class Supply

Asynchronous data stream with multiple subscribers

```
    class Supply does Awaitable {}
```

A supply is a thread-safe, asynchronous data stream like a Channel , but it can have multiple subscribers (*taps*) that all get the same values flowing through the supply.

It is a thread-safe implementation of the Observer Pattern , and central to supporting reactive programming in Raku.

There are two types of Supplies: `live` and `on demand` . When tapping into a `live` supply, the tap will only see values that are flowing through the supply **after** the tap has been created. Such supplies are normally infinite in nature, such as mouse movements. Closing such a tap does not stop mouse events from occurring, it just means that the values will go by unseen. All tappers see the same flow of values.

A tap on an `on demand` supply will initiate the production of values, and tapping the supply again may result in a new set of values. For example, `Supply.interval` produces a fresh timer with the appropriate interval each time it is tapped. If the tap is closed, the timer simply stops emitting values to that tap.

A `live Supply` is obtained from the Supplier factory method `Supply` . New values are emitted by calling `emit` on the `Supplier` object.

```
my $supplier = Supplier.new;
my $supply = $supplier.Supply;
$supply.tap(-> $v { say "$v" });
$supplier.emit(42); # Will cause the tap to output "42"
```

The live method returns `True` on live supplies. Factory methods such as interval , from-list will return *on demand* supplies.

A live `Supply` that keeps values until tapped the first time can be created with Supplier::Preserving .

Further examples can be found in the concurrency page .

Methods that return Taps

method tap

```
method tap(Supply:D: &emit = -> $ { },
        :&done = -> {},
        :&quit = -> $ex { $ex.throw },
        :&tap = -> $ {} )
```

Creates a new tap (a kind of subscription if you will), in addition to all existing taps. The first positional argument is a piece of code that will be called when a new value becomes available through the `emit` call.

The `&done` callback can be called in a number of cases: if a supply block is being tapped, when a `done` routine is reached; if a supply block is being tapped, it will be automatically triggered if the supply block reaches the end; if the `done` method is called on the parent `Supplier` (in the case of a supply block, if there are multiple Suppliers referenced by `whenever`, they must all have their `done` method invoked for this to trigger the `&done` callback of the tap as the block will then reach its end).

The `&quit` callback is called if the tap is on a supply block which exits with an error. It is also called if the `quit` method is invoked on the parent `Supplier` (in the case of a supply block any one `Supplier` quitting with an uncaught exception will call the `&quit` callback as the block will exit with an error). The error is passed as a parameter to the callback.

The `&tap` callback is called once the Tap object is created, which is passed as a parameter to the callback. The callback is called ahead of `emit` / `done` / `quit`, providing a reliable way to get the `Tap` object. One case where this is useful is when the `Supply` begins emitting values synchronously, since the call to `.tap` won't return the `Tap` object until it is done emitting, preventing it from being stopped if needed.

Method `tap` returns an object of type Tap, on which you can call the `close` method to cancel the subscription.

```
my $s = Supply.from-list(0 .. 5);
my $t = $s.tap(-> $v { say $v }, done => { say "no more ticks" });
```

Produces:

```
0
1
2
3
4
5
no more ticks
```

method act

```
method act(Supply:D: &actor, *%others)
```

Creates a tap on the given supply with the given code. Differently from `tap`, the given code is guaranteed to be executed by only one thread at a time.

Utility methods

method Capture

Defined as:

```
method Capture(Supply:D: --> Capture:D)
```

Equivalent to calling .List.Capture on the invocant.

method Channel

```
method Channel(Supply:D: --> Channel:D)
```

Returns a Channel object that will receive all future values from the supply, and will be `close` d when the Supply is done, and quit (shut down with error) when the supply is quit.

method Promise

```
method Promise(Supply:D: --> Promise:D)
```

Returns a Promise that will be kept when the `Supply` is `done` . If the `Supply` also emits any values, then the `Promise` will be kept with the final value. Otherwise, it will be kept with `Nil` . If the `Supply` ends with a `quit` instead of a `done` , then the `Promise` will be broken with that exception.

```
my $supplier = Supplier.new;
my $s = $supplier.Supply;
my $p = $s.Promise;
$p.then(-> $v { say "got $v.result()" });
$supplier.emit('cha');         # not output yet
$supplier.done();              # got cha
```

The `Promise` method is most useful when dealing with supplies that will tend to produce just one value, when only the final value is of interest, or when only completion (successful or not) is relevant.

method live

```
method live(Supply:D: --> Bool:D)
```

Returns `True` if the supply is "live", that is, values are emitted to taps as soon as they arrive. Always returns `True` in the default `Supply` (but for example on the supply returned from `Supply.from-list` it's `False`).

```
say Supplier.new.Supply.live;    # OUTPUT: «True»
```

method schedule-on

```
method schedule-on(Supply:D: Scheduler $scheduler)
```

Runs the emit, done and quit callbacks on the specified scheduler.

This is useful for GUI toolkits that require certain actions to be run from the GUI thread.

Methods that wait until the supply is done

method wait

```
method wait(Supply:D:)
```

Taps the `Supply` it is called on, and blocks execution until the either the supply is `done` (in which case it evaluates to the final value that was emitted on the `Supply`, or `Nil` if not value was emitted) or `quit` (in which case it will throw the exception that was passed to `quit`).

```
my $s = Supplier.new;
start {
  sleep 1;
  say "One second: running.";
  sleep 1;
  $s.emit(42);
  $s.done;
}
$s.Supply.wait;
say "Two seconds: done";
```

method list

```
multi method list(Supply:D:)
```

Taps the `Supply` it is called on, and returns a lazy list that will be reified as the `Supply` emits values. The list will be terminated once the `Supply` is `done`. If the `Supply` `quit`s, then an exception will be thrown once that point in the lazy list is reached.

method Seq

```
method Seq(Supply:D:)
```

Returns a `Seq` with an iterator containing the values that the `Supply` contains.

method grab

```
method grab(Supply:D: &when-done --> Supply:D)
```

Taps the `Supply` it is called on. When it is `done`, calls `&when-done` and then emits the list of values that it returns on the result `Supply`. If the original `Supply` `quit`s, then the exception is immediately conveyed on the return `Supply`.

```
my $s = Supply.from-list(4, 10, 3, 2);
my $t = $s.grab(&sum);
$t.tap(&say);           # OUTPUT: «19»
```

method reverse

```
method reverse(Supply:D: --> Supply:D)
```

Taps the `Supply` it is called on. Once that `Supply` emits `done`, all of the values it emitted will be emitted on the returned `Supply` in reverse order. If the original `Supply` `quit`s, then the exception is immediately conveyed on the return `Supply`.

```
my $s = Supply.from-list(1, 2, 3);
my $t = $s.reverse;
$t.tap(&say);           # OUTPUT: «321»
```

method sort

```
method sort(Supply:D: &custom-routine-to-use? --> Supply:D)
```

Taps the `Supply` it is called on. Once that `Supply` emits `done`, all of the values that it emitted will be sorted, and the results emitted on the returned `Supply` in the sorted order. Optionally accepts a comparator Block. If the original `Supply` `quit`s, then the exception is immediately conveyed on the return `Supply`.

```
my $s = Supply.from-list(4, 10, 3, 2);
my $t = $s.sort();
$t.tap(&say);           # OUTPUT: «23410»
```

method collate

```
method collate(Supply:D:)
```

Taps the `Supply` it is called on. Once that `Supply` emits `done`, all of the values that it emitted will be sorted taking into account Unicode grapheme characteristics. A new `Supply` is returned with the sorted values emitted. See Any.collate for more details on the collated sort.

```
my $s = Supply.from-list(<ä a o ö>);
my $t = $s.collate();
$t.tap(&say);           # OUTPUT: «aäoö»
```

method reduce

Defined as:

```
method reduce(Supply:D: &with --> Supply:D)
```

Creates a "reducing" supply, which will emit a single value with the same semantics as List.reduce.

```
my $supply = Supply.from-list(1..5).reduce({$^a + $^b});
$supply.tap(-> $v { say "$v" }); # OUTPUT: «15»
```

Methods that return another Supply

method from-list

```
method from-list(Supply:U: +@values --> Supply:D)
```

Creates an on-demand supply from the values passed to this method.

```
my $s = Supply.from-list(1, 2, 3);
$s.tap(&say);           # OUTPUT: «123»
```

method share

```
method share(Supply:D: --> Supply:D)
```

Creates a live supply from an on-demand supply, thus making it possible to share the values of the on-demand supply on multiple taps, instead of each tap seeing its own copy of all values from the on-demand supply.

```
# this says in turn: "first 1" "first 2" "second 2" "first 3" "second 3"
my $s = Supply.interval(1).share;
$s.tap: { "first $_".say };
sleep 1.1;
$s.tap: { "second $_".say };
sleep 2
```

method flat

```
method flat(Supply:D: --> Supply:D)
```

Creates a supply on which all of the values seen in the given supply are flattened before being emitted again.

method do

```
method do(Supply:D: &do --> Supply:D)
```

Creates a supply to which all values seen in the given supply, are emitted again. The given code, executed for its side-effects only, is guaranteed to be only executed by one thread at a time.

method on-close

```
method on-close(Supply:D: &on-close --> Supply:D)
```

Returns a new `Supply` which will run `&on-close` whenever a Tap of that `Supply` is closed. This includes if further operations are chained on to the `Supply`. (for example, `$supply.on-close(&on-close).map(*.uc)`). When using a `react` or `supply` block, using the CLOSE phaser is usually a better choice.

```
my $s = Supplier.new;
my $tap = $s.Supply.on-close({ say "Tap closed" }).tap(
    -> $v { say "the value is $v" },
    done    => { say "Supply is done" },
    quit    => -> $ex { say "Supply finished with error $ex" },
);

$s.emit('Raku');
$tap.close;        # OUTPUT: «Tap closed»
```

method interval

```
method interval(Supply:U: $interval, $delay = 0, :$scheduler = $*SCHEDULER --> Supply:D)
```

Creates a supply that emits a value every `$interval` seconds, starting `$delay` seconds from the call. The emitted value is an integer, starting from 0, and is incremented by one for each value emitted.

Implementations may treat too-small and negative values as lowest resolution they support, possibly warning in such situations; e.g. treating `0.0001` as `0.001` . For 6.d language version, the minimal value specified is `0.001` .

method grep

```
method grep(Supply:D: Mu $test --> Supply:D)
```

Creates a new supply that only emits those values from the original supply that smartmatch against `$test` .

```
my $supplier = Supplier.new;
my $all      = $supplier.Supply;
my $ints     = $all.grep(Int);
$ints.tap(&say);
$supplier.emit($_) for 1, 'a string', 3.14159;   # prints only 1
```

method map

```
method map(Supply:D: &mapper --> Supply:D)
```

Returns a new supply that maps each value of the given supply through `&mapper` and emits it to the new supply.

```
my $supplier = Supplier.new;
my $all      = $supplier.Supply;
my $double   = $all.map(-> $value { $value * 2 });
$double.tap(&say);
$supplier.emit(4);           # OUTPUT: «8»
```

method batch

```
method batch(Supply:D: :$elems, :$seconds --> Supply:D)
```

Creates a new supply that batches the values of the given supply by either the number of elements in the batch (using `:elems`) or a duration (using `:seconds`) or both. Any remaining values are emitted in a final batch when the supply is done.

Note : Since Rakudo version 2020.12, the `:seconds` parameter has a millisecond granularity: for example a 1 millisecond duration could be specified as `:seconds(0.001)` . Before Rakudo version 2020.12, the `:seconds` parameter had a second granularity.

method elems

```
method elems(Supply:D: $seconds? --> Supply:D)
```

Creates a new supply in which changes to the number of values seen are emitted. It optionally also takes an interval (in seconds) if you only want to be updated every so many seconds.

method head

Defined as:

```
multi method head(Supply:D:)
multi method head(Supply:D: Callable:D $limit)
multi method head(Supply:D: \limit)
```

Creates a "head" supply with the same semantics as List.head .

```
my $s = Supply.from-list(4, 10, 3, 2);
my $hs = $s.head(2);
$hs.tap(&say);           # OUTPUT: «410»
```

Since release 2020.07, A `WhateverCode` can be used also, again with the same semantics as `List.head`

```
my $s = Supply.from-list(4, 10, 3, 2, 1);
my $hs = $s.head( * - 2);
$hs.tap(&say);           # OUTPUT: «4103»
```

method tail

```
multi method tail(Supply:D:)
multi method tail(Supply:D: Callable:D $limit)
multi method tail(Supply:D: \limit)
```

Creates a "tail" supply with the same semantics as List.tail .

```
my $s = Supply.from-list(4, 10, 3, 2);
my $ts = $s.tail(2);
$ts.tap(&say);           # OUTPUT: «32»
```

You can call `.tail` with `Whatever` or `Inf` ; which will return a new supply equivalent to the initial one. Calling it with a `WhateverCode` will be equivalent to skipping until that number.

```
my $s = Supply.from-list(4, 10, 3, 2);
my $ts = $s.tail( * - 2 );
$ts.tap(&say);           # OUTPUT: «32»
```

This feature is only available from the 2020.07 release of Raku.

method first

```
method first(Supply:D: :$end, |c)
```

This method creates a supply of the first element, or the last element if the optional named parameter `:end` is truthy, from a supply created by calling the `grep` method on the invocant, with any remaining arguments as parameters. If there is no remaining argument, this method is equivalent to calling on the invocant, without parameter, the `head` or the `tail` method, according to named parameter `:end` .

```
my $rand = supply { emit (rand × 100).floor for ^∞ };
my $first-prime = $rand.first: &is-prime;
# output the first prime from the endless random number supply $rand,
# then the $first-prime supply reaches its end
$first-prime.tap: &say;
```

method split

```
multi method split(Supply:D: \delimiter)
multi method split(Supply:D: \delimiter, \limit)
```

This method creates a supply of the values returned by the `Str.split` method called on the string collected from the invocant. See Str.split

for details on the `\delimiter` argument as well as available extra named parameters. The created supply can be limited with the `\limit` argument, see .head .

```
my $words = Supply.from-list(<Hello World From Raku!>);
my $s = $words.split(/ <?upper> /, 2, :skip-empty);
$s.tap(&say); # OUTPUT: «HelloWorld»
```

method rotate

```
method rotate(Supply:D: $rotate = 1)
```

Creates a supply with elements rotated to the left when `$rotate` is positive or to the right otherwise, in which case the invocant is tapped on before a new supply is returned.

```
my $supply = Supply.from-list(<a b c d e>).rotate(2);
$supply.tap(&say); # OUTPUT: «cdeab»
```

Note : Available since Rakudo 2020.06.

method rotor

```
method rotor(Supply:D: @cycle --> Supply:D)
```

Creates a "rotoring" supply with the same semantics as List.rotor .

method delayed

```
method delayed(Supply:D: $seconds, :$scheduler = $*SCHEDULER --> Supply:D)
```

Creates a new supply in which all values flowing through the given supply are emitted, but with the given delay in seconds.

method throttle

Defined as

```
multi method throttle(Supply:D:
      Int()  $elems,
      Real() $seconds,
      Real() $delay  = 0,
      :$scheduler    = $*SCHEDULER,
      :$control,
      :$status,
      :$bleed,
      :$vent-at,
    )
```

```
multi method throttle(Supply:D:
      Int()  $elems,
      Callable:D $process,
      Real() $delay = 0,
      :$scheduler   = $*SCHEDULER,
      :$control,
      :$status,
      :$bleed,
      :$vent-at,
    )
```

Arguments to `.throttle` are defined as follows:

Argument	Meaning
$limit,	values / time or simultaneous processing
$seconds or $process	time-unit / code to process simultaneously
$delay = 0,	initial delay before starting, in seconds
:$control,	supply to emit control messages on (optional)
:$status,	supply to tap status messages from (optional)
:$bleed,	supply to bleed messages to (optional)
:$vent-at,	bleed when so many buffered (optional)
:$scheduler,	scheduler to use, default $*SCHEDULER

This method produces a `Supply` from a given one, but makes sure the number of messages passed through is limited.

It has two modes of operation: per time-unit or by maximum number of executions of a block of code: this is determined by the type of the second positional parameter.

The first positional parameter specifies the limit that should be applied.

If the second positional parameter is a `Callable`, then the limit indicates the maximum number of parallel processes executing the Callable, which is given the value that was received. The emitted values in this case will be the `Promise`s that were obtained from `start`ing the Callable.

If the second positional parameter is a real number, it is interpreted as the time-unit (in seconds). If you specify **.1** as the value, then it makes sure you don't exceed the limit for every tenth of a second.

If the limit is exceeded, then incoming messages are buffered until there is room to pass on / execute the Callable again.

The third positional parameter is optional: it indicates the number of seconds the throttle will wait before passing on any values.

The `:control` named parameter optionally specifies a Supply that you can use to control the throttle while it is in operation. Messages that can be sent, are strings in the form of "key:value". Please see below for the types of messages that you can send to control the throttle.

The `:status` named parameter optionally specifies a Supply that will receive any status messages. If specified, it will at least send one status message after the original Supply is exhausted. See status message below.

The `:bleed` named parameter optionally specifies a Supply that will receive any values that were either explicitly bled (with the **bleed** control message), or automatically bled (if there's a **vent-at** active).

The `:vent-at` named parameter indicates the number of values that may be buffered before any additional value will be routed to the `:bleed` Supply. Defaults to 0 if not specified (causing no automatic bleeding to happen). Only makes sense if a `:bleed` Supply has also been specified.

The `:scheduler` named parameter indicates the scheduler to be used. Defaults to `$*SCHEDULER`.

control messages

These messages can be sent to the `:control` Supply. A control message consists of a string of the form "key: value", e.g. "limit: 4".

- limit

Change the number of messages (as initially given in the first positional) to the value given.

- bleed

Route the given number of buffered messages to the `:bleed` Supply.

- vent-at

Change the maximum number of buffered values before automatic bleeding takes place. If the value is lower than before, will cause immediate rerouting of buffered values to match the new maximum.

- status

Send a status message to the `:status` Supply with the given id.

status message

The status return message is a hash with the following keys:

- allowed

The current number of messages / callables that is still allowed to be passed / executed.

- bled

The number of messages routed to the `:bleed` Supply.

- buffered

The number of messages currently buffered because of overflow.

- emitted

The number of messages emitted (passed through).

- id

The id of this status message (a monotonically increasing number). Handy if you want to log status messages.

- limit

The current limit that is being applied.

- vent-at

The maximum number of messages that may be buffered before they're automatically re-routed to the `:bleed` Supply.

Examples

Have a simple piece of code announce when it starts running asynchronously, wait a random amount of time, then announce when it is done. Do this 6 times, but don't let more than 3 of them run simultaneously.

```
my $s = Supply.from-list(^6);  # set up supply
my $t = $s.throttle: 3,        # only allow 3 at a time
{                              # code block to run
    say "running $_";          # announce we've started
    sleep rand;                # wait some random time
    say "done $_"              # announce we're done
}                              # don't need ; because } at end of line
$t.wait;                       # wait for the supply to be done
```

and the result of one run will be:

```
running 0
running 1
running 2
done 2
running 3
done 1
running 4
done 4
running 5
done 0
done 3
done 5
```

method stable

```
method stable(Supply:D: $time, :$scheduler = $*SCHEDULER --> Supply:D)
```

Creates a new supply that only passes on a value flowing through the given supply if it wasn't superseded by another value in the given `$time` (in seconds). Optionally uses another scheduler than the default scheduler, using the `:scheduler` parameter.

To clarify the above, if, during the timeout `$time`, additional values are emitted to the `Supplier` all but the last one will be thrown away. Each time an additional value is emitted to the `Supplier`, during the timeout, `$time` is reset.

This method can be quite useful when handling UI input, where it is not desired to perform an operation until the user has stopped typing for a while rather than on every keystroke.

```
my $supplier = Supplier.new;
my $supply1 = $supplier.Supply;
$supply1.tap(-> $v { say "Supply1 got: $v" });
$supplier.emit(42);

my Supply $supply2 = $supply1.stable(5);
$supply2.tap(-> $v { say "Supply2 got: $v" });
sleep(3);
$supplier.emit(43);  # will not be seen by $supply2 but will reset $time
$supplier.emit(44);
sleep(10);
# OUTPUT: «Supply1 got: 42Supply1 got: 43Supply1 got: 44Supply2 got: 44»
```

As can be seen above, `$supply1` received all values emitted to the `Supplier` while `$supply2` only received one value. The 43 was thrown away because it was followed by another 'last' value 44 which was retained and sent to `$supply2` after approximately eight seconds, this due to the fact that the timeout `$time` was reset after three seconds.

method produce

```
method produce(Supply:D: &with --> Supply:D)
```

Creates a "producing" supply with the same semantics as List.produce .

```
my $supply = Supply.from-list(1..5).produce({$^a + $^b});
$supply.tap(-> $v { say "$v" }); # OUTPUT: «1361015»
```

method lines

```
method lines(Supply:D: :$chomp = True --> Supply:D)
```

Creates a supply that will emit the characters coming in line by line from a supply that's usually created by some asynchronous I/O operation. The optional `:chomp` parameter indicates whether to remove line separators: the default is `True`.

method words

```
method words(Supply:D: --> Supply:D)
```

Creates a supply that will emit the characters coming in word for word from a supply that's usually created by some asynchronous I/O operation.

```
my $s = Supply.from-list("Hello Word!".comb);
my $ws = $s.words;
$ws.tap(&say);           # OUTPUT: «HelloWord!»
```

method unique

```
method unique(Supply:D: :$as, :$with, :$expires --> Supply:D)
```

Creates a supply that only provides unique values, as defined by the optional `:as` and `:with` parameters (same as with unique). The optional `:expires` parameter how long to wait (in seconds) before "resetting" and not considering a value to have been seen, even if it's the same as an old value.

method repeated

```
method repeated(Supply:D: :&as, :&with)
```

Creates a supply that only provides repeated values, as defined by the optional `:as` and `:with` parameters (same as with unique).

```
my $supply = Supply.from-list(<a A B b c b C>).repeated(:as(&lc));
$supply.tap(&say);           # OUTPUT: «AbbC»
```

See repeated for more examples that use its sub form.

Note : Available since version 6.e (Rakudo 2020.01 and later).

method squish

```
method squish(Supply:D: :$as, :$with --> Supply:D)
```

Creates a supply that only provides unique values, as defined by the optional `:as` and `:with` parameters (same as with squish).

method max

```
method max(Supply:D: &custom-routine-to-use = &infix:<cmp> --> Supply:D)
```

Creates a supply that only emits values from the given supply if they are larger than any value seen before. In other words, from a continuously ascending supply it will emit all the values. From a continuously descending supply it will only emit the first value. The optional parameter specifies the comparator, just as with Any.max .

method min

```
method min(Supply:D: &custom-routine-to-use = &infix:<cmp> --> Supply:D)
```

Creates a supply that only emits values from the given supply if they are smaller than any value seen before. In other words, from a continuously descending supply it will emit all the values. From a continuously ascending supply it will only emit the first value. The optional parameter specifies the comparator, just as with Any.min .

method minmax

```
method minmax(Supply:D: &custom-routine-to-use = &infix:<cmp> --> Supply:D)
```

Creates a supply that emits a Range every time a new minimum or maximum values is seen from the given supply. The optional parameter specifies the comparator, just as with Any.minmax .

method skip

```
method skip(Supply:D: Int(Cool) $number = 1 --> Supply:D)
```

Returns a new `Supply` which will emit all values from the given `Supply` except for the first `$number` values, which will be thrown away.

```
my $supplier = Supplier.new;
my $supply = $supplier.Supply;
$supply = $supply.skip(3);
$supply.tap({ say $_ });
$supplier.emit($_) for 1..10; # OUTPUT: «45678910»
```

method start

```
method start(Supply:D: &startee --> Supply:D)
```

Creates a supply of supplies. For each value in the original supply, the code object is scheduled on another thread, and returns a supply either of a single value (if the code succeeds), or one that quits without a value (if the code fails).

This is useful for asynchronously starting work that you don't block on.

Use `migrate` to join the values into a single supply again.

method migrate

```
method migrate(Supply:D: --> Supply:D)
```

Takes a `Supply` which itself has values that are of type `Supply` as input. Each time the outer `Supply` emits a new `Supply`, this will be tapped and its values emitted. Any previously tapped `Supply` will be closed. This is useful for migrating between different data sources, and only paying attention to the latest one.

For example, imagine an application where the user can switch between different stocks. When they switch to a new one, a connection is established to a web socket to get the latest values, and any previous connection should be closed. Each stream of values coming over the web socket would be represented as a Supply, which themselves are emitted into a Supply of latest data sources to watch. The `migrate` method could be used to flatten this supply of supplies into a single Supply of the current values that the user cares about.

Here is a simple simulation of such a program:

```
my Supplier $stock-sources .= new;

sub watch-stock($symbol) {
    $stock-sources.emit: supply {
        say "Starting to watch $symbol";
        whenever Supply.interval(1) {
            emit "$symbol: 111." ~ 99.rand.Int;
        }
        CLOSE say "Lost interest in $symbol";
    }
}

$stock-sources.Supply.migrate.tap: *.say;

watch-stock('GOOG');
sleep 3;
watch-stock('AAPL');
sleep 3;
```

Which produces output like:

```
Starting to watch GOOG
GOOG: 111.67
GOOG: 111.20
GOOG: 111.37
Lost interest in GOOG
Starting to watch AAPL
AAPL: 111.55
```

```
AAPL:111.55AAPL:111.55AAPL:111.6
```

Methods that combine supplies

method merge

```
method merge(Supply @*supplies --> Supply:D)
```

Creates a supply to which any value seen from the given supplies, is emitted. The resulting supply is done only when all given supplies are done. Can also be called as a class method.

method zip

Defined as:

```
method zip(**@s, :&with)
```

Creates a supply that emits combined values as soon as there is a new value seen on **all** of the supplies. By default, Lists are created, but this can be changed by specifying your own combiner with the `:with` parameter. The resulting supply is done as soon as **any** of the given supplies are done. Can also be called as a class method.

This can also be used as a class method; in case it's used as an object method the corresponding supply will be one of the supplies combined (with no special treatment).

method zip-latest

Defined as:

```
method zip-latest(**@s, :&with, :$initial )
```

Creates a supply that emits combined values as soon as there is a new value seen on **any** of the supplies. By default, Lists are created, but this can be changed by specifying your own combiner with the `:with` parameter. The optional :initial parameter can be used to indicate the initial state of the combined values. By default, all supplies have to have at least one value emitted on them before the first combined values is emitted on the resulting supply. The resulting supply is done as soon as **any** of the given supplies are done. Can also be called as a class method.

I/O features exposed as supplies

sub signal

```
sub signal(*@signals, :$scheduler = $*SCHEDULER)
```

Creates a supply for the Signal enums (such as SIGINT) specified, and an optional `:scheduler` parameter. Any signals received, will be emitted on the supply. For example:

```
signal(SIGINT).tap( { say "Thank you for your attention"; exit 0 } );
```

would catch Control-C, thank you, and then exit.

To go from a signal number to a Signal , you can do something like this:

```
signal(Signal(2)).tap( -> $sig { say "Received signal: $sig" } );
```

The list of supported signals can be found by checking `Signal::.keys` (as you would any enum). For more details on how enums work see enum .

Note: Rakudo versions up to 2018.05 had a bug due to which numeric values of signals were incorrect on some systems. For example, `Signal(10)` was returning `SIGBUS` even if it was actually `SIGUSR1` on a particular system. That being said, using `signal(SIGUSR1)` was working as expected on all Rakudo versions except 2018.04, 2018.04.1 and 2018.05, where the intended behavior can be achieved by using `signal(SIGBUS)` instead. These issues are resolved in Rakudo releases after 2018.05.

method IO::Notification.watch-path

```
method watch-path($path --> Supply:D)
```

Creates a supply to which the OS will emit values to indicate changes on the filesystem for the given path. Also has a shortcut with the `watch` method on an IO object, like this:

```
IO::Notification.watch-path(".").act( { say "$^file changed" } );
".".IO.watch.act(                     { say "$^file changed" } );   # same
```