Dataflow

Russell Winder

Version 1.0, 2015-10-01

Table of Contents

Dataflow concurrency offers an alternative concurrency model, which is inherently safe and robust.

Introduction

Check out this small example written in **Groovy** using **GPars** to sum results of calculations performed by three concurrently run tasks:

A Simple Sample

```
import static groovyx.gpars.dataflow.Dataflow.task

final def x = new DataflowVariable()
final def y = new DataflowVariable()
final def z = new DataflowVariable()

task {
    z << x.val + y.val
}

task {
    x << 10
}

task {
    y << 5
}

println "Result: ${z.val}"
```

Or the same algorithm rewritten using the *Dataflows* class.

A Dataflows Sample

```
import static groovyx.gpars.dataflow.Dataflow.task

final def df = new Dataflows()

task {
    df.z = df.x + df.y
}

task {
    df.x = 10
}

task {
    df.y = 5
}

println "Result: ${df.z}"
```

We start three logical tasks, which can run in parallel and perform their particular activities. The tasks need to exchange data and they do so using `Dataflow Variables`. Think of `Dataflow Variables` as one-shot channels safely and reliably transferring data from producers to their consumers.

The `Dataflow Variables` have pretty straightforward semantics. When a task needs to read a value from a `DataflowVariable` (through the val property), it will block until the value has been set by another task or thread (using the '<<' operator). Each `Dataflow Variable` can be set **only once** in its lifetime.

Notice that you don't have to bother with ordering and synchronizing the tasks or threads and their access to shared variables. The values are magically transferred among tasks at the right time without your intervention. The data flow seamlessly among tasks / threads without your intervention or care.

Implementation Detail

The three tasks in the example **do not necessarily need to be mapped to three physical threads**. Tasks represent so-called "green" or "logical" threads and can be mapped under the covers to any number of physical threads. The actual mapping depends on the scheduler, but the outcome of dataflow algorithms doesn't depend on the actual scheduling.

Re-binding Is Possible

The *bind* operation of `dataflow variables` silently accepts re-binding to a value, which is equal to an already bound value. We can call the *bindUnique* method to reject equal values on already-bound variables.

Benefits

Here's what you gain by using `Dataflow Concurrency` (by Jonas Bonér):

- No race-conditions
- No live-locks
- Deterministic deadlocks
- Completely deterministic programs
- BEAUTIFUL code.

This doesn't sound bad, does it?

Concepts

Dataflow Programming

Quoting Wikipedia

Operations (in `Dataflow` programs) consist of "black boxes" with inputs and outputs, all of which are always explicitly defined. They run as soon as all of their inputs become valid, as opposed to when the program encounters them. Whereas a traditional program essentially consists of a series of statements saying "do this, now do this", a `dataflow` program is more like a series of workers on an assembly line, who will do their assigned task as soon as the materials arrive.

This is why dataflow languages are inherently parallel: the operations have no hidden state to keep track of, and the operations are all "ready" at the same time.

Principles

With `Dataflow Concurrency`, you can safely share variables across tasks. These variable (in **Groovy** instances of the `DataflowVariable` class) can only be assigned (using the '<<' operator) a value once in their lifetime. The values of the variables, on the other hand, can be read multiple times (in **Groovy** through the `val` property), even before the value has been assigned. In such cases, the reading task is suspended until the value is set by another task. So you can simply write your code for each task sequentially using `Dataflow Variables` and the underlying mechanics will make sure you get all the values you need in a thread-safe manner.

In brief, you generally perform three operations with `Dataflow variables`:

- Create a `dataflow variable`
- Wait for the variable to be bound (read it)
- Bind the variable (write to it)

And these are the three essential rules your programs have to follow:

- When the program encounters an unbound variable it waits for a value.
- It's not possible to change the value of a dataflow variable once it's bound.
- `Dataflow variables` makes it easy to create concurrent stream agents.

Dataflow Queues and Broadcasts

Before you check our samples of **Dataflow Variables**, **Tasks** and **Operators**, you should learn a bit about streams and queues to have a full picture of `Dataflow Concurrency`. Except for `dataflow variables`,

there are also the concepts of *DataflowQueues* and *DataflowBroadcast* that you can leverage in your code.

You may think of them as thread-safe buffers or queues for message transfer among concurrent tasks or threads. Check out a typical producer-consumer demo:

A Producer-Consumer Demo

```
import static groovyx.gpars.dataflow.Dataflow.task

def words = ['Groovy', 'fantastic', 'concurrency', 'fun', 'enjoy', 'safe', 'GPars', '
data', 'flow']
final def buffer = new DataflowQueue()

task {
    for (word in words) {
        buffer << word.toUpperCase()  //add to the buffer
    }
}

task {
    while(true) println buffer.val  //read from the buffer in a loop
}
```

Both *DataflowBroadcasts* and *DataflowQueues* , just like `DataflowVariables` , implement the `DataflowChannel` interface with common methods allowing us to write to them and read values from them.

The ability to treat both types identically through the *DataflowChannel* interface comes in handy once you start using them to wire *tasks* , *operators* or *selectors* together.

DataflowChannels Combine Two Interfaces

The *DataflowChannel* interface combines two interfaces, each serving its purpose:

- DataflowReadChannel holds all the methods necessary for reading values from a channel - **getVal()**, **getValAsync()**, **whenBound()**, etc.
- DataflowWriteChannel holds all the methods necessary for writing values into a channel - **bind()**, '<<'

You may prefer using these dedicated interfaces instead of the general *DataflowChannel* interface, to better express your intended usage.

Please refer to the API doc for more details on the channel interfaces.

Point-to-point Communication

The *DataflowQueue* class can be viewed as a point-to-point (1 to 1, many to 1) communication channel. It allows one or more producers send messages to one reader. If multiple readers read from the same *DataflowQueue* , they will each consume different messages. Or to put it a different way, each message is consumed by exactly one reader. You can easily imagine a simple load-balancing scheme built around a shared *DataflowQueue* with readers being added dynamically when the consumer part of your algorithm needs to scale up. This is also a useful default choice when connecting tasks or operators.

Publish-subscribe Communication

The *DataflowBroadcast* class offers a publish-subscribe (1 to many, many to many) communication model. One or more producers write messages, while all registered readers will receive all the messages. Each message is thus consumed by all readers with a valid subscription at the point when the message is written to the channel. The readers subscribe by calling the *createReadChannel()* method.

A Pub-Sub Sample

```
DataflowWriteChannel broadcastStream = new DataflowBroadcast()
DataflowReadChannel stream1 = broadcastStream.createReadChannel()
DataflowReadChannel stream2 = broadcastStream.createReadChannel()

broadcastStream << 'Message1'
broadcastStream << 'Message2'
broadcastStream << 'Message3'

assert stream1.val == stream2.val
assert stream1.val == stream2.val
assert stream1.val == stream2.val
```

Under the covers, *DataflowBroadcast* uses the *DataflowStream* class to implement the message delivery.

DataflowStream

The *DataflowStream* class represents a deterministic dataflow channel. It's built around the concept of a functional queue and so provides a lock-free thread-safe implementation for message passing. Essentially, you may think of *DataflowStream* mechanisms as a 1-to-many communication channel, since when a reader consumes a messages, other readers will still be able to read the same message. Also, all messages arrive to all readers in the same order. Since the *DataflowStream* is implemented as a functional queue, its API requires users to traverse the values in the stream themselves. On the other

hand, *DataflowStream* offers handy methods for value filtering or transformation together with interesting performance characteristics.

The Semantics of The *DataflowStream* is Different From The *DataflowChannel* Interface

The *DataflowStream* class, unlike the other communication elements, does not implement the *DataflowChannel* interface, since the semantics of its use is different. Use *DataflowStreamReadAdapter* and *DataflowStreamWriteAdapter* classes to wrap instances of the *DataflowChannel* class in a *DataflowReadChannel* or *DataflowWriteChannel* implementations.

A Sample of DataflowStream Usage

```
import groovyx.gpars.dataflow.stream.DataflowStream
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.scheduler.ResizeablePool

/**
 * Demonstrates concurrent implementation of the Sieve of Eratosthenes using dataflow
tasks
 *
 * In principle, the algorithm consists of a concurrently run chained filters,
 * each of which detects whether the current number can be divided by a single prime
number.
 * (generate nums 1, 2, 3, 4, 5, ...) -> (filter by mod 2) -> (filter by mod 3) ->
(filter by mod 5) -> (filter by mod 7) -> (filter by mod 11) -> (caution! Primes falling
out here)
 * The chain is built (grows) on the fly, whenever a new prime is found
 */

/**
 * We need a resizeable thread pool, since tasks consume threads while waiting, blocked
for values from the DataflowQueue.val
 */
group = new DefaultPGroup(new ResizeablePool(true))

final int requestedPrimeNumberCount = 100

/**
 * Generating candidate numbers
 */
final DataflowStream candidates = new DataflowStream()
group.task {
    candidates.generate(2, {it + 1}, {it < 1000})
}
```

```
/**
 * Chain a new filter for a particular prime number to the end of the Sieve
 * @param inChannel The current end channel to consume
 * @param prime The prime number to divide future prime candidates with
 * @return A new channel ending the whole chain
 */
def filter(DataflowStream inChannel, int prime) {
    inChannel.filter { number ->
        group.task {
            number % prime != 0
        }
    }
}

/**
 * Consume Sieve output and add additional filters for all found primes
 */
def currentOutput = candidates
requestedPrimeNumberCount.times {

    int prime = currentOutput.first
    println "Found: $prime"
    currentOutput = filter(currentOutput, prime)
}
```

For convenience and for the ability to use *DataflowStream* objects with other dataflow constructs, like e.g. operators, you can wrap it with *DataflowReadAdapter* for read access or *DataflowWriteAdapter* for write access.

The *DataflowStream* class is designed for single-threaded producers and consumers. If multiple threads are supposed to read or write values to the stream, their access to the stream must be serialized externally or adapters should be used.

DataflowStream Adapters

The *DataflowStream* API as well as the semantics of its use are very different from the one defined by *Dataflow(Read/Write)Channel*. Adapters have to be used in order to allow *DataflowStreams* to work with other dataflow elements. The *DataflowStreamReadAdapter* class will wrap a *DataflowStream* with the necessary methods to read values, while the *DataflowStreamWriteAdapter* class provides write methods around the wrapped *DataflowStream* method.

Thread Safety

It's important to mention that the *DataflowStreamWriteAdapter* is thread safe. It allows multiple threads to add values to the wrapped *DataflowStream* through the adapter. On the other hand, the

DataflowStreamReadAdapter is designed to be used by a single thread.

The *DataflowStreamWriteAdapter* is thread safe

To minimize overhead and stay in-line with *DataflowStream* semantics, the *DataflowStreamReadAdapter* class is not thread-safe and should only be used from within a single thread.

If multiple threads need to read from a *DataflowStream*, they should create their own wrapping of *DataflowStreamReadAdapter* .

Thanks to the adapters, *DataflowStream* can be used for communications between operators or selectors, as these expect *Dataflow(Read/Write)Channels* .

DataflowStreamAdapters Sample

```
import groovyx.gpars.dataflow.DataflowQueue
import groovyx.gpars.dataflow.stream.DataflowStream
import groovyx.gpars.dataflow.stream.DataflowStreamReadAdapter
import groovyx.gpars.dataflow.stream.DataflowStreamWriteAdapter
import static groovyx.gpars.dataflow.Dataflow.selector
import static groovyx.gpars.dataflow.Dataflow.operator

/**
 * Demonstrates the use of DataflowStreamAdapters to allow dataflow operators to use
DataflowStreams
 */

final DataflowStream a = new DataflowStream()
final DataflowStream b = new DataflowStream()
def aw = new DataflowStreamWriteAdapter(a)
def bw = new DataflowStreamWriteAdapter(b)
def ar = new DataflowStreamReadAdapter(a)
def br = new DataflowStreamReadAdapter(b)

def result = new DataflowQueue()

def op1 = operator(ar, bw) {
    bindOutput it
}
def op2 = selector([br], [result]) {
    result << it
}

aw << 1
aw << 2
aw << 3
assert([1, 2, 3] == [result.val, result.val, result.val])
op1.stop()
op2.stop()
op1.join()
op2.join()
```

Also the ability to select a value from multiple *DataflowChannels* can only be used through an adapter around a *DataflowStream*.

A DataflowStream Sample

```
import groovyx.gpars.dataflow.Select
import groovyx.gpars.dataflow.stream.DataflowStream
import groovyx.gpars.dataflow.stream.DataflowStreamReadAdapter
import groovyx.gpars.dataflow.stream.DataflowStreamWriteAdapter
import static groovyx.gpars.dataflow.Dataflow.select
import static groovyx.gpars.dataflow.Dataflow.task

/**
 * Demonstrates the use of DataflowStreamAdapters to allow dataflow select to select on
DataflowStreams
 */

final DataflowStream a = new DataflowStream()
final DataflowStream b = new DataflowStream()

def aw = new DataflowStreamWriteAdapter(a)
def bw = new DataflowStreamWriteAdapter(b)
def ar = new DataflowStreamReadAdapter(a)
def br = new DataflowStreamReadAdapter(b)

final Select<?> select = select(ar, br)
task {
    aw << 1
    aw << 2
    aw << 3
}

assert 1 == select().value
assert 2 == select().value
assert 3 == select().value

task {
    bw << 4
    aw << 5
    bw << 6
}

def result = (1..3).collect{select()}.sort{it.value}

assert result*.value == [4, 5, 6]
assert result*.index == [1, 0, 1]
```

If you don't need any of the functional queue *DataflowStream-special* functionality, like generation, filtering or mapping, you might consider using the *DataflowBroadcast* class instead.

This class offers the *publish-subscribe* communication model through the *DataflowChannel* interface.

Bind Handlers

What A Bind

```
def a = new DataflowVariable()

a >> {println "The variable has just been bound to $it"}

a.whenBound {println "Just to confirm that the variable has been really set to $it"}
...
```

`Bind handlers` can be registered on all dataflow channels (variables, queues or broadcasts) either using the '>>' operator and/or the *then()* or the *whenBound()* methods. They will be run only after a value is bound to the variable.

`Dataflow queues` and `broadcasts` also support a *wheneverBound* method to register a closure or a message handler to run each time a value is bound to them.

A DataflowQueue().wheneverBound Sample

```
def queue = new DataflowQueue()
queue.wheneverBound {println "A value $it arrived to the queue"}
```

Obviously, nothing prevents you from having more than a single handler for a single promise: They will all trigger in parallel once the **promise** has a concrete value:

A `wheneverBound` *Sample*

```
Promise bookingPromise = task {
    final data = collectData()
    return broker.makeBooking(data)
}

bookingPromise.whenBound {booking -> printAgenda booking}
bookingPromise.whenBound {booking -> sendMeAnEmailTo booking}
bookingPromise.whenBound {booking -> updateTheCalendar booking}
```

Parallel Speculations Anyone ?

Dataflow variables and broadcasts are one of several possible ways to implement *Parallel Speculations* . For details, please check out *Parallel Speculations* in the *Parallel Collections* section of the **User Guide**.

Bind Handlers Grouping

When you need to wait for multiple `DataflowVariables Promises` to be bound, we can benefit from calling the *whenAllBound()* function. It's available on the *Dataflow* class as well as on *PGroup* instances.

whenAllBound() Sample

```
    final group = new NonDaemonPGroup()

    //Calling asynchronous services and receiving back promises for the reservations
    Promise flightReservation = flightBookingService('PAR <-> BRU')
    Promise hotelReservation = hotelBookingService('BRU:Feb 24 20015 - Feb 29 2015')
    Promise taxiReservation = taxiBookingService('BRU:Feb 24 2015 10:31')

    //when all reservations have been made, we need to build an agenda for our trip
    Promise agenda = group.whenAllBound(flightReservation, hotelReservation,
taxiReservation) {flight, hotel, taxi ->
        "Agenda: $flight | $hotel | $taxi"
    }

    //since this is a demo, we only print the agenda and block when it's ready
    println agenda.val
```

If you don't know the number of parameters the *whenAllBound()* handler needs, then use a closure with one argument of type *List*:

whenAllBound() Sample

```
Promise module1 = task {
    compile(module1Sources)
}
Promise module2 = task {
    compile(module2Sources)
}

//We don't know the number of modules that will be jarred together, so use a List
final jarCompiledModules = {List modules -> ...}

whenAllBound([module1, module2], jarCompiledModules)
```

Bind Handler Chaining

All dataflow channels also support the *then()* method to register a callback handler to invoke when a value becomes available. Unlike *whenBound()*, the *then()* method allows us to use chaining, giving us the option to transfer resulting values between functions asynchronously.

Groovy allows us to leave out some of the *dots* in the *then()* method chains.

A Pointless Sample - No Need To Join The Dots !

```
final DataflowVariable variable = new DataflowVariable()
final DataflowVariable result = new DataflowVariable()

variable.then {it * 2} then {it + 1} then {result << it}
variable << 4
assert 9 == result.val
```

This could be nicely combined with Asynchronous functions

```
final DataflowVariable variable = new DataflowVariable()
final DataflowVariable result = new DataflowVariable()

final doubler = {it * 2}
final adder = {it + 1}

variable.then doubler then adder then {result << it}

Thread.start {variable << 4}

assert 9 == result.val
```

or ActiveObjects

```
@ActiveObject
class ActiveDemoCalculator {
    @ActiveMethod
    def doubler(int value) {
        value * 2
    }

    @ActiveMethod
    def adder(int value) {
        value + 1
    }
}

final DataflowVariable result = new DataflowVariable()
final calculator = new ActiveDemoCalculator();

calculator.doubler(4).then {calculator.adder it}.then {result << it}

assert 9 == result.val
```

Motivation for Chaining Promises

Chaining can save quite some code when calling other asynchronous services from within *whenBound()* handlers.

Asynchronous services, such as *Asynchronous Functions* or *Active Methods*, return **Promises** for their results. To obtain the actual results, your handlers would have to block to wait for the value to be bound. This locks the current thread in an unproductive state.

An Unproductive Sample

```
variable.whenBound {value ->
    Promise promise = asyncFunction(value)
    println promise.get()
}
```

or, alternatively, it could register another (nested) *whenBound()* handler, which would result in unnecessarily complex code.

An Unnecessarily Complex Nested Sample

```
variable.whenBound {value ->
    asyncFunction(value).whenBound {
        println it
    }
}
```

For an illustration, compare the following two code snippets. One is using *whenBound()* and one using *then()* chaining. They're both equivalent in terms of functionality and behavior.

A whenBound() Sample Plus a then() Example

```
final DataflowVariable variable = new DataflowVariable()

final doubler = {it * 2}
final inc = {it + 1}

//Using whenBound()
variable.whenBound {value ->
    task {
        doubler(value)
    }.whenBound {doubledValue ->
        task {
            inc(doubledValue)
        }.whenBound {incrementedValue ->
            println incrementedValue
        }
    }
}

//Using then() chaining
variable.then doubler then inc then this.&println

Thread.start {variable << 4}
```

Chaining Promises solves both of these issues elegantly:

```
variable >> asyncFunction >> {println it}
```

The *RightShift* '>>' operator has been overloaded to call *then()* method and, therefore, can be chained the same way:

A Chaining Sample

```
final DataflowVariable variable = new DataflowVariable()
final DataflowVariable result = new DataflowVariable()

final doubler = {it * 2}
final adder = {it + 1}

variable >> doubler >> adder >> {result << it}

Thread.start {variable << 4}

assert 9 == result.val
```

Error Handling for Promise Chaining

Asynchronous operations may obviously throw exceptions. It's important to be able to handle them easily and with little effort. **GPars romise** objects can implicitly propagate exceptions from asynchronous calculations across **promise** chains.

- **Promises** propagate result values as well as exceptions. The blocking *get()* method re-throws any exception that was bound to the **Promise** so the caller can handle it.
- For `asynchronous notifications` - the *whenBound()* handler closure - gets the exception passed in as an argument.
- The *then()* method accepts two arguments - a **value handler** and an optional **error handler**. These will be invoked depending on whether the result is a regular value or an exception. If no errorHandler is specified, the exception is re-thrown to the **Promise** returned by *then()* .
- Exactly the same behavior for *then()* methods holds true for the *whenAllBound()* method, which listens on multiple **Promises** to get bound.

A Sample of Error Handling

```
Promise<Integer> initial = new DataflowVariable<Integer>()
Promise<String> result = initial.then {it * 2} then {100 / it}  // Will throw exception
for 0
.then {println "Log the value $it as it passes by"; return it}  // No error handler is
defined,
                                                                // so exceptions are
ignored
                                                                // and silently re-thrown
to the next handler in chain
.then({"The result for $num is $it"}, {"Error detected for $num: $it"}) // Here the
exception is caught

initial << 0

println result.get()
```

ErrorHandler is a closure that accepts instances of *Throwable* as its' only (optional) argument. It returns a value that should be bound to the result of the *then()* method call, i.e. the returned **Promise**. If an exception is thrown from within an error handler, it's bound to the resulting **Promise** as an error.

Re-throwing Potential Exceptions

```
promise.then({it+1})                 // Implicitly re-throws potential exceptions bound
to promise
promise.then({it+1}, {e -> throw e})  // Explicitly re-throws potential exceptions bound
to promise

promise.then({it+1}, {e -> throw new RuntimeException('Error occurred', e})
// Explicitly re-throws a new exception wrapping a potential exception bound to a
*Promise*
```

Where Do You Want This Exception ?

Exception handling in **Java** has try-catch statements. The behavior of **GPars Promise** objects gives an asynchronous invocation freedom to handle exceptions at anywhere it's most convenient. You can freely ignore exceptions in your code if you want to, then just assume things work. Even so, remember that exceptions are not accidentally swallowed.

A Exceptional Sample

```
task {
    'gpars.org'.toURL().text  //should throw MalformedURLException
}

.then {page -> page.toUpperCase()}
.then {page -> page.contains('GROOVY')}
.then({mentionsGroovy -> println "Groovy found: $mentionsGroovy"}, {error -> println
"Error: $error"}).join()
```

Handling Concrete Exception Types

You may also be more specific about the handled exception types like this :

A Specific Exception Handling Example

```
url.then(download)
    .then(calculateHash, {MalformedURLException e -> return 0}) // <- specific !
    .then(formatResult)
    .then(printResult, printError)
    .then(sendNotificationEmail);
```

Customer-site Exception Handling

You may wish to leave an exception completely un-handled, then let clients (consumers) handle it:

A Delayed Exception Handling Example

```
Promise<Object> result = url.then(download).then(calculateHash).then(formatResult).then
(printResult);
try {
    result.get()
} catch (Exception e) {
    //handle exceptions here
}
```

Putting It All Together

By combining *whenAllBound()* and *then* (or '>>') methods, we can easily manage large asynchronous scenarios in a convenient way:

A Large Asynchronous Sample

```
withPool {

    Closure download = {String url ->
        sleep 3000  //Simulate a web read
        'web content'
    }.asyncFun()

    Closure loadFile = {String fileName ->
        'file content'  //simulate a local file read
    }.asyncFun()

    Closure hash = {s -> s.hashCode()}

    Closure compare = {int first, int second ->
        first == second
    }

    Closure errorHandler = {println "Error detected: $it"}

    def all = whenAllBound([
                download('http://www.gpars.org') >> hash,
                loadFile('/coolStuff/gpars/website/index.html') >> hash
            ], compare).then({println it}, errorHandler)
    all.join()  //optionally block until the calculation is all done
```

Notice that only the initial action (function) needs to be asynchronous. The functions further down the pipeline will be invoked asynchronously by your **Promise**, even if they are synchronous.

Implementing the Fork/join Pattern With Promises

Promises are very flexible and can be used as an implementation vehicle for many different scenarios. Here's one handy additional capability of a **Promise**.

The _thenForkAndJoin() method triggers one or several activities once the current **Promise** becomes bound and returns a completed **Promise** object, bound only after all the activities finish.

Let's see how this fits into the picture:

- *then()* - permits activity chaining, so that one activity is performed after another
- *whenAllBound()* - allows joining multiple activities; a new activity is started only after they all finish
- *task()* - allows us to create (fork) multiple asynchronous activities
- *thenForkAndJoin()* - a short-hand syntax for forking several activities and joining on them

So with *thenForkAndJoin()* you simply create multiple activities that should be triggered by a shared (triggering) **Promise**.

A Sample of Multiple Activities

```
promise.thenForkAndJoin(task1, task2, task3).then{...}
```

Once all the activities return a result, they're collected into a list and bound into the **Promise** returned by *thenForkAndJoin()* .

A thenForkAndJoin() Sample

```
task {
    2
}.thenForkAndJoin({ it ** 2 }, { it**3 }, { it**4 }, { it**5 }).then({ println it}).join
()
```

Lazy Dataflow Tasks and Variables

Sometimes you may need to combine the qualities of `Dataflow Variables` with a lazy initialization.

A Lazy Sample

```
Closure<String> download = {url ->
    println "Downloading"
    url.toURL().text
}

def pageContent = new LazyDataflowVariable(download.curry("http://gpars.org"))
```

Instances of *LazyDataflowVariable* have an initializer declared at construction time. An instance is only triggered when someone asks for its value, either through the blocking *get()* method or using any of the non-blocking callback methods, such as *then()* . Since *LazyDataflowVariables* preserve all the goodness of ordinary *DataflowVariables* , you can chain them together easily with other *lazy* or *ordinary* `Dataflow Variables`.

A Bigger Example

This discussion deserves a more practical example. So, taking inspiration from this long post, the following piece of code demonstrates how to use *LazyDataflowVariables* to lazily and asynchronously load mutually dependent components into memory. The component modules will be loaded in the

order of their dependencies and concurrently, if possible.

Each module will only be loaded once, irrespective of the number of modules that depend on it. Thanks to `laziness`, only the modules that are transitively needed will be loaded. Our example uses a simple "diamond" dependency scheme:

- D depends on B and C
- C depends on A
- B depends on A

When loading D, A will get loaded first. B and C will be loaded concurrently once A has been loaded. D will start loading once both B and C have been loaded.

A Diamond Sample

```
def moduleA = new LazyDataflowVariable({->
    println "Loading moduleA into memory"
    sleep 3000
    println "Loaded moduleA into memory"
    return "moduleA"
})

def moduleB = new LazyDataflowVariable({->
    moduleA.then {
        println "->Loading moduleB into memory, since moduleA is ready"
        sleep 3000
        println "  Loaded moduleB into memory"
        return "moduleB"
    }
})

def moduleC = new LazyDataflowVariable({->
    moduleA.then {
        println "->Loading moduleC into memory, since moduleA is ready"
        sleep 3000
        println "  Loaded moduleC into memory"
        return "moduleC"
    }
})

def moduleD = new LazyDataflowVariable({->
    whenAllBound(moduleB, moduleC) { b, c ->
        println "-->Loading moduleD into memory, since moduleB and moduleC are ready"
        sleep 3000
        println "   Loaded moduleD into memory"
        return "moduleD"
    }
})

println "Nothing loaded so far"
println "=========================================================================="
println "Load module: " + moduleD.get()
println "=========================================================================="
println "All requested modules loaded"
```

Making Tasks Lazy

The *lazyTask()* method is available alongside the *task()* method to give the us a task-oriented

abstraction for delayed activities. A **Lazy Task** returns an instance of a *LazyDataflowVariable* (like a **Promise**) with the initializer set by the provided closure. As soon as someone asks for the value, the task will start asynchronously and eventually deliver a value into the *LazyDataflowVariable* .

A Lazy Sample

```
import groovyx.gpars.dataflow.Dataflow

def pageContent = Dataflow.lazyTask {
        println "Downloading"
        "http://gpars.org".toURL().text
    }

println "No-one has asked for the value just yet. Bound = ${pageContent.bound}"
sleep 1000
println "Now going to ask for a value"
println pageContent.get().size()
println "Repetitive requests will receive the already calculated value. No additional
downloading."
println pageContent.get().size()
```

Dataflow Expressions

Look at the magic below:

A Dataflow Sample

```
def initialDistance = new DataflowVariable()
def acceleration = new DataflowVariable()
def time = new DataflowVariable()

task {
    initialDistance << 100
    acceleration << 2
    time << 10
}

def result = initialDistance + acceleration*0.5*time**2
println 'Total distance ' + result.val
```

We use **DataflowVariables** that represent several parameters to a mathematical equation calculating total distance of an accelerating object. In the equation itself, however, we use the **DataflowVariable** directly. We do not refer to the values they represent and yet we are able to do the math correctly. This shows that **DataflowVariables** can be very flexible.

For example, you can call methods on them and these methods are dispatched to the bound values:

A **DataflowVariable** *Sample*

```
def name = new DataflowVariable()
task {
    name << '  adam   '
}
println name.toUpperCase().trim().val
```

You can pass other **DataflowVariables** as arguments to such methods and the real values will be passed automatically instead:

Another **DataflowVariable** *as An Argument Sample*

```
def title = new DataflowVariable()
def searchPhrase = new DataflowVariable()
task {
    title << ' Groovy in Action 2nd edition   '
}

task {
    searchPhrase << '2nd'
}

println title.trim().contains(searchPhrase).val
```

And you can also query properties of the bound value using directly the **DataflowVariable**:

A **DataflowVariable** *Sample To Query Book Title Properties*

```
def book = new DataflowVariable()
def searchPhrase = new DataflowVariable()
task {
    book << [
              title:'Groovy in Action 2nd edition   ',
              author:'Dierk Koenig',
              publisher:'Manning']
}

task {
    searchPhrase << '2nd'
}

book.title.trim().contains(searchPhrase).whenBound {println it}  //Asynchronous waiting

println book.title.trim().contains(searchPhrase).val  //Synchronous waiting
```

Please note that the result is still a **DataflowVariable** (`DataflowExpression` to be precise), from which you can get the real value from both synchronously and asynchronously.

Bind Error Notification

DataflowVariables offer the ability to send notifications to registered listeners whenever a bind operation fails. The *getBindErrorManager()* method allows listeners to be added and removed. The listeners are notified in case of a failed attempt to bind a value (through `bind()`, `bindSafely()`, `bindUnique()` or `leftShift()`) or an error (through **bindError()**).

Reporting Bind Operation FailED

```
        final DataflowVariable variable = new DataflowVariable()

        variable.getBindErrorManager().addBindErrorListener(new BindErrorListener() {
            @Override
            void onBindError(final Object oldValue, final Object failedValue, final
boolean uniqueBind) {
                println "Bind failed!"
            }

            @Override
            void onBindError(final Object oldValue, final Throwable failedError) {
                println "Binding an error failed!"
            }

            @Override
            public void onBindError(final Throwable oldError, final Object failedValue,
final boolean uniqueBind) {
                println "Bind failed!"
            }

            @Override
            public void onBindError(final Throwable oldError, final Throwable
failedError) {
                println "Binding an error failed!"
            }

        })
```

This lets us customize responses to any attempt to bind an already bound **Dataflow Variable**. For example, using *bindSafely()*, you do not receive bind exceptions back to the caller, but rather, a registered *BindErrorListener* is notified.

Further Reading

- Scala Dataflow library by Jonas Bonér
- JVM concurrency presentation slides by Jonas Bonér
- Dataflow Concurrency library for Ruby

Tasks

The **Dataflow Task** give us an easy-to-grasp abstraction of mutually-independent logical tasks or threads. These can run concurrently and exchange data solely through `Dataflow Variables`, `Queues`, `Broadcasts` and `Streams`. A **Dataflow Task** with it's easy-to-express mutual dependencies and inherently sequential body could also be used as a practical implementation of a UML *Activity Diagrams* .

Check out the examples.

A Simple Mashup Example

In this example, we're downloading the front pages of three popular web sites, each in their own task, while in a separate task we're filtering out sites talking about **Groovy** today and forming the output. The output task synchronizes automatically with the three download tasks on the three Dataflow variables through which the content of each website is passed to the output task.

What Wonderful A Mashup !

```
import static groovyx.gpars.GParsPool.withPool
import groovyx.gpars.dataflow.DataflowVariable
import static groovyx.gpars.dataflow.Dataflow.task


/**
 * A simple mashup sample, downloads content of three websites
 * and checks how many of them refer to Groovy.
 */

def dzone = new DataflowVariable()
def jroller = new DataflowVariable()
def theserverside = new DataflowVariable()

task {
    println 'Started downloading from DZone'
    dzone << 'http://www.dzone.com'.toURL().text
    println 'Done downloading from DZone'
}

task {
    println 'Started downloading from JRoller'
    jroller << 'http://www.jroller.com'.toURL().text
    println 'Done downloading from JRoller'
}

task {
    println 'Started downloading from TheServerSide'
    theserverside << 'http://www.theserverside.com'.toURL().text
    println 'Done downloading from TheServerSide'
}

task {
    withPool {
        println "Number of Groovy sites today: " +
                ([dzone, jroller, theserverside].findAllParallel {
                    it.val.toUpperCase().contains 'GROOVY'
                }).size()
    }
}.join()
```

Grouping Tasks

Dataflow tasks can be organized into groups for performance fine-tuning. Groups provide a handy *task()* factory method to create tasks attached to these groups. Using groups allows us to organize tasks or operators around different thread pools (wrapped inside the group). While the Dataflow.task() command schedules the task on a default thread pool (`java.util.concurrent.Executor, fixed size=#cpu+1, daemon threads`), we might prefer defining our own thread pool(s) to run these tasks.

A Personal Thread Pool Sample

```
import groovyx.gpars.group.DefaultPGroup

def group = new DefaultPGroup()

group.with {
    task {
        ...
    }

    task {
        ...
    }
}
```

Custom Thread Pools for Dataflow

The default thread pool for dataflow tasks has daemon threads. This means our application will exit as soon as the main thread finishes and **won't** wait for all tasks to complete!

When grouping tasks, make sure the custom thread pools either :

1. use daemon threads (achieved by using **DefaultPGroup**)
2. provide a thread factory to a thread pool constructor
3. or in case the thread pools use non-daemon threads, (from the **NonDaemonPGroup** group class), we must shutdown the group or the thread pool explicitly by calling its **shutdown()** method, otherwise our application will not exit.

We can selectively override the default group used for tasks, operators, callbacks and other dataflow elements inside a code block using the *Dataflow.usingGroup()* method:

A Sample

```
Dataflow.usingGroup(group) {
    task {
        'http://gpars.codehaus.org'.toURL().text  //should throw MalformedURLException
    }
    .then {page -> page.toUpperCase()}
    .then {page -> page.contains('GROOVY')}
    .then({mentionsGroovy -> println "Groovy found: $mentionsGroovy"}, {error -> println
"Error: $error"}).join()
}
```

You can always override the default group by being specific:

A Sample

```
Dataflow.usingGroup(group) {
    anotherGroup.task {
        'http://gpars.codehaus.org'.toURL().text  //should throw MalformedURLException
    }
    .then(anotherGroup) {page -> page.toUpperCase()}
    .then(anotherGroup) {page -> page.contains('GROOVY')}.then(anotherGroup) {println
Dataflow.retrieveCurrentDFPGroup();it}
    .then(anotherGroup, {mentionsGroovy -> println "Groovy found: $mentionsGroovy"},
{error -> println "Error: $error"}).join()
}
```

A Mashup Variant With Methods

To avoid giving you wrong impression about structuring the Dataflow code, here's a rewrite of the mashup example, with a *downloadPage()* method performing the actual download in a separate task and returning a DataflowVariable instance, so that the main application thread could eventually get hold of the downloaded content. Dataflow variables can obviously be passed around as parameters or return values.

A Sample

```
package groovyx.gpars.samples.dataflow

import static groovyx.gpars.GParsExecutorsPool.withPool
import groovyx.gpars.dataflow.DataflowVariable
import static groovyx.gpars.dataflow.Dataflow.task


/**
 * A simple mashup sample, downloads content of three websites and checks how many of
them refer to Groovy.
 */
final List urls = ['http://www.dzone.com', 'http://www.jroller.com',
'http://www.theserverside.com']

task {
    def pages = urls.collect { downloadPage(it) }
    withPool {
        println "Number of Groovy sites today: " +
                (pages.findAllParallel {
                    it.val.toUpperCase().contains 'GROOVY'
                }).size()
    }
}.join()

def downloadPage(def url) {
    def page = new DataflowVariable()
    task {
        println "Started downloading from $url"
        page << url.toURL().text
        println "Done downloading from $url"
    }
    return page
}
```

A Physical Calculation Example

Dataflow programs naturally scale with the number of processors. Up to a certain level, the more processors you have the faster the program runs. Check out, for example, the following script, which calculates parameters of a simple physical experiment and prints out the results. Each task performs its part of the calculation and may depend on values calculated by some other tasks and its' results might be needed by some of the other tasks. With `Dataflow Concurrency` you can split the work between tasks or reorder the tasks themselves as you like and the dataflow mechanics will ensure the

calculation is accomplished correctly.

A DataflowVariable Sample

```
import groovyx.gpars.dataflow.DataflowVariable
import static groovyx.gpars.dataflow.Dataflow.task

final def mass = new DataflowVariable()
final def radius = new DataflowVariable()
final def volume = new DataflowVariable()
final def density = new DataflowVariable()
final def acceleration = new DataflowVariable()
final def time = new DataflowVariable()
final def velocity = new DataflowVariable()
final def decelerationForce = new DataflowVariable()
final def deceleration = new DataflowVariable()
final def distance = new DataflowVariable()

def t = task {
    println """

Calculating distance required to stop a moving ball.
.....................................................
The ball has a radius of ${radius.val} meters and is made of a material with ${density
.val} kg/m3 density,
which means that the ball has a volume of ${volume.val} m3 and a mass of ${mass.val} kg.
The ball has been accelerating with ${acceleration.val} m/s2 from 0 for ${time.val}
seconds and so reached a velocity of ${velocity.val} m/s.

Given our ability to push the ball backwards with a force of ${decelerationForce.val} N
(Newton), we can cause a deceleration
of ${deceleration.val} m/s2 and so stop the ball at a distance of ${distance.val} m.
.....................................................

This example has been calculated asynchronously in multiple tasks using *GPars* Dataflow
concurrency in Groovy.
Author: ${author.val}
"""

    System.exit 0
}

task {
    mass << volume.val * density.val
}

task {
    volume << Math.PI * (radius.val ** 3)
```

```
}

task {
    radius << 2.5
    density <<  998.2071  //water
    acceleration << 9.80665 //free fall
    decelerationForce << 900
}

task {
    println 'Enter your name:'
    def name = new InputStreamReader(System.in).readLine()
    author << (name?.trim()?.size()>0 ? name : 'anonymous')
}

task {
    time << 10
    velocity << acceleration.val * time.val
}

task {
    deceleration << decelerationForce.val / mass.val
}

task {
    distance << deceleration.val * ((velocity.val/deceleration.val) ** 2) * 0.5
}

t.join()
```

I did my best to make all the physical calculations right. Feel free to change the values and see how long a distance you need to stop the rolling ball.

Deterministic Deadlocks

If you happen to introduce a deadlock in your dependencies, the deadlock will occur each time you run the code. No randomness is allowed. That's one of the benefits of `Dataflow Concurrency`. Irrespective of the actual thread scheduling scheme, if you don't get a deadlock in tests, you won't get them in production.

A Deterministic Deadlock Sample

```
task {
    println a.val
    b << 'Hi there'
}

task {
    println b.val
    a << 'Hello man'
}
```

Dataflows Map

As a handy shortcut the *Dataflows* class can help you reduce the amount of code you need to leverage `Dataflow Variables`.

A Convenience Example

```
def df = new Dataflows()
df.x = 'value1'

assert df.x == 'value1'

Dataflow.task {df.y = 'value2}

assert df.y == 'value2'
```

Think of **Dataflows** as a map with `Dataflow Variables` as keys storing their bound values as appropriate map values. The semantics of reading a value (e.g. df.x) and binding a value (e.g. df.x = 'value') are identical to the semantics of plain `Dataflow Variables` (x.val and x << 'value' respectively).

Mixing *Dataflows* and Groovy *with* blocks

When inside a *with* block of a **Dataflows** instance, the `Dataflow Variables` stored inside the **Dataflows** instance can be accessed directly without the need to prefix them with the **Dataflows** instance identifier.

A with block Makes Coding Easier

```
new Dataflows().with {
    x = 'value1'
    assert x == 'value1'

    Dataflow.task {y = 'value2}

    assert y == 'value2'
}
```

Returning Values From A Task

Typically, **Dataflow** tasks communicate through `Dataflow Variables`. On top of that, tasks can also return values, again through a `Dataflow Variable`. When you invoke the *task()* factory method, you get back an instance of a **Promise** (implemented as `DataflowVariable`), through which you can listen for the task's return value, just like when using any other **Promise** or `DataflowVariable`.

A Task Returns A Value Using a **Promise**

```
    final Promise t1 = task {
        return 10
    }
    final Promise t2 = task {
        return 20
    }

    def results = [t1, t2]*.val

    println 'Both sub-tasks finished and returned values: ' + results
```

The value can also be obtained without blocking the caller using the *whenBound()* method.

A Sample

```
def task = task {
    println 'The task is running and calculating the return value'
    30  // the value to be returned
}
task >> {value -> println "The task finished and returned $value"}
```

Joining Tasks

Using the *join()* operation on the resulting `Dataflow Variable` of a task, you can block until the task finishes.

A Bloacking Sample

```
task {
    final Promise t1 = task {
        println 'First sub-task running.'
    }
    final Promise t2 = task {
        println 'Second sub-task running'
    }
    [t1, t2]*.join()
    println 'Both sub-tasks finished'
}.join()
```

Selects

Frequently, a value needs to be obtained from one of several dataflow channels (such as variables, queues, broadcasts or streams). The *Select* class is suitable for these scenarios.

Select can scan several dataflow channels and pick one channel from all the input channels, with a value that's ready to be read. The value from that chosen channel is read and returned to the caller together with the index of the originating channel. Picking the channel is either random, or based on channel priority, in which case, channels with a lower position index in the *Select* constructor have higher priority.

Selecting A Value From Multiple Channels

```
import groovyx.gpars.dataflow.DataflowQueue
import groovyx.gpars.dataflow.DataflowVariable
import static groovyx.gpars.dataflow.Dataflow.select
import static groovyx.gpars.dataflow.Dataflow.task

/**
 * Shows a basic use of Select, which monitors a set of input channels for values and
makes these values
 * available on its output irrespective of their original input channel.
 * Note that dataflow variables and queues can be combined for Select.
 *
 * You might also consider checking out the prioritySelect method, which prioritizes
values by the index of their input channel
 */
def a = new DataflowVariable()
def b = new DataflowVariable()
def c = new DataflowQueue()

task {
    sleep 3000
    a << 10
}

task {
    sleep 1000
    b << 20
}

task {
    sleep 5000
    c << 30
}

def select = select([a, b, c])

println "The fastest result is ${select().value}"
```

A *select()* Method Returns What ?

Note that the return type from *select()* is *SelectResult* , holding the value as well as the original channel index.

There are several ways to read values from a **Select**:

How Do I **Select** *Thee ? Let Me Count The Ways !*

```
def sel = select(a, b, c, d)
def result = sel.select()                                    //Random selection
def result = sel()                                           //Random selection (a
short-hand variant)
def result = sel.select([true, true, false, true])           //Random selection with
guards specified
def result = sel([true, true, false, true])                  //Random selection with
guards specified (a short-hand variant)

def result = sel.prioritySelect()                            //Priority selection
def result = sel.prioritySelect([true, true, false, true])   //Priority selection with
guards specifies
```

By default, the *Select* method blocks processing of the caller until a value is available to be read. The alternative *selectToPromise()* and *prioritySelectToPromise()* methods give us a way to obtain a **Promise** of a value that can be selected later. Through the returned **Promise**, you can register a callback to invoke asynchronously whenever the next value is selected.

Random And Priority Seletions

```
def sel = select(a, b, c, d)

Promise result = sel.selectToPromise()                              //Random
selection
Promise result = sel.selectToPromise([true, true, false, true])     //Random
selection with guards specified

Promise result = sel.prioritySelectToPromise()                      //Priority
selection
Promise result = sel.prioritySelectToPromise([true, true, false, true])   //Priority
selection with guards specifies
```

Another Way ?

Alternatively, the *Select* method can have it's value sent to a declared *MessageStream* (e.g. an **Actor**) without blocking the caller.

A Sample

```
def handler = actor {...}
def sel = select(a, b, c, d)

sel.select(handler)                                    //Random selection
sel(handler)                                           //Random selection (a short-
hand variant)
sel.select(handler, [true, true, false, true])         //Random selection with
guards specified
sel(handler, [true, true, false, true])                //Random selection with
guards specified (a short-hand variant)

sel.prioritySelect(handler)                            //Priority selection
sel.prioritySelect(handler, [true, true, false, true]) //Priority selection with
guards specifies
```

Guards

Guards let the caller omit some input channels from the selection. **Guards** are specified as a List of boolean flags passed to the *select()* or *prioritySelect()* methods.

A Useful Filter Tool

```
def sel = select(leaders, seniors, experts, juniors)
def teamLead = sel([true, true, false, false]).value        //Only 'leaders' and
'seniors' qualify for becoming a teamLead here
```

A typical use for **Guards** is to make **Selects** flexible enough to adapt to changes in user state.

Guards Enable/Disable Channels During Value Selection

```
import groovyx.gpars.dataflow.DataflowQueue
import static groovyx.gpars.dataflow.Dataflow.select
import static groovyx.gpars.dataflow.Dataflow.task

/**
 * Demonstrates the ability to enable/disable channels during a value selection on a
Select by providing boolean guards.
 */
final DataflowQueue operations = new DataflowQueue()
final DataflowQueue numbers = new DataflowQueue()

def t = task {
    final def select = select(operations, numbers)
    3.times {
        def instruction = select([true, false]).value
        def num1 = select([false, true]).value
        def num2 = select([false, true]).value
        final def formula = "$num1 $instruction $num2"
        println "$formula = ${new GroovyShell().evaluate(formula)}"
    }
}

task {
    operations << '+'
    operations << '+'
    operations << '*'
}

task {
    numbers << 10
    numbers << 20
    numbers << 30
    numbers << 40
    numbers << 50
    numbers << 60
}

t.join()
```

Priority Select

When certain channels should have precedence over others when selecting, the **prioritySelect**

methods should be used instead.

A **prioritySelect** *Sample*

```
/**
 * Here's a simply usecase for Priority Select. It monitors a set of input channels for
values and makes these values
 * available on its output irrespective of their original input channel.
 *
 * Note that dataflow variables, queues and broadcasts can be combined for Select.
 *
 * Unlike plain select method call, the prioritySelect call gives precedence to input
channels with lower index.
 * Available messages from high priority channels will be served before messages from
lower-priority channels.
 * Messages received through a single input channel will have their mutual order
preserved.
 *
 */
def critical = new DataflowVariable()
def ordinary = new DataflowQueue()
def whoCares = new DataflowQueue()

task {
    ordinary << 'All working fine'
    whoCares << 'I feel a bit tired'
    ordinary << 'We are on target'
}

task {
    ordinary << 'I have just started my work. Busy. Will come back later...'
    sleep 5000
    ordinary << 'I am done for now'
}

task {
    whoCares << 'Huh, what is that noise'
    ordinary << 'Here I am to do some clean-up work'
    whoCares << 'I wonder whether unplugging this cable will eliminate that nasty sound.'
    critical << 'The server room runs on UPS!'
    whoCares << 'The sound has disappeared'
}

def select = select([critical, ordinary, whoCares])

println 'Starting to monitor our IT department'

sleep 3000
10.times {println "Received: ${select.prioritySelect().value}"}
```

Collecting Results of Asynchronous Computations

No matter whether they are **dataflow tasks** , **active objects' methods** or **asynchronous functions**, asynchronous activities always return a **Promise**. **Promises** implement the *SelectableChannel* interface and so can be passed in *selects* for selection together with other **Promises** as well as *read channels* .

Similarly to **Java**'s *CompletionService* , our **GPars** *Select* method enables you to obtain results of asynchronous activities as soon as each becomes available. Also, we can use a *Select* to give us the first/fastest result from the first of several computations running in parallel.

How To Pick The Fastest Result

```
import groovyx.gpars.dataflow.Promise
import groovyx.gpars.dataflow.Select
import groovyx.gpars.group.DefaultPGroup
/**
 * Demonstrates the use of dataflow tasks and selects to pick the fastest result of
concurrently run calculations.
 */

final group = new DefaultPGroup()
group.with {
    Promise p1 = task {
        sleep(1000)
        10 * 10 + 1
    }
    Promise p2 = task {
        sleep(1000)
        5 * 20 + 2
    }
    Promise p3 = task {
        sleep(1000)
        1 * 100 + 3
    }

    final alt = new Select(group, p1, p2, p3)

    def result = alt.select()
    println "Result: " + result
}
```

Timeouts

The *Select.createTimeout()* method will create a **DataflowVariable** bound to a value after a declared

time period. This can be leveraged in *Selects* so that they unblock(resume processing) after a desired delay, if none of the other channels delivers a value before that time. Simply pass a **timeout channel** as another input channel to the *Select* .

A **Timeout Channel** *Helps Pick The Fastest Answer*

```
import groovyx.gpars.dataflow.Promise
import groovyx.gpars.dataflow.Select
import groovyx.gpars.group.DefaultPGroup
/**
 * Demonstrates the use of dataflow tasks and selects to pick the fastest result of
concurrently run calculations.
 */

final group = new DefaultPGroup()
group.with {
    Promise p1 = task {
        sleep(1000)
        10 * 10 + 1
    }
    Promise p2 = task {
        sleep(1000)
        5 * 20 + 2
    }
    Promise p3 = task {
        sleep(1000)
        1 * 100 + 3
    }

    final timeoutChannel = Select.createTimeout(500)

    final alt = new Select(group, p1, p2, p3, timeoutChannel)

    def result = alt.select()
    println "Result: " + result
}
```

Cancellations

Ok, so we have our answer. What bout the other tasks that continue to work on their answer? If we need to cancel those other tasks once an answer was found or maybe a timeout expired, then the best way is to set a flag that our tasks periodically monitor.

Intentionally, There's no cancellation machinery built into *DataflowVariables* or *Tasks*

A Sample To Pick The Fastest Answer And Cancel The Others

```
import groovyx.gpars.dataflow.Promise
import groovyx.gpars.dataflow.Select
import groovyx.gpars.group.DefaultPGroup

import java.util.concurrent.atomic.AtomicBoolean

/**
 * Demonstrates the use of dataflow tasks and selects to pick the fastest result of
concurrently run calculations.
 * It shows a waY to cancel the slower tasks once a result is known
 */

final group = new DefaultPGroup()
final done = new AtomicBoolean()

group.with {
    Promise p1 = task {
        sleep(1000)
        if (done.get()) return
        10 * 10 + 1
    }
    Promise p2 = task {
        sleep(1000)
        if (done.get()) return
        5 * 20 + 2
    }
    Promise p3 = task {
        sleep(1000)
        if (done.get()) return
        1 * 100 + 3
    }

    final alt = new Select(group, p1, p2, p3, Select.createTimeout(500))

    def result = alt.select()
    done.set(true)

    println "Result: " + result
}
```

Operators

`Dataflow Operators` and `Selectors` provide a full **Dataflow** implementation with all the usual ceremony.

Concepts

Full `Dataflow Concurrency` builds on the concept of channels connecting operators and selectors. These objects consume values coming through input channels, transform them into new values and output the new values into their output channels.

Operators wait for **every** input channel to have a value before starting to process them but *Selectors* only wait for the first available value on **any** of its' input channels.

An Operator Sample

```
operator(inputs: [a, b, c], outputs: [d]) {x, y, z ->
    ...
    bindOutput 0, x + y + z
}
```

A Sample Cache

```
/**
 * CACHE
 *
 * Caches sites' contents. Accepts requests for url content, outputs the content. Outputs
requests for download
 * if the site is not in cache yet.
 */
operator(inputs: [urlRequests], outputs: [downloadRequests, sites]) {request ->

    if (!request.content) {
        println "[Cache] Retrieving ${request.site}"

        def content = cache[request.site]

        if (content) {
            println "[Cache] Found in cache"
            bindOutput 1, [site: request.site, word:request.word, content: content]
        } else {
            def downloads = pendingDownloads[request.site]
            if (downloads != null) {
                println "[Cache] Awaiting download"
                downloads << request
            } else {
```

```
                pendingDownloads[request.site] = []
                println "[Cache] Asking for download"
                bindOutput 0, request
            }
        }

    } else {
        println "[Cache] Caching ${request.site}"

        cache[request.site] = request.content
        bindOutput 1, request

        def downloads = pendingDownloads[request.site]

        if (downloads != null) {
            for (downloadRequest in downloads) {
                println "[Cache] Waking up"
                bindOutput 1, [site: downloadRequest.site, word:downloadRequest.word,
content: request.content]
            }
            pendingDownloads.remove(request.site)
        }
    }
}
```

Exception Handling Explained

Standard error handling prints an error message to the standard error output and terminates the operator in case an uncaught exception is thrown from withing the operator's body. To alter the behavior, you can register your own event listener. See the *Operator Lifecycle* section for more details.

Exception Handling Sample

```
def listener = new DataflowEventAdapter() {

    @Override
    boolean onException(final DataflowProcessor processor, final Throwable e) {
        logChannel << e
        return false   //Indicate whether to terminate the operator or not
    }
}

op = group.operator(inputs: [a, b], outputs: [c], listeners: [listener]) {x, y ->
    ...
}
```

Types of Operators

There are specialized versions of operator methods for specific purposes:

- operator - the basic general-purpose operator
- selector - operator triggered by a value being available on any input channel
- prioritySelector - a selector that prefers delivering messages from lower-indexed input channels over higher-indexed ones
- splitter - a single-input operator copying its input values to all of its output channels

Wiring Operators Together

Operators are typically combined into networks, like when some operators consume output produced by other operators.

Using Several Operators

```
operator(inputs:[a, b], outputs:[c, d]) {...}
splitter(c, [e, f])
selector(inputs:[e, d]: outputs:[]) {...}
```

You may alternatively refer to output channels through operators themselves:

A More Complex Operator Example

```
def op1 = operator(inputs:[a, b], outputs:[c, d]) {...}
def sp1 = splitter(op1.outputs[0], [e, f])                          //takes the first
output of op1

selector(inputs:[sp1.outputs[0], op1.outputs[1]]: outputs:[]) {...}   //takes the first
output of sp1 and the second output of op1
```

Grouping Operators

Dataflow operators can be organized into groups for performance fine-tuning. Groups provide a handy *operator()* factory method to create tasks attached to the groups.

For Performance Fine-tuning ? Use **Groups**

```
import groovyx.gpars.group.DefaultPGroup

def group = new DefaultPGroup()

group.with {
    operator(inputs: [a, b, c], outputs: [d]) {x, y, z ->
        ...
        bindOutput 0, x + y + z
    }
}
```

Custom Thread Pools For Dataflow

The default thread pool for dataflow operators contains daemon threads, which means your application will exit as soon as the main thread finishes and won't wait for all tasks to complete.

When grouping operators, make sure your custom thread pools use either daemon threads, too, which can be achieved by using DefaultPGroup or by providing your own thread factory to a thread pool constructor, or in case your thread pools use non-daemon threads, such as when using the NonDaemonPGroup group class, make sure you shutdown the group or the thread pool explicitly by calling its shutdown() method, otherwise your applications will not exit.

You may selectively override the default group used for tasks, operators, callbacks and other dataflow elements inside a code block using the *Dataflow.usingGroup()* method:

A Sample

```
Dataflow.usingGroup(group) {
    operator(inputs: [a, b, c], outputs: [d]) {x, y, z ->
        ...
        bindOutput 0, x + y + z
    }
}
```

You can always override the default group by being specific:

A Sample

```
Dataflow.usingGroup(group) {
    anotherGroup.operator(inputs: [a, b, c], outputs: [d]) {x, y, z ->
        ...
        bindOutput 0, x + y + z
    }
}
```

Constructing Operators

The construction properties of an operator, such as *inputs*, *outputs*, *stateObject* or *maxForks* cannot be modified once the operator has been build. You may find the *groovyx.gpars.dataflow.ProcessingNode* class helpful when gradually collecting channels and values into lists before you finally build an operator.

A Sample

```
import groovyx.gpars.dataflow.Dataflow
import groovyx.gpars.dataflow.DataflowQueue
import static groovyx.gpars.dataflow.ProcessingNode.node

/**
 * Shows how to build operators using the ProcessingNode class
 */

final DataflowQueue aValues = new DataflowQueue()
final DataflowQueue bValues = new DataflowQueue()
final DataflowQueue results = new DataflowQueue()

//Create a config and gradually set the required properties - channels, code, etc.
def adderConfig = node {valueA, valueB ->
    bindOutput valueA + valueB
}
adderConfig.inputs << aValues
adderConfig.inputs << bValues
adderConfig.outputs << results

//Build the operator
final adder = adderConfig.operator(Dataflow.DATA_FLOW_GROUP)

//Now the operator is running and processing the data
aValues << 10
aValues << 20
bValues << 1
bValues << 2

assert [11, 22] == (1..2).collect {
    results.val
}
```

Holding State in Operators

Although operators can frequently do without keeping state between subsequent invocations, **GPars** allows operators to maintain state, if desired by the developer. One obvious way is to leverage the **Groovy** closure capabilities to close-over their context:

A Sample

```
int counter = 0
operator(inputs: [a], outputs: [b]) {value ->
    counter += 1
}
```

Another way, which allows you to avoid declaring the state object outside of the operator definition, is to pass the state object into the operator as a *stateObject* parameter at construction time:

A Sample

```
operator(inputs: [a], outputs: [b], stateObject: [counter: 0]) {value ->
    stateObject.counter += 1
}
```

Parallelize Operators

By default an operator's body is processed by a single thread at a time. While this is a safe setting allowing the operator's body to be written in a non-thread-safe manner, once an operator becomes "hot" and data start to accumulate in the operator's input queues, you might consider allowing multiple threads to run the operator's body concurrently. Bear in mind that in such a case you need to avoid or protect shared resources from multi-threaded access. To enable multiple threads to run the operator's body concurrently, pass an extra *maxForks* parameter when creating an operator:

A Sample

```
def op = operator(inputs: [a, b, c], outputs: [d, e], maxForks: 2) {x, y, z ->
    bindOutput 0, x + y + z
    bindOutput 1, x * y * z
}
```

The value of the *maxForks* parameter indicates the maximum of threads running the operator concurrently. Only positive numbers are allowed with value 1 being the default.

Thread Starvation

Please always make sure the **group** serving the operator holds enough threads to support all requested forks. Using groups allows you to organize tasks or operators around different thread pools (wrapped inside the group). While the Dataflow.task() command schedules the task on a default thread pool (java.util.concurrent.Executor, fixed size=#cpu+1, daemon threads), you may prefer being able to define your own thread pool(s) to run your tasks.

A Sample

```
def group = new DefaultPGroup(10)
group.operator((inputs: [a, b, c], outputs: [d, e], maxForks: 5) {x, y, z -> ...}
```

The default group uses a resizeable thread pool as so will never run out of threads.

Synchronizing Outputs

When enabling internal parallelization of an operator by setting the value for *maxForks* to a value greater than 1 it is important to remember that without explicit or implicit synchronization in the operators' body race-conditions may occur. Especially bear in mind that values written to multiple output channels are not guarantied to be written atomically in the same order to all the channels

A Sample

```
operator(inputs:[inputChannel], outputs:[a, b], maxForks:5) {msg ->
    bindOutput 0, msg
    bindOutput 1, msg
}
inputChannel << 1
inputChannel << 2
inputChannel << 3
inputChannel << 4
inputChannel << 5
```

```
May result in output channels having the values mixed-up something like:
```

A Sample

```
a -> 1, 3, 2, 4, 5
b -> 2, 1, 3, 5, 4
```

```
Explicit synchronization is one way to get correctly bound all output channels and
protect operator not-thread local state:
```

A Sample

```
def lock = new Object()
operator(inputs:[inputChannel], outputs:[a, b], maxForks:5) {msg ->
    doStuffThatIsThreadSafe()

    synchronized(lock) {
        doSomethingThatMustNotBeAccessedByMultipleThreadsAtTheSameTime()
        bindOutput 0, msg
        bindOutput 1, 2*msg
    }
}
```

Obviously you need to weight the pros and cons here, since synchronization may defeat the purpose of setting *maxForks* to a value greater than 1.

To set values of all the operator's output channels in one atomic step, you may also consider calling either the *bindAllOutputsAtomically* method, passing in a single value to write to all output channels or the *bindAllOutputsAtomically* method, which takes a multiple values, each of which will be written to the output channel with the same position index.

A Sample

```
operator(inputs:[inputChannel], outputs:[a, b], maxForks:5) {msg ->
    doStuffThatIsThreadSafe()
        bindAllOutputValuesAtomically msg, 2*msg
    }
}
```

Which Bind Do I Use ?

```
Using the _bindAllOutputs_ or the _bindAllOutputValues_ methods will not guarantee
atomicity of writes across al the output channels when using internal parallelism.
```

If preserving the order of messages in multiple output channels is not an issue, *bindAllOutputs* as well as *bindAllOutputValues* will provide better performance over the atomic variants.

Operator Lifecycle

Dataflow operators and selectors fire several events during their lifecycle, which allows the interested parties to obtain notifications and potential alter operator's behavior. The *DataflowEventListener* interface offers a couple of callback methods:

A Sample

```
public interface DataflowEventListener {
    /**
     * Invoked immediately after the operator starts by a pooled thread before the first
message is obtained
     *
     * @param processor The reporting dataflow operator/selector
     */
    void afterStart(DataflowProcessor processor);

    /**
     * Invoked immediately after the operator terminates
     *
     * @param processor The reporting dataflow operator/selector
     */
    void afterStop(DataflowProcessor processor);

    /**
     * Invoked if an exception occurs.
     * If any of the listeners returns true, the operator will terminate.
     * Exceptions outside of the operator's body or listeners' messageSentOut() handlers
will terminate the operator irrespective of the listeners' votes.
     *
     * @param processor The reporting dataflow operator/selector
     * @param e         The thrown exception
     * @return True, if the operator should terminate in response to the exception, false
otherwise.
     */
    boolean onException(DataflowProcessor processor, Throwable e);

    /**
     * Invoked when a message becomes available in an input channel.
     *
     * @param processor The reporting dataflow operator/selector
     * @param channel   The input channel holding the message
     * @param index     The index of the input channel within the operator
     * @param message   The incoming message
     * @return The original message or a message that should be used instead
     */
    Object messageArrived(DataflowProcessor processor, DataflowReadChannel<Object>
```

```
channel, int index, Object message);

    /**
     * Invoked when a control message (instances of ControlMessage) becomes available in
an input channel.
     *
     * @param processor The reporting dataflow operator/selector
     * @param channel   The input channel holding the message
     * @param index     The index of the input channel within the operator
     * @param message   The incoming message
     * @return The original message or a message that should be used instead
     */
    Object controlMessageArrived(DataflowProcessor processor, DataflowReadChannel<Object>
channel, int index, Object message);

    /**
     * Invoked when a message is being bound to an output channel.
     *
     * @param processor The reporting dataflow operator/selector
     * @param channel   The output channel to send the message to
     * @param index     The index of the output channel within the operator
     * @param message   The message to send
     * @return The original message or a message that should be used instead
     */
    Object messageSentOut(DataflowProcessor processor, DataflowWriteChannel<Object>
channel, int index, Object message);

    /**
     * Invoked when all messages required to trigger the operator become available in the
input channels.
     *
     * @param processor The reporting dataflow operator/selector
     * @param messages  The incoming messages
     * @return The original list of messages or a modified/new list of messages that
should be used instead
     */
    List<Object> beforeRun(DataflowProcessor processor, List<Object> messages);

    /**
     * Invoked when the operator completes a single run
     *
     * @param processor The reporting dataflow operator/selector
     * @param messages  The incoming messages that have been processed
     */
    void afterRun(DataflowProcessor processor, List<Object> messages);

    /**
     * Invoked when the fireCustomEvent() method is triggered manually on a dataflow
```

```
operator/selector
     *
     * @param processor The reporting dataflow operator/selector
     * @param data      The custom piece of data provided as part of the event
     * @return A value to return from the fireCustomEvent() method to the caller (event
initiator)
     */
    Object customEvent(DataflowProcessor processor, Object data);
}
```

A default implementation is provided through the *DataflowEventAdapter* class.

Listeners provide a way to handle exceptions, when they occur inside operators. A listener may typically log such exceptions, notify a supervising entity, generate an alternative output or perform any steps required to recover from the situation. If there's no listener registered or if any of the listeners returns *true* the operator will terminate, preserving the contract of *afterStop()* . Exceptions that occur outside the actual operator's body, i.e. at the parameter preparation phase before the body is triggered or at the clean-up and channel subscription phase, after the body finishes, always lead to operator termination.

The *fireCustomEvent()* method available on operators and selectors may be used to communicate back and forth between operator's body and the interested listeners:

A Sample

```
final listener = new DataflowEventAdapter() {
    @Override
    Object customEvent(DataflowProcessor processor, Object data) {
        println "Log: Getting quite high on the scale $data"
        return 100  //The value to use instead
    }
}

op = group.operator(inputs: [a, b], outputs: [c], listeners: [listener]) {x, y ->
    final sum = x + y
    if (sum > 100) bindOutput(fireCustomEvent(sum))  //Reporting that the sum is too
high, binding the lowered value that comes back
    else bindOutput sum
}
```

Selectors

Selector's body should be a closure consuming either one or two arguments.

A Sample

```
selector (inputs : [a, b, c], outputs : [d, e]) {value ->
    ....
}
```

The two-argument closure will get a value plus an index of the input channel, the value of which is currently being processed. This allows the selector to distinguish between values coming through different input channels.

A Sample

```
selector (inputs : [a, b, c], outputs : [d, e]) {value, index ->
    ....
}
```

Priority Selector

When priorities need to be preserved among input channels, a *DataflowPrioritySelector* should be used.

A Sample

```
prioritySelector(inputs : [a, b, c], outputs : [d, e]) {value, index ->
    ...
}
```

The priority selector will always prefer values from channels with lower position index over values coming through the channels with higher position index.

Join Selector

A selector without a body closure specified will copy all incoming values to all of its output channels.

A Sample

```
def join = selector (inputs : [programmers, analysis, managers], outputs : [employees,
colleagues])
```

Internal Parallelism

The *maxForks* attribute allowing for internal selectors parallelism is also available.

A Sample

```
selector (inputs : [a, b, c], outputs : [d, e], maxForks : 5) {value ->
    ....
}
```

Guards

Just like *Selects* , *Selectors* also allow the users to temporarily include/exclude individual input channels from selection. The *guards* input property can be used to set the initial mask on all input channels and the *setGuards* and *setGuard* methods are then available in the selector's body.

A Sample

```
import groovyx.gpars.dataflow.DataflowQueue
import static groovyx.gpars.dataflow.Dataflow.selector
import static groovyx.gpars.dataflow.Dataflow.task

/**
 * Demonstrates the ability to enable/disable channels during a value selection on a
select by providing boolean guards.
 */
final DataflowQueue operations = new DataflowQueue()
final DataflowQueue numbers = new DataflowQueue()

def instruction
def nums = []

selector(inputs: [operations, numbers], outputs: [], guards: [true, false]) {value, index
->   //initial guards is set here
    if (index == 0) {
        instruction = value
        setGuard(0, false)  //setGuard() used here
        setGuard(1, true)
    }
    else nums << value
    if (nums.size() == 2) {
        setGuards([true, false])                                    //setGuards() used
here
        final def formula = "${nums[0]} $instruction ${nums[1]}"
        println "$formula = ${new GroovyShell().evaluate(formula)}"
        nums.clear()
    }
}

task {
    operations << '+'
    operations << '+'
    operations << '*'
}

task {
    numbers << 10
    numbers << 20
    numbers << 30
    numbers << 40
    numbers << 50
    numbers << 60
}
```

Warning

Avoid combining *guards* and *maxForks* greater than 1. Although the *Selector* is thread-safe and won't be damaged in any way, the guards are likely not to be set the way you expect. The multiple threads running the selector's body concurrently will tend to over-write each-other's settings to the *guards* property.

Shutting Down Dataflow Networks

Shutting down a network of dataflow processors (operators and selectors) may sometimes be a non-trivial task, especially if you need a generic mechanism that will not leave any messages unprocessed.

Dataflow operators and selectors can be terminated in three ways:

- by calling the terminate() method on all operators that need to be terminated
- by sending a poisson message
- by setting up a network of activity monitors that will shutdown the network after all messages have been processed

Check out the details on the ways that **GPars** provides.

Shutting down the thread pool

If you use a custom *PGroup* to maintain a thread pool for your dataflow network, you should not forget to shutdown the pool once the network is terminated. Otherwise the thread pool will consume system resources and, in case of using non-daemon threads, it will prevent JVM from exit.

Emergency Shutdown

You can call *terminate()* on any operator/selector to immediately shut it down. Provided you keep track of all your processors, perhaps by adding them to a list, the fastest way to stop the network would be:

A Sample

```
allMyProcessors*.terminate()
```

This should, however, be treated as an emergency exit, since no guarantees can be given regarding messages processed nor finished work. Operators will simply terminate instantly leaving work unfinished and abandoning messages in the input channels. Certainly, the lifecycle event listeners hooked to the operators/selectors will have their *afterStop()* event handlers invoked in order to, for example, release resources or output a note into the log.

A Sample

```
def op1 = operator(inputs: [a, b, c], outputs: [d, e]) {x, y, z -> }

def op2 = selector(inputs: [d], outputs: [f, out]) { }

def op3 = prioritySelector(inputs: [e, f], outputs: [b]) {value, index -> }

[op1, op2, op3]*.terminate()  //Terminate all operators by calling the terminate() method
on them
op1.join()
op2.join()
op3.join()
```

Shutting down the whole JVM through *System.exit()* will obviously shutdown the dataflow network, however, no lifecycle listeners will be invoked.

Stopping Operators Gently

Operators handle incoming messages repeatedly. The only safe moment for stopping an operator without the risk of loosing any messages is right after the operator has finished processing messages and is just about to look for more messages in its incoming pipes. This is exactly what the *terminateAfterNextRun()* method does. It will schedule the operator for shutdown after the next set of messages gets handled.

The unprocessed messages will stay in the input channels, which allows you to handle them later, perhaps with a different operator/selector or in some other way. Using *terminateAfterNextRun()* you will not loose any input messages. This may be particularly handy when you use a group of operators/selectors to load-balance messages coming from a channel. Once the work-load decreases, the terminateAfterNextRun() method may be used to safely reduce the pool of load-balancing operators.

Detecting shutdown

Operators and electors offer a handy *join()* method for those who need to block until the operator terminates.

A Sample

```
allMyProcessors*.join()
```

This is the easies way to wait until the whole dataflow network shuts down, irrespective of the

shutdown method used.

PoisonPill

PoisonPill is a common term for a strategy that uses special-purpose messages to stop entities that receive it. **GPars** offers the *PoisonPill* class, which has exactly such effect or operators and selectors. Since *PoisonPill* is a *ControlMessage*, it is invisible to operator's body and custom code does not need to handle it in any way. *DataflowEventListeners* may react to *ControlMessages* through the *controlMessageArrived()* handler method.

A Sample

```
def op1 = operator(inputs: [a, b, c], outputs: [d, e]) {x, y, z -> }

def op2 = selector(inputs: [d], outputs: [f, out]) { }

def op3 = prioritySelector(inputs: [e, f], outputs: [b]) {value, index -> }

a << PoisonPill.instance  //Send the poisson

op1.join()
op2.join()
op3.join()
```

After receiving a poisson an operator terminates, right after it finishes the current calculation and makes sure the poisson is sent to all its output channels, so that the poisson can spread to the connected operators. Also, although operators typically wait for all inputs to have a value, in case of *PoisonPills*, the operator will terminate immediately as soon as a *PoisonPill* appears on any of its inputs. The values already obtained from the other channels will be lost. It can be considered an error in the design of the network, if these messages were supposed to be processed. They would need a proper value as their peer and not a PoisonPill in order to be processes normally.

Selectors, on the other hand, will patiently wait for *PoisonPill* to be received from all their input channels before sending it on the the output channels. This behavior prevents networks containing **feed-back loops involving selectors** from being shutdown using *PoisonPill* . A selector would never receive a *PoisonPill* from the channel that comes back from behind the selector. A different shutdown strategy should be used for such networks.

Operators and Selectors Should Only Terminate Themselves

Given the potential variety of operator networks and their asynchronous nature, a good termination strategy is that operators and selectors should only ever terminate themselves. All ways of terminating them from outside (either by calling the terminate() method or by sending poisson down the stream) may result in messages being lost somewhere in the pipes, when the reading operators terminate before they fully handle the messages waiting in their input channels.

Immediate Poison Pill

Especially for selectors to shutdown immediately after receiving a poison pill, a notion of **immediate poison pill** has been introduced. Since normal, non-immediate poison pills merely close the input channel leaving the selector alive until at least one input channel remains open, the immediate poison pill closes the selector instantly. Obviously, unprocessed messages from the other selector's input channels will not be handled by the selector, once it reads an immediate poison pill.

With immediate poison pill you can safely shutdown networks with selectors involved in feedback loops.

A Sample

```
def op1 = selector(inputs: [a, b, c], outputs: [d, e]) {value, index -> }
def op2 = selector(inputs: [d], outputs: [f, out]) { }
def op3 = prioritySelector(inputs: [e, f], outputs: [b]) {value, index -> }

a << PoisonPill.immediateInstance

[op1, op2, op3]*.join()
```

Poison With Counting

When sending a poison pill down the operator network you may need to be notified when all the operators or a specified number of them have been stopped. The *CountingPoisonPill* class serves exactly this purpose:

A Sample

```
operator(inputs: [a, b, c], outputs: [d, e]) {x, y, z -> }
selector(inputs: [d], outputs: [f, out]) { }
prioritySelector(inputs: [e, f], outputs: [b]) {value, index -> }

//Send the poisson indicating the number of operators than need to be terminated before
we can continue
final pill = new CountingPoisonPill(3)
a << pill

//Wait for all operators to terminate
pill.join()
//At least 3 operators should be terminated by now
```

The *termination* property of the *CountingPoisonPill* class is a regular *Promise<Boolean>* and so has a lot of handy properties.

A Sample

```
//Send the poisson indicating the number of operators than need to be terminated before
we can continue
final pill = new CountingPoisonPill(3)
pill.termination.whenBound {println "Reporting asynchronously that the network has been
stopped"}
a << pill

if (pill.termination.bound) println "Wow, that was quick. We are done already!"
else println "Things are being slow today. The network is still running."

//Wait for all operators to terminate
assert pill.termination.get()
//At least 3 operators should be terminated by now
```

An immediate variant of *CountingPoisonPill* is also available - *ImmediateCountingPoisonPill* .

A Sample

```
def op1 = selector(inputs: [a, b, c], outputs: [d, e]) {value, index -> }
def op2 = selector(inputs: [d], outputs: [f, out]) { }
def op3 = prioritySelector(inputs: [e, f], outputs: [b]) {value, index -> }

final pill = new ImmediateCountingPoisonPill(3)
a << pill
pill.join()
```

ImmediateCountingPoisonPill will safely and instantly shutdown dataflow networks even with selectors involved in feedback loops, which normal non-immediate poison pill would not be able to.

Poison Strategies

To correctly shutdown a network using *PoisonPill* you must identify the appropriate set of channels to send *PoisonPill* to. *PoisonPill* will spread in the network the usual way through the channels and processors down the stream. Typically the right channels to send *PoisonPill* to will be those that serve as **data sources** for the network. This may be difficult to achieve for general cases or for complex networks. On the other hand, for networks with a prevalent direction of message flow *PoisonPill* provides a very straightforward way to shutdown the whole network gracefully.

Load-balancing Prevents Poison Shutdown

Load-balancing architectures, which use multiple operators reading messages off a shared channel (queue), will also prevent poison shutdown to work properly, since only one of the reading operators will get to read the poison message. You may consider using **forked operators** instead, by setting the *maxForks* property to a value greater than 1. Another alternative is to manually split the message stream into multiple channels, each of which would be consumed by one of the original operators.

Termination Tips and Tricks

Notice that **GPars** *tasks* return a *DataflowVariable*, which gets bound to a value as soon as the task finishes. The 'terminator' operator below leverages the fact that *DataflowVariables* are implementations of the *DataflowReadChannel* interface and thus can be consumed by operators. As soon as both tasks finish, the operator will send a *PoisonPill* down the *q* channel to stop the consumer as soon as it processes all data.

A Sample

```
import groovyx.gpars.dataflow.DataflowQueue
import groovyx.gpars.group.NonDaemonPGroup


def group = new NonDaemonPGroup()

final DataflowQueue q = new DataflowQueue()

// final destination
def customs = group.operator(inputs: [q], outputs: []) { value ->
    println "Customs received $value"
}

// big producer
def green = group.task {
    (1..100).each {
        q << 'green channel ' + it
        sleep 10
    }
}

// little producer
def red = group.task {
    (1..10).each {
        q << 'red channel ' + it
        sleep 15
    }
}

def terminator = group.operator(inputs: [green, red], outputs: []) { t1, t2 ->
    q << PoisonPill.instance
}

customs.join()
group.shutdown()
```

Keeping PoisonPill Inside a Given Network

If your network passed values through channels to entities outside of it, you may need to stop the *PoisonPill* messages on the network boundaries. This can be easily achieved by putting a single-input single-output filtering operator on each such channel.

A Sample

```
operator(networkLeavingChannel, otherNetworkEnteringChannel) {value ->
    if (!(value instanceOf PoisonPill)) bindOutput it
}
```

The *Pipeline* DSL may be also helpful here:

A Sample

```
networkLeavingChannel.filter { !(it instanceOf PoisonPill) } into
otherNetworkEnteringChannel
```

Check out the *Pipeline DSL* section to find out more on pipelines.

Graceful Shutdown

GPars provides a generic way to shutdown a dataflow network. Unlike the previously mentioned mechanisms this approach will keep the network running until all the messages get handled and than gracefully shuts all operators down letting you know when this happens. You have to pay a modest performance penalty, though. This is unavoidable since we need to keep track of what's happening inside the network.

A Sample

```
import groovyx.gpars.dataflow.DataflowBroadcast
import groovyx.gpars.dataflow.DataflowQueue
import groovyx.gpars.dataflow.operator.component.GracefulShutdownListener
import groovyx.gpars.dataflow.operator.component.GracefulShutdownMonitor
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.group.PGroup

PGroup group = new DefaultPGroup(10)
final a = new DataflowQueue()
final b = new DataflowQueue()
final c = new DataflowQueue()
final d = new DataflowQueue<Object>()
final e = new DataflowBroadcast<Object>()
final f = new DataflowQueue<Object>()
final result = new DataflowQueue<Object>()

final monitor = new GracefulShutdownMonitor(100);
```

```
def op1 = group.operator(inputs: [a, b], outputs: [c], listeners: [new
GracefulShutdownListener(monitor)]) {x, y ->
    sleep 5
    bindOutput x + y
}
def op2 = group.operator(inputs: [c], outputs: [d, e], listeners: [new
GracefulShutdownListener(monitor)]) {x ->
    sleep 10
    bindAllOutputs 2*x
}
def op3 = group.operator(inputs: [d], outputs: [f], listeners: [new
GracefulShutdownListener(monitor)]) {x ->
    sleep 5
    bindOutput x + 40
}
def op4 = group.operator(inputs: [e.createReadChannel(), f], outputs: [result],
listeners: [new GracefulShutdownListener(monitor)]) {x, y ->
    sleep 5
    bindOutput x + y
}

100.times{a << 10}
100.times{b << 20}

final shutdownPromise = monitor.shutdownNetwork()

100.times{assert 160 == result.val}

shutdownPromise.get()
[op1, op2, op3, op4]*.join()

group.shutdown()
```

First, we need an instance of *GracefulShutdownMonitor* , which will orchestrate the shutdown process. It relies on instances of *GracefulShutdownListener* attached to all operators/selectors. These listeners observe their respective processors together with their input channels and report to the shared *GracefulShutdownMonitor*. Once *shutdownNetwork()* is called on *GracefulShutdownMonitor* , it will periodically check for reported activities, query the state of operators as well as the number of messages in their input channels.

Please make sure that no new messages enter the dataflow network after the shutdown has been initiated, since this may cause the network to never terminate. The shutdown process should only be started after all data producers have ceased sending additional messages to the monitored network.

The *shutdownNetwork()* method returns a **Promise** so that you can do the usual set of tricks with it - block waiting for the network to terminate using the *get()* method, register a callback using the

whenBound() method or make it trigger a whole set of activities through the *then()* method.

Limitations of Graceful Sshutdown

- For *GracefulShutdownListener* to work correctly, its *messageArrived()* event handler must see the original value that has arrived through the input channel. Since some event listeners may alter the messages as they pass through the listeners it is advisable to add the *GracefulShutdownListener* first to the list of listeners on each dataflow processor.
- Also, graceful shutdown will not work for those rare operators that have listeners, which turn control messages into plain value messages in the *controlMessageArrived()* event handler.
- Third and last, load-balancing architectures, which use multiple operators reading messages off a shared channel (queue), will also prevent graceful shutdown to work properly. You may consider using **forked operators** instead, by setting the *maxForks* property to a value greater than 1. Another alternative is to manually split the message stream into multiple channels, each of which would be consumed by one of the original operators.

Application Frameworks

Dataflow Operators and Selectors can be successfully used to build high-level domain-specific frameworks for problems that naturally fit the flow model.

Building Flow Frameworks on Top of GPars Dataflow

GPars dataflow can be viewed as bottom-line language-level infrastructure. Operators, selectors, channels and event listeners can be very useful at language level to combine, for example, with **actors** or parallel collections. Whenever a need comes for asynchronous handling of events that come through one of more channels, a dataflow operator or a small dataflow network could be a very good fit. Unlike tasks, operators are lightweight and release threads when there's no message to process. Unlike **actors**, operators are addressed indirectly through channels and may easily combine messages from multiple channels into one action.

Alternatively, operators can be looked at as continuous functions, which instantly and repeatedly transform their input values into output. We believe that a concurrency-friendly general-purpose programming language should provide this type of abstraction.

At the same time, dataflow elements can be easily used as building blocks for constructing domain-specific workflow-like frameworks. These frameworks can offer higher-level abstractions specialized to a single problem domain, which would be inappropriate for a general-purpose language-level library. Each of the higher-level concepts is then mapped to (potentially several) **GPars** concepts.

For example, a network solving data-mining problems may consist of several data sources, data cleaning nodes, categorization nodes, reporting nodes and others. Image processing network, on the other hand, may need nodes specialized in image compression and format transformation. Similarly, networks for data encryption, mp3 encoding, work-flow management as well as many other domains that would benefit from dataflow-based solutions, will differ in many aspects - the type of nodes in the network, the type and frequency of events, the load-balancing scheme, potential constraints on branching, the need for visualization, debugging and logging, the way users define the networks and interact with them as well as many others.

The higher-level application-specific frameworks should put effort into providing abstractions best suited for the given domain and hide **GPars** complexities. For example, the visual graph of the network that the user manipulates on the screen should typically not show all the channels that participate in the network. Debugging or logging channels, which rarely contribute to the core of the solution, are among the first good candidates to consider for exclusion. Also channels and lifecycle-event listeners, which orchestrate aspects such as load balancing or graceful shutdown, will probably be not exposed to the user, although they will be part of the generated and executed network. Similarly, a single channel in the domain-specific model will in reality translate into multiple channels perhaps with one or more logging/transforming/filtering operators connecting them together. The function associated with a node will most likely be wrapped with some additional infrastructural code to form the operator's body.

GPars gives you the underlying components that the end user may be abstracted away completely by the application-specific framework. This keeps **GPars** domain-agnostic and universal, yet useful at the implementation level.

Pipeline DSL

A DSL for Building Operators Pipelines

Building dataflow networks can be further simplified. **GPars** offers handy shortcuts for the common scenario of building (mostly linear) pipelines of operators.

A Sample

```
def toUpperCase = {s -> s.toUpperCase()}

final encrypt = new DataflowQueue()
final DataflowReadChannel encrypted = encrypt | toUpperCase | {it.reverse()} | {
'###encrypted###' + it + '###'}

encrypt << "I need to keep this message secret!"
encrypt << "GPars can build linear operator pipelines really easily"

println encrypted.val
println encrypted.val
```

This saves you from directly creating, wiring and manipulating all the channels and operators that are to form the pipeline. The *pipe* operator lets you hook an output of one function/operator/process to the input of another one. Just like chaining system processes on the command line.

The *pipe* operator is a handy shorthand for a more generic *chainWith()* method:

A Sample

```
def toUpperCase = {s -> s.toUpperCase()}

final encrypt = new DataflowQueue()
final DataflowReadChannel encrypted = encrypt.chainWith toUpperCase chainWith {it.
reverse()} chainWith {'###encrypted###' + it + '###'}

encrypt << "I need to keep this message secret!"
encrypt << "GPars can build linear operator pipelines really easily"

println encrypted.val
println encrypted.val
```

Combining Pipelines with Straight Operators

Since each operator pipeline has an entry and an exit channel, pipelines can be wired into more complex operator networks. Only your imagination can limit your ability to mix pipelines with channels and operators in the same network definitions.

A Sample

```
def toUpperCase = {s -> s.toUpperCase()}
def save = {text ->
    //Just pretending to be saving the text to disk, database or whatever
    println 'Saving ' + text
}

final toEncrypt = new DataflowQueue()
final DataflowReadChannel encrypted = toEncrypt.chainWith toUpperCase chainWith {it
.reverse()} chainWith {'###encrypted###' + it + '###'}

final DataflowQueue fork1 = new DataflowQueue()
final DataflowQueue fork2 = new DataflowQueue()
splitter(encrypted, [fork1, fork2])  //Split the data flow

fork1.chainWith save  //Hook in the save operation

//Hook in a sneaky decryption pipeline
final DataflowReadChannel decrypted = fork2.chainWith {it[15..-4]} chainWith {it.reverse
()} chainWith {it.toLowerCase()}
      .chainWith {'Groovy leaks! Check out a decrypted secret message: ' + it}

toEncrypt << "I need to keep this message secret!"
toEncrypt << "GPars can build operator pipelines really easy"

println decrypted.val
println decrypted.val
```

Type of Channel Preservation

The type of the channel is preserved across the whole pipeline. E.g. if you start chaining off a synchronous channel, all the channels in the pipeline will be synchronous. In that case, obviously, the whole chain blocks, including the writer who writes into the channel at head, until someone reads data off the tail of the pipeline.

A Sample

```
final SyncDataflowQueue queue = new SyncDataflowQueue()
final result = queue.chainWith {it * 2}.chainWith {it + 1} chainWith {it * 100}

Thread.start {
    5.times {
        println result.val
    }
}

queue << 1
queue << 2
queue << 3
queue << 4
queue << 5
```

Joining Pipelines

Two pipelines (or channels) can be connected using the *into()* method:

A Sample

```
final encrypt = new DataflowQueue()
final DataflowWriteChannel messagesToSave = new DataflowQueue()
encrypt.chainWith toUpperCase chainWith {it.reverse()} into messagesToSave

task {
    encrypt << "I need to keep this message secret!"
    encrypt << "GPars can build operator pipelines really easy"
}

task {
    2.times {
        println "Saving " + messagesToSave.val
    }
}
```

The output of the *encryption* pipeline is directly connected to the input of the *saving* pipeline (a single channel in out case).

Forking the Dataflow

When a need comes to copy the output of a pipeline/channel into more than one following

pipeline/channel, the *split()* method will help you:

A Sample

```
final encrypt = new DataflowQueue()
final DataflowWriteChannel messagesToSave = new DataflowQueue()
final DataflowWriteChannel messagesToLog = new DataflowQueue()

encrypt.chainWith toUpperCase chainWith {it.reverse()}.split(messagesToSave,
messagesToLog)
```

Tapping into a Pipeline

Like *split()* the *tap()* method allows you to fork the data flow into multiple channels. Tapping, however, is slightly more convenient in some scenarios, since it treats one of the two new forks as the successor of the pipeline.

A Sample

```
queue.chainWith {it * 2}.tap(logChannel).chainWith{it + 1}.tap(logChannel).into
(PrintChannel)
```

Merging Channels

Merging allows you to join multiple read channels as inputs for a single dataflow operator. The function passed as the second argument needs to accept as many arguments as there are channels being merged - each will hold a value of the corresponding channel.

A Sample

```
maleChannel.merge(femaleChannel) {m, f -> m.marry(f)}.into(mortgageCandidatesChannel)
```

Separation

Separation is the opposite operation to *merge*. The supplied closure returns a list of values, each of which will be output into an output channel with the corresponding position index.

A Sample

```
queue1.separate([queue2, queue3, queue4]) {a -> [a-1, a, a+1]}
```

Choices

The *binaryChoice()* and *choice()* methods allow you to send a value to one out of two (or many) output channels, as indicated by the return value from a closure.

A Sample

```
queue1.binaryChoice(queue2, queue3) {a -> a > 0}
queue1.choice([queue2, queue3, queue4]) {a -> a % 3}
```

Filtering

The *filter()* method allows to filter data in the pipeline using boolean predicates.

A Sample

```
        final DataflowQueue queue1 = new DataflowQueue()
        final DataflowQueue queue2 = new DataflowQueue()

        final odd = {num -> num % 2 != 0 }

        queue1.filter(odd) into queue2
        (1..5).each {queue1 << it}
        assert 1 == queue2.val
        assert 3 == queue2.val
        assert 5 == queue2.val
```

Null Values

If a chained function returns a *null* value, it is normally passed along the pipeline as a valid value. To indicate to the operator that no value should be passed further down the pipeline, a *NullObject.nullObject* instance must be returned.

A Sample

```
        final DataflowQueue queue1 = new DataflowQueue()
        final DataflowQueue queue2 = new DataflowQueue()

        final odd = {num ->
            if (num == 5) return null  //null values are normally passed on
            if (num % 2 != 0) return num
            else return NullObject.nullObject  //this value gets blocked
        }

        queue1.chainWith odd into queue2
        (1..5).each {queue1 << it}
        assert 1 == queue2.val
        assert 3 == queue2.val
        assert null == queue2.val
```

Customizing Thread Pools

All of the Pipeline DSL methods allow for custom thread pools or *PGroups* to be specified:

A Sample

```
channel | {it * 2}

channel.chainWith(closure)
channel.chainWith(pool) {it * 2}
channel.chainWith(group) {it * 2}

channel.into(otherChannel)
channel.into(pool, otherChannel)
channel.into(group, otherChannel)

channel.split(otherChannel1, otherChannel2)
channel.split(otherChannels)
channel.split(pool, otherChannel1, otherChannel2)
channel.split(pool, otherChannels)
channel.split(group, otherChannel1, otherChannel2)
channel.split(group, otherChannels)

channel.tap(otherChannel)
channel.tap(pool, otherChannel)
channel.tap(group, otherChannel)

channel.merge(otherChannel)
channel.merge(otherChannels)
channel.merge(pool, otherChannel)
channel.merge(pool, otherChannels)
channel.merge(group, otherChannel)
channel.merge(group, otherChannels)

channel.filter( otherChannel)
channel.filter(pool, otherChannel)
channel.filter(group, otherChannel)

channel.binaryChoice( trueBranch, falseBranch)
channel.binaryChoice(pool, trueBranch, falseBranch)
channel.binaryChoice(group, trueBranch, falseBranch)

channel.choice( branches)
channel.choice(pool, branches)
channel.choice(group, branches)

channel.separate( outputs)
channel.separate(pool, outputs)
channel.separate(group, outputs)
```

Overriding the Default PGroup

To avoid the necessity to specify PGroup for each Pipeline DSL method separately you may override the value of the default Dataflow PGroup.

A Sample

```
Dataflow.usingGroup(group) {
    channel.choice(branches)
}
//Is identical to
channel.choice(group, branches)
```

The *Dataflow.usingGroup()* method resets the value of the default dataflow PGroup for the given code block to the value specified.

The Pipeline Builder

The *Pipeline* class offers an intuitive builder for operator pipelines. The greatest benefit of using the *Pipeline* class compared to chaining the channels directly is the ease with which a custom thread pool/group can be applied to all the operators along the constructed chain. The available methods and overloaded operators are identical to the ones available on channels directly.

The greatest benefit of using the *Pipeline* class is easy usage

---

A Sample

```
import groovyx.gpars.dataflow.DataflowQueue
import groovyx.gpars.dataflow.operator.Pipeline
import groovyx.gpars.scheduler.DefaultPool
import groovyx.gpars.scheduler.Pool

final DataflowQueue queue = new DataflowQueue()
final DataflowQueue result1 = new DataflowQueue()
final DataflowQueue result2 = new DataflowQueue()
final Pool pool = new DefaultPool(false, 2)

final negate = {-it}

final Pipeline pipeline = new Pipeline(pool, queue)

pipeline | {it * 2} | {it + 1} | negate
pipeline.split(result1, result2)

queue << 1
queue << 2
queue << 3

assert -3 == result1.val
assert -5 == result1.val
assert -7 == result1.val

assert -3 == result2.val
assert -5 == result2.val
assert -7 == result2.val

pool.shutdown()
```

Passing Construction Parameters Through the Pipeline DSL

You are likely to frequently need the ability to pass additional initialization parameters to the operators, such as the listeners to attach or the value for *maxForks*. Just like when building operators directly, the Pipeline DSL methods accept an optional map of parameters to pass in.

A Sample

```
new Pipeline(group, queue1).merge([maxForks: 4, listeners: [listener]], queue2) {a, b ->
a + b}.into queue3
```

Implementation

The Dataflow Concurrency in **GPars** builds on the same principles as the **Actor** support. All of the dataflow tasks share a thread pool and so the number threads created through *Dataflow.task()* factory method don't need to correspond to the number of physical threads required from the system. The *PGroup.task()* factory method can be used to attach the created task to a group. Since each group defines its own thread pool, you can easily organize tasks around different thread pools just like you do with **actors**.

Combining Actors and `Dataflow Concurrency`

The good news is that you can combine **actors** and `Dataflow Concurrency` in any way you feel fit for your particular problem at hands. You can freely you use Dataflow Variables from **Actors**.

A Sample

```
final DataflowVariable a = new DataflowVariable()

final Actor doubler = Actors.actor {
    react {message->
        a << 2 * message
    }
}

final Actor fakingDoubler = actor {
    react {
        doubler.send it  //send a number to the doubler
        println "Result ${a.val}"  //wait for the result to be bound to 'a'
    }
}

fakingDoubler << 10
```

In the example you see the "fakingDoubler" using both messages and a *DataflowVariable* to communicate with the *doubler* **Actor**.

Using Plain Java Threads

The *DataflowVariable* as well as the *DataflowQueue* classes can obviously be used from any thread of your application, not only from the tasks created by *Dataflow.task()* . Consider the following example:

A Sample

```
import groovyx.gpars.dataflow.DataflowVariable

final DataflowVariable a = new DataflowVariable<String>()
final DataflowVariable b = new DataflowVariable<String>()

Thread.start {
    println "Received: $a.val"
    Thread.sleep 2000
    b << 'Thank you'
}

Thread.start {
    Thread.sleep 2000
    a << 'An important message from the second thread'
    println "Reply: $b.val"
}
```

We're creating two plain *java.lang.Thread* instances, which exchange data using the two data flow variables. Obviously, neither the **Actor** lifecycle methods, nor the send/react functionality or thread pooling take effect in such scenarios.

Synchronous Variables and Channels

When using asynchronous dataflow channels, apart from the fact that readers have to wait for a value to be available for consumption, the communicating parties remain completely independent. Writers don't wait for their messages to get consumed. Readers obtain values immediately as they come and ask. Synchronous channels, on the other hand, can synchronize writers with the readers as well as multiple readers among themselves. This is particularly useful when you need to increase the level of determinism. The writer-to-reader partial ordering imposed by asynchronous communication is complemented with reader-to-writer partial ordering, when using synchronous communication. In other words, you are guaranteed that whatever the reader did before reading a value from a synchronous channel preceded whatever the writer did after writing the value. Also, with synchronous communication writers can never get too far ahead of readers, which simplifies reasoning about the system and reduces the need to manage data production speed in order to avoid system overload.

Synchronous Dataflow Queue

The *SyncDataflowQueue* class should be used for point-to-point (1:1 or n:1) communication. Each message written to the queue will be consumed by exactly one reader. Writers are blocked until their message is consumed, readers are blocked until there's a value available for them to read.

A Sample

```
import groovyx.gpars.dataflow.SyncDataflowQueue
import groovyx.gpars.group.NonDaemonPGroup

/**
 * Shows how synchronous dataflow queues can be used to throttle fast producer when
serving data to a slow consumer.
 * Unlike when using asynchronous channels, synchronous channels block both the writer
and the readers until all parties are ready to exchange messages.
 */

def group = new NonDaemonPGroup()

final SyncDataflowQueue channel = new SyncDataflowQueue()

def producer = group.task {
    (1..30).each {
        channel << it
        println "Just sent $it"
    }
    channel << -1
}

def consumer = group.task {
    while (true) {
        sleep 500  //simulating a slow consumer
        final Object msg = channel.val
        if (msg == -1) return
        println "Received $msg"
    }
}

consumer.join()

group.shutdown()
```

Synchronous Dataflow Broadcast

The *SyncDataflowBroadcast* class should be used for publish-subscribe (1:n or n:m) communication. Each message written to the broadcast will be consumed by all subscribed readers. Writers are blocked until their message is consumed by all readers, readers are blocked until there's a value available for them to read and all the other subscribed readers ask for the message as well. With *SyncDataflowBroadcast* you get all readers processing the same message at the same time and waiting

for one-another before getting the next one.

A Sample

```
import groovyx.gpars.dataflow.SyncDataflowBroadcast
import groovyx.gpars.group.NonDaemonPGroup

/**
 * Shows how synchronous dataflow broadcasts can be used to throttle fast producer when
serving data to slow consumers.
 * Unlike when using asynchronous channels, synchronous channels block both the writer
and the readers until all parties are ready to exchange messages.
 */

def group = new NonDaemonPGroup()

final SyncDataflowBroadcast channel = new SyncDataflowBroadcast()

def subscription1 = channel.createReadChannel()
def fastConsumer = group.task {
    while (true) {
        sleep 10  //simulating a fast consumer
        final Object msg = subscription1.val
        if (msg == -1) return
        println "Fast consumer received $msg"
    }
}

def subscription2 = channel.createReadChannel()
def slowConsumer = group.task {
    while (true) {
        sleep 500  //simulating a slow consumer
        final Object msg = subscription2.val
        if (msg == -1) return
        println "Slow consumer received $msg"
    }
}

def producer = group.task {
    (1..30).each {
        println "Sending $it"
        channel << it
        println "Sent $it"
    }
    channel << -1
}

[fastConsumer, slowConsumer]*.join()
```

```
group.shutdown()
```

Synchronous Dataflow Variable

Unlike *DataflowVariable*, which is asynchronous and only blocks the readers until a value is bound to the variable, the *SyncDataflowVariable* class provides a one-shot data exchange mechanism that blocks the writer and all readers until a specified number of waiting parties is reached.

A Sample

```
import groovyx.gpars.dataflow.SyncDataflowVariable
import groovyx.gpars.group.NonDaemonPGroup

final NonDaemonPGroup group = new NonDaemonPGroup()

final SyncDataflowVariable value = new SyncDataflowVariable(2)  //two readers required to
exchange the message

def writer = group.task {
    println "Writer about to write a value"
    value << 'Hello'
    println "Writer has written the value"
}

def reader = group.task {
    println "Reader about to read a value"
    println "Reader has read the value: ${value.val}"
}

def slowReader = group.task {
    sleep 5000
    println "Slow reader about to read a value"
    println "Slow reader has read the value: ${value.val}"
}

[reader, slowReader]*.join()

group.shutdown()
```

Kanban Flow

APIs: [KanbanFlow|api:groovyx.gpars.dataflow.KanbanFlow] | [KanbanLink|api:groovyx.gpars.dataflow.KanbanLink] | [KanbanTray|api:groovyx.gpars.dataflow.KanbanTray] |

KanbanFlow

A *KanbanFlow* is a composed object that uses dataflow abstractions to define dependencies between multiple concurrent producer and consumer operators.

Each link between a producer and a consumer is defined by a *KanbanLink*.

Inside each KanbanLink, the communication between producer and consumer follows the KanbanFlow pattern as described in The KanbanFlow Pattern. They use objects of type *KanbanTray* to send products downstream and signal requests for further products back to the producer.

The figure below shows a *KanbanLink* with one producer, one consumer and five trays numbered 0 to 4. Tray number 0 has been used to take a product from producer to consumer, has been emptied by the consumer and is now sent back to the producer's input queue. Trays 1 and 2 wait carry products waiting for consumption, trays 3 and 4 wait to be used by producers.

A *KanbanFlow* object links producers to consumers thus creating *KanbanLink* objects. In the course of this activity, a second link may be constructed where the producer is the same object that acted as the consumer in a formerly created link such that the two links become connected to build a chain.

Here is an example of a *KanbanFlow* with only one link, e.g. one producer and one consumer. The producer always sends the number 1 downstream and the consumer prints this number.

A Sample

```
import static groovyx.gpars.dataflow.ProcessingNode.node
import groovyx.gpars.dataflow.KanbanFlow

def producer = node { down -> down 1 }
def consumer = node { up   -> println up.take() }

new KanbanFlow().with {
    link producer to consumer
    start()
    // run for a while
    stop()
}
```

For putting a product into a tray and sending the tray downstream, one can either use the @send()@

method, the @<<@ operator, or use the tray as a method object. The following lines are equivalent:

A Sample

```
node { down -> down.send 1 }
node { down -> down << 1 }
node { down -> down 1 }
```

When a product is taken from the input tray with the @take()@ method, the empty tray is automatically released.

You should call @take()@ only once!

If you prefer to not using an empty tray for sending products downstream (as typically the case when a *ProcessingNode* acts as a filter), you must release the tray in order to keep it in play. Otherwise, the number of trays in the system decreases. You can release a tray either by calling the @release()@ method or by using the @~@ operator (think "shake it off"). The following lines are equivalent:

A Sample

```
node { down -> down.release() }
node { down -> ~down }
```

Trays are automatically released, if you call any of the @take()@ or @send()@ methods.

Various Linking Structures

In addition to a linear chains, a *KanbanFlow* can also link a single producer to multiple consumers (tree) or multiple producers to a single consumer (collector) or any combination of the above that results in a directed acyclic graph (DAG).

The *KanbanFlowTest* class has many examples for such structures, including scenarios where a single producer delegates work to multiple consumers with:

- a **work-stealing** strategy where all consumers get their pick from the downstream,
- a **master-slave** strategy where a producer chooses from the available consumers, and
- a **broadcast** strategy where a producer sends all products to all consumers.

Cycles are forbidden by default but when enabled, they can be used as so-called generators. A producer can even be his own consumer that increases a product value in every cycle. The generator itself remains state-free since the value is only stored as a product riding on a tray. Such a generator can be used for e.g. lazy sequences or as a the "heartbeat" of a subsequent flow.

The approach of generator "loops" can equally be applied to collectors, where a collector does not maintain any internal state but sends a collection onto itself, adding products at each call.

Generally speaking, a *ProcessingNode* can link to itself for exporting state to the tray/product that it sends to itself. Access to the product is then **thread-safe by design**.

Composing KanbanFlows

Just as *KanbanLink* objects can be chained together to form a *KanbanFlow*, flows themselves can be composed again to form new greater flows from existing smaller ones.

A Sample

```
def firstFlow = new KanbanFlow()
def producer  = node(counter)
def consumer  = node(repeater)
firstFlow.link(producer).to(consumer)

def secondFlow = new KanbanFlow()
def producer2  = node(repeater)
def consumer2  = node(reporter)
secondFlow.link(producer2).to(consumer2)

flow = firstFlow + secondFlow

flow.start()
```

Customizing Concurrency Characteristics

The amount of concurrency in a kanban system is determined by the number of trays (sometimes called **WIP** = work in progress). With no trays in the streams, the system does nothing:

- With one tray only, the system is confined to sequential execution.
- With more trays, concurrency begins.
- With more trays than available processing units, the system begins to waste resources.

The number of trays can be controlled in various ways. They are typically set when starting the flow.

A Sample

```
flow.start(0) // start without trays
flow.start(1) // start with one tray per link in the flow
flow.start()  // start with the optimal number of trays
```

In addition to the trays, the *KanbanFlow* may also be constrained by its underlying *ThreadPool*. A pool of size 1 for example will not allow much concurrency.

KanbanFlows use a default pool that is dimensioned by the number of available cores. This can be customized by setting the @pooledGroup@ property.

Tests

- Kanban Flow Test in **Groovy**

Demos

- Demo Kanban Flow
- Demo Kanban Flow Broadcast
- Demo Kanban Flow Cycle
- Demo Kanban Lazy Prime Sequence Loops

Classic Examples

The Sieve of Eratosthenes Implementation using Dataflow Tasks

A Sample

```
import groovyx.gpars.dataflow.DataflowQueue
import static groovyx.gpars.dataflow.Dataflow.task

/**
 * Demonstrates concurrent implementation of the Sieve of Eratosthenes using dataflow
tasks
 */

final int requestedPrimeNumberCount = 1000

final DataflowQueue initialChannel = new DataflowQueue()

/**
 * Generating candidate numbers
 */
task {
    (2..10000).each {
        initialChannel << it
    }
}

/**
 * Chain a new filter for a particular prime number to the end of the Sieve
 * @param inChannel The current end channel to consume
 * @param prime The prime number to divide future prime candidates with
 * @return A new channel ending the whole chain
 */
def filter(inChannel, int prime) {
    def outChannel = new DataflowQueue()

    task {
        while (true) {
            def number = inChannel.val
            if (number % prime != 0) {
                outChannel << number
            }
        }
    }
```

```
    return outChannel
}

/**
 * Consume Sieve output and add additional filters for all found primes
 */
def currentOutput = initialChannel
requestedPrimeNumberCount.times {
    int prime = currentOutput.val
    println "Found: $prime"
    currentOutput = filter(currentOutput, prime)
}
```

The Sieve of Eratosthenes using both Dataflow Tasks and Operators

A Sample

```
        import groovyx.gpars.dataflow.DataflowQueue
        import static groovyx.gpars.dataflow.Dataflow.operator
        import static groovyx.gpars.dataflow.Dataflow.task

        /**
         * Demonstrates concurrent implementation of the Sieve of Eratosthenes using
dataflow tasks and operators
         */

        final int requestedPrimeNumberCount = 100

        final DataflowQueue initialChannel = new DataflowQueue()

        /**
         * Generating candidate numbers
         */
        task {
            (2..1000).each {
                initialChannel << it
            }
        }

        /**
         * Chain a new filter for a particular prime number to the end of the Sieve
         * @param inChannel The current end channel to consume
         * @param prime The prime number to divide future prime candidates with
         * @return A new channel ending the whole chain
         */
```

```
        def filter(inChannel, int prime) {
            def outChannel = new DataflowQueue()

            operator([inputs: [inChannel], outputs: [outChannel]]) {
                if (it % prime != 0) {
                    bindOutput it
                }
            }
            return outChannel
        }

        /**
         * Consume Sieve output and add additional filters for all found primes
         */
        def currentOutput = initialChannel
        requestedPrimeNumberCount.times {
            int prime = currentOutput.val
            println "Found: $prime"
            currentOutput = filter(currentOutput, prime)
        }
```