the guard array. We then make them all true except for the one that corresponds to the CSTimer. Until this changes the CSTimer can *never* be selected by a call to **fairSelect(preConds)**.

Each time round the loop we update the boolean that relates to the CSTimer. If the requisite number of objects have been sent, we set it to **true** so the timeout can be selected. Then we call **fairSelect(preConds)** on the Alternative to make the channel selection.

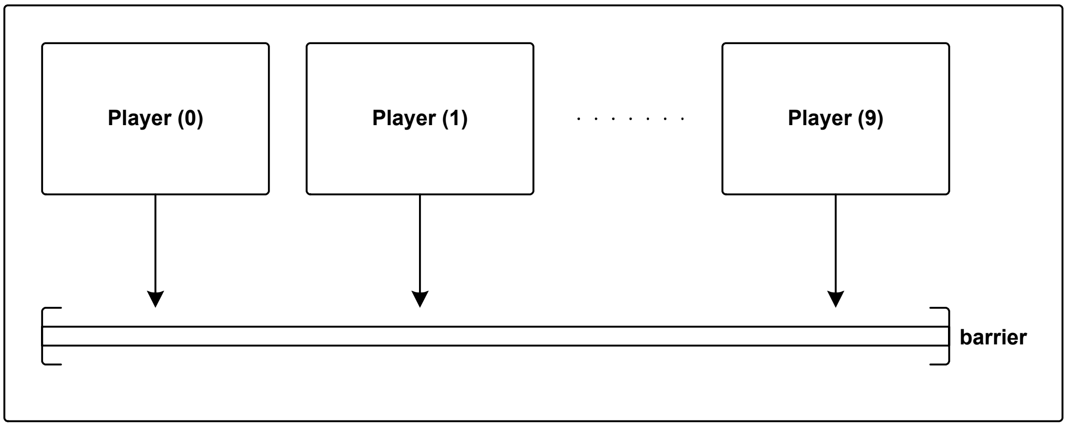
Advanced Synchronization Methods

As mentioned right at the start, channels aren't the only way to synchronize processes. Whilst a channel can only ever synchronize two processes (the writer and the reader) `Barriers` and `Buckets` can synchronize multiple processes. `Crew` (Concurrent Read Exclusive Write) locks are used to provide safe access by multiple processes to a single shared resource. JCSP also includes CALL channels which provide an extended rendezvous mechanism that allows a more object-oriented programming style.

Barriers

Any process synchronizing on a `Barrier` will block until all processes associated with that `Barrier` have synchronized. The example below shows a turn-based game where ten `Player` processes simulate taking a turn (including a random amount of 'thinking' time from 1-10 seconds) and then synchronize with a `Barrier` to wait for all the other `Players` to take their turn. Once all the

Players have taken their turn and synchronized with the Barrier, the Barrier continues the game and releases them all to take their next turn.



The code for the process network looks like this:

```
import com.quickstone.jcsp.lang.*;

public class BarrierExample1 {

  public static void main (String[] args) {

    final int nPlayers = 10;
    final Barrier barrier = new Barrier (nPlayers);
    final Player[] players = new Player[nPlayers];
    final long seed = new CSTimer().read();

    for (int i = 0; i < players.length; i++) {
      players[i] = new Player (i, seed+i, barrier);
    }

    new Parallel (players).run ();
  }
}
```

As the number of `Players` doesn't change we initialise the `Barrier` with that number.

Each `Player` needs to be able to see the `Barrier`, so we pass a reference in the constructor.

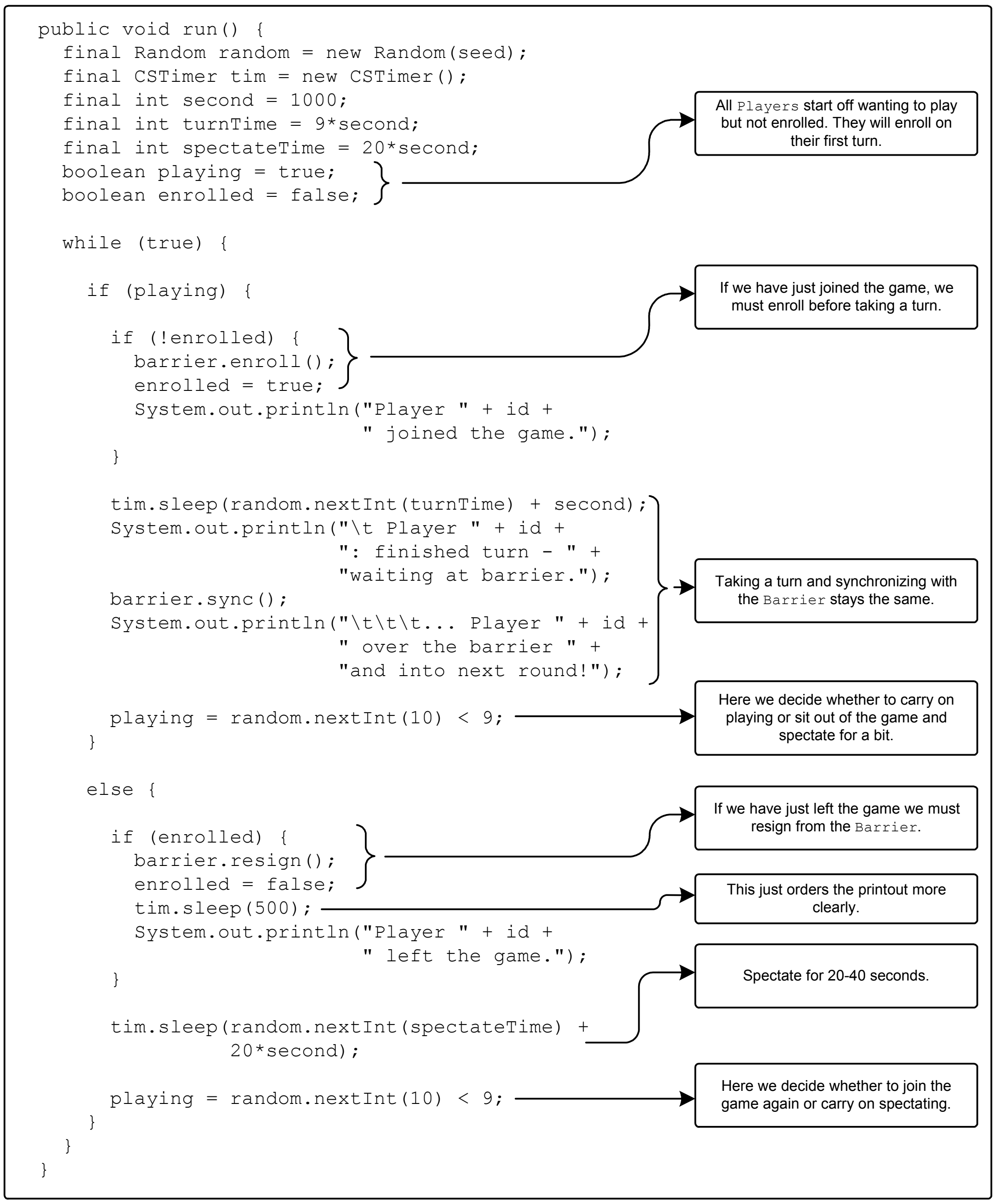
And here's the Player class. Notice the use of the Barrier's sync method:

```
import com.quickstone.jcsp.lang.*;
import java.util.*;

public class Player implements CSProcess {
  private final int id;
  private final long seed;
  private final Barrier barrier;

  public Player (int id, long seed, Barrier barrier) {
    this.id = id;
    this.seed = seed;
    this.barrier = barrier;
  }

  public void run () {
    final Random random = new Random(seed);
    final CSTimer tim = new CSTimer ();
    final int second = 1000;
    final int turnTime = 9*second;

    while (true) {
      tim.sleep (random.nextInt(turnTime)+ second);
      System.out.println ("Player " + id +
                          ": finished turn - " +
                          "waiting at barrier.");
      barrier.sync();
      System.out.println ("\t\t\t... Player " + id +
                          " over the barrier" +
                          " and into next round!");
    }
  }
}
```

This simulates the `Player` thinking and taking a turn by putting the thread to sleep for between 1 and 10 seconds.

We synchronize on the `Barrier` by calling it's `sync` method. This method blocks until all the other players have synchronized.

Processes can also join or leave a `Barrier` by calling either `enroll` or `resign` on the `Barrier` object. To demonstrate this, we can modify the `Player` class' run method so that each `Player` can make an internal decision to leave the game for a period of time:



```java
public void run() {
  final Random random = new Random(seed);
  final CSTimer tim = new CSTimer();
  final int second = 1000;
  final int turnTime = 9*second;
  final int spectateTime = 20*second;
  boolean playing = true;
  boolean enrolled = false;

  while (true) {

    if (playing) {

      if (!enrolled) {
        barrier.enroll();
        enrolled = true;
        System.out.println("Player " + id +
                           " joined the game.");
      }

      tim.sleep(random.nextInt(turnTime) + second);
      System.out.println("\t Player " + id +
                         ": finished turn - " +
                         "waiting at barrier.");
      barrier.sync();
      System.out.println("\t\t\t... Player " + id +
                         " over the barrier " +
                         "and into next round!");

      playing = random.nextInt(10) < 9;
    }

    else {

      if (enrolled) {
        barrier.resign();
        enrolled = false;
        tim.sleep(500);
        System.out.println("Player " + id +
                           " left the game.");
      }

      tim.sleep(random.nextInt(spectateTime) +
                20*second);

      playing = random.nextInt(10) < 9;
    }
  }
}
```

All `Players` start off wanting to play but not enrolled. They will enroll on their first turn.

If we have just joined the game, we must enroll before taking a turn.

Taking a turn and synchronizing with the `Barrier` stays the same.

Here we decide whether to carry on playing or sit out of the game and spectate for a bit.

If we have just left the game we must resign from the `Barrier`.

This just orders the printout more clearly.

Spectate for 20-40 seconds.

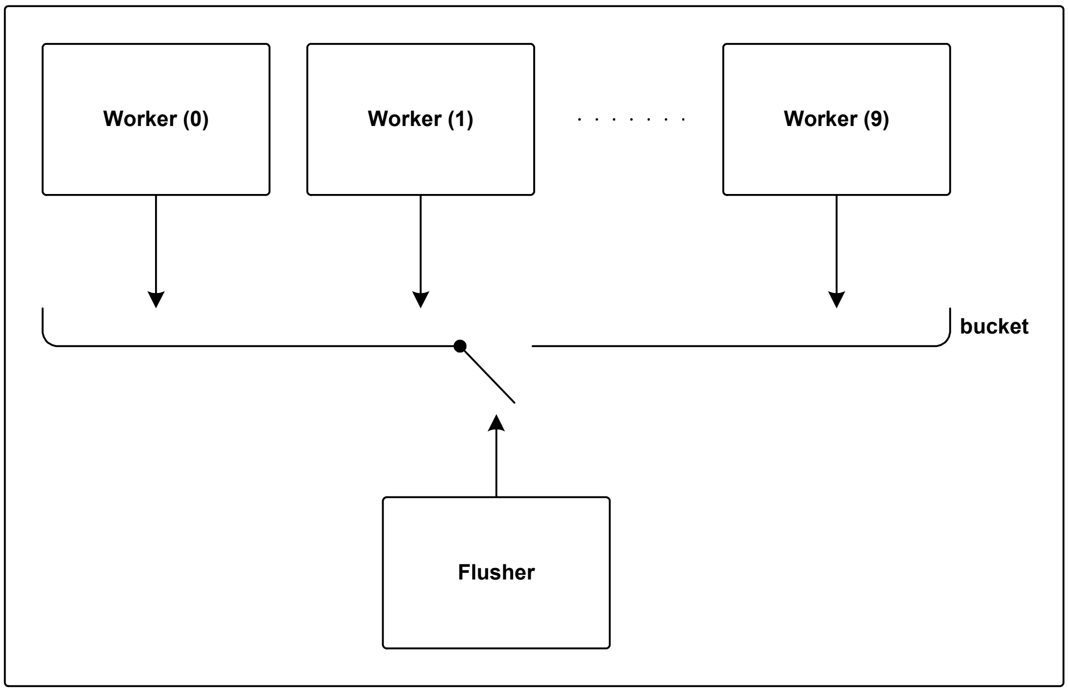
Here we decide whether to join the game again or carry on spectating.

We also have to make a minor change in the main class so that the `Barrier` is not associated with any processes when it is created - we do this by just using the `Barrier()` constructor instead of the `Barrier(int nEnrolled)` constructor.

Buckets

A `Bucket` is somewhere to `fallInto` when a process needs somewhere to park itself. There is no limit on the number of processes that can `fallInto` a `Bucket` - and all are blocked when they do. Release happens when a process chooses to flush that bucket, although obviously this must be a process which has not fallen into the bucket. When the bucket is flushed, any processes in it are rescheduled for execution.

The following example demonstrates the simple use of a bucket:



Each Worker does a shift which takes a random amount of time (up to 10 seconds) and then falls into the bucket. The Flusher flushes the bucket every five seconds, thereby triggering the beginning of a shift. This allows all the workers to keep accurately to the timeslices set by the Flusher process.

Here's the main class:

```
import com.quickstone.jcsp.lang.*;

public class BucketExample1 {

  public static void main (String[] args) {

    final int nWorkers = 10;
    final int second = 1000;
    final int flushInterval = 5*second;
    final int maxWork = 10*second;
    final long seed = new CSTimer().read ();

    final Bucket bucket = new Bucket ();
    final Flusher flusher = new Flusher (flushInterval, bucket);
    final Worker[] workers = new Worker[nWorkers];

    for (int i = 0; i < workers.length; i++) {
      workers[i] = new Worker (i, i + seed, maxWork, bucket);
    }

    System.out.println ("*** Flusher: interval = " +
                        flushInterval + " milliseconds");
    new Parallel (
      new CSProcess[] {
        flusher,
        new Parallel (workers)
      }
    ).run ();
  }
}
```

And here's the Worker class:

```
import com.quickstone.jcsp.lang.*;
import java.util.*;

public class Worker implements CSProcess {

  private final int id;
  private final long seed;
  private final int maxWork;
  private final Bucket bucket;

  public Worker (int id, long seed, int maxWork,
                 Bucket bucket) {
    this.id = id;
    this.seed = seed;
    this.maxWork = maxWork;
    this.bucket = bucket;
  }

  public void run () {

    final Random random = new Random (seed);

    final CSTimer tim = new CSTimer ();
    final int second = 1000;

    final String working = "\t... Worker " + id +
                           " working ...";
    final String falling = "\t\t\t... Worker " +
                           id + " falling ...";
    final String flushed = "\t\t\t\t\t... Worker " +
                           id + " flushed ...";
    while (true) {
      System.out.println (working);
      tim.sleep (random.nextInt (maxWork));
      System.out.println (falling);
      bucket.fallInto ();
      System.out.println (flushed);
    }
  }
}
```

As with a Barrier, each process using the Bucket must be able to get a handle on it, which is why we pass it in as a constructor argument.

Here we use a CSTimer to simulate some work again.

When we're done we fall into the bucket. This method blocks until another process flushes the bucket.

At which point we report this and loop back round to do some more work.

The Flusher class is very simple - it simply flushes the Bucket at regular intervals and reports how many workers were in the Bucket when it was flushed. It gets this information from the Bucket's flush method which not only flushes the Bucket but also returns an `int` representing the number of processes flushed.

```
import com.quickstone.jcsp.lang.*;
import java.util.*;

public class Flusher implements CSProcess {

  private final int interval;
  private final Bucket bucket;

  public Flusher (int interval, Bucket bucket) {
    this.interval = interval;
    this.bucket = bucket;
  }

  public void run () {

    final CSTimer tim = new CSTimer ();
    long timeout = tim.read () + interval;

    while (true) {
      tim.after (timeout);
      System.out.println ("*** Flusher: about to flush ...");
      final int n = bucket.flush ();
      System.out.println ("*** Flusher: number flushed = " + n);
      timeout += interval;
    }
  }
}
```

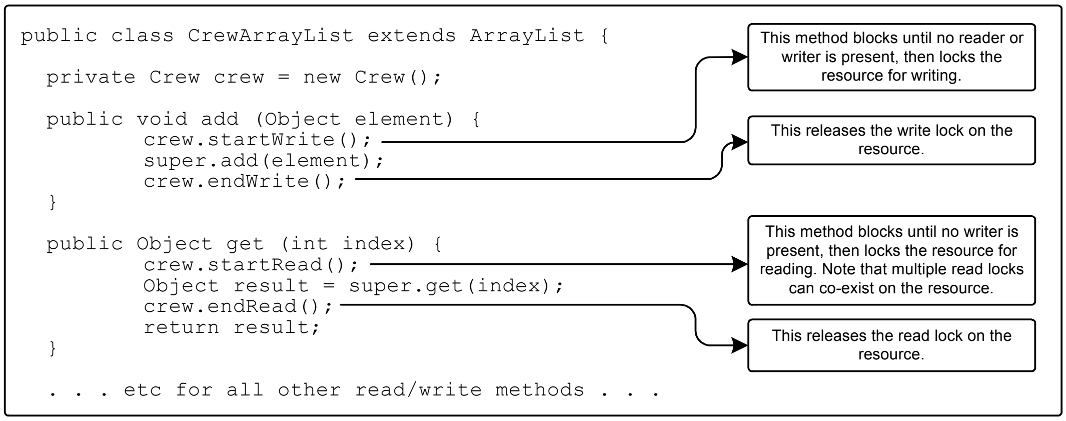
Note the use of the `after (long milliseconds)` method on the `CSTimer` object. Unlike the `sleep (long milliseconds)` method which blocks for a specified number of milliseconds starting from when the method is called, `after` blocks until the *absolute* time specified has been reached. If the absolute time has already passed the method will not block at all. This prevents our timing slipping when things get busy.

CREW (Concurrent Read Exclusive Write) Locks

Where possible, data should be kept wrapped up in a process and accessed via a channel interface - this will always be safe. However, this also serialises access to the data so that it can be used by only one process at a time, regardless of whether that usage is read-only or read-write. This access pattern is an example of Exclusive Read Exclusive Write (EREW).

Parallel write and read/write operations on a single resource are always dangerous - for example, a reader of a resource may observe partially updated (and, hence, invalid) data because of the activities of a concurrent writer. Or two writers may interfere with each other's updating to leave a resource in an invalid state (as well as confusing themselves). However, parallel read operations on a resource do not lead to race hazards, so long as no writer is present, and many applications need to be able to exploit this. This principle is known as Concurrent Read Exclusive Write (CREW).

The Crew class allows us to easily implement this functionality into any resource. Perhaps the easiest and safest way to do so is to extend the resource class to contain its own private Crew object and override any write and read methods to check with the Crew before proceeding, for example:



```
public class CrewArrayList extends ArrayList {

  private Crew crew = new Crew();

  public void add (Object element) {
          crew.startWrite();
          super.add(element);
          crew.endWrite();
  }

  public Object get (int index) {
          crew.startRead();
          Object result = super.get(index);
          crew.endRead();
          return result;
  }

  . . . etc for all other read/write methods . . .
```

This method blocks until no reader or writer is present, then locks the resource for writing.

This releases the write lock on the resource.

This method blocks until no writer is present, then locks the resource for reading. Note that multiple read locks can co-exist on the resource.

This releases the read lock on the resource.

This might get somewhat tedious for a class like ArrayList which has quite a few read/write methods, but you only have to do it once. Of course, one can also build resource classes from scratch with the Crew lock built-in. Alternatively, the Crew object can be passed to the processes using it along with the resource object and then used like this:

```
crew.startRead();
Object o = resource.read();
crew.endRead();

...

crew.startWrite();
resource.write(o);
crew.endWrite();
```

Note: If either the read or write methods of the resource class might throw a runtime exception it is prudent to use a try... catch... finally clause to prevent the resource being locked indefinitely, e.g:

```
try {
  crew.startWrite();
  resource.write(o);
}
catch (Exception e) {
  ...report exception...
}
finally {
  crew.endWrite();
}
```

Now the lock is always released, whether the write method terminates normally or with an exception.

CALL Channels

CALL Channels are primarily a device which allow a more object-oriented style of programming to be used with JCSP. With regular JCSP channels, the operations equivalent to calling a method on another process are rather long-winded. First, the process which wants to call the method (the client) has to write some data (its request) to the process which can perform the method (the server). Then the server has to read that request, act accordingly, and perhaps respond to the request by returning some data with a write on another channel. The client then has to read that data. All in all it requires 2 channels, 2 write operations and 2 read operations. CALL channels let you do all this with 1 channel, 1 method invocation by the client, and 1 accept operation by the server. They also provide a more object-oriented syntax.

However there's no such thing as a free lunch, and setting up a CALL Channel requires a bit more effort than a regular channel. Say we intend to create a process, `counter`, which we wish to call methods on. It will contain three methods: `increment()`, `set(int i)` and `get()`. In order to create a CALL Channel we can plug into this process we first create an Interface listing the methods we want to be able to call via the channel:

```
interface CounterCallChannelInterface {
  public void increment();
  public void set(int i);
  public int get();
}
```

All the client processes need to see is this interface, so they can be completely unconcerned with the mechanics of the CALL Channel and simply call methods in the usual way - the only difference from regular object-oriented programming is that rather than giving the client process a reference to `counter` to call methods on directly, we give it a reference to the CALL Channel:

```
if (counterCallChannel.get() > 100) {
  counterCallChannel.set(0);
}
else {
  counterCallChannel.increment();
}
```

This brings us on to the channel itself. Each process that requires a CALL channel must implement its own unique CALL channel class by extending one of the CALL channel classes and implementing the interface we created earlier:

```
import com.quickstone.jcsp.lang.*;

public class One2OneCounterCallChannel extends
One2OneCallChannel implements CounterCallChannelInterface {

  public void increment() {
    join ();
    ((CounterCallChannelInterface)server).increment();
    fork ();
  }

  public int get() {
    join ();
    int result =
      ((CounterCallChannelInterface)server).get();
    fork ();
    return result;
  }

  public void set(int i) {
    join ();
    ((CounterCallChannelInterface)server).set(i);
    fork ();
  }
}
```

This instructs the CALL Channel to wait for the server to `accept` the call

When the server process accepts, it passes a reference to itself to the CALL Channel and this is stored in the protected `server` field. We have to cast to `CounterCallChannel Interface` in order to call the required method.

This tells the CALL Channel that the method call is finished.

This example extends `One2OneCallChannel` so the resulting channel can only be used in a One2One context. If we want to plug CALL Channels of different types into the server process we must also create separate classes extending `One2AnyCallChannel`, `Any2AnyCallChannel` or `Any2OneCallChannel` as appropriate. Fortunately, this would be a simple copy and paste job.

Note that a client process remains blind to the type of channel it is calling because it only sees the interface not the channel. This means that a client can be connected to its server(s) via a One2One, Any2One, One2Any or Any2Any channel without any change to the client's coding.

Finally, we write the `Counter` class itself, which provides the 'real' implementation of `CounterCallChannelInterface`:

```
import com.quickstone.jcsp.lang.*;

public class Counter implements
       CounterCallChannelInterface, CSProcess {

  private ChannelAccept callChanIn;

  public Counter (One2OneCounterChannel callChanIn) {
    this.callChanIn = callChanIn;
  }

  private int count = 0;

  public void increment() {
    count++;
  }
  public void set(int i) {
    count = i;
  }
  public int get() {
    return count;
  }

  public void run() {
    while (true) {
      callChanIn.accept(this);
    }
  }
}
```

ChannelAccept is the CALL Channel equivalent of the ChannelInput interface.

Here's the 'real' implementation of `CounterCallChannelInterface`

Now all our run method has to do is sit back and accept the calls (passing itself back to the Call Channel as a parameter). The actual calling of the methods is handled by the Call Channel class.

Now we have all the elements we need to use a CALL Channel to call methods on our `Counter` process. The main class below brings them all together in a little example program:

```
import com.quickstone.jcsp.lang.*;

public class CounterCallChannelExample {

  public static void main(String[] args) {

    final One2OneCounterCallChannel countChan = new One2OneCounterCallChannel();

    new Parallel(new CSProcess[] {
      new Counter(countChan),
      new CSProcess() {
        public void run() {
          while (true) {
            int i = countChan.get();
            System.out.println("count: " + i);
            if (i > 99) {
              countChan.set(0);
            }
            else {
              countChan.increment();
            }
          }
        }
      }
    }).run();
  }
}
```

As you can see, CALL Channels do require quite a bit of code to get going, but if you plan to use them a lot it is quite possible to automate most of this by writing a class which uses reflection to generate the Interface and the CALL channel and server classes from an existing java class. In this case it is probably easier to create a separate server class which simply calls an instance of the existing class to implement the methods, rather than modifying the existing class to act as a server.

Tip: Where a process has a large number of methods to be called via a CALL Channel it may be desirable to split the methods across several CALL Channel classes. These channels can then be ALTed over and/or connected to different groups of client processes in order to provide greater control over the servicing of the channels. The JCSP Server Builder, available as part of the JCSP Toolset from Quickstone Technologies, allows the graphical creation of complex CALL Channel servers from existing Java classes and takes the donkey work out of using CALL Channels by autogenerating most of the code.

An Introduction to JCSP Network Edition

JCSP Network Edition contains all the features of JCSP Base Edition but adds powerful networking capabilities. Network channels have been designed to plug into processes in exactly the same way as regular channels, and implement the same `ChannelInput` and `ChannelOutput` interfaces so process networks can be 're-wired' in no time, whether they run across physical networks or not.

Using Network Channels

The main difference between using regular JCSP channels and using network channels is that you only ever have a handle on one 'end' of a network channel, rather than having a single channel object from which you obtain both the `ChannelInput` and `ChannelOutput` ends. This allows the channel ends to be on separate computers but it also leaves us with a problem - how will the channel ends be able to find each other?

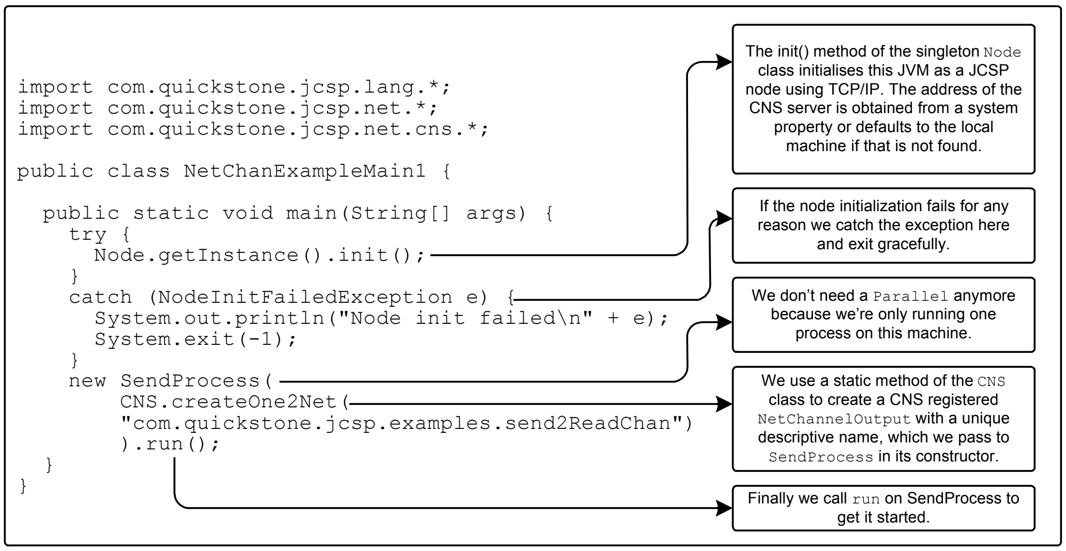
The Channel Name Server

JCSP Network Edition provides several solutions to this problem. The simplest one involves using a Channel Name Server, or CNS, which resolves textual channel names to network addresses. This allows us to simply give our matching channel ends the same unique name and then treat them just like a regular JCSP channel. For convenience, JCSP Network Edition comes with a standalone TCP/IP CNS class (`com.quickstone.jcsp.net.tcpip.TCPIPCNSServer`). However, it is quite possible to run the CNS as part of another process or use the rest of the CNS framework with your own custom CNS class which supports a different network protocol.

For most purposes, it is easiest to run the standalone CNS in a separate JVM on the same machine as another JCSP program - so before we move on to our first example make sure you've started the CNS by running `com.quickstone.jcsp.net.tcpip.TCPIPCNSServer`. This starts up a CNS on the default port which is 7890. The CNS requires no more attention - it will just run happily away in the background and deal with any CNS requests.

For our first example of networked channels using a CNS we'll return to our very first non-networked example - `SendProcess` and `ReadProcess` - but this time the two processes will be running on separate computers. Of course, this means that we need two main classes too - one for each machine.

Here's the main class for the computer running `SendProcess`:



```
import com.quickstone.jcsp.lang.*;
import com.quickstone.jcsp.net.*;
import com.quickstone.jcsp.net.cns.*;

public class NetChanExampleMain1 {

  public static void main(String[] args) {
    try {
      Node.getInstance().init();
    }
    catch (NodeInitFailedException e) {
      System.out.println("Node init failed\n" + e);
      System.exit(-1);
    }
    new SendProcess(
        CNS.createOne2Net(
        "com.quickstone.jcsp.examples.send2ReadChan")
        ).run();
  }
}
```

The init() method of the singleton `Node` class initialises this JVM as a JCSP node using TCP/IP. The address of the CNS server is obtained from a system property or defaults to the local machine if that is not found.

If the node initialization fails for any reason we catch the exception here and exit gracefully.

We don't need a `Parallel` anymore because we're only running one process on this machine.

We use a static method of the `CNS` class to create a CNS registered `NetChannelOutput` with a unique descriptive name, which we pass to `SendProcess` in its constructor.

Finally we call `run` on SendProcess to get it started.

The main class for the computer running `ReadProcess` is almost identical:

```
import com.quickstone.jcsp.lang.*;
import com.quickstone.jcsp.net.*;
import com.quickstone.jcsp.net.cns.*;

public class NetChanExampleMain2 {

  public static void main(String[] args) {
    try {
      Node.getInstance().init();
    }
    catch (NodeInitFailedException e) {
      System.out.println("Node init failed\n" + e);
      System.exit(-1);
    }

    new ReadProcess(
    CNS.createNet2One(
    "com.quickstone.jcsp.examples.send2ReadChan"
    )).run();
  }
}
```

All we need to change is the name of the process we're running and the type of channel end we pass to it. In this case we create a `NetChannelInput`, give it the same name as the `NetChannelOutput` at the other end and the CNS will do the rest.

Initializing JCSP Nodes

One of the first and most important things to notice when looking at the example code above is that each JVM must be initialized as a JCSP `Node` before it can use any of the networking features of JCSP Network Edition.

To initialize a TCP/IP node we get the instance of the singleton `Node` class using its static `getInstance` method and then call `init` on it. If we don't supply any parameters, as in the above example, the `Node` will be initialized using the default `NodeFactory`. This is a static instance of `com.quickstone.jcsp.tcpip.TCPIPNodeFactory` which obtains the CNS address by looking for the java system property `com.quickstone.jcsp.tcpip.DefaultCNSServer`.

This can be set by using the `-D` option when running the program from the command line. Alternatively it can be set within the program using the `System.setProperty(String key, String value)` method before we initialize the node. If the system property cannot be found, the NodeFactory defaults to using the local machine as the CNS address.

So, make sure you let the example programs know where to find the CNS server one way or another before running them, otherwise you will get a `NodeInitFailedException` (unless of course you're running both nodes and the CNS on the same machine, but that would surely defeat the object of using the Network Edition!)

Creating CNS Channels

The CNS class provides static factory methods to create channel ends that are automatically named and registered with the CNS. In the example above we used `CNS.createOne2Net("com.quickstone.jcsp.examples.send2ReadChan")` to create a `NetChannelOutput` which would send data to a `NetChannelInput` called `com.quickstone.jcsp.examples.send2ReadChan`.

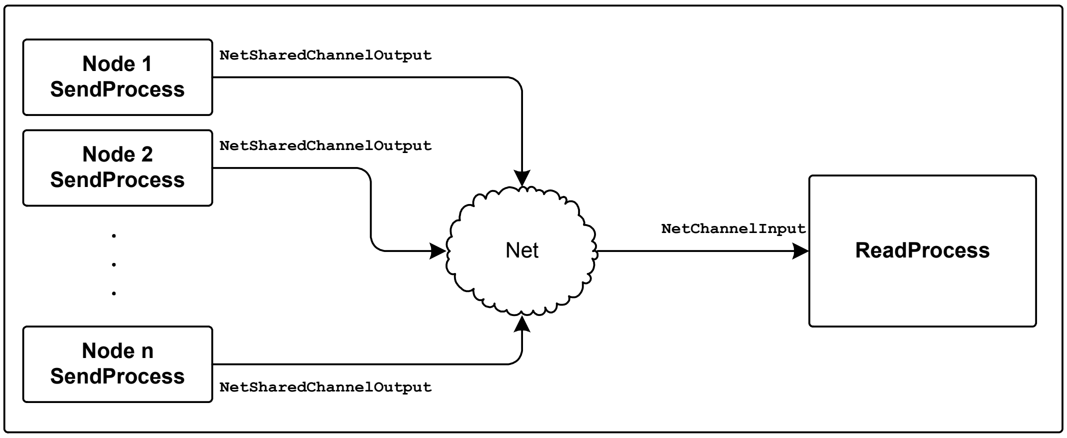
CNS channel names need to be globally unique, so whilst any `String` will be a valid channel name it is recommended that each name begins with the package or class that defines it, e.g:

`com.quickstone.jcsp.examples.testChan` (used throughout package)

`com.quickstone.jcsp.examples.TestClass.testChan` (used in that class)

Networked channel ends can be either One2Net, Net2One, Any2Net or Net2Any and can be plugged together in any combination to create channels equivalent to all the regular JCSP channels.

For example, in order to reproduce the second non-networked example, where multiple `SendProcesses` send to a single `ReadProcess`, we can just change the `NetChanExampleMain1` class to use an Any2Net `NetSharedChannelOutput` instead of a One2Net `NetChannelOutput`. Now we can run the main class on as many different nodes as we like, and they will all use the shared channel to send to the single `ReadProcess` with the matching `NetChannelInput` name.

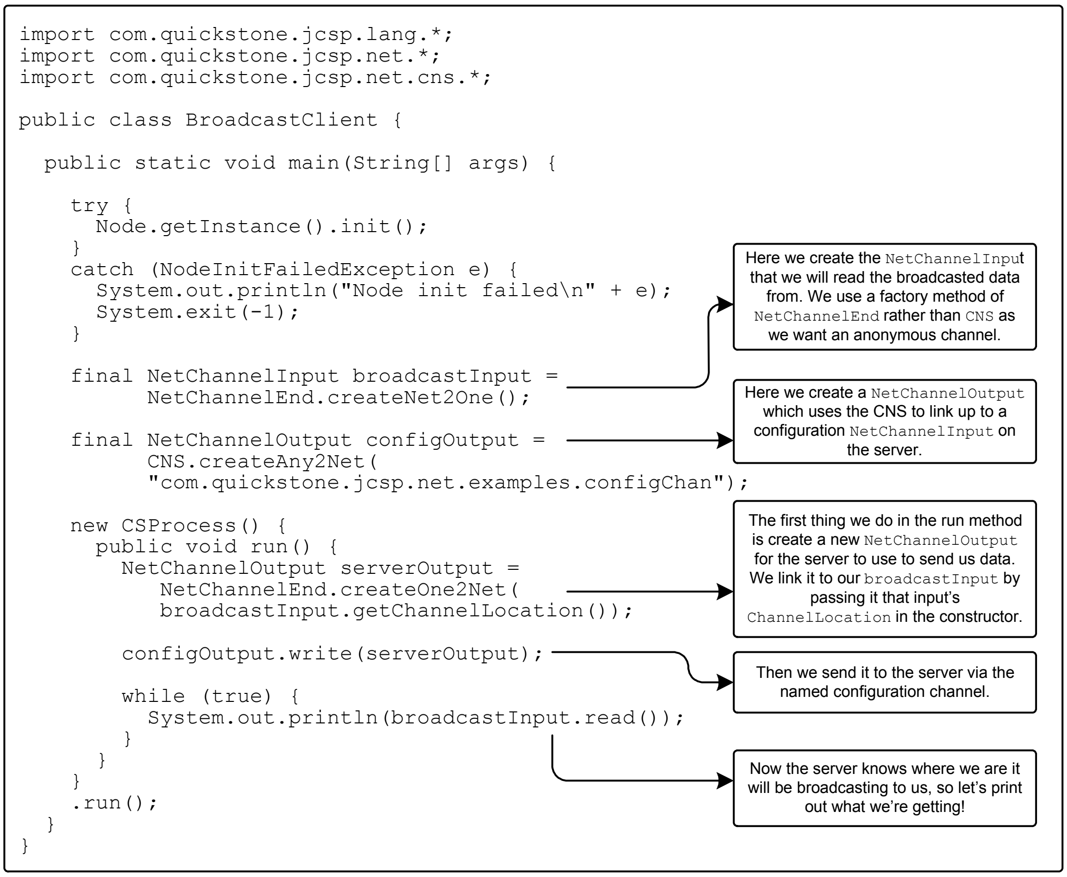


Note: It doesn't matter if we start up a sending node before the reading node because even though the channel name will not be able to be resolved at that point, the CNS keeps a note of it and will resolve it as soon as we start up the reading node and create the matching channel end.

Anonymous Network Channels

Once we have established one channel between two nodes we can start using anonymous network channels. An example application for this would be if you had a server process broadcasting to a large, changing number of client processes. It would be very irritating to have to uniquely name every channel that runs from the server to a client so instead we can create a matched `NetChannelInput` and `NetChannelOutput` on the client (e.g. the equivalent of a regular, two-ended, JCSP channel) and then send the `NetChannelOutput` to the server via a CNS registered configuration channel. The server can then use the `NetChannelOutput` to send data to the client without having to go via the CNS.

The code for the client might look something like this:



```
import com.quickstone.jcsp.lang.*;
import com.quickstone.jcsp.net.*;
import com.quickstone.jcsp.net.cns.*;

public class BroadcastClient {

  public static void main(String[] args) {

    try {
      Node.getInstance().init();
    }
    catch (NodeInitFailedException e) {
      System.out.println("Node init failed\n" + e);
      System.exit(-1);
    }

    final NetChannelInput broadcastInput =
          NetChannelEnd.createNet2One();

    final NetChannelOutput configOutput =
          CNS.createAny2Net(
          "com.quickstone.jcsp.net.examples.configChan");

    new CSProcess() {
      public void run() {
        NetChannelOutput serverOutput =
           NetChannelEnd.createOne2Net(
           broadcastInput.getChannelLocation());

        configOutput.write(serverOutput);

        while (true) {
          System.out.println(broadcastInput.read());
        }
      }
    }
    .run();
  }
}
```

Here we create the `NetChannelInput` that we will read the broadcasted data from. We use a factory method of `NetChannelEnd` rather than `CNS` as we want an anonymous channel.

Here we create a `NetChannelOutput` which uses the CNS to link up to a configuration `NetChannelInput` on the server.

The first thing we do in the run method is create a new `NetChannelOutput` for the server to use to send us data. We link it to our `broadcastInput` by passing it that input's `ChannelLocation` in the constructor.

Then we send it to the server via the named configuration channel.

Now the server knows where we are it will be broadcasting to us, so let's print out what we're getting!

As you can see, creating an anonymous networked channel is a simple matter of using `NetChannelEnd` instead of `CNS` to create your channel ends (linking the ends by passing the input's `NetChannelLocation` to the output end as a parameter when you create it). Then send the appropriate end to the remote process down an existing channel and use the new anonymous channel as normal.

To complete the example, this `BroadcastServer` process will accept new clients by receiving a `NetChannelOutput` on its CNS-registered configuration channel. The `NetChannelOutput` for each client is then added to the server's output array and the client will be able to receive the numbers that the server is broadcasting.

```
import com.quickstone.jcsp.lang.*;
import com.quickstone.jcsp.net.*;
import com.quickstone.jcsp.net.cns.*;
import com.quickstone.jcsp.plugNplay.*;

public class BroadcastServer {

  public static void main(String[] args) {
    try {
      Node.getInstance().init();
    }
    catch (NodeInitFailedException e) {
      System.out.println("Node failed to initialise - terminating\n" + e);
      System.exit(-1);
    }

    One2OneChannel numbers2DeltaChan = Channel.createOne2One();
```

```
            new Parallel(
              new CSProcess[] {
                new DynamicDelta(numbers2DeltaChan.in(),
                    CNS.createNet2One("com.quickstone.jcsp.net.examples.configChan")),
                new Numbers(numbers2DeltaChan.out())})
              .run();
        }
      }
```

Advanced Network Edition Features

JCSP Network Edition provides many more advanced features such as remote classloading over channels and migratable channel ends. Further documentation for advanced users can be found in the API documentation for the relevant classes.