Table of contents

Groovy Parallel Systems

The GPars Framework - Reference Documentation

Authors: The Whole GPars Gang

Version: 1.1.0

Table of Contents

1 Introduction
1.1 Enter GPars
1.2 Credits
2 Getting Started
2.1 Downloading and Installing
2.2 A Hello World Example
2.3 Code conventions
2.4 Getting Set Up in an IDE
2.5 Applicability of Concepts
2.6 What's New
2.7 Java API - Using GPars from Java
3 Data Parallelism
3.1 Parallel Collections
3.1.1 GParsPool
3.1.2 GParsExecutorsPool
3.1.3 Memoize
3.2 Map-Reduce
3.3 Parallel Arrays
3.4 Asynchronous Invocation
3.5 Composable Asynchronous Functions
3.6 Fork-Join
3.7 Parallel Speculations
4 Groovy CSP
5 Actors
5.1 Actors Principles
5.2 Stateless Actors
5.3 Tips and Tricks
5.4 Active Objects
5.5 Classic Examples
6 Agents
7 Dataflow
7.1 Tasks

7.2 Selects
7.3 Operators
7.4 Shutting Down Dataflow Networks
7.5 Application Frameworks
7.6 Pipeline DSL
7.7 Implementation
7.8 Synchronous Variables and Channels
7.9 Kanban Flow
7.10 Classic Examples
8 STM
9 Google App Engine Integration
10 Tips
10.1 Performance
10.2 Integration into hosted environment
11 Conclusion

1 Introduction

The world of mainstream computing is changing rapidly these days. If you open the hood and loc your computer, you'll most likely see a dual-core processor there. Or a quad-core one, if you hav computer. We all now run our software on multi-processor systems. The code we write today anc probably never run on a single processor system: parallel hardware has become standard. Not s though, at least not yet. People still create single-threaded code, even though it will not be able t power of current and future hardware. Some developers experiment with low-level concurrency p and locks or synchronized blocks. However, it has become obvious that the shared-memory mult used at the application level causes more trouble than it solves. Low-level concurrency handling right, and it's not much fun either. With such a radical change in hardware, software inevitably ha dramatically too. Higher-level concurrency and parallelism concepts like map/reduce, fork/join, ac provide natural abstractions for different types of problem domains while leveraging the multi-cor

1.1 Enter GPars

Meet GPars - an open-source concurrency and parallelism library for Java and Groovy that gives high-level abstractions for writing concurrent and parallel code in Groovy (map/reduce, fork/join, a closures, actors, agents, dataflow concurrency and other concepts), which can make your Java a concurrent and/or parallel with little effort. With GPars your Java and/or Groovy code can easily u processors on the target system. You can run multiple calculations at the same time, request net parallel, safely solve hierarchical divide-and-conquer problems, perform functional style map/redu collection processing or build your applications around the actor or dataflow model.

The project is open sourced under the Apache 2 License . If you're working on a commercial, ope or any other type of software project in Groovy, download the binaries or integrate them from the get going. The way to writing highly concurrent and/or parallel Java and Groovy code is wide ope

1.2 Credits

This project could not have reached the point where it stands currently without all the great help a many individuals, who have devoted their time, energy and expertise to make GPars a solid prod people in the core team who should be mentioned:

- Václav Pech
- Dierk Koenig
- Alex Tkachman
- Russel Winder
- Paul King
- Jon Kerridge

Over time, many people have contributed their ideas, provided useful feedback or helped GPars There are many people in this group, too many to name them all, but let's list at least the most ac

- Hamlet d'Arcy
- Hans Dockter
- Guillaume Laforge
- Robert Fischer
- Johannes Link
- Graeme Rocher
- Alex Miller
- Jeff Gortatowsky
- Jií Kropáek

Many thanks to everyone!

2 Getting Started

Let's set out a few assumptions before we get started:

1. You know and use Groovy and Java: otherwise you'd not be investing your valuable time stu and parallelism library for Groovy and Java.
2. You definitely want to write your codes employing concurrency and parallelism using Groovy
3. If you are not using Groovy for your code, you are prepared to pay the inevitable verbosity ta
4. You target multi-core hardware with your code.
5. You appreciate that in concurrent and parallel code things can happen at any time, in any or with than one thing happening at once.

With those assumptions in place, we get started.

It's becoming more and more obvious that dealing with concurrency and parallelism at the threac level, as provided by the JVM, is far too low a level to be safe and comfortable. Many high-level c actors and dataflow have been around for quite some time: parallel computers have been in use, centres if not on the desktop, long before multi-core chips hit the hardware mainstream. Now the these higher-level abstractions in the mainstream software industry. This is what **GPars** enables Java languages, allowing Groovy and Java programmers to use higher-level abstractions and the developing concurrent and parallel software easier and less error prone.

The concepts available in **GPars** can be categorized into three groups:

1. *Code-level helpers* Constructs that can be applied to small parts of the code-base such as ir data structures without any major changes in the overall project architecture
    1. Parallel Collections
    2. Asynchronous Processing
    3. Fork/Join (Divide/Conquer)
2. *Architecture-level concepts* Constructs that need to be taken into account when designing th
    1. Actors
    2. Communicating Sequential Processes (CSP)
    3. Dataflow
    4. Data Parallelism
3. *Shared Mutable State Protection* Although about 95% of current use of shared mutable state proper abstractions, good abstractions are still necessary for the remaining 5% use cases, w state cannot be avoided
    1. Agents
    2. Software Transactional Memory (not fully implemented in GPars as yet)

2.1 Downloading and Installing

GPars is now distributed as standard with Groovy. So if you have a Groovy installation, you shou already. The exact version of GPars you have will, of course, depend of which version of Groovy have GPars, and you do have Groovy, then perhaps you should upgrade your Groovy!

If you do not have a Groovy installation, but get Groovy by using dependencies or just having the then you will need to get GPars. Also if you want to use a version of GPars different from the one an old GPars-less Groovy you cannot upgrade, you will need to get GPars. The ways of getting (

- Download the artifact from a repository and add it and all the transitive dependencies manua
- Specify a dependency in Gradle, Maven, or Ivy (or Gant, or Ant) build files.
- Use Grapes (especially useful for Groovy scripts).

If you're building a Grails or a Griffon application, you can use the appropriate plugins to fetch the

The GPars Artifact

As noted above GPars is now distributed as standard with Groovy. If however, you have to mana manually, the GPars artifact is in the main Maven repository and in the Codehaus main and snap The released versions are in the Maven and Codehaus main repositories, the current developme (SNAPSHOT) is in the Codehaus snapshots repository. To use from Gradle or Grapes use the sp

```
"org.codehaus.gpars:gpars:1.1.0"
```

for the release version, and:

```
"org.codehaus.gpars:gpars:1.2-SNAPSHOT"
```

for the development version. You will likely need to add the Codehaus snapshots repository man in this latter case. Using Maven the dependency is:

```
<dependency>
    <groupId>org.codehaus.gpars</groupId>
    <artifactId>gpars</artifactId>
    <version>1.1.0</version>
</dependency>
```

or version 1.2-SNAPSHOT if using the latest snapshot.

Transitive Dependencies

GPars as a library depends on Groovy version equal or greater than 2.0. Also, the Fork/Join con jsr166y (an artifact from the JSR-166 Project) must be on the classpath the programs, which use and execute. Released versions of this artifact are in the main Maven and Codehaus repositories versions of the artifact are available in the Codehaus snapshots repository. Using Gradle or Grap following dependency specification:

```
"org.codehaus.jsr166-mirror:jsr166y:1.7.0"
```

For Maven, the specification would be:

```
<dependency>
    <groupId>org.codehaus.jsr166-mirror</groupId>
    <artifactId>jsr166y</artifactId>
    <version>1.7.0</version>
</dependency>
```

The development versions have version number 1.7.0.1-SNAPSHOT.

GPars defines this dependency in its own descriptor, so ideally all dependency management sho automatically, if you use Gradle, Grails, Griffon, Maven, Ivy or other type of automatic dependenc

Please visit the page Integration on the GPars website for more details.

2.2 A Hello World Example

Once you are setup, try the following Groovy script to test that your setup is functioning as it shou

```
import static groovyx.gpars.actor.Actors.actor

/**
 * A demo showing two cooperating actors. The decryptor decrypts received messages
 * and replies them back.  The console actor sends a message to decrypt, prints out
 * the reply and terminates both actors.  The main thread waits on both actors to
 * finish using the join() method to prevent premature exit, since both actors use
 * the default actor group, which uses a daemon thread pool.
 * @author Dierk Koenig, Vaclav Pech
 */

def decryptor = actor {
    loop {
        react { message ->
            if (message instanceof String) reply message.reverse()
            else stop()
        }
    }
}

def console = actor {
    decryptor.send 'lellarap si yvoorG'
    react {
        println 'Decrypted message: ' + it
        decryptor.send false
    }
}

[decryptor, console]*.join()
```

You should get a message "Decrypted message: Groovy is parallel" printed out on the console w

> GPars has been designed primarily for use with the Groovy programming language. (all Java and Groovy programs are just bytecodes running on the JVM, so GPars can with Java source. Despite being aimed at Groovy code use, the solid technical founda the good performance characteristics, of GPars make it an excellent library for Java p In fact most of GPars is written in Java, so there is no performance penalty for Java applications using GPars.
>
> For details please refer to the Java API section.

To quick-test using GPars via the Java API, you can compile and run the following Java code:

```
import groovyx.gpars.MessagingRunnable;
import groovyx.gpars.actor.DynamicDispatchActor;

public class StatelessActorDemo {
    public static void main(String[] args) throws InterruptedException {
        final MyStatelessActor actor = new MyStatelessActor();
        actor.start();
        actor.send("Hello");
        actor.sendAndWait(10);
        actor.sendAndContinue(10.0, new MessagingRunnable<String>() {
            @Override protected void doRun(final String s) {
                System.out.println("Received a reply " + s);
            }
        });
    }
}

class MyStatelessActor extends DynamicDispatchActor {
    public void onMessage(final String msg) {
        System.out.println("Received " + msg);
        replyIfExists("Thank you");
    }

public void onMessage(final Integer msg) {
        System.out.println("Received a number " + msg);
        replyIfExists("Thank you");
    }

public void onMessage(final Object msg) {
        System.out.println("Received an object " + msg);
        replyIfExists("Thank you");
    }
}
```

Remember though that you will almost certainly have to add the Groovy artifact to the build as we artifact. GPars may well work at Java speeds with Java applications, but it still has some compila Groovy.

2.3 Code conventions

We follow certain conventions in the code samples. Understanding these may help you read and code samples better.

- The *leftShift* operator << has been overloaded on actors, agents and dataflow expressions (streams) to mean *send* a message or *assign* a value.

```
myActor << 'message'

myAgent << {account -> account.add('5 USD')}

myDataflowVariable << 120332
```

- On actors and agents the default *call()* method has been also overloaded to mean *send* . So an actor or agent may look like a regular method call.

```
myActor "message"

myAgent {house -> house.repair()}
```

- The *rightShift* operator >> in GPars has the *when bound* meaning. So

```
myDataflowVariable >> {value -> doSomethingWith(value)}
```

will schedule the closure to run only after *myDataflowVariable* is bound to a value, with the value

In samples we tend to statically import frequently used factory methods:

- GParsPool.withPool()
- GParsPool.withExistingPool()
- GParsExecutorsPool.withPool()
- GParsExecutorsPool.withExistingPool()
- Actors.actor()
- Actors.reactor()
- Actors.fairReactor()
- Actors.messageHandler()
- Actors.fairMessageHandler()
- Agent.agent()
- Agent.fairAgent()
- Dataflow.task()
- Dataflow.operator()

It is more a matter of style preferences and personal taste, but we think static imports make the c and readable.

2.4 Getting Set Up in an IDE

Adding the GPars jar files to your project or defining the appropriate dependencies in pom.xml sh you started with GPars in your IDE.

GPars DSL recognition

IntelliJ IDEA in both the free *Community Edition* and the commercial *Ultimate Edition* will recogn specific languages, complete methods like *eachParallel()* , *reduce()* or *callAsync()* and validate th GroovyDSL mechanism, which teaches IntelliJ IDEA the DSLs as soon as the GPars jar file is ac

2.5 Applicability of Concepts

GPars provides a lot of concepts to pick from. We're continuously building and updating a page t choose the right abstraction for their tasks at hands. Please, refer to the Concepts compared pag

To briefly summarize the suggestions, below you can find the basic guide-lines:

1. You're looking at a collection, which needs to be **iterated** or processed using one of the mar collections method, like *each()* , *collect()* , *find()* and such. Proposing that processing each e collection is independent of the other items, using GPars **parallel collections** can be recom

2. If you have a **long-lasting calculation** , which may safely run in the background, use the **as invocation support** in GPars. Since the GPars asynchronous functions can be composed, parallelize complex functional calculations without having to mark independent calculations

3. You need to **parallelize** an algorithm at hand. You can identify a set of **tasks** with their mutu tasks typically do not need to share data, but instead some tasks may need to wait for other starting. You're ready to express these dependencies explicitly in code. With GPars **dataflo** internally sequential tasks, each of which can run concurrently with the others. Dataflow vari provide the tasks with the capability to express their dependencies and to exchange data sa

4. You can't avoid using **shared mutable state** in your algorithm. Multiple threads will be acce (some of them) modifying it. Traditional locking and synchronized approach feels too risky o **agents**, which will wrap your data and serialize all access to it.

5. You're building a system with high concurrency demands. Tweaking a data structure here o You need to build the architecture from the ground up with concurrency in mind. **Message-p** way to go.

    1. **Groovy CSP** will give you highly deterministic and composable model for concurrent pr organized around the concept of **calculations** or **processes**, which run concurrently ar through synchronous channels.

    2. If you're trying to solve a complex data-processing problem, consider GPars **dataflow c** data flow network. The concept is organized around event-driven transformations wired asynchronous channels.

    3. **Actors** and **Active Objects** will shine if you need to build a general-purpose, highly co architecture following the object-oriented paradigm.

Now you may have a better idea of what concepts to use on your current project. Go and check them in the User Guide.

2.6 What's New

The new GPars 1.1.0 release introduces several enhancements and improvements on top of the mainly in the dataflow area.

Check out the [JIRA release notes]

Project changes

> ⚠ See [the Breaking Changes listing] for the list of breaking changes.

Asynchronous functions

Parallel collections

- Deprecated foldParallel, renamed to injectParallel

Fork / Join

Actors

Dataflow

- LazyDataflowVariable added to allow for lazy asynchronous values
- Timeout channels on Selects
- Added a Promise-based API for value selection through the Select class
- Enabled listening for bind errors on DataflowVariables
- Minor API improvement affecting Promise and DataflowReadChannel

Agent

- Protecting agent's blocking methods from being called from within commands

Stm

- Updated to the latest 0.7.0 GA version of Multiverse

Other

- Migrated to Groovy 2.0
- Used @CompileStatic where appropriate

Renaming hints

2.7 Java API - Using GPars from Java

Using GPars is very addictive, I guarantee. Once you get hooked you won't be able to code witho
force you to write code in Java, you will still be able to benefit from most of GPars features.

Java API specifics

Some parts of GPars are irrelevant in Java and it is better to use the underlying Java libraries dir

- Parallel Collection - use jsr-166y library's Parallel Array directly
- Fork/Join - use jsr-166y library's Fork/Join support directly
- Asynchronous functions - use Java executor services directly

The other parts of GPars can be used from Java just like from Groovy, although most will miss th
capabilities.

GPars Closures in Java API

To overcome the lack of closures as a language element in Java and to avoid forcing users to us
directly through the Java API, a few handy wrapper classes have been provided to help you defir
body or dataflow tasks.

- groovyx.gpars.MessagingRunnable - used for single-argument callbacks or actor body
- groovyx.gpars.ReactorMessagingRunnable - used for ReactiveActor body
- groovyx.gpars.DataflowMessagingRunnable - used for dataflow operators' body

These classes can be used in all places GPars API expects a Groovy closure.

Actors

The *DynamicDispatchActor* as well as the *ReactiveActor* classes can be used just like in Groovy:

```
import groovyx.gpars.MessagingRunnable;
 import groovyx.gpars.actor.DynamicDispatchActor;

public class StatelessActorDemo {
     public static void main(String[] args) throws InterruptedException {
         final MyStatelessActor actor = new MyStatelessActor();
         actor.start();
         actor.send("Hello");
         actor.sendAndWait(10);
         actor.sendAndContinue(10.0, new MessagingRunnable<String>() {
             @Override protected void doRun(final String s) {
                 System.out.println("Received a reply " + s);
             }
         });
     }
 }

class MyStatelessActor extends DynamicDispatchActor {
     public void onMessage(final String msg) {
         System.out.println("Received " + msg);
         replyIfExists("Thank you");
     }

public void onMessage(final Integer msg) {
         System.out.println("Received a number " + msg);
         replyIfExists("Thank you");
     }

public void onMessage(final Object msg) {
         System.out.println("Received an object " + msg);
         replyIfExists("Thank you");
     }
 }
```

Although there are not many differences between Groovy and Java GPars use, notice, the callba
MessagingRunnable class in place for a groovy closure.

```
import groovy.lang.Closure;
import groovyx.gpars.ReactorMessagingRunnable;
import groovyx.gpars.actor.Actor;
import groovyx.gpars.actor.ReactiveActor;

public class ReactorDemo {
    public static void main(final String[] args) throws InterruptedException {
        final Closure handler = new ReactorMessagingRunnable<Integer, Integer>() {
            @Override protected Integer doRun(final Integer integer) {
                return integer * 2;
            }
        };
        final Actor actor = new ReactiveActor(handler);
        actor.start();

System.out.println("Result: " +  actor.sendAndWait(1));
        System.out.println("Result: " +  actor.sendAndWait(2));
        System.out.println("Result: " +  actor.sendAndWait(3));
    }
}
```

Convenience factory methods

Obviously, all the essential factory methods to build actors quickly are available where you'd exp

```
import groovy.lang.Closure;
import groovyx.gpars.ReactorMessagingRunnable;
import groovyx.gpars.actor.Actor;
import groovyx.gpars.actor.Actors;

public class ReactorDemo {
    public static void main(final String[] args) throws InterruptedException {
        final Closure handler = new ReactorMessagingRunnable<Integer, Integer>() {
            @Override protected Integer doRun(final Integer integer) {
                return integer * 2;
            }
        };
        final Actor actor = Actors.reactor(handler);

System.out.println("Result: " +  actor.sendAndWait(1));
        System.out.println("Result: " +  actor.sendAndWait(2));
        System.out.println("Result: " +  actor.sendAndWait(3));
    }
}
```

Agents

```
import groovyx.gpars.MessagingRunnable;
 import groovyx.gpars.agent.Agent;

public class AgentDemo {
     public static void main(final String[] args) throws InterruptedException {
         final Agent counter = new Agent<Integer>(0);
         counter.send(10);
         System.out.println("Current value: " + counter.getVal());
         counter.send(new MessagingRunnable<Integer>() {
             @Override protected void doRun(final Integer integer) {
                 counter.updateValue(integer + 1);
             }
         });
         System.out.println("Current value: " + counter.getVal());
     }
 }
```

Dataflow Concurrency

Both *DataflowVariables* and *DataflowQueues* can be used from Java without any hiccups. Just a overloaded operators and go straight to the methods, like *bind* , *whenBound* , *getVal* and other. ` using dataflow *tasks* passing to them instances of *Runnable* or *Callable* just like groovy *Closure* .

```
import groovyx.gpars.MessagingRunnable;
import groovyx.gpars.dataflow.DataflowVariable;
import groovyx.gpars.group.DefaultPGroup;

import java.util.concurrent.Callable;

public class DataflowTaskDemo {
    public static void main(final String[] args) throws InterruptedException {
        final DefaultPGroup group = new DefaultPGroup(10);

final DataflowVariable a = new DataflowVariable();

group.task(new Runnable() {
            public void run() {
                a.bind(10);
            }
        });

final Promise result = group.task(new Callable() {
            public Object call() throws Exception {
                return (Integer)a.getVal() + 10;
            }
        });

result.whenBound(new MessagingRunnable<Integer>() {
            @Override protected void doRun(final Integer integer) {
                System.out.println("arguments = " + integer);
            }
        });

System.out.println("result = " + result.getVal());
    }
}
```

Dataflow operators

The sample below should illustrate the main differences between Groovy and Java API for datafl

1. Use the convenience factory methods accepting list of channels to create operators or selec
2. Use *DataflowMessagingRunnable* to specify the operator body
3. Call *getOwningProcessor()* to get hold of the operator from within the body in order to e.g. bi

```
import groovyx.gpars.DataflowMessagingRunnable;
import groovyx.gpars.dataflow.Dataflow;
import groovyx.gpars.dataflow.DataflowQueue;
import groovyx.gpars.dataflow.operator.DataflowProcessor;

import java.util.Arrays;
import java.util.List;

public class DataflowOperatorDemo {
    public static void main(final String[] args) throws InterruptedException {
        final DataflowQueue stream1 = new DataflowQueue();
        final DataflowQueue stream2 = new DataflowQueue();
        final DataflowQueue stream3 = new DataflowQueue();
        final DataflowQueue stream4 = new DataflowQueue();

final DataflowProcessor op1 = Dataflow.selector(Arrays.asList(stream1), Arrays.asList(stream2), new DataflowMe
            @Override protected void doRun(final Object… objects) {
                getOwningProcessor().bindOutput(2*(Integer)objects[0]);
            }
        });

final List secondOperatorInput = Arrays.asList(stream2, stream3);

final DataflowProcessor op2 = Dataflow.operator(secondOperatorInput, Arrays.asList(stream4), new DataflowMessa
            @Override protected void doRun(final Object… objects) {
                getOwningProcessor().bindOutput((Integer) objects[0] + (Integer) objects[1]);
            }
        });

stream1.bind(1);
        stream1.bind(2);
        stream1.bind(3);
        stream3.bind(100);
        stream3.bind(100);
        stream3.bind(100);
        System.out.println("Result: " + stream4.getVal());
        System.out.println("Result: " + stream4.getVal());
        System.out.println("Result: " + stream4.getVal());
        op1.stop();
        op2.stop();
    }
}
```

Performance

In general, GPars overhead is identical irrespective of whether you use it from Groovy or Java ar
low. GPars actors, for example, can compete head-to-head with other JVM actor options, like Sc

Since Groovy code in general runs slower than Java code, mainly due to dynamic method invoca
consider writing your code in Java to improve performance. Typically numeric operations or frequ
method calls within a task or actor body may benefit from a rewrite into Java.

Prerequisites

All the GPars integration rules apply to Java projects just like they do to Groovy projects. You onl
groovy distribution jar file in your project and all is clear to march ahead. You may also want to ch
Java Maven project to get tips on how to integrate GPars into a maven-based pure Java applicat
Maven Project

3 Data Parallelism

Focusing on data instead of processes helps a great deal to create robust concurrent programs. define your data together with functions that should be applied to it and then let the underlying m the data. Typically a set of concurrent tasks will be created and then they will be submitted to a tl processing.

In **GPars** the *GParsPool* and *GParsExecutorsPool* classes give you access to low-level data par While the *GParsPool* class relies on the jsr-166y Fork/Join framework and so offers greater funct performance, the *GParsExecutorsPool* uses good old Java executors and so is easier to setup ir restricted environment.

There are three fundamental domains covered by the GPars low-level data parallelism:

1. Processing collections concurrently
2. Running functions (closures) asynchronously
3. Performing Fork/Join (Divide/Conquer) algorithms

3.1 Parallel Collections

Dealing with data frequently involves manipulating collections. Lists, arrays, sets, maps, iterators other data types can be viewed as collections of items. The common pattern to process such coll elements sequentially, one-by-one, and make an action for each of the items in row.

Take, for example, the *min()* function, which is supposed to return the smallest element of a colle the *min()* method on a collection of numbers, the caller thread will create an *accumulator* or *so-fa* initialized to the minimum value of the given type, let say to zero. And then the thread will iterate of the collection and compare them with the value in the *accumulator* . Once all elements have b minimum value is stored in the *accumulator* .

This algorithm, however simple, is **totally wrong** on multi-core hardware. Running the *min()* func chip can leverage **at most 50%** of the computing power of the chip. On a quad-core it would be c algorithm effectively **wastes 75% of the computing power** of the chip.

Tree-like structures proved to be more appropriate for parallel processing. The *min()* function in c need to iterate through all the elements in row and compare their values with the *accumulator* . V is relying on the multi-core nature of your hardware. A *parallel_min()* function could, for example, tuples of certain size) of neighboring values in the collection and promote the smallest value from round of comparison. Searching for minimum in different tuples can safely happen in parallel and round can be processed by different cores at the same time without races or contention among tl

Meet Parallel Arrays

The jsr-166y library brings a very convenient abstraction called Parallel Arrays . GPars leverages implementation in several ways. The **GParsPool** and **GParsExecutorsPool** classes provide par common Groovy iteration methods like *each()* , *collect()* , *findAll()* and such.

```
def selfPortraits = images.findAllParallel{it.contains me}.collectParallel {it.resize()}
```

It also allows for a more functional style map/reduce collection processing.

```
def smallestSelfPortrait = images.parallel.filter{it.contains me}.map{it.resize()}.min{it.sizeInMB}
```

3.1.1 GParsPool

Use of *GParsPool* - the JSR-166y based concurrent collection processor

Usage of GParsPool

The *GParsPool* class enables a ParallelArray-based (from JSR-166y) concurrency DSL for collec

Examples of use:

```
//summarize numbers concurrently
 GParsPool.withPool {
     final AtomicInteger result = new AtomicInteger(0)
     [1, 2, 3, 4, 5].eachParallel {result.addAndGet(it)}
     assert 15 == result
 }

//multiply numbers asynchronously
 GParsPool.withPool {
     final List result = [1, 2, 3, 4, 5].collectParallel {it * 2}
     assert ([2, 4, 6, 8, 10].equals(result))
 }
```

The passed-in closure takes an instance of a ForkJoinPool as a parameter, which can be then us
closure.

```
//check whether all elements within a collection meet certain criteria
 GParsPool.withPool(5) {ForkJoinPool pool ->
     assert [1, 2, 3, 4, 5].everyParallel {it > 0}
     assert ![1, 2, 3, 4, 5].everyParallel {it > 1}
 }
```

The *GParsPool.withPool()* method takes optional parameters for number of threads in the create
unhandled exception handler.

```
withPool(10) {...}
withPool(20, exceptionHandler) {...}
```

The *GParsPool.withExistingPool()* takes an already existing ForkJoinPool instance to reuse. The
within the associated block of code and only for the thread that has called the *withPool()* or *withE*
The *withPool()* method returns only after all the worker threads have finished their tasks and the
destroyed, returning back the return value of the associated block of code. The *withExistingPool(*
for the pool threads to finish.

Alternatively, the *GParsPool* class can be statically imported *import static groovyx.gpars.GParsP*
allow omitting the *GParsPool* class name.

```
withPool {
     assert [1, 2, 3, 4, 5].everyParallel {it > 0}
     assert ![1, 2, 3, 4, 5].everyParallel {it > 1}
 }
```

The following methods are currently supported on all objects in Groovy:

- eachParallel()
- eachWithIndexParallel()
- collectParallel()
- collectManyParallel()
- findAllParallel()
- findAnyParallel
- findParallel()
- everyParallel()
- anyParallel()
- grepParallel()
- groupByParallel()
- foldParallel()
- minParallel()
- maxParallel()
- sumParallel()
- splitParallel()
- countParallel()
- foldParallel()

Meta-class enhancer

As an alternative you can use the *ParallelEnhancer* class to enhance meta-classes of any classe instances with the parallel methods.

```
import groovyx.gpars.ParallelEnhancer

def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
ParallelEnhancer.enhanceInstance(list)
println list.collectParallel {it * 2 }

def animals = ['dog', 'ant', 'cat', 'whale']
ParallelEnhancer.enhanceInstance animals
println (animals.anyParallel {it ==~ /ant/} ? 'Found an ant' : 'No ants found')
println (animals.everyParallel {it.contains('a')} ? 'All animals contain a' : 'Some animals can live without a
```

When using the *ParallelEnhancer* class, you're not restricted to a *withPool()* block with the use of The enhanced classed or instances remain enhanced till they get garbage collected.

Exception handling

If an exception is thrown while processing any of the passed-in closures, the first exception gets xxxParallel methods and the algorithm stops as soon as possible.

> ⚠ The exception handling mechanism of GParsPool builds on the one built into the Fork
> framework. Since Fork/Join algorithms are by nature hierarchical, once any part of the
> algorithm fails, there's usually little benefit from continuing the computation, since son
> branches of the algorithm will never return a result.
>
> Bear in mind that the GParsPool implementation doesn't give any guarantees about it
> behavior after a first unhandled exception occurs, beyond stopping the algorithm and
> re-throwing the first detected exception to the caller. This behavior, after all, is consis
> what the traditional sequential iteration methods do.

Transparently parallel collections

On top of adding new xxxParallel() methods, **GPars** can also let you change the semantics of the methods. For example, you may be passing a collection into a library method, which will process sequential way, let say using the *collect()* method. By changing the semantics of the *collect()* me you can effectively parallelize the library sequential code.

```
GParsPool.withPool {

//The selectImportantNames() will process the name collections concurrently
    assert ['ALICE', 'JASON'] == selectImportantNames(['Joe', 'Alice', 'Dave', 'Jason'].makeConcurrent())
}

/**
 * A function implemented using standard sequential collect() and findAll() methods.
 */
def selectImportantNames(names) {
    names.collect {it.toUpperCase()}.findAll{it.size() > 4}
}
```

The *makeSequential()* method will reset the collection back to the original sequential semantics.

```
import static groovyx.gpars.GParsPool.withPool

def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

println 'Sequential: '
list.each { print it + ',' }
println()

withPool {

println 'Sequential: '
    list.each { print it + ',' }
    println()

list.makeConcurrent()

println 'Concurrent: '
    list.each { print it + ',' }
    println()

list.makeSequential()

println 'Sequential: '
    list.each { print it + ',' }
    println()
}

println 'Sequential: '
list.each { print it + ',' }
println()
```

The *asConcurrent()* convenience method will allow you to specify code blocks, in which the colle concurrent semantics.

```
import static groovyx.gpars.GParsPool.withPool

def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

println 'Sequential: '
list.each { print it + ',' }
println()

withPool {

println 'Sequential: '
    list.each { print it + ',' }
    println()

list.asConcurrent {
        println 'Concurrent: '
        list.each { print it + ',' }
        println()
    }

println 'Sequential: '
    list.each { print it + ',' }
    println()
}

println 'Sequential: '
list.each { print it + ',' }
println()
```

Transparent parallelizm, including the *makeConcurrent()* , *makeSequential()* and *asConcurrent()* available in combination with *ParallelEnhancer* .

```
/**
 * A function implemented using standard sequential collect() and findAll() methods.
 */
def selectImportantNames(names) {
    names.collect {it.toUpperCase()}.findAll{it.size() > 4}
}

def names = ['Joe', 'Alice', 'Dave', 'Jason']
ParallelEnhancer.enhanceInstance(names)
//The selectImportantNames() will process the name collections concurrently
assert ['ALICE', 'JASON'] == selectImportantNames(names.makeConcurrent())
```

```
import groovyx.gpars.ParallelEnhancer

def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

println 'Sequential: '
list.each { print it + ',' }
println()

ParallelEnhancer.enhanceInstance(list)

println 'Sequential: '
list.each { print it + ',' }
println()

list.asConcurrent {
    println 'Concurrent: '
    list.each { print it + ',' }
    println()

}
list.makeSequential()

println 'Sequential: '
list.each { print it + ',' }
println()
```

Avoid side-effects in functions

We have to warn you. Since the closures that are provided to the parallel methods like *eachPara* may be run in parallel, you have to make sure that each of the closures is written in a thread-safe must hold no internal state, share data nor have side-effects beyond the boundaries the single el been invoked on. Violations of these rules will open the door for race conditions and deadlocks, t enemies of a modern multi-core programmer.

Don't do this:

```
def thumbnails = []
images.eachParallel {thumbnails << it.thumbnail}  //Concurrently accessing a not-thread-safe collection of thu
```

At least, you've been warned.

3.1.2 GParsExecutorsPool

Use of GParsExecutorsPool - the Java Executors' based concurrent collection processor

Usage of GParsExecutorsPool

The *GParsPool* class enables a Java Executors-based concurrency DSL for collections and obje

The *GParsExecutorsPool* class can be used as a pure-JDK-based collection parallel processor. U class, *GParsExecutorsPool* doesn't require jsr-166y jar file, but leverages the standard JDK exec parallelize closures processing a collections or an object iteratively. It needs to be states, howeve performs typically much better than *GParsExecutorsPool* does.

Examples of use:

```
//multiply numbers asynchronously
 GParsExecutorsPool.withPool {
     Collection<Future> result = [1, 2, 3, 4, 5].collectParallel{it * 10}
     assert new HashSet([10, 20, 30, 40, 50]) == new HashSet((Collection)result*.get())
 }
//multiply numbers asynchronously using an asynchronous closure
 GParsExecutorsPool.withPool {
     def closure={it * 10}
     def asyncClosure=closure.async()
     Collection<Future> result = [1, 2, 3, 4, 5].collect(asyncClosure)
     assert new HashSet([10, 20, 30, 40, 50]) == new HashSet((Collection)result*.get())
 }
```

The passed-in closure takes an instance of a ExecutorService as a parameter, which can be the closure.

```
//find an element meeting specified criteria
 GParsExecutorsPool.withPool(5) {ExecutorService service ->
     service.submit({performLongCalculation()} as Runnable)
 }
```

The *GParsExecutorsPool.withPool()* method takes optional parameters for number of threads in thread factory.

```
withPool(10) {...}
withPool(20, threadFactory) {...}
```

The *GParsExecutorsPool.withExistingPool()* takes an already existing executor service instance valid only within the associated block of code and only for the thread that has called the *withPool* method. The *withPool()* method returns only after all the worker threads have finished their tasks service has been destroyed, returning back the return value of the associated block of code. The method doesn't wait for the executor service threads to finish.

Alternatively, the *GParsExecutorsPool* class can be statically imported *import static groovyx.gpars.GParsExecutorsPool.`*`* , which will allow omitting the *GParsExecutorsPool* class

```
withPool {
     def result = [1, 2, 3, 4, 5].findParallel{Number number -> number > 2}
     assert result in [3, 4, 5]
 }
```

The following methods on all objects, which support iterations in Groovy, are currently supported

- eachParallel()
- eachWithIndexParallel()
- collectParallel()
- findAllParallel()
- findParallel()
- allParallel()
- anyParallel()
- grepParallel()
- groupByParallel()

Meta-class enhancer

As an alternative you can use the *GParsExecutorsPoolEnhancer* class to enhance meta-classes individual instances with asynchronous methods.

```
import groovyx.gpars.GParsExecutorsPoolEnhancer

def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
GParsExecutorsPoolEnhancer.enhanceInstance(list)
println list.collectParallel {it * 2 }

def animals = ['dog', 'ant', 'cat', 'whale']
GParsExecutorsPoolEnhancer.enhanceInstance animals
println (animals.anyParallel {it ==~ /ant/} ? 'Found an ant' : 'No ants found')
println (animals.allParallel {it.contains('a')} ? 'All animals contain a' : 'Some animals can live without an
```

When using the *GParsExecutorsPoolEnhancer* class, you're not restricted to a *withPool()* block v GParsExecutorsPool DSLs. The enhanced classed or instances remain enhanced till they get ga

Exception handling

If exceptions are thrown while processing any of the passed-in closures, an instance of *AsyncEx* the original exceptions gets re-thrown from the xxxParallel methods.

Avoid side-effects in functions

Once again we need to warn you about using closures with side-effects effecting objects beyond single currently processed element or closures which keep state. Don't do that! It is dangerous tc the *xxxParallel()* methods.

3.1.3 Memoize

The *memoize* function enables caching of function's return values. Repeated calls to the memoiz
same argument values will, instead of invoking the calculation encoded in the original function, re
from an internal transparent cache. Provided the calculation is considerably slower than retrievin
the cache, this allows users to trade-off memory for performance. Checkout out the example, wh
multiple websites for particular content:

The memoize functionality of GPars has been contributed to Groovy in version 1.8 and if you run
later, it is recommended to use the Groovy functionality. Memoize in GPars is almost identical, ex
the memoize caches concurrently using the surrounding thread pool and so may give performan
scenarios.

> ⚠ The GPars memoize functionality has been renamed to avoid future conflicts with the
> functionality in Groovy. GPars now calls the methods with a preceding letter *g* , such
> gmemoize().

Examples of use

```
GParsPool.withPool {
    def urls = ['http://www.dzone.com', 'http://www.theserverside.com', 'http://www.infoq.com']
    Closure download = {url ->
        println "Downloading $url"
        url.toURL().text.toUpperCase()
    }
    Closure cachingDownload = download.gmemoize()

println 'Groovy sites today: ' + urls.findAllParallel {url -> cachingDownload(url).contains('GROOVY')}
    println 'Grails sites today: ' + urls.findAllParallel {url -> cachingDownload(url).contains('GRAILS')}
    println 'Griffon sites today: ' + urls.findAllParallel {url -> cachingDownload(url).contains('GRIFFON')}
    println 'Gradle sites today: ' + urls.findAllParallel {url -> cachingDownload(url).contains('GRADLE')}
    println 'Concurrency sites today: ' + urls.findAllParallel {url -> cachingDownload(url).contains('CONCURRE
    println 'GPars sites today: ' + urls.findAllParallel {url -> cachingDownload(url).contains('GPARS')}
}
```

Notice closures are enhanced inside the *GParsPool.withPool()* blocks with a *memoize()* function,
closure wrapping the original closure with a cache. In the example we're calling the *cachingDowr*
several places in the code, however, each unique url gets downloaded only once - the first time i
values are then cached and available for subsequent calls. And also to all threads, no matter whi
came first with a download request for the particular url and had to handle the actual calculation/

So, to wrap up, memoize shields a function by a cache of past return values. However, *memoize*
some algorithms adding a little memory may have dramatic impact on the computational comple
Let's look at a classical example of Fibonacci numbers.

Fibonacci example

A purely functional, recursive implementation, following closely the definition of Fibonacci numbe
complex:

```
Closure fib = {n -> n > 1 ? call(n - 1) + call(n - 2) : n}
```

Try calling the *fib* function with numbers around 30 and you'll see how slow it is.

Now with a little twist and added memoize cache the algorithm magically turns into a linearly com

```
Closure fib
fib = {n -> n > 1 ? fib(n - 1) + fib(n - 2) : n}.gmemoize()
```

The extra memory we added cut off all but one recursive branches of the calculation. And all sub same *fib* function will also benefit from the cached values.

Also, see below, how the *memoizeAtMost* variant can reduce memory consumption in our examp linear complexity of the algorithm.

Available variants

memoize

The basic variant, which keeps values in the internal cache for the whole lifetime of the memoize the best performance characteristics of all the variants.

memoizeAtMost

Allows the user to set a hard limit on number of items cached. Once the limit has been reached, a values will eliminate the oldest value from the cache using the LRU (Last Recently Used) strateg

So for our Fibonacci number example, we could safely reduce the cache size to two items:

```
Closure fib
fib = {n -> n > 1 ? fib(n - 1) + fib(n - 2) : n}.memoizeAtMost(2)
```

Setting an upper limit on the cache size may have two purposes:

1. Keep the memory footprint of the cache within defined boundaries
2. Preserve desired performance characteristics of the function. Too large caches may take lon cached value than it would have taken to calculate the result directly.

memoizeAtLeast

Allows unlimited growth of the internal cache until the JVM's garbage collector decides to step in SoftReferences, used by our implementation, from the memory. The single parameter value to th method specifies the minimum number of cached items that should be protected from gc eviction shrink below the specified number of entries. The cache ensures it only protects the most recentl eviction using the LRU (Last Recently Used) strategy.

memoizeBetween

Combines memoizeAtLeast and memoizeAtMost and so allowing the cache to grow and shrink in the two parameter values depending on available memory and the gc activity, yet the cache size upper size limit to preserve desired performance characteristics of the cache.

3.2 Map-Reduce

The Parallel Collection Map/Reduce DSL gives GPars a more functional flavor. In general, the M be used for the same purpose as the *xxxParallel()* family methods and has very similar semantic Map/Reduce can perform considerably faster, if you need to chain multiple methods to process a multiple steps:

```
println 'Number of occurrences of the word GROOVY today: ' + urls.parallel
            .map {it.toURL().text.toUpperCase()}
            .filter {it.contains('GROOVY')}
            .map{it.split()}
            .map{it.findAll{word -> word.contains 'GROOVY'}.size()}
            .sum()
```

The *xxxParallel()* methods have to follow the contract of their non-parallel peers. So a *collectPara* return a legal collection of items, which you can again treat as a Groovy collection. Internally the builds an efficient parallel structure, called parallel array, performs the required operation concurr returning destroys the Parallel Array building the collection of results to return to you. A potential *findAllParallel()* on the resulting collection would repeat the whole process of construction and de Array instance under the covers.

With Map/Reduce you turn your collection into a Parallel Array and back only once. The Map/Re do not return Groovy collections, but are free to pass along the internal Parallel Arrays directly. Ir property on a collection will build a Parallel Array for the collection and return a thin wrapper arou instance. Then you can chain all required methods like:

- map()
- reduce()
- filter()
- size()
- sum()
- min()
- max()
- sort()
- groupBy()
- combine()

Returning back to a plain Groovy collection instance is always just a matter of retrieving the *colle*

```
def myNumbers = (1..1000).parallel.filter{it % 2 == 0}.map{Math.sqrt it}.collection
```

Avoid side-effects in functions

Once again we need to warn you. To avoid nasty surprises, please, keep your closures, which yo Map/Reduce functions, stateless and clean from side-effects.

Availability

This feature is only available when using in the Fork/Join-based *GParsPool* , not in *GParsExecut*

Classical Example

A classical example, inspired by http://github.com/thevery, counting occurrences of words in a str

```
import static groovyx.gpars.GParsPool.withPool

def words = "This is just a plain text to count words in"
print count(words)

def count(arg) {
  withPool {
    return arg.parallel
      .map{[it, 1]}
      .groupBy{it[0]}.getParallel()
      .map {it.value=it.value.size();it}
      .sort{-it.value}.collection
  }
}
```

The same example, now implemented the more general *combine* operation:

```
def words = "This is just a plain text to count words in"
print count(words)

def count(arg) {
  withPool {
    return arg.parallel
      .map{[it, 1]}
      .combine(0) {sum, value -> sum + value}.getParallel()
      .sort{-it.value}.collection
  }
}
```

Combine

The *combine* operation expects on its input a list of tuples (two-element lists) considered to be ke
[key1, value1, key2, value2, key1, value3, key3, value4 …]) with potentially repeating keys. Wh
merges the values for identical keys using the provided accumulator function and produces a ma
(unique) keys to their accumulated values. E.g. [a, b, c, d, a, e, c, f] will be combined into a : b+e,
operation on the values needs to be provided by the user as the accumulation closure.

The *accumulation function* argument needs to specify a function to use for combining (accumulat
belonging to the same key. An *initial accumulator value* needs to be provided as well. Since the c
processes items in parallel, the *initial accumulator value* will be reused multiple times. Thus the p
allow for reuse. It should be either a **cloneable** or **immutable** value or a **closure** returning a fres
each time requested. Good combinations of accumulator functions and reusable initial values inc

```
accumulator = {List acc, value -> acc << value} initialValue = []
accumulator = {List acc, value -> acc << value} initialValue = {-> []}
accumulator = {int sum, int value -> acc + value} initialValue = 0
accumulator = {int sum, int value -> sum + value} initialValue = {-> 0}
accumulator = {ShoppingCart cart, Item value -> cart.addItem(value)} initialValue = {-> new ShoppingCart()}
```

The return type is a map. E.g. ['he', 1, 'she', 2, 'he', 2, 'me', 1, 'she, 5, 'he', 1 with the initial value p
combined into 'he' : 4, 'she' : 7, 'he', : 2, 'me' : 1

⚠ The keys will be mutually compared using their equals and hashCode methods. Cons *@Canonical* or *@EqualsAndHashCode* to annotate classes that you use as keys. Jus all hash maps in Groovy, be sure you're using a String not a GString as a key!

For more involved scenarios when you *combine()* complex objects, a good strategy here is to ha used as a key for the common use cases and apply different keys for uncommon cases.

```
import groovy.transform.ToString
import groovy.transform.TupleConstructor

import static groovyx.gpars.GParsPool.withPool

TupleConstructor ToString
class PricedCar implements Cloneable {
    String model
    String color
    Double price

boolean equals(final o) {
        if (this.is(o)) return true
        if (getClass() != o.class) return false

final PricedCar pricedCar = (PricedCar) o

if (color != pricedCar.color) return false
        if (model != pricedCar.model) return false

return true
    }

int hashCode() {
        int result
        result = (model != null ? model.hashCode() : 0)
        result = 31 * result + (color != null ? color.hashCode() : 0)
        return result
    }

@Override
    protected Object clone() {
        return super.clone()
    }
}

def cars = [new PricedCar('F550', 'blue', 2342.223),
        new PricedCar('F550', 'red', 234.234),
        new PricedCar('Da', 'white', 2222.2),
        new PricedCar('Da', 'white', 1111.1)]

withPool {
    //Combine by model
    def result =
        cars.parallel.map {
            [it.model, it]
        }.combine(new PricedCar('', 'N/A', 0.0)) {sum, value ->
            sum.model = value.model
            sum.price += value.price
            sum
        }.values()

println result


    //Combine by model and color (the PricedCar's equals and hashCode))
    result =
        cars.parallel.map {
            [it, it]
        }.combine(new PricedCar('', 'N/A', 0.0)) {sum, value ->
            sum.model = value.model
            sum.color = value.color
            sum.price += value.price
            sum
        }.values()

println result
}
```

3.3 Parallel Arrays

As an alternative, the efficient tree-based data structures defines in JSR-166y can be used direct property on any collection or object will return a *jsr166y.forkjoin.ParallelArray* instance holding th original collection, which then can be manipulated through the jsr166y API. Please refer to the jsr for the API details.

```
import groovyx.gpars.extra166y.Ops

groovyx.gpars.GParsPool.withPool {
    assert 15 == [1, 2, 3, 4, 5].parallelArray.reduce({a, b -> a + b} as Ops.Reducer, 0)
//summarize
    assert 55 == [1, 2, 3, 4, 5].parallelArray.withMapping({it ** 2} as Ops.Op).reduce({a, b -> a + b} as Ops.
//summarize squares
    assert 20 == [1, 2, 3, 4, 5].parallelArray.withFilter({it % 2 == 0} as Ops.Predicate)
//summarize squares of even numbers
            .withMapping({it ** 2} as Ops.Op)
            .reduce({a, b -> a + b} as Ops.Reducer, 0)

assert 'aa:bb:cc:dd:ee' == 'abcde'.parallelArray
//concatenate duplicated characters with separator
            .withMapping({it * 2} as Ops.Op)
            .reduce({a, b -> "$a:$b"} as Ops.Reducer, "")
```

3.4 Asynchronous Invocation

Running long-lasting tasks in the background belongs to the activities, the need for which arises main thread of execution wants to initialize a few calculations, downloads, searches or such, how not be needed immediately. **GPars** gives the developers the tools to schedule the asynchronous processing in the background and collect the results once they're needed.

Usage of GParsPool and GParsExecutorsPool asynchronous process

Both *GParsPool* and *GParsExecutorsPool* provide almost identical services in this domain, altho different underlying machinery, based on which of the two classes the user chooses.

Closures enhancements

The following methods are added to closures inside the *GPars(Executors)Pool.withPool()* blocks

- async() - Creates an asynchronous variant of the supplied closure, which when invoked retu potential return value
- callAsync() - Calls a closure in a separate thread supplying the given arguments, returning a return value,

Examples:

```
GParsPool.withPool() {
    Closure longLastingCalculation = {calculate()}
    Closure fastCalculation = longLastingCalculation.async()  //create a new closure, which starts the origina
pool
    Future result=fastCalculation()                           //returns almost immediately
    //do stuff while calculation performs …
    println result.get()
}
```

```
GParsPool.withPool() {
    /**
     * The callAsync() method is an asynchronous variant of the default call() method to invoke a closure.
     * It will return a Future for the result value.
     */
    assert 6 == {it * 2}.call(3)
    assert 6 == {it * 2}.callAsync(3).get()
}
```

Timeouts

The *callTimeoutAsync()* methods, taking either a long value or a Duration instance, allow the use
calculation cancelled after a given time interval.

```
{->
    while(true) {
        Thread.sleep 1000  //Simulate a bit of interesting calculation
        if (Thread.currentThread().isInterrupted()) break;  //We've been cancelled
    }
}.callTimeoutAsync(2000)
```

In order to allow cancellation, the asynchronously running code must keep checking the *interrupt*
thread and cease the calculation once the flag is set to true.

Executor Service enhancements

The ExecutorService and jsr166y.forkjoin.ForkJoinPool class is enhanced with the << (leftShift) c
tasks to the pool and return a *Future* for the result.

Example:

```
GParsExecutorsPool.withPool {ExecutorService executorService ->
    executorService << {println 'Inside parallel task'}
}
```

Running functions (closures) in parallel

The *GParsPool* and *GParsExecutorsPool* classes also provide handy methods *executeAsync()* a
executeAsyncAndWait() to easily run multiple closures asynchronously.

Example:

```
GParsPool.withPool {
    assert [10, 20] == GParsPool.executeAsyncAndWait({calculateA()}, {calculateB()}         //waits for result
    assert [10, 20] == GParsPool.executeAsync({calculateA()}, {calculateB()})*.get()  //returns Futures inste
results to be calculated
}
```

3.5 Composable Asynchronous Functions

Functions are to be composed. In fact, composing side-effect-free functions is very easy. Much e
composing objects, for example. Given the same input, functions always return the same result,
their behavior unexpectedly nor they break when multiple threads call them at the same time.

Functions in Groovy

We can treat Groovy closures as functions. They take arguments, do their calculation and return
don't let your closures touch anything outside their scope, your closures are well-behaved pure fu
that you can combine for a better good.

```
def sum = (0..100000).inject(0, {a, b -> a + b})
```

For example, by combining a function adding two numbers with the *inject* function, which iterates
collection, you can quickly summarize all items. Then, replacing the *adding* function with a *compa*
immediately give you a combined function calculating maximum.

```
def max = myNumbers.inject(0, {a, b -> a>b?a:b})
```

You see, functional programming is popular for a reason.

Are we concurrent yet?

This all works just fine until you realize you're not utilizing the full power of your expensive hardw
plain sequential. No parallelism in here. All but one processor core do nothing, they're idle, totally

> ⚠ Those paying attention would suggest to use the *Parallel Collection* techniques descr
> earlier and they would certainly be correct. For our scenario described here, where w
> a collection, using those *parallel* methods would be the best choice. However, we're r
> looking for a **generic way to create and combine asynchronous functions** , which
> help us not only for collection processing but mostly in other more generic cases, like
> right below.

To make things more obvious, here's an example of combining four functions, which are suppose
particular web page matches the contents of a local file. We need to download the page, load the
of both and finally compare the resulting numbers.

```
Closure download = {String url ->
    url.toURL().text
}

Closure loadFile = {String fileName ->
    …  //load the file here
}

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

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

def result = compare(hash(download('http://www.gpars.org')), hash(loadFile('/coolStuff/gpars/website/index.htm
println "The result of comparison: " + result
```

We need to download the page, load up the file, calculate hashes of both and finally compare the
Each of the functions is responsible for one particular job. One downloads the content, second lo
calculates the hashes and finally the fourth one will do the comparison. Combining the functions
their calls.

Making it all asynchronous

The downside of our code is that we don't leverage the independence of the *download()* and the
Neither we allow the two hashes to be run concurrently. They could well run in parallel, but our w
functions restricts any parallelism.

Obviously not all of the functions can run concurrently. Some functions depend on results of othe
before the other function finishes. We need to block them till their parameters are available. The
a string to work on. The *compare()* function needs two numbers to compare.

So we can only parallelize some functions, while blocking parallelism of others. Seems like a cha

Things are bright in the functional world

Luckily, the dependencies between functions are already expressed implicitly in the code. There' duplicate the dependency information. If one functions takes parameters and the parameters nee by another function, we implicitly have a dependency here. The *hash()* function depends on the *l* the *download()* functions in our example. The *inject* function in our earlier example depends on th *addition* functions invoked gradually on all the elements of the collection.

> However difficult it may seem at first, our task is in fact very simple. We only need to functions to return *promises* of their future results. And we need to teach the other fur accept those *promises* as parameters so that they wait for the real values before they work. And if we convince the functions to release the threads they hold while waiting values, we get directly to where the magic can happen.

In the good tradition of *GPars* we've made it very straightforward for you to convince any functior functions' promises. Call the *asyncFun()* function on a closure and you're asynchronous.

```
withPool {
    def maxPromise = numbers.inject(0, {a, b -> a>b?a:b}.asyncFun())
    println "Look Ma, I can talk to the user while the math is being done for me!"
    println maxPromise.get()
}
```

The *inject* function doesn't really care what objects are being returned from the *addition* function, surprised that each call to the *addition* function returns so fast, but doesn't moan much, keeps ite returns the overall result to you.

Now, this is the time you should stand behind what you say and do what you want others to do. D and just accepts that you got back just a promise. A **promise** to get the result delivered as soon done. The extra heat coming out of your laptop is an indication the calculation exploits natural pa functions and makes its best effort to deliver the result to you quickly.

> The *promise* is a good old *DataflowVariable* , so you may query its status, register no hooks or make it an input to a Dataflow algorithm.

```
withPool {
    def sumPromise = (0..100000).inject(0, {a, b -> a + b}.asyncFun())
    println "Are we done yet? " + sumPromise.bound
    sumPromise.whenBound {sum -> println sum}
}
```

> The *get()* method has also a variant with a timeout parameter, if you want to avoid the waiting indefinitely.

Can things go wrong?

Sure. But you'll get an exception thrown from the result promise *get()* method.

```
try {
    sumPromise.get()
} catch (MyCalculationException e) {
    println "Guess, things are not ideal today."
}
```

This is all fine, but what functions can be really combined?

There are no limits. Take any sequential functions you need to combine and you should be able t
asynchronous variants as well.

Back to our initial example comparing content of a file with a web page, we simply make all the fu
by calling the *asyncFun()* method on them and we are ready to set off.

```
Closure download = {String url ->
        url.toURL().text
    }.asyncFun()

Closure loadFile = {String fileName ->
        …  //load the file here
    }.asyncFun()

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

Closure compare = {int first, int second ->
        first == second
    }.asyncFun()

def result = compare(hash(download('http://www.gpars.org')), hash(loadFile('/coolStuff/gpars/website/index.htm
    println 'Allowed to do something else now'
    println "The result of comparison: " + result.get()
```

Calling asynchronous functions from within asynchronous functions

Another very valuable characteristics of asynchronous functions is that their result promises can

```
import static groovyx.gpars.GParsPool.withPool

withPool {
      Closure plus = {Integer a, Integer b ->
          sleep 3000
          println 'Adding numbers'
          a + b
      }.asyncFun()

Closure multiply = {Integer a, Integer b ->
          sleep 2000
          a * b
      }.asyncFun()

Closure measureTime = {->
          sleep 3000
          4
      }.asyncFun()

Closure distance = {Integer initialDistance, Integer velocity, Integer time ->
          plus(initialDistance, multiply(velocity, time))
      }.asyncFun()

Closure chattyDistance = {Integer initialDistance, Integer velocity, Integer time ->
          println 'All parameters are now ready - starting'
          println 'About to call another asynchronous function'
          def innerResultPromise = plus(initialDistance, multiply(velocity, time))
          println 'Returning the promise for the inner calculation as my own result'
          return innerResultPromise
      }.asyncFun()

println "Distance = " + distance(100, 20, measureTime()).get() + ' m'
      println "ChattyDistance = " + chattyDistance(100, 20, measureTime()).get() + ' m'
  }
```

If an asynchronous function (e.f. the *distance* function in the example) in its body calls another as
(e.g. *plus*) and returns the the promise of the invoked function, the inner function's (*plus*) result
with the outer function's (*distance*) result promise. The inner function (*plus*) will now bind its re
function's (*distance*) promise, once the inner function (plus) finishes its calculation. This ability c
allows functions to cease their calculation without blocking a thread not only when waiting for par
whenever they call another asynchronous function anywhere in their body.

Methods as asynchronous functions

Methods can be referred to as closures using the *.&* operator. These closures can then be transf
into composable asynchronous functions just like ordinary closures.

```
class DownloadHelper {
    String download(String url) {
        url.toURL().text
    }

int scanFor(String word, String text) {
        text.findAll(word).size()
    }

String lower(s) {
        s.toLowerCase()
    }
}
//now we'll make the methods asynchronous
withPool {
    final DownloadHelper d = new DownloadHelper()
    Closure download = d.&download.asyncFun()
    Closure scanFor = d.&scanFor.asyncFun()
    Closure lower = d.&lower.asyncFun()

//asynchronous processing
    def result = scanFor('groovy', lower(download('http://www.infoq.com')))
    println 'Allowed to do something else now'
    println result.get()
}
```

Using annotation to create asynchronous functions

Instead of calling the *asyncFun()* function, the *@AsyncFun* annotation can be used to annotate C
The fields have to be initialized in-place and the containing class needs to be instantiated withing

```
import static groovyx.gpars.GParsPool.withPool
import groovyx.gpars.AsyncFun

class DownloadingSearch {
    @AsyncFun Closure download = {String url ->
        url.toURL().text
    }

@AsyncFun Closure scanFor = {String word, String text ->
        text.findAll(word).size()
    }

@AsyncFun Closure lower = {s -> s.toLowerCase()}

void scan() {
        def result = scanFor('groovy', lower(download('http://www.infoq.com')))  //synchronous processing
        println 'Allowed to do something else now'
        println result.get()
    }
}

withPool {
    new DownloadingSearch().scan()
}
```

Alternative pools

The *AsyncFun* annotation by default uses an instance of *GParsPool* from the wrapping withPool
however, specify the type of pool explicitly:

```
@AsyncFun(GParsExecutorsPoolUtil) def sum6 = {a, b -> a + b }
```

Blocking functions through annotations

The *AsyncFun* also allows the user to specify, whether the resulting function should have blockin non-blocking (false - default) semantics.

```
@AsyncFun(blocking = true)
def sum = {a, b -> a + b }
```

Explicit and delayed pool assignment

When using the *GPars(Executors)PoolUtil.asyncFun()* function directly to create an asynchronou two additional options to assign a thread pool to the function.

1. The thread pool to use by the function can be specified explicitly as an additional argument a
2. The implicit thread pool can be obtained from the surrounding scope at invocation rather at c

When specifying the thread pool explicitly, the call doesn't need to be wrapped in an *withPool()* b

```
Closure sPlus = {Integer a, Integer b ->
    a + b
}
Closure sMultiply = {Integer a, Integer b ->
    sleep 2000
    a * b
}
println "Synchronous result: " + sMultiply(sPlus(10, 30), 100)
final pool = new FJPool()
Closure aPlus = GParsPoolUtil.asyncFun(sPlus, pool)
Closure aMultiply = GParsPoolUtil.asyncFun(sMultiply, pool)
def result = aMultiply(aPlus(10, 30), 100)
println "Time to do something else while the calculation is running"
println "Asynchronous result: " + result.get()
```

With delayed pool assignment only the function invocation must be surrounded with a *withPool()*

```
Closure aPlus = GParsPoolUtil.asyncFun(sPlus)
Closure aMultiply = GParsPoolUtil.asyncFun(sMultiply)
withPool {
    def result = aMultiply(aPlus(10, 30), 100)
println "Time to do something else while the calculation is running"
    println "Asynchronous result: " + result.get()
}
```

On our side this is a very interesting domain to explore, so any comments, questions or suggesti asynchronous functions or hints about its limits are welcome.

3.6 Fork-Join

Fork/Join or Divide and Conquer is a very powerful abstraction to solve hierarchical problems.

The abstraction

When talking about hierarchical problems, think about quick sort, merge sort, file system or gene such.

- Fork / Join algorithms essentially split a problem at hands into several smaller sub-problems
  the same algorithm to each of the sub-problems.
- Once the sub-problem is small enough, it is solved directly.
- The solutions of all sub-problems are combined to solve their parent problem, which in turn l
  parent problem.

> ⚠ Check out the fancy interactive Fork/Join visualization demo , which will show you ho
> cooperate to solve a common divide-and-conquer algorithm.

The mighty **JSR-166y** library solves Fork / Join orchestration pretty nicely for us, but leaves a co
which can hurt you, if you don't pay attention enough. You still deal with threads, pools or synchr

The GPars abstraction convenience layer

GPars can hide the complexities of dealing with threads, pools and recursive tasks from you, yet
powerful Fork/Join implementation in jsr166y.

```
import static groovyx.gpars.GParsPool.runForkJoin
import static groovyx.gpars.GParsPool.withPool

withPool() {
    println """Number of files: ${
        runForkJoin(new File("./src")) {file ->
            long count = 0
            file.eachFile {
                if (it.isDirectory()) {
                    println "Forking a child task for $it"
                    forkOffChild(it)           //fork a child task
                } else {
                    count++
                }
            }
            return count + (childrenResults.sum(0))
            //use results of children tasks to calculate and store own result
        }
    }"""
}
```

The *runForkJoin()* factory method will use the supplied recursive code together with the provided
hierarchical Fork/Join calculation. The number of values passed to the *runForkJoin()* method mu
expected parameters of the closure as well as the number of arguments passed into the *forkOffC*
runChildDirectly() methods.

```
def quicksort(numbers) {
    withPool {
        runForkJoin(0, numbers) {index, list ->
            def groups = list.groupBy {it <=> list[list.size().intdiv(2)]}
            if ((list.size() < 2) || (groups.size() == 1)) {
                return [index: index, list: list.clone()]
            }
            (-1..1).each {forkOffChild(it, groups[it] ?: [])}
            return [index: index, list: childrenResults.sort {it.index}.sum {it.list}]
        }.list
    }
}
```

> ⚠ The important piece of the puzzle that needs to be mentioned here is that *forkOffChil*
> wait for the child to run. It merely schedules it for execution some time in the future. If
> fails by throwing an exception, you should not expect the exception to be fired from th
> forkOffChild() method itself. The exception ise likely to happen long after the parent h
> returned from the call to the *forkOffChild()* method.
>
> It is the *getChildrenResults()* method that will re-throw exceptions that happened in th
> sub-tasks back to the parent task.

Alternative approach

Alternatively, the underlying mechanism of nested Fork/Join worker tasks can be used directly. C workers can eliminate the performance overhead associated with parameter spreading imposed generic workers. Also, custom workers can be implemented in Java and so further increase the p algorithm.

```
public final class FileCounter extends AbstractForkJoinWorker<Long> {
    private final File file;

def FileCounter(final File file) {
        this.file = file
    }

@Override
    protected Long computeTask() {
        long count = 0;
        file.eachFile {
            if (it.isDirectory()) {
                println "Forking a thread for $it"
                forkOffChild(new FileCounter(it))           //fork a child task
            } else {
                count++
            }
        }
        return count + ((childrenResults)?.sum() ?: 0)  //use results of children tasks to calculate and store
    }
}

withPool(1) {pool ->  //feel free to experiment with the number of fork/join threads in the pool
    println "Number of files: ${runForkJoin(new FileCounter(new File("..")))}"
}
```

The AbstractForkJoinWorker subclasses may be written both in Java or Groovy, giving you the o for execution speed, if row performance of the worker becomes a bottleneck.

Fork / Join saves your resources

Fork/Join operations can be safely run with small number of threads thanks to internally using the synchronize the threads. While a thread is blocked inside an algorithm waiting for its sub-problen thread is silently returned to the pool to take on any of the available sub-problems from the task q them. Although the algorithm creates as many tasks as there are sub-directories and tasks wait f tasks to complete, as few as one thread is enough to keep the computation going and eventually result.

Mergesort example

```
import static groovyx.gpars.GParsPool.runForkJoin
import static groovyx.gpars.GParsPool.withPool

/**
 * Splits a list of numbers in half
 */
def split(List<Integer> list) {
    int listSize = list.size()
    int middleIndex = listSize / 2
    def list1 = list[0..<middleIndex]
    def list2 = list[middleIndex..listSize - 1]
    return [list1, list2]
}

/**
 * Merges two sorted lists into one
 */
List<Integer> merge(List<Integer> a, List<Integer> b) {
    int i = 0, j = 0
    final int newSize = a.size() + b.size()
    List<Integer> result = new ArrayList<Integer>(newSize)

while ((i < a.size()) && (j < b.size())) {
        if (a[i] <= b[j]) result << a[i++]
        else result << b[j++]
    }

if (i < a.size()) result.addAll(a[i..-1])
    else result.addAll(b[j..-1])
    return result
}

final def numbers = [1, 5, 2, 4, 3, 8, 6, 7, 3, 4, 5, 2, 2, 9, 8, 7, 6, 7, 8, 1, 4, 1, 7, 5, 8, 2, 3, 9, 5, 7,

withPool(3) {  //feel free to experiment with the number of fork/join threads in the pool
    println """Sorted numbers: ${
        runForkJoin(numbers) {nums ->
            println "Thread ${Thread.currentThread().name[-1]}: Sorting $nums"
            switch (nums.size()) {
                case 0..1:
                    return nums                                   //store own result
                case 2:
                    if (nums[0] <= nums[1]) return nums     //store own result
                    else return nums[-1..0]                       //store own result
                default:
                    def splitList = split(nums)
                    [splitList[0], splitList[1]].each {forkOffChild it}  //fork a child task
                    return merge(* childrenResults)      //use results of children tasks to calculate and stor
            }
        }
    }"""
}
```

Mergesort example using a custom-tailored worker class

```
public final class SortWorker extends AbstractForkJoinWorker<List<Integer>> {
    private final List numbers

def SortWorker(final List<Integer> numbers) {
        this.numbers = numbers.asImmutable()
    }

/**
     * Splits a list of numbers in half
     */
    def split(List<Integer> list) {
        int listSize = list.size()
        int middleIndex = listSize / 2
        def list1 = list[0..<middleIndex]
        def list2 = list[middleIndex..listSize - 1]
        return [list1, list2]
    }

/**
     * Merges two sorted lists into one
     */
    List<Integer> merge(List<Integer> a, List<Integer> b) {
        int i = 0, j = 0
        final int newSize = a.size() + b.size()
        List<Integer> result = new ArrayList<Integer>(newSize)

while ((i < a.size()) && (j < b.size())) {
            if (a[i] <= b[j]) result << a[i++]
            else result << b[j++]
        }

if (i < a.size()) result.addAll(a[i..-1])
        else result.addAll(b[j..-1])
        return result
    }

/**
     * Sorts a small list or delegates to two children, if the list contains more than two elements.
     */
    @Override
    protected List<Integer> computeTask() {
        println "Thread ${Thread.currentThread().name[-1]}: Sorting $numbers"
        switch (numbers.size()) {
            case 0..1:
                return numbers                                   //store own result
            case 2:
                if (numbers[0] <= numbers[1]) return numbers     //store own result
                else return numbers[-1..0]                       //store own result
            default:
                def splitList = split(numbers)
                [new SortWorker(splitList[0]), new SortWorker(splitList[1])].each{forkOffChild it}  //fork a c
                return merge(* childrenResults)      //use results of children tasks to calculate and store ow
        }
    }
}

final def numbers = [1, 5, 2, 4, 3, 8, 6, 7, 3, 4, 5, 2, 2, 9, 8, 7, 6, 7, 8, 1, 4, 1, 7, 5, 8, 2, 3, 9, 5, 7,

withPool(1) {  //feel free to experiment with the number of fork/join threads in the pool
    println "Sorted numbers: ${runForkJoin(new SortWorker(numbers))}"
}
```

Running child tasks directly

The *forkOffChild()* method has a sibling - the *runChildDirectly()* method, which will run the child ta immediately within the current thread instead of scheduling the child task for asynchronous proce pool. Typically you'll call _forkOffChild() on all sub-tasks but the last, which you invoke directly wi overhead.

```
Closure fib = {number ->
            if (number <= 2) {
                return 1
            }
            forkOffChild(number - 1)                           //  This task will run asynchronously, probabl
            final def result = runChildDirectly(number - 2)    //  This task is run directly within the curre
            return (Integer) getChildrenResults().sum() + result
        }

withPool {
            assert 55 == runForkJoin(10, fib)
        }
```

Availability

This feature is only available when using in the Fork/Join-based *GParsPool* , not in *GParsExecut*

3.7 Parallel Speculations

With processor cores having become plentiful, some algorithms might benefit from brutal-force p
Instead of deciding up-front about how to solve a problem, what algorithm to use or which locatio
run all potential solutions in parallel.

Parallel speculations

Imagine you need to perform a task like e.g. calculate an expensive function or read data from a
internet. Luckily, you know of several good ways (e.g. functions or urls) to achieve your goal. Ho
equal. Although they return back the same (as far as your needs are concerned) result, they may
amount of time to complete and some of them may even fail (e.g. network issues). What's worse
you which path gives you the solution first nor which paths lead to no solution at all. Shall I run *q*
on my list? Which url will work best? Is this service available at its primary location or should I us

GPars speculations give you the option to try all the available alternatives in parallel and so get t
fastest functional path, silently ignoring the slow or broken ones.

This is what the *speculate()* methods on *GParsPool* and *GParsExecutorsPool()* can do.

```
def numbers = …
def quickSort = …
def mergeSort = …
def sortedNumbers = speculate(quickSort, mergeSort)
```

Here we're performing both *quick sort* and *merge sort* **concurrently**, while getting the result of th
parallel resources available these days on mainstream hardware, running the two functions in pa
dramatic impact on speed of calculation of either one, and so we get the result in about the same
solely the faster of the two calculations. And we get the result sooner than when running the slow
have to know up-front, which of the two sorting algorithms would perform better on our data. Thu

Similarly, downloading a document from multiple sources of different speed and reliability would

```
import static groovyx.gpars.GParsPool.speculate
import static groovyx.gpars.GParsPool.withPool

def alternative1 = {
    'http://www.dzone.com/links/index.html'.toURL().text
}

def alternative2 = {
    'http://www.dzone.com/'.toURL().text
}

def alternative3 = {
    'http://www.dzzzzzone.com/'.toURL().text  //wrong url
}

def alternative4 = {
    'http://dzone.com/'.toURL().text
}

withPool(4) {
    println speculate([alternative1, alternative2, alternative3, alternative4]).contains('groovy')
}
```

⚠ Make sure the surrounding thread pool has enough threads to process all alternatives
parallel. The size of the pool should match the number of closures supplied.

Alternatives using dataflow variables and streams

In cases, when stopping unsuccessful alternatives is not needed, dataflow variables or streams r
the result value from the winning speculation.

> ⚠ Please refer to the Dataflow Concurrency section of the User Guide for details on Dat
> variables and streams.

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

def alternative1 = {
    'http://www.dzone.com/links/index.html'.toURL().text
}

def alternative2 = {
    'http://www.dzone.com/'.toURL().text
}

def alternative3 = {
    'http://www.dzzzzzone.com/'.toURL().text  //will fail due to wrong url
}

def alternative4 = {
    'http://dzone.com/'.toURL().text
}

//Pick either one of the following, both will work:
final def result = new DataflowQueue()
//  final def result = new DataflowVariable()

[alternative1, alternative2, alternative3, alternative4].each {code ->
    task {
        try {
            result << code()
        } catch (ignore) { }  //We deliberately ignore unsuccessful urls
    }
}

println result.val.contains('groovy')
```

4 Groovy CSP

The CSP (Communicating Sequential Processes) abstraction builds on independent composable exchange messages in a synchronous manner. GPars leverages the JCSP library developed at t UK.

Jon Kerridge, the author of the CSP implementation in GPars, provides exhaustive examples on his website:

> ⚠ The GroovyCSP implementation leverages JCSP, a Java-based CSP library, which is under LGPL. There are some differences between the Apache 2 license, which GPar and LGPL. Please make sure your application conforms to the LGPL rules before ena use of JCSP in your code.

If the LGPL license is not adequate for your use, you might consider checking out the Dataflow C this User Guide to learn about *tasks* , *selectors* and *operators* , which may help you resolve conc similar to the CSP approach. In fact the dataflow and CSP concepts, as implemented in GPars, s each other.

> ⚠ By default, without actively adding an explicit dependency on JCSP in your build file c downloading and including the JCSP jar file in your project, the standard commercial-software-friendly Apache 2 License terms apply to your project. GPars di depends on software licensed under licenses compatible with the Apache 2 License.

The CSP model principles

In essence, the CSP model builds on independent concurrent processes, which mutually commu channels using synchronous (i.e. rendezvous) message passing. Unlike actors or dataflow opera around the event-processing pattern, CSP processes place focus the their activities (aka sequen communication to stay mutually in sync along the way.

Since the addressing is indirect through channels, the processes do not need to know about one typically consist of a set of input and output channels and a body. Once a CSP process is startec from a thread pool and starts processing its body, pausing only when reading from a channel or v Some implementations (e.g. GoLang) can also detach the thread from the CSP process when blo

CSP programs are deterministic. The same data on the program's input will always generate the irrespective of the actual thread-scheduling scheme used. This helps a lot when debugging CSP analyzing deadlocks.

Determinism combined with indirect addressing result in a great level of composability of CSP pro combine small CSP processes into bigger ones just by connecting their input and output channel them by another, bigger containing process.

The CSP model introduces non-determinism using *Alternatives* . A process can attempt to read a channels at the same time through a construct called *Alternative* or *Select* . The first value that b any of the channels involved in the *Select* will be read and consumed by the process. Since the c received through a *Select* depends on unpredictable conditions during program run-time, the valu non-deterministic.

CSP with GPars dataflow

GPars provides all the necessary building blocks to create CSP processes.

- **CSP Processes** can be modelled through GPars tasks using a *Closure* , a *Runnable* or a *C*
actual implementation of the process
- **CSP Channels** should be modelled with *SyncDataflowQueue* and *SyncDataflowBroadcast* c
- **CSP Alternative** is provided through the *Select* class with its *select* and *prioritySelect* metho

Processes

To start a process simply use the *task* factory method.

```
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.scheduler.ResizeablePool

group = new DefaultPGroup(new ResizeablePool(true))

def t = group.task {
    println "I am a process"
}

t.join()
```

> Since each process consumes a thread for its lifetime, it is advisable to use resizeabl
> pools as in the example above.

A process can also be created from a Runnable or Callable object:

```
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.scheduler.ResizeablePool

group = new DefaultPGroup(new ResizeablePool(true))

class MyProcess implements Runnable {

@Override
    void run() {
        println "I am a process"
    }
}
def t = group.task new MyProcess()

t.join()
```

Using Callable allows for values to be returned through the *get()* method:

```
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.scheduler.ResizeablePool

import java.util.concurrent.Callable

group = new DefaultPGroup(new ResizeablePool(true))

class MyProcess implements Callable<String> {

@Override
    String call() {
        println "I am a process"
        return "CSP is great!"
    }
}
def t = group.task new MyProcess()

println t.get()
```

Channels

Processes typically need channels to communicate with the other processes as well as with the c

```
import groovy.transform.TupleConstructor
import groovyx.gpars.dataflow.DataflowReadChannel
import groovyx.gpars.dataflow.DataflowWriteChannel
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.scheduler.ResizeablePool

import java.util.concurrent.Callable
import groovyx.gpars.dataflow.SyncDataflowQueue

group = new DefaultPGroup(new ResizeablePool(true))

@TupleConstructor
class Greeter implements Callable<String> {
    DataflowReadChannel names
    DataflowWriteChannel greetings

@Override
    String call() {
        while(!Thread.currentThread().isInterrupted()) {
            String name = names.val
            greetings << "Hello " + name
        }
        return "CSP is great!"
    }
}

def a = new SyncDataflowQueue()
def b = new SyncDataflowQueue()

group.task new Greeter(a, b)

a << "Joe"
a << "Dave"
println b.val
println b.val
```

⚠ The CSP model uses synchronous messaging, however, in GPars you may consider asynchronous channels as well as synchronous ones. You can also combine these tw of channels within the same process.

Composition

Grouping processes is then just a matter of connecting them with channels:

```
group = new DefaultPGroup(new ResizeablePool(true))

@TupleConstructor
class Formatter implements Callable<String> {
    DataflowReadChannel rawNames
    DataflowWriteChannel formattedNames

@Override
    String call() {
        while(!Thread.currentThread().isInterrupted()) {
            String name = rawNames.val
            formattedNames << name.toUpperCase()
        }
    }
}

@TupleConstructor
class Greeter implements Callable<String> {
    DataflowReadChannel names
    DataflowWriteChannel greetings

@Override
    String call() {
        while(!Thread.currentThread().isInterrupted()) {
            String name = names.val
            greetings << "Hello " + name
        }
    }
}

def a = new SyncDataflowQueue()
def b = new SyncDataflowQueue()
def c = new SyncDataflowQueue()

group.task new Formatter(a, b)
group.task new Greeter(b, c)

a << "Joe"
a << "Dave"
println c.val
println c.val
```

Alternatives

To introduce non-determinist GPars offers the *Select* class with its *select* and *prioritySelect* meth

```
import groovy.transform.TupleConstructor
import groovyx.gpars.dataflow.SyncDataflowQueue
import groovyx.gpars.dataflow.DataflowReadChannel
import groovyx.gpars.dataflow.DataflowWriteChannel
import groovyx.gpars.dataflow.Select
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.scheduler.ResizeablePool

import static groovyx.gpars.dataflow.Dataflow.select

group = new DefaultPGroup(new ResizeablePool(true))

@TupleConstructor
class Receptionist implements Runnable {
    DataflowReadChannel emails
    DataflowReadChannel phoneCalls
    DataflowReadChannel tweets
    DataflowWriteChannel forwardedMessages

private final Select incomingRequests = select([phoneCalls, emails, tweets])  //prioritySelect() would give hi
phone calls

@Override
    void run() {
        while(!Thread.currentThread().isInterrupted()) {
            String msg = incomingRequests.select()
            forwardedMessages << msg.toUpperCase()
        }
    }
}

def a = new SyncDataflowQueue()
def b = new SyncDataflowQueue()
def c = new SyncDataflowQueue()
def d = new SyncDataflowQueue()

group.task new Receptionist(a, b, c, d)

a << "my email"
b << "my phone call"
c << "my tweet"

//The values come in random order since the process uses a Select to read its input
3.times{
    println d.val.value
}
```

Components

CSP processes can be composed into larger entities. Suppose you already have a set of CSP pr
Runnable/Callable classes), you can compose them into a larger process:

```
final class Prefix implements Callable {
    private final DataflowChannel inChannel
    private final DataflowChannel outChannel
    private final def prefix

def Prefix(final inChannel, final outChannel, final prefix) {
        this.inChannel = inChannel;
        this.outChannel = outChannel;
        this.prefix = prefix
    }

public def call() {
        outChannel << prefix
        while (true) {
            sleep 200
            outChannel << inChannel.val
        }
    }
}
```

```
final class Copy implements Callable {
    private final DataflowChannel inChannel
    private final DataflowChannel outChannel1
    private final DataflowChannel outChannel2

def Copy(final inChannel, final outChannel1, final outChannel2) {
        this.inChannel = inChannel;
        this.outChannel1 = outChannel1;
        this.outChannel2 = outChannel2;
    }

public def call() {
        final PGroup group = Dataflow.retrieveCurrentDFPGroup()
        while (true) {
            def i = inChannel.val
            group.task {
                outChannel1 << i
                outChannel2 << i
            }.join()
        }
    }
}
```

```
import groovyx.gpars.dataflow.DataflowChannel
import groovyx.gpars.dataflow.SyncDataflowQueue
import groovyx.gpars.group.DefaultPGroup

group = new DefaultPGroup(6)

def fib(DataflowChannel out) {
    group.task {
        def a = new SyncDataflowQueue()
        def b = new SyncDataflowQueue()
        def c = new SyncDataflowQueue()
        def d = new SyncDataflowQueue()
        [new Prefix(d, a, 0L), new Prefix(c, d, 1L), new Copy(a, b, out), new StatePairs(b, c)].each { group.t
    }
}

final SyncDataflowQueue ch = new SyncDataflowQueue()
group.task new Print('Fibonacci numbers', ch)
fib(ch)

sleep 10000
```

5 Actors

The actor support in GPars was originally inspired by the Actors library in Scala, but has since gc Scala offers as standard.

Actors allow for a message passing-based concurrency model: programs are collections of indep that exchange messages and have no mutable shared state. Actors can help developers avoid is deadlock, live-lock and starvation, which are common problems for shared memory based appro way of leveraging the multi-core nature of today's hardware without all the problems traditionally shared-memory multi-threading, which is why programming languages such as Erlang and Scala model.

A nice article summarizing the key concepts behind actors was written recently by Ruben Verme guarantee that **at most one thread processes the actor's body** at any one time and also, unde memory gets synchronized each time a thread gets assigned to an actor so the actor's state **can** by code in the body **without any other extra (synchronization or locking) effort** . Ideally actor **be invoked** directly from outside so all the code of the actor class can only be executed by the th received message and so all the actor's code is **implicitly thread-safe** . If any of the actor's metl called by other objects directly, the thread-safety guarantee for the actor's code and state are **no**

Types of actors

In general, you can find two types of actors in the wild - ones that hold **implicit state** and those, \ you both options. **Stateless** actors, represented in **GPars** by the *DynamicDispatchActor* and the classes, keep no track of what messages have arrived previously. You may thing of these as flat which process messages as they come. Any state-based behavior has to be implemented by the

The **stateful** actors, represented in GPars by the *DefaultActor* class (and previously also by the / class), allow the user to handle implicit state directly. After receiving a message the actor moves different ways to handle future messages. To give you an example, a freshly started actor may o of messages, e.g. encrypted messages for decryption, only after it has received the encryption k allow to encode such dependencies directly in the structure of the message-handling code. Impli however, comes at a slight performance cost, mainly due to the lack of continuations support on

Actor threading model

Since actors are detached from the system threads, a great number of actors can share a relativ This can go as far as having many concurrent actors that share a single pooled thread. This arch avoid some of the threading limitations of the JVM. In general, while the JVM can only give you a threads (typically around a couple of thousands), the number of actors is only limited by the avail actor has no work to do, it doesn't consume threads.

Actor code is processed in chunks separated by quiet periods of waiting for new events (messag naturally modeled through *continuations* . As JVM doesn't support continuations directly, they ha the actors frameworks, which has slight impact on organization of the actors' code. However, the outweigh the difficulties.

```
import groovyx.gpars.actor.Actor
import groovyx.gpars.actor.DefaultActor

class GameMaster extends DefaultActor {
    int secretNum

void afterStart() {
        secretNum = new Random().nextInt(10)
    }

void act() {
        loop {
            react { int num ->
                if (num > secretNum)
                reply 'too large'
                else if (num < secretNum)
                reply 'too small'
                else {
                    reply 'you win'
                    terminate()
                }
            }
        }
    }
}

class Player extends DefaultActor {
    String name
    Actor server
    int myNum

void act() {
        loop {
            myNum = new Random().nextInt(10)
            server.send myNum
            react {
                switch (it) {
                    case 'too large': println "$name: $myNum was too large"; break
                    case 'too small': println "$name: $myNum was too small"; break
                    case 'you win': println "$name: I won $myNum"; terminate(); break
                }
            }
        }
    }
}

def master = new GameMaster().start()
def player = new Player(name: 'Player', server: master).start()

//this forces main thread to live until both actors stop
[master, player]*.join()
```

example by *Jordi Campos i Miralles, Departament de Matem tica Aplicada i An lisi, MAiA Faculta Universitat de Barcelona*

Usage of Actors

Gpars provides consistent Actor APIs and DSLs. Actors in principal perform three specific operat receive messages and create new actors. Although not specifically enforced by **GPars** message or at least follow the **hands-off** policy when the sender never touches the messages after the me off.

Sending messages

Messages can be sent to actors using the *send()* method.

```
def passiveActor = Actors.actor{
    loop {
        react { msg -> println "Received: $msg"; }
    }
}
passiveActor.send 'Message 1'
passiveActor << 'Message 2'    //using the << operator
passiveActor 'Message 3'       //using the implicit call() method
```

Alternatively, the << operator or the implicit *call()* method can be used. A family of *sendAndWait(* to block the caller until a reply from the actor is available. The *reply* is returned from the *sendAnc* return value. The *sendAndWait()* methods may also return after a timeout expires or in case of te actor.

```
def replyingActor = Actors.actor{
    loop {
        react { msg ->
            println "Received: $msg";
            reply "I've got $msg"
        }
    }
}
def reply1 = replyingActor.sendAndWait('Message 4')
def reply2 = replyingActor.sendAndWait('Message 5', 10, TimeUnit.SECONDS)
use (TimeCategory) {
    def reply3 = replyingActor.sendAndWait('Message 6', 10.seconds)
}
```

The *sendAndContinue()* method allows the caller to continue its processing while the supplied clo reply from the actor.

```
friend.sendAndContinue 'I need money!', {money -> pocket money}
println 'I can continue while my friend is collecting money for me'
```

The *sendAndPromise()* method returns a *Promise* (aka Future) to the final reply and so allows th processing while the actor is handling the submitted message.

```
Promise loan = friend.sendAndPromise 'I need money!'
println 'I can continue while my friend is collecting money for me'
loan.whenBound {money -> pocket money}  //asynchronous waiting for a reply
println "Received ${loan.get()}"  //synchronous waiting for a reply
```

All *send()* , *sendAndWait()* or *sendAndContinue()* methods will throw an exception if invoked on a

Receiving messages

Non-blocking message retrieval

Calling the *react()* method, optionally with a timeout parameter, from within the actor's code will c
message from the actor's inbox, potentially waiting, if there is no message to be processed imme

```
println 'Waiting for a gift'
react {gift ->
    if (myWife.likes gift) reply 'Thank you!'
}
```

Under the covers the supplied closure is not invoked directly, but scheduled for processing by an pool once a message is available. After scheduling the current thread will then be detached from process any other actor, which has received a message already.

To allow detaching actors from the threads the *react()* method demands the code to be written in **Continuation-style**.

```
Actors.actor {
    loop {
        println 'Waiting for a gift'
        react {gift ->
            if (myWife.likes gift) reply 'Thank you!'
            else {
                reply 'Try again, please'
                react {anotherGift ->
                    if (myChildren.like gift) reply 'Thank you!'
                }
                println 'Never reached'
            }
        }
        println 'Never reached'
    }
    println 'Never reached'
}
```

The *react()* method has a special semantics to allow actors to be detached from threads when nc available in their mailbox. Essentially, *react()* schedules the supplied code (closure) to be execut arrival and returns. The closure supplied to the *react()* methods is the code where the computatic Thus **continuation style** .

Since actor has to preserve the guarantee of at most one thread active within the actor's body, th cannot be handled before the current message processing finishes. Typically, there shouldn't be after calls to *react()* . Some actor implementations even enforce this, however, GPars does not fc reasons. The *loop()* method allows iteration within the actor body. Unlike typical looping construc loops, *loop()* cooperates with nested *react()* blocks and will ensure looping across subsequent m

Sending replies

The *reply/replyIfExists* methods are not only defined on the actors themselves, but for *AbstractPo* available in *DefaultActor* , *DynamicDispatchActor* nor *ReactiveActor* classes) also on the process themselves upon their reception, which is particularly handy when handling multiple messages in cases *reply()* invoked on the actor sends a reply to authors of all the currently processed messag whereas *reply()* called on messages sends a reply to the author of the particular message only.

See demo here

The sender property

Messages upon retrieval offer the sender property to identify the originator of the message. The p
inside the Actor's closure:

```
react {tweet ->
    if (isSpam(tweet)) ignoreTweetsFrom sender
    sender.send 'Never write me again!'
}
```

Forwarding

When sending a message, a different actor can be specified as the sender so that potential replie
be forwarded to the specified actor and not to the actual originator.

```
def decryptor = Actors.actor {
    react {message ->
        reply message.reverse()
//        sender.send message.reverse()    //An alternative way to send replies
    }
}

def console = Actors.actor {  //This actor will print out decrypted messages, since the replies are forwarded
    react {
        println 'Decrypted message: ' + it
    }
}

decryptor.send 'lellarap si yvoorG', console  //Specify an actor to send replies to
console.join()
```

Creating Actors

Actors share a **pool** of threads, which are dynamically assigned to actors when the actors need t
sent to them. The threads are returned to back the pool once a message has been processed an
waiting for some more messages to arrive.

For example, this is how you create an actor that prints out all messages that it receives.

```
def console = Actors.actor {
    loop {
        react {
            println it
        }
    }
}
```

Notice the *loop()* method call, which ensures that the actor doesn't stop after having processed tl

Here's an example with a decryptor service, which can decrypt submitted messages and send th
back to the originators.

```
final def decryptor = Actors.actor {
    loop {
        react {String message ->
            if ('stopService' == message) {
                println 'Stopping decryptor'
                stop()
            }
            else reply message.reverse()
        }
    }
}

Actors.actor {
    decryptor.send 'lellarap si yvoorG'
    react {
        println 'Decrypted message: ' + it
        decryptor.send 'stopService'
    }
}.join()
```

Here's an example of an actor that waits for up to 30 seconds to receive a reply to its message.

```
def friend = Actors.actor {
    react {
        //this doesn't reply -> caller won't receive any answer in time
        println it
        //reply 'Hello' //uncomment this to answer conversation
        react {
            println it
        }
    }
}

def me = Actors.actor {
    friend.send('Hi')
    //wait for answer 1sec
    react(1000) {msg ->
        if (msg == Actor.TIMEOUT) {
            friend.send('I see, busy as usual. Never mind.')
            stop()
        } else {
            //continue conversation
            println "Thank you for $msg"
        }
    }
}

me.join()
```

Undelivered messages

Sometimes messages cannot be delivered to the target actor. When special action needs to be t messages, at actor termination all unprocessed messages from its queue have their *onDeliveryE* The *onDeliveryError()* method or closure defined on the message can, for example, send a notifi original sender of the message.

```
final DefaultActor me
me = Actors.actor {
    def message = 1

message.metaClass.onDeliveryError = {->
        //send message back to the caller
        me << "Could not deliver $delegate"
    }

def actor = Actors.actor {
        react {
            //wait 2sec in order next call in demo can be emitted
            Thread.sleep(2000)
            //stop actor after first message
            stop()
        }
    }

actor << message
    actor << message

react {
        //print whatever comes back
        println it
    }

}
me.join()
```

Alternatively the *onDeliveryError()* method can be specified on the sender itself. The method can dynamically

```
final DefaultActor me
me = Actors.actor {
    def message1 = 1
    def message2 = 2

def actor = Actors.actor {
        react {
            //wait 2sec in order next call in demo can be emitted
            Thread.sleep(2000)
            //stop actor after first message
            stop()
        }
    }

me.metaClass.onDeliveryError = {msg ->
        //callback on actor inaccessibility
        println "Could not deliver message $msg"
    }

actor << message1
    actor << message2

actor.join()

}
me.join()
```

and statically in actor definition:

```
class MyActor extends DefaultActor {
    public void onDeliveryError(msg) {
        println "Could not deliver message $msg"
    }
    …
}
```

Joining actors

Actors provide a *join()* method to allow callers to wait for the actor to terminate. A variant accepti available. The Groovy *spread-dot* operator comes in handy when joining multiple actors at a time

```
def master = new GameMaster().start()
def player = new Player(name: 'Player', server: master).start()

[master, player]*.join()
```

Conditional and counting loops

The *loop()* method allows for either a condition or a number of iterations to be specified, optionall closure to invoke once the loop finishes - *After Loop Termination Code Handler* .

The following actor will loop three times to receive 3 messages and then prints out the maximum messages.

```
final Actor actor = Actors.actor {
    def candidates = []
    def printResult = {-> println "The best offer is ${candidates.max()}"}

loop(3, printResult) {
        react {
            candidates << it
        }
    }
}

actor 10
actor 30
actor 20
actor.join()
```

The following actor will receive messages until a value greater then 30 arrives.

```
final Actor actor = Actors.actor {
    def candidates = []
    final Closure printResult = {-> println "Reached best offer - ${candidates.max()}"}

loop({-> candidates.max() < 30}, printResult) {
        react {
            candidates << it
        }
    }
}

actor 10
actor 20
actor 25
actor 31
actor 20
actor.join()
```

> The *After Loop Termination Code Handler* can use actor's *react{}* but not *loop()* .

> *DefaultActor* can be set to behave in a fair on non-fair (default) manner. Depending o strategy chosen, the actor either makes the thread available to other actors sharing tl parallel group (fair), or keeps the thread fot itself until the message queue gets empty Generally, non-fair actors perform 2 - 3 times better than fair ones.
>
> Use either the *fairActor()* factory method or the actor's makeFair() method.

Custom schedulers

Actors leverage the standard JDK concurrency library by default. To provide a custom thread sch appropriate constructor parameter when creating a parallel group (PGroup class). The supplied s orchestrate threads in the group's thread pool.

Please also see the numerous [Actor Demos](#) .

5.1 Actors Principles

Actors share a **pool** of threads, which are dynamically assigned to actors when the actors need t sent to them. The threads are returned back to the pool once a message has been processed an waiting for some more messages to arrive. Actors become detached from the underlying threads small thread pool can serve potentially unlimited number of actors. Virtually unlimited scalability i the main advantage of *event-based actors* , which are detached from the underlying physical thre

Here are some examples of how to use actors. This is how you create an actor that prints out all receives.

```
import static groovyx.gpars.actor.Actors.actor

def console = actor {
    loop {
        react {
            println it
        }
    }
```

Notice the *loop()* method call, which ensures that the actor doesn't stop after having processed th

As an alternative you can extend the *DefaultActor* class and override the *act()* method. Once you you need to start it so that it attaches itself to the thread pool and can start accepting messages. method will take care of starting the actor.

```
class CustomActor extends DefaultActor {
    @Override
    protected void act() {
        loop {
            react {
                println it
            }
        }
    }
}

def console=new CustomActor()
console.start()
```

Messages can be sent to the actor using multiple methods

```
console.send('Message')
console 'Message'
console.sendAndWait 'Message'                                                     //Wait for a reply
console.sendAndContinue 'Message', {reply -> println "I received reply: $reply"}  //Forward the reply to a fur
```

Creating an asynchronous service

```
import static groovyx.gpars.actor.Actors.actor

final def decryptor = actor {
    loop {
        react {String message->
            reply message.reverse()
        }
    }
}

def console = actor {
    decryptor.send 'lellarap si yvoorG'
    react {
        println 'Decrypted message: ' + it
    }
}

console.join()
```

As you can see, you create new actors with the *actor()* method passing in the actor's body as a c
Inside the actor's body you can use *loop()* to iterate, *react()* to receive messages and *reply()* to s
actor, which has sent the currently processed message. The sender of the current message is al
the actor's *sender* property. When the decryptor actor doesn't find a message in its message que
react() is called, the *react()* method gives up the thread and returns it back to the thread pool for
up. Only after a new message arrives to the actor's message queue, the closure of the *react()* m
for processing with the pool. Event-based actors internally simulate continuations - actor's work i
run chunks, which get invoked once a message is available in the inbox. Each chunk for a single
performed by a different thread from the thread pool.

Groovy flexible syntax with closures allows our library to offer multiple ways to define actors. For
example of an actor that waits for up to 30 seconds to receive a reply to its message. Actors allo
org.codehaus.groovy.runtime.TimeCategory class to be used for timeout specification to the *reac*
the user wraps the call within a *TimeCategory* use block.

```
def friend = Actors.actor {
    react {
        //this doesn't reply -> caller won't receive any answer in time
        println it
        //reply 'Hello' //uncomment this to answer conversation
        react {
            println it
        }
    }
}

def me = Actors.actor {
    friend.send('Hi')
    //wait for answer 1sec
    react(1000) {msg ->
        if (msg == Actor.TIMEOUT) {
            friend.send('I see, busy as usual. Never mind.')
            stop()
        } else {
            //continue conversation
            println "Thank you for $msg"
        }
    }
}

me.join()
```

When a timeout expires when waiting for a message, the Actor.TIMEOUT message arrives inste
onTimeout() handler is invoked, if present on the actor:

```
def friend = Actors.actor {
    react {
        //this doesn't reply -> caller won't receive any answer in time
        println it
        //reply 'Hello' //uncomment this to answer conversation
        react {
            println it
        }
    }
}
def me = Actors.actor {
    friend.send('Hi')
delegate.metaClass.onTimeout = {->
        friend.send('I see, busy as usual. Never mind.')
        stop()
    }
//wait for answer 1sec
    react(1000) {msg ->
        if (msg != Actor.TIMEOUT) {
            //continue conversation
            println "Thank you for $msg"
        }
    }
}
me.join()
```

Notice the possibility to use Groovy meta-programming to define actor's lifecycle notification metl) dynamically. Obviously, the lifecycle methods can be defined the usual way when you decide tc your actor.

```
class MyActor extends DefaultActor {
    public void onTimeout() {
        …
    }
protected void act() {
       …
    }
}
```

Actors guarantee thread-safety for non-thread-safe code

Actors guarantee that always at most one thread processes the actor's body at a time and also u memory gets synchronized each time a thread gets assigned to an actor so the actor's state **can** by code in the body **without any other extra (synchronization or locking) effort** .

```
class MyCounterActor extends DefaultActor {
    private Integer counter = 0
protected void act() {
        loop {
            react {
                counter++
            }
        }
    }
}
```

Ideally actor's code should **never be invoked** directly from outside so all the code of the actor cla executed by the thread handling the last received message and so all the actor's code is **implicit** of the actor's methods is allowed to be called by other objects directly, the thread-safety guarante and state are **no longer valid** .

Simple calculator

A little bit more realistic example of an event-driven actor that receives two numeric messages, s
sends the result to the console actor.

```
import groovyx.gpars.group.DefaultPGroup

//not necessary, just showing that a single-threaded pool can still handle multiple actors
def group = new DefaultPGroup(1);

final def console = group.actor {
    loop {
        react {
            println 'Result: ' + it
        }
    }
}

final def calculator = group.actor {
    react {a ->
        react {b ->
            console.send(a + b)
        }
    }
}

calculator.send 2
calculator.send 3

calculator.join()
group.shutdown()
```

Notice that event-driven actors require special care regarding the *react()* method. Since *event_dı*
split the code into independent chunks assignable to different threads sequentially and **continua**
supported on JVM, the chunks are created artificially. The *react()* method creates the next messa
as the current message handler finishes, the next message handler (continuation) gets schedule

Concurrent Merge Sort Example

For comparison I'm also including a more involved example performing a concurrent merge sort
using actors. You can see that thanks to flexibility of Groovy we came pretty close to the Scala m
miss Scala pattern matching for message handling.

```
import groovyx.gpars.group.DefaultPGroup
import static groovyx.gpars.actor.Actors.actor

Closure createMessageHandler(def parentActor) {
    return {
        react {List<Integer> message ->
            assert message != null
            switch (message.size()) {
                case 0..1:
                    parentActor.send(message)
                    break
                case 2:
                    if (message[0] <= message[1]) parentActor.send(message)
                    else parentActor.send(message[-1..0])
                    break
                default:
                    def splitList = split(message)

def child1 = actor(createMessageHandler(delegate))
                    def child2 = actor(createMessageHandler(delegate))
                    child1.send(splitList[0])
                    child2.send(splitList[1])

react {message1 ->
                        react {message2 ->
                            parentActor.send merge(message1, message2)
                        }
                    }
            }
        }
    }
}
def console = new DefaultPGroup(1).actor {
    react {
        println "Sorted array:t${it}"
        System.exit 0
    }
}

def sorter = actor(createMessageHandler(console))
sorter.send([1, 5, 2, 4, 3, 8, 6, 7, 3, 9, 5, 3])
console.join()

def split(List<Integer> list) {
    int listSize = list.size()
    int middleIndex = listSize / 2
    def list1 = list[0..<middleIndex]
    def list2 = list[middleIndex..listSize - 1]
    return [list1, list2]
}

List<Integer> merge(List<Integer> a, List<Integer> b) {
    int i = 0, j = 0
    final int newSize = a.size() + b.size()
    List<Integer> result = new ArrayList<Integer>(newSize)

while ((i < a.size()) && (j < b.size())) {
        if (a[i] <= b[j]) result << a[i++]
        else result << b[j++]
    }

if (i < a.size()) result.addAll(a[i..-1])
    else result.addAll(b[j..-1])
    return result
}
```

Since *actors* reuse threads from a pool, the script will work with virtually **any size of a thread po** many actors are created along the way.

Actor lifecycle methods

Each Actor can define lifecycle observing methods, which will be called whenever a certain lifecy

- afterStart() - called right after the actor has been started.
- afterStop(List undeliveredMessages) - called right after the actor is stopped, passing in all th messages from the queue.
- onInterrupt(InterruptedException e) - called when the actor's thread gets interrupted. Thread in the stopping the actor in any case.
- onTimeout() - called when no messages are sent to the actor within the timeout specified for react method.
- onException(Throwable e) - called when an exception occurs in the actor's event handler. A return from this method.

You can either define the methods statically in your Actor class or add them dynamically to the a

```
class MyActor extends DefaultActor {
    public void afterStart() {
        …
    }
    public void onTimeout() {
        …
    }
protected void act() {
       …
    }
}
```

```
def myActor = actor {
    delegate.metaClass.onException = {
        log.error('Exception occurred', it)
    }

…
}
```

⚠ To help performance, you may consider using the *silentStart()* method instead of *star* starting a *DynamicDispatchActor* or a *ReactiveActor* . Calling *silentStart()* will by-pass the start-up machinery and as a result will also avoid calling the *afterStart()* method. D stateful nature, *DefaultActor* cannot be started silently.

Pool management

Actors can be organized into groups and as a default there's always an application-wide pooled a And just like the *Actors* abstract factory can be used to create actors in the default group, custom as abstract factories to create new actors instances belonging to these groups.

```
def myGroup = new DefaultPGroup()

def actor1 = myGroup.actor {
…
}

def actor2 = myGroup.actor {
…
}
```

The *parallelGroup* property of an actor points to the group it belongs to. It by default points to the which is *Actors.defaultActorPGroup* , and can only be changed before the actor is started.

```
class MyActor extends StaticDispatchActor<Integer> {
    private static PGroup group = new DefaultPGroup(100)

MyActor(...) {
        this.parallelGroup = group
        …
    }
}
```

The actors belonging to the same group share the **underlying thread pool** of that group. The pc **n + 1 threads**, where **n** stands for the number of **CPUs** detected by the JVM. The **pool size** can either by setting the *gpars.poolsize* system property or individually for each actor group by specif constructor parameter.

```
def myGroup = new DefaultPGroup(10)  //the pool will contain 10 threads
```

The thread pool can be manipulated through the appropriate *DefaultPGroup* class, which **delega** interface of the thread pool. For example, the *resize()* method allows you to change the pool size *resetDefaultSize()* sets it back to the default value. The *shutdown()* method can be called when y finish all tasks, destroy the pool and stop all the threads in order to exit JVM in an organized mar

```
… (n+1 threads in the default pool after startup)

Actors.defaultActorPGroup.resize 1  //use one-thread pool

… (1 thread in the pool)

Actors.defaultActorPGroup.resetDefaultSize()

… (n+1 threads in the pool)

Actors.defaultActorPGroup.shutdown()
```

As an alternative to the *DefaultPGroup* , which creates a pool of daemon threads, the *NonDaem* used when non-daemon threads are required.

```
def daemonGroup = new DefaultPGroup()

def actor1 = daemonGroup.actor {
…
}

def nonDaemonGroup = new NonDaemonPGroup()

def actor2 = nonDaemonGroup.actor {
…
}

class MyActor {
    def MyActor() {
        this.parallelGroup = nonDaemonGroup
    }

void act() {...}
}
```

Actors belonging to the same group share the **underlying thread pool**. With pooled actor group actors to leverage multiple thread pools of different sizes and so assign resources to different co system and tune their performance.

```
def coreActors = new NonDaemonPGroup(5)  //5 non-daemon threads pool
def helperActors = new DefaultPGroup(1)  //1 daemon thread pool

def priceCalculator = coreActors.actor {
…
}

def paymentProcessor = coreActors.actor {
…
}

def emailNotifier = helperActors.actor {
…
}

def cleanupActor = helperActors.actor {
…
}

//increase size of the core actor group
coreActors.resize 6

//shutdown the group's pool once you no longer need the group to release resources
helperActors.shutdown()
```

Do not forget to shutdown custom pooled actor groups, once you no longer need them and their system resources.

The default actor group

Actors that didn't have their parallelGroup property changed or that were created through any of t the *Actors* class share a common group *Actors.defaultActorPGroup* . This group uses a **resizeab** an upper limit of **1000 threads** . This gives you the comfort of having the pool automatically adjus actors. On the other hand, with a growing number of actors the pool may become too big an ineffi group your actors into your own PGroups with fixed size thread pools for all but trivial application

Common trap: App terminates while actors do not receive messages

Most likely you're using daemon threads and pools, which is the default setting, and your main th *actor.join()* on any, some or all of your actors would block the main thread until the actor terminat your actors running. Alternatively use instances of *NonDaemonPGroup* and assign some of your groups.

```
def nonDaemonGroup = new NonDaemonPGroup()
def myActor = nonDaemonGroup.actor {...}
```

alternatively

```
def nonDaemonGroup = new NonDaemonPGroup()

class MyActor extends DefaultActor {
    def MyActor() {
        this.parallelGroup = nonDaemonGroup
    }

void act() {...}
}

def myActor = new MyActor()
```

Blocking Actors

Instead of event-driven continuation-styled actors, you may in some scenarios prefer using block
actors hold a single pooled thread for their whole life-time including the time when waiting for me
some of the thread management overhead, since they never fight for threads after start, and also
straight code without the necessity of continuation style, since they only do blocking message rea
method. Obviously the number of blocking actors running concurrently is limited by the number o
the shared pool. On the other hand, blocking actors typically provide better performance compare
actors, especially when the actor's message queue rarely gets empty.

```
def decryptor = blockingActor {
    while (true) {
        receive {message ->
            if (message instanceof String) reply message.reverse()
            else stop()
        }
    }
}

def console = blockingActor {
    decryptor.send 'lellarap si yvoorG'
    println 'Decrypted message: ' + receive()
    decryptor.send false
}

[decryptor, console]*.join()
```

Blocking actors increase the number of options to tune performance of your applications. They m
good candidates for high-traffic positions in your actor network.

5.2 Stateless Actors

Dynamic Dispatch Actor

The *DynamicDispatchActor* class is an actor allowing for an alternative structure of the message
general *DynamicDispatchActor* repeatedly scans for messages and dispatches arrived message
onMessage(message) methods defined on the actor. The *DynamicDispatchActor* leverages the (
method dispatch mechanism under the covers. Since, unlike *DefaultActor* descendants, a *Dynan*
ReactiveActor (discussed below) do not need to implicitly remember actor's state between subse
receptions, they provide much better performance characteristics, generally comparable to other
e.g. Scala Actors.

```
import groovyx.gpars.actor.Actors
import groovyx.gpars.actor.DynamicDispatchActor

final class MyActor extends DynamicDispatchActor {

void onMessage(String message) {
        println 'Received string'
    }

void onMessage(Integer message) {
        println 'Received integer'
        reply 'Thanks!'
    }

void onMessage(Object message) {
        println 'Received object'
        sender.send 'Thanks!'
    }

void onMessage(List message) {
        println 'Received list'
        stop()
    }
}

final def myActor = new MyActor().start()

Actors.actor {
    myActor 1
    myActor ''
    myActor 1.0
    myActor(new ArrayList())
    myActor.join()
}.join()
```

In some scenarios, typically when no implicit conversation-history-dependent state needs to be p
the dynamic dispatch code structure may be more intuitive than the traditional one using nested
statements.

The *DynamicDispatchActor* class also provides a handy facility to add message handlers dynami
construction time or any time later using the *when* handlers, optionally wrapped inside a *become*

```
final Actor myActor = new DynamicDispatchActor().become {
    when {String msg -> println 'A String'; reply 'Thanks'}
    when {Double msg -> println 'A Double'; reply 'Thanks'}
    when {msg -> println 'A something ...'; reply 'What was that?';stop()}
}
myActor.start()
Actors.actor {
    myActor 'Hello'
    myActor 1.0d
    myActor 10 as BigDecimal
    myActor.join()
}.join()
```

Obviously the two approaches can be combined:

```
final class MyDDA extends DynamicDispatchActor {

void onMessage(String message) {
        println 'Received string'
    }

void onMessage(Integer message) {
        println 'Received integer'
    }

void onMessage(Object message) {
        println 'Received object'
    }

void onMessage(List message) {
        println 'Received list'
        stop()
    }
}

final def myActor = new MyDDA().become {
    when {BigDecimal num -> println 'Received BigDecimal'}
    when {Float num -> println 'Got a float'}
}.start()
Actors.actor {
    myActor 'Hello'
    myActor 1.0f
    myActor 10 as BigDecimal
    myActor.send([])
    myActor.join()
}.join()
```

The dynamic message handlers registered via *when* take precedence over the static *onMessage*

> *DynamicDispatchActor* can be set to behave in a fair on non-fair (default) manner. De
> on the strategy chosen, the actor either makes the thread available to other actors sh
> same parallel group (fair), or keeps the thread fot itself until the message queue gets
> (non-fair). Generally, non-fair actors perform 2 - 3 times better than fair ones.
>
> Use either the *fairMessageHandler()* factory method or the actor's makeFair() methoc

```
def fairActor = Actors.fairMessageHandler {...}
```

Static Dispatch Actor

While *DynamicDispatchActor* dispatches messages based on their run-time type and so pays ext
penalty for each message, *StaticDispatchActor* avoids run-time message checks and dispatches
based on the compile-time information.

```
final class MyActor extends StaticDispatchActor<String> {
    void onMessage(String message) {
        println 'Received string ' + message

switch (message) {
            case 'hello':
                reply 'Hi!'
                break
            case 'stop':
                stop()
        }
    }
}
```

Instances of *StaticDispatchActor* have to override the *onMessage* method appropriate for the act
parameter. The *onMessage(T message)* method is then invoked with every received message.

A shorter route towards both fair and non-fair static dispatch actors is available through the helpe

```
final actor = staticMessageHandler {String message ->
    println 'Received string ' + message

switch (message) {
        case 'hello':
            reply 'Hi!'
            break
        case 'stop':
            stop()
    }
}

println 'Reply: ' + actor.sendAndWait('hello')
actor 'bye'
actor 'stop'
actor.join()
```

Although when compared to *DynamicDispatchActor* the *StaticDispatchActor* class is limited to a s
the simplified creation without any *when* handlers plus the considerable performance benefits sh
StaticDispatchActor your default choice for straightforward message handlers, when dispatching
run-time type is not necessary. For example, *StaticDispatchActors* make dataflow operators four
to when using *DynamicDispatchActor* .

Reactive Actor

The *ReactiveActor* class, constructed typically by calling *Actors.reactor()* or *DefaultPGroup.react*
event-driven like approach. When a reactive actor receives a message, the supplied block of cod
reactive actor's body, is run with the message as a parameter. The result returned from the code

```
final def group = new DefaultPGroup()

final def doubler = group.reactor {
    2 * it
}

group.actor {
    println 'Double of 10 = ' + doubler.sendAndWait(10)
}

group.actor {
    println 'Double of 20 = ' + doubler.sendAndWait(20)
}

group.actor {
    println 'Double of 30 = ' + doubler.sendAndWait(30)
}

for(i in (1..10)) {
    println "Double of $i = ${doubler.sendAndWait(i)}"
}

doubler.stop()
doubler.join()
```

Here's an example of an actor, which submits a batch of numbers to a *ReactiveActor* for process
results gradually as they arrive.

```
import groovyx.gpars.actor.Actor
import groovyx.gpars.actor.Actors

final def doubler = Actors.reactor {
    2 * it
}

Actor actor = Actors.actor {
    (1..10).each {doubler << it}
    int i = 0
    loop {
        i += 1
        if (i > 10) stop()
        else {
            react {message ->
                println "Double of $i = $message"
            }
        }
    }
}

actor.join()
doubler.stop()
doubler.join()
```

Essentially reactive actors provide a convenience shortcut for an actor that would wait for messa
them and send back the result. This is schematically how the reactive actor looks inside:

```
public class ReactiveActor extends DefaultActor {
    Closure body

void act() {
        loop {
            react {message ->
                reply body(message)
            }
        }
    }
}
```

⚠ *ReactiveActor* can be set to behave in a fair on non-fair (default) manner. Depending
strategy chosen, the actor either makes the thread available to other actors sharing th
parallel group (fair), or keeps the thread fot itself until the message queue gets empty
Generally, non-fair actors perform 2 - 3 times better than fair ones.

Use either the *fairReactor()* factory method or the actor's makeFair() method.

```
def fairActor = Actors.fairReactor {...}
```

5.3 Tips and Tricks

Structuring actor's code

When extending the *DefaultActor* class, you can call any actor's methods from within the *act()* me
react() or *loop()* methods in them.

```
class MyDemoActor extends DefaultActor {

protected void act() {
        handleA()
    }

private void handleA() {
        react {a ->
            handleB(a)
        }
    }

private void handleB(int a) {
        react {b ->
            println a + b
            reply a + b
        }
    }
}

final def demoActor = new MyDemoActor()
demoActor.start()

Actors.actor {
    demoActor 10
    demoActor 20
    react {
        println "Result: $it"
    }
}.join()
```

Bear in mind that the methods *handleA()* and *handleB()* in all our examples will only schedule the
handlers to run as continuations of the current calculation in reaction to the next message arrivin

Alternatively, when using the *actor()* factory method, you can add event-handling code through th
closures.

```
Actor demoActor = Actors.actor {
    delegate.metaClass {
        handleA = {->
            react {a ->
                 handleB(a)
            }
        }

handleB = {a ->
            react {b ->
                println a + b
                reply a + b
            }
        }
    }

handleA()
}

Actors.actor {
    demoActor 10
    demoActor 20
    react {
        println "Result: $it"
    }
}.join()
```

Closures, which have the actor set as their delegate can also be used to structure event-handling

```
Closure handleB = {a ->
    react {b ->
        println a + b
        reply a + b
    }
}

Closure handleA = {->
    react {a ->
        handleB(a)
    }
}

Actor demoActor = Actors.actor {
    handleA.delegate = delegate
    handleB.delegate = delegate

handleA()
}

Actors.actor {
    demoActor 10
    demoActor 20
    react {
        println "Result: $it"
    }
}.join()
```

Event-driven loops

When coding event-driven actors you have to have in mind that calls to *react()* and *loop()* methoc different semantics. This becomes a bit of a challenge once you try to implement any types of loc the other hand, if you leverage the fact that *react()* only schedules a continuation and returns, yo recursively without fear to fill up the stack. Look at the examples below, which respectively use th techniques for structuring actor's code.

A subclass of *DefaultActor*

```
class MyLoopActor extends DefaultActor {

protected void act() {
        outerLoop()
    }

private void outerLoop() {
        react {a ->
            println 'Outer: ' + a
            if (a != 0) innerLoop()
            else println 'Done'
        }
    }

private void innerLoop() {
        react {b ->
            println 'Inner ' + b
            if (b == 0) outerLoop()
            else innerLoop()
        }
    }
}

final def actor = new MyLoopActor().start()
actor 10
actor 20
actor 0
actor 0
actor.join()
```

Enhancing the actor's metaClass

```
Actor actor = Actors.actor {

delegate.metaClass {
      outerLoop = {->
          react {a ->
              println 'Outer: ' + a
              if (a!=0) innerLoop()
              else println 'Done'
          }
      }

innerLoop = {->
          react {b ->
              println 'Inner ' + b
              if (b==0) outerLoop()
              else innerLoop()
          }
      }
  }

outerLoop()
}

actor 10
actor 20
actor 0
actor 0
actor.join()
```

Using Groovy closures

```
Closure innerLoop

Closure outerLoop = {->
    react {a ->
        println 'Outer: ' + a
        if (a!=0) innerLoop()
        else println 'Done'
    }
}

innerLoop = {->
    react {b ->
        println 'Inner ' + b
        if (b==0) outerLoop()
        else innerLoop()
    }
}

Actor actor = Actors.actor {
    outerLoop.delegate = delegate
    innerLoop.delegate = delegate

outerLoop()
}

actor 10
actor 20
actor 0
actor 0
actor.join()
```

Plus don't forget about the possibility to use the actor's *loop()* method to create a loop that runs u
terminates.

```
class MyLoopingActor extends DefaultActor {

protected void act() {
      loop {
          outerLoop()
      }
  }

private void outerLoop() {
      react {a ->
          println 'Outer: ' + a
          if (a!=0) innerLoop()
          else println 'Done for now, but will loop again'
      }
  }

private void innerLoop() {
      react {b ->
          println 'Inner ' + b
          if (b == 0) outerLoop()
          else innerLoop()
      }
  }
}

final def actor = new MyLoopingActor().start()
actor 10
actor 20
actor 0
actor 0
actor 10
actor.stop()
actor.join()
```

5.4 Active Objects

Active objects provide an OO facade on top of actors, allowing you to avoid dealing directly with t
having to match messages, wait for results and send replies.

Actors with a friendly facade

```
import groovyx.gpars.activeobject.ActiveObject
import groovyx.gpars.activeobject.ActiveMethod

@ActiveObject
class Decryptor {
    @ActiveMethod
    def decrypt(String encryptedText) {
        return encryptedText.reverse()
    }

@ActiveMethod
    def decrypt(Integer encryptedNumber) {
        return -1*encryptedNumber + 142
    }
}

final Decryptor decryptor = new Decryptor()
def part1 = decryptor.decrypt(' noitcA ni yvoorG')
def part2 = decryptor.decrypt(140)
def part3 = decryptor.decrypt('noitide dn')

print part1.get()
print part2.get()
println part3.get()
```

You mark active objects with the *@ActiveObject* annotation. This will ensure a hidden actor insta
instance of your class. Now you can mark methods with the *@ActiveMethod* annotation indicatin
method to be invoked asynchronously by the target object's internal actor. An optional boolean *b*
the *@ActiveMethod* annotation specifies, whether the caller should block until a result is availabl
the caller should only receive a *promise* for a future result in a form of a *DataflowVariable* and so
blocked waiting.

> ⚠ By default, all active methods are set to be **non-blocking** . However, methods, which
> their return type explicitly, must be configured as blocking, otherwise the compiler will
> error. Only *def* , *void* and *DataflowVariable* are allowed return types for non-blocking

Under the covers, GPars will translate your method call to **a message being sent to the interna** eventually handle that message by invoking the desired method on behalf of the caller and once sent back to the caller. Non-blocking methods return promises for results, aka *DataflowVariables*

But blocking means we're not really asynchronous, are we?

Indeed, if you mark your active methods as *blocking* , the caller will be blocked waiting for the res doing normal plain method invocation. All we've achieved is being thread-safe inside the Active c access. Something the *synchronized* keyword could give you as well. So it is the **non-blocking** r drive your decision towards using active objects. Blocking methods will then provide the usual sy yet give the consistency guarantees across concurrent method invocations. The blocking methoc useful when used in combination with non-blocking ones.

```
import groovyx.gpars.activeobject.ActiveMethod
import groovyx.gpars.activeobject.ActiveObject
import groovyx.gpars.dataflow.DataflowVariable

@ActiveObject
class Decryptor {
    @ActiveMethod(blocking=true)
    String decrypt(String encryptedText) {
        encryptedText.reverse()
    }

@ActiveMethod(blocking=true)
    Integer decrypt(Integer encryptedNumber) {
        -1*encryptedNumber + 142
    }
}

final Decryptor decryptor = new Decryptor()
print decryptor.decrypt(' noitcA ni yvoorG')
print decryptor.decrypt(140)
println decryptor.decrypt('noitide dn')
```

Non-blocking semantics

Now calling the non-blocking active method will return as soon as the actor has been sent a mes now allowed to do whatever he likes, while the actor is taking care of the calculation. The state o polled using the *bound* property on the promise. Calling the *get()* method on the returned promis until a value is available. The call to *get()* will eventually return a value or throw an exception, de outcome of the actual calculation.

> ⚠ The *get()* method has also a variant with a timeout parameter, if you want to avoid the waiting indefinitely.

Annotation rules

There are a few rules to follow when annotating your objects:

1. The *ActiveMethod* annotations are only accepted in classes annotated as *ActiveObject*
2. Only instance (non-static) methods can be annotated as *ActiveMethod*
3. You can override active methods with non-active ones and vice versa
4. Subclasses of active objects can declare additional active methods, provided they are thems *ActiveObject*
5. Combining concurrent use of active and non-active methods may result in race conditions. Io active objects as completely encapsulated classes with all non-private methods marked as a

Inheritance

The *@ActiveObject* annotation can appear on any class in an inheritance hierarchy. The actor fie in top-most annotated class in the hierarchy, the subclasses will reuse the field.

```
import groovyx.gpars.activeobject.ActiveObject
import groovyx.gpars.activeobject.ActiveMethod
import groovyx.gpars.dataflow.DataflowVariable

@ActiveObject
class A {
    @ActiveMethod
    def fooA(value) {
        …
    }
}

class B extends A {
}

@ActiveObject
class C extends B {
    @ActiveMethod
    def fooC(value1, value2) {
        …
    }
}
```

In our example the actor field will be generated into class *A* . Class *C* has to be annotated with @ holds the *@ActiveMethod* annotation on method *fooC()* , while class *B* does not need the annota methods is active.

Groups

Just like actors can be grouped around thread pools, active objects can be configured to use thre parallel groups.

```
@ActiveObject("group1")
class MyActiveObject {
    …
}
```

The *value* parameter to the *@ActiveObject* annotation specifies a name of parallel group to bind Only threads from the specified group will be used to run internal actors of instances of the class need to be created and registered prior to creation of any of the active object instances belonging specified explicitly, an active object will use the default actor group - *Actors.defaultActorPGroup* .

```
final DefaultPGroup group = new DefaultPGroup(10)
ActiveObjectRegistry.instance.register("group1", group)
```

Alternative names for the internal actor

You will probably only rarely run into name collisions with the default name for the active object's
May you need to change the default name *internalActiveObjectActor* , use the *actorName* param
@ActiveObject annotation.

```
@ActiveObject(actorName = "alternativeActorName")
class MyActiveObject {
    …
}
```

⚠ Alternative names for internal actors as well as their desired groups cannot be overric
subclasses. Make sure you only specify these values in the top-most active objects ir
inheritance hierarchy. Obviously, the top most active object is still allowed to subclass
classes, just none of the predecessors must be an active object.

5.5 Classic Examples

A few examples on Actors use

Examples

- The Sieve of Eratosthenes
- Sleeping Barber
- Dining Philosophers
- Word Sort
- Load Balancer

The Sieve of Eratosthenes

[Problem description]

```
import groovyx.gpars.actor.DynamicDispatchActor

/**
 * Demonstrates concurrent implementation of the Sieve of Eratosthenes using actors
 *
 * In principle, the algorithm consists of 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) -> (filte
by mod 11) -> (caution! Primes falling out here)
 * The chain is built (grows) on the fly, whenever a new prime is found.
 */

int requestedPrimeNumberBoundary = 1000

final def firstFilter = new FilterActor(2).start()

/**
 * Generating candidate numbers and sending them to the actor chain
 */
(2..requestedPrimeNumberBoundary).each {
    firstFilter it
}
firstFilter.sendAndWait 'Poison'

/**
 * Filter out numbers that can be divided by a single prime number
 */
final class FilterActor extends DynamicDispatchActor {
    private final int myPrime
    private def follower

def FilterActor(final myPrime) { this.myPrime = myPrime; }

/**
     * Try to divide the received number with the prime. If the number cannot be divided, send it along the ch
     * If there's no-one to send it to, I'm the last in the chain, the number is a prime and so I will create
     * a new actor responsible for filtering by this newly found prime number.
     */
    def onMessage(int value) {
        if (value % myPrime != 0) {
            if (follower) follower value
            else {
                println "Found $value"
                follower = new FilterActor(value).start()
            }
        }
    }

/**
     * Stop the actor on poisson reception
     */
    def onMessage(def poisson) {
        if (follower) {
            def sender = sender
            follower.sendAndContinue(poisson, {this.stop(); sender?.send('Done')})  //Pass the poisson along a
        } else {  //I am the last in the chain
            stop()
            reply 'Done'
        }
    }
}
```

Sleeping Barber

Problem description

```
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.actor.DefaultActor
import groovyx.gpars.group.DefaultPGroup
import groovyx.gpars.actor.Actor

final def group = new DefaultPGroup()

final def barber = group.actor {
    final def random = new Random()
    loop {
        react {message ->
            switch (message) {
                case Enter:
                    message.customer.send new Start()
                    println "Barber: Processing customer ${message.customer.name}"
                    doTheWork(random)
                    message.customer.send new Done()
                    reply new Next()
                    break
                case Wait:
                    println "Barber: No customers. Going to have a sleep"
                    break
            }
        }
    }
}
```

```
private def doTheWork(Random random) {
    Thread.sleep(random.nextInt(10) * 1000)
}

final Actor waitingRoom

waitingRoom = group.actor {
    final int capacity = 5
    final List<Customer> waitingCustomers = []
    boolean barberAsleep = true

loop {
        react {message ->
            switch (message) {
                case Enter:
                    if (waitingCustomers.size() == capacity) {
                        reply new Full()
                    } else {
                        waitingCustomers << message.customer
                        if (barberAsleep) {
                            assert waitingCustomers.size() == 1
                            barberAsleep = false
                            waitingRoom.send new Next()
                        }
                        else reply new Wait()
                    }
                    break
                case Next:
                    if (waitingCustomers.size()>0) {
                        def customer = waitingCustomers.remove(0)
                        barber.send new Enter(customer:customer)
                    } else {
                        barber.send new Wait()
                        barberAsleep = true
                    }
            }
        }
    }

}

class Customer extends DefaultActor {
    String name
    Actor localBarbers

void act() {
        localBarbers << new Enter(customer:this)
        loop {
            react {message ->
                switch (message) {
                    case Full:
                        println "Customer: $name: The waiting room is full. I am leaving."
                        stop()
                        break
                    case Wait:
                        println "Customer: $name: I will wait."
                        break
                    case Start:
                        println "Customer: $name: I am now being served."
                        break
                    case Done:
                        println "Customer: $name: I have been served."
                        stop();
                        break

}
            }
        }
    }
}

class Enter { Customer customer }
class Full {}
class Wait {}
class Next {}
class Start {}
class Done {}

def customers = []
customers << new Customer(name:'Joe', localBarbers:waitingRoom).start()
customers << new Customer(name:'Dave', localBarbers:waitingRoom).start()
customers << new Customer(name:'Alice', localBarbers:waitingRoom).start()

sleep 15000
customers << new Customer(name: 'James', localBarbers: waitingRoom).start()
sleep 5000
customers*.join()
barber.stop()
waitingRoom.stop()
```

Dining Philosophers

Problem description

```
import groovyx.gpars.actor.DefaultActor
import groovyx.gpars.actor.Actors

Actors.defaultActorPGroup.resize 5

final class Philosopher extends DefaultActor {
    private Random random = new Random()

String name
    def forks = []

void act() {
        assert 2 == forks.size()
        loop {
            think()
            forks*.send new Take()
            def messages = []
            react {a ->
                messages << [a, sender]
                react {b ->
                    messages << [b, sender]
                    if ([a, b].any {Rejected.isCase it}) {
                        println "$name: tOops, can't get my forks! Giving up."
                        final def accepted = messages.find {Accepted.isCase it[0]}
                        if (accepted!=null) accepted[1].send new Finished()
                    } else {
                        eat()
                        reply new Finished()
                    }
                }
            }
        }
    }

void think() {
        println "$name: tI'm thinking"
        Thread.sleep random.nextInt(5000)
        println "$name: tI'm done thinking"
    }

void eat() {
        println "$name: tI'm EATING"
        Thread.sleep random.nextInt(2000)
        println "$name: tI'm done EATING"
    }
}

final class Fork extends DefaultActor {

String name
    boolean available = true

void act() {
        loop {
            react {message ->
                switch (message) {
                    case Take:
                        if (available) {
                            available = false
                            reply new Accepted()
                        } else reply new Rejected()
                        break
                    case Finished:
                        assert !available
                        available = true
                        break
                    default: throw new IllegalStateException("Cannot process the message: $message")
                }
            }
        }
    }
}

final class Take {}
final class Accepted {}
final class Rejected {}
final class Finished {}

def forks = [
        new Fork(name:'Fork 1'),
        new Fork(name:'Fork 2'),
        new Fork(name:'Fork 3'),
        new Fork(name:'Fork 4'),
        new Fork(name:'Fork 5')
]

def philosophers = [
        new Philosopher(name:'Joe', forks:[forks[0], forks[1]]),
        new Philosopher(name:'Dave', forks:[forks[1], forks[2]]),
        new Philosopher(name:'Alice', forks:[forks[2], forks[3]]),
        new Philosopher(name:'James', forks:[forks[3], forks[4]]),
        new Philosopher(name:'Phil', forks:[forks[4], forks[0]]),
]

forks*.start()
philosophers*.start()

sleep 10000
forks*.stop()
philosophers*.stop()
```

Word sort

Given a folder name, the script will sort words in all files in the folder. The *SortMaster* actor creat
WordSortActors , splits among them the files to sort words in and collects the results.

Inspired by Scala Concurrency blog post by Michael Galpin

```
//Messages
private final class FileToSort { String fileName }
private final class SortResult { String fileName; List<String> words }

//Worker actor
class WordSortActor extends DefaultActor {

private List<String> sortedWords(String fileName) {
        parseFile(fileName).sort {it.toLowerCase()}
    }

private List<String> parseFile(String fileName) {
        List<String> words = []
        new File(fileName).splitEachLine(' ') {words.addAll(it)}
        return words
    }

void act() {
        loop {
            react {message ->
                switch (message) {
                    case FileToSort:
                        println "Sorting file=${message.fileName} on thread ${Thread.currentThread().name}"
                        reply new SortResult(fileName: message.fileName, words: sortedWords(message.fileName))
                }
            }
        }
    }
}

//Master actor
final class SortMaster extends DefaultActor {

String docRoot = '/'
    int numActors = 1

List<List<String>> sorted = []
    private CountDownLatch startupLatch = new CountDownLatch(1)
    private CountDownLatch doneLatch

private void beginSorting() {
        int cnt = sendTasksToWorkers()
        doneLatch = new CountDownLatch(cnt)
    }

private List createWorkers() {
        return (1..numActors).collect {new WordSortActor().start()}
    }

private int sendTasksToWorkers() {
        List<Actor> workers = createWorkers()
        int cnt = 0
        new File(docRoot).eachFile {
            workers[cnt % numActors] << new FileToSort(fileName: it)
            cnt += 1
        }
        return cnt
    }

public void waitUntilDone() {
        startupLatch.await()
        doneLatch.await()
    }

void act() {
        beginSorting()
        startupLatch.countDown()
        loop {
            react {
                switch (it) {
                    case SortResult:
                        sorted << it.words
                        doneLatch.countDown()
                        println "Received results for file=${it.fileName}"
                }
            }
        }
    }
}

//start the actors to sort words
def master = new SortMaster(docRoot: 'c:/tmp/Logs/', numActors: 5).start()
master.waitUntilDone()
println 'Done'

File file = new File("c:/tmp/Logs/sorted_words.txt")
file.withPrintWriter { printer ->
    master.sorted.each { printer.println it }
}
```

Load Balancer

Demonstrates work balancing among adaptable set of workers. The load balancer receives tasks temporary task queue. When a worker finishes his assignment, it asks the load balancer for a ne

If the load balancer doesn't have any tasks available in the task queue, the worker is stopped. If t
the task queue exceeds certain limit, a new worker is created to increase size of the worker pool.

```
import groovyx.gpars.actor.Actor
import groovyx.gpars.actor.DefaultActor

/**
 * Demonstrates work balancing among adaptable set of workers.
 * The load balancer receives tasks and queues them in a temporary task queue.
 * When a worker finishes his assignment, it asks the load balancer for a new task.
 * If the load balancer doesn't have any tasks available in the task queue, the worker is stopped.
 * If the number of tasks in the task queue exceeds certain limit, a new worker is created
 * to increase size of the worker pool.
 */

final class LoadBalancer extends DefaultActor {
    int workers = 0
    List taskQueue = []
    private static final QUEUE_SIZE_TRIGGER = 10

void act() {
        loop {
            react { message ->
                switch (message) {
                    case NeedMoreWork:
                        if (taskQueue.size() == 0) {
                            println 'No more tasks in the task queue. Terminating the worker.'
                            reply DemoWorker.EXIT
                            workers -= 1
                        } else reply taskQueue.remove(0)
                        break
                    case WorkToDo:
                        taskQueue << message
                        if ((workers == 0) || (taskQueue.size() >= QUEUE_SIZE_TRIGGER)) {
                            println 'Need more workers. Starting one.'
                            workers += 1
                            new DemoWorker(this).start()
                        }
                }
                println "Active workers=${workers}tTasks in queue=${taskQueue.size()}"
            }
        }
    }
}

final class DemoWorker extends DefaultActor {
    final static Object EXIT = new Object()
    private static final Random random = new Random()

Actor balancer

def DemoWorker(balancer) {
        this.balancer = balancer
    }

void act() {
        loop {
            this.balancer << new NeedMoreWork()
            react {
                switch (it) {
                    case WorkToDo:
                        processMessage(it)
                        break
                    case EXIT: terminate()
                }
            }
        }

}

private void processMessage(message) {
        synchronized (random) {
            Thread.sleep random.nextInt(5000)
        }
    }
}
final class WorkToDo {}
final class NeedMoreWork {}

final Actor balancer = new LoadBalancer().start()

//produce tasks
for (i in 1..20) {
    Thread.sleep 100
    balancer << new WorkToDo()
}

//produce tasks in a parallel thread
Thread.start {
    for (i in 1..10) {
        Thread.sleep 1000
        balancer << new WorkToDo()
    }
}

Thread.sleep 35000  //let the queues get empty
balancer << new WorkToDo()
balancer << new WorkToDo()
Thread.sleep 10000

balancer.stop()
balancer.join()
```

6 Agents

The Agent class, which is a thread-safe non-blocking shared mutable state wrapper implementat
in Clojure.

> A lot of the concurrency problems disappear when you eliminate the need for Shared State with your architecture. Indeed, concepts like actors, CSP or dataflow concurren or isolate mutable state completely. In some cases, however, sharing mutable data is inevitable or makes the design more natural and understandable. Think, for example, shopping cart in a typical e-commerce application, when multiple AJAX requests may cart with read or write requests concurrently.

Introduction

In the Clojure programing language you can find a concept of Agents, the purpose of which is to that need to be shared across threads. Agents hide the data and protect it from direct access. Cli commands (functions) to the agent. The commands will be serialized and processed against the turn. With the commands being executed serially the commands do not need to care about concu assume the data is all theirs when run. Although implemented differently, GPars Agents, called A behave like actors. They accept messages and process them asynchronously. The messages, h commands (functions or Groovy closures) and will be executed inside the agent. After reception run against the internal state of the Agent and the return value of the function is considered to be of the Agent.

Essentially, agents safe-guard mutable values by allowing only a single **agent-managed thread** to them. The mutable values are **not directly accessible** from outside, but instead **requests ha agent** and the agent guarantees to process the requests sequentially on behalf of the callers. Ag sequential execution of all requests and so consistency of the values.

Schematically:

```
agent = new Agent(0)  //created a new Agent wrapping an integer with initial value 0
agent.send {increment()}  //asynchronous send operation, sending the increment() function
…
//after some delay to process the message the internal Agent's state has been updated
…
assert agent.val== 1
```

To wrap integers, we can certainly use AtomicXXX types on the Java platform, but when the stat object we need more support.

Concepts

GPars provides an Agent class, which is a special-purpose thread-safe non-blocking implementa Agents in Clojure.

An Agent wraps a reference to mutable state, held inside a single field, and accepts code (closur messages, which can be sent to the Agent just like to any other actor using the '<<' operator, the implicit *call()* method. At some point after reception of a closure / command, the closure is invoke mutable field and can make changes to it. The closure is guaranteed to be run without interventic and so may freely alter the internal state of the Agent held in the internal <i>data</i> field.

The whole update process is of the fire-and-forget type, since once the message (closure) is sen caller thread can go off to do other things and come back later to check the current value with Ag Agent.valAsync(closure).

Basic rules

- When executed, the submitted commands obtain the agent's state as a parameter.
- The submitted commands /closures can call any methods on the agent's state.
- Replacing the state object with a new one is also possible and is done using the **updateValu**
- The **return value** of the submitted closure doesn't have a special meaning and is ignored.
- If the message sent to an *Agent* is **not a closure**, it is considered to be a **new value** for the
- The *val* property of an *Agent* will wait until all preceding commands in the agent's queue are safely return the value of the Agent.
- The *valAsync()* method will do the same **without blocking** the caller.
- The *instantVal* property will return an immediate snapshot of the internal agent's state.
- All Agent instances share a default daemon thread pool. Setting the *threadPool* property of a allow it to use a different thread pool.
- Exceptions thrown by the commands can be collected using the *errors* property.

Examples

Shared list of members

The Agent wraps a list of members, who have been added to the jug. To add a new member a m add a member) has to be sent to the *jugMembers* Agent.

```
import groovyx.gpars.agent.Agent
import java.util.concurrent.ExecutorService
import java.util.concurrent.Executors

/**
 * Create a new Agent wrapping a list of strings
 */
def jugMembers = new Agent<List<String>>(['Me'])  //add Me

jugMembers.send {it.add 'James'}  //add James

final Thread t1 = Thread.start {
    jugMembers.send {it.add 'Joe'}  //add Joe
}

final Thread t2 = Thread.start {
    jugMembers << {it.add 'Dave'}  //add Dave
    jugMembers {it.add 'Alice'}    //add Alice (using the implicit call() method)
}

[t1, t2]*.join()
println jugMembers.val
jugMembers.valAsync {println "Current members: $it"}

jugMembers.await()
```

Shared conference counting number of registrations

The Conference class allows registration and un-registration, however these methods can only b commands sent to the *conference* Agent.

```
import groovyx.gpars.agent.Agent

/**
 * Conference stores number of registrations and allows parties to register and unregister.
 * It inherits from the Agent class and adds the register() and unregister() private methods,
 * which callers may use it the commands they submit to the Conference.
 */
class Conference extends Agent<Long> {
    def Conference() { super(0) }
    private def register(long num) { data += num }
    private def unregister(long num) { data -= num }
}

final Agent conference = new Conference()  //new Conference created

/**
 * Three external parties will try to register/unregister concurrently
 */

final Thread t1 = Thread.start {
    conference << {register(10L)}               //send a command to register 10 attendees
}

final Thread t2 = Thread.start {
    conference << {register(5L)}                //send a command to register 5 attendees
}

final Thread t3 = Thread.start {
    conference << {unregister(3L)}              //send a command to unregister 3 attendees
}

[t1, t2, t3]*.join()

assert 12L == conference.val
```

Factory methods

Agent instances can also be created using the *Agent.agent()* factory method.

```
def jugMembers = Agent.agent ['Me']  //add Me
```

Listeners and validators

Agents allow the user to add listeners and validators. While listeners will get notified each time th
changes, validators get a chance to reject a coming change by throwing an exception.

```
final Agent counter = new Agent()

counter.addListener {oldValue, newValue -> println "Changing value from $oldValue to $newValue"}
counter.addListener {agent, oldValue, newValue -> println "Agent $agent changing value from $oldValue to $new\

counter.addValidator {oldValue, newValue -> if (oldValue > newValue) throw new IllegalArgumentException('Thing
Groovy')}
counter.addValidator {agent, oldValue, newValue -> if (oldValue == newValue) throw new IllegalArgumentExceptic
same for $agent')}

counter 10
counter 11
counter {updateValue 12}
counter 10  //Will be rejected
counter {updateValue it - 1}  //Will be rejected
counter {updateValue it}  //Will be rejected
counter {updateValue 11}  //Will be rejected
counter 12  //Will be rejected
counter 20
counter.await()
```

Both listeners and validators are essentially closures taking two or three arguments. Exceptions t
validators will be logged inside the agent and can be tested using the *hasErrors()* method or retri
errors property.

```
assert counter.hasErrors()
assert counter.errors.size() == 5
```

Validator gotchas

With Groovy being not very strict on data types and immutability, agent users should be aware of the road. If the submitted code modifies the state directly, validators will not be able to un-do the validation rule violation. There are two possible solutions available:

1. Make sure you never change the supplied object representing current agent state
2. Use custom copy strategy on the agent to allow the agent to create copies of the internal sta

In both cases you need to call *updateValue()* to set and validate the new state properly.

The problem as well as both of the solutions are shown below:

```
//Create an agent storing names, rejecting 'Joe'
final Closure rejectJoeValidator = {oldValue, newValue -> if ('Joe' in newValue) throw new IllegalArgumentExce
allowed to enter our list.')}

Agent agent = new Agent([])
agent.addValidator rejectJoeValidator

agent {it << 'Dave'}                    //Accepted
agent {it << 'Joe'}                     //Erroneously accepted, since by-passes the validation mechanism
println agent.val

//Solution 1 - never alter the supplied state object
agent = new Agent([])
agent.addValidator rejectJoeValidator

agent {updateValue(['Dave', * it])}      //Accepted
agent {updateValue(['Joe', * it])}       //Rejected
println agent.val

//Solution 2 - use custom copy strategy on the agent
agent = new Agent([], {it.clone()})
agent.addValidator rejectJoeValidator

agent {updateValue it << 'Dave'}        //Accepted
agent {updateValue it << 'Joe'}         //Rejected, since 'it' is now just a copy of the internal agent's stat
println agent.val
```

Grouping

By default all Agent instances belong to the same group sharing its daemon thread pool.

Custom groups can also create instances of Agent. These instances will belong to the group, wh will share a thread pool. To create an Agent instance belonging to a group, call the *agent()* factor group. This way you can organize and tune performance of agents.

```
final def group = new NonDaemonPGroup(5)  //create a group around a thread pool
def jugMembers = group.agent(['Me'])  //add Me
```

> The default thread pool for agents contains daemon threads. Make sure that your cus thread pools either use daemon threads, too, which can be achieved either by using DefaultPGroup or by providing your own thread factory to a thread pool constructor, c your thread pools use non-daemon threads, such as when using the NonDaemonPG group class, make sure you shutdown the group or the thread pool explicitly by callin shutdown() method, otherwise your applications will not exit.

Direct pool replacement

Alternatively, by calling the *attachToThreadPool()* method on an Agent instance a custom thread
for it.

```
def jugMembers = new Agent<List<String>>(['Me'])  //add Me

final ExecutorService pool = Executors.newFixedThreadPool(10)
jugMembers.attachToThreadPool(new DefaultPool(pool))
```

> ⚠ Remember, like actors, a single Agent instance (aka agent) can never use more than
> thread at a time

The shopping cart example

```
import groovyx.gpars.agent.Agent

class ShoppingCart {
    private def cartState = new Agent([:])
//----------------- public methods below here ----------------------------------
    public void addItem(String product, int quantity) {
        cartState << {it[product] = quantity}  //the << operator sends
                                               //a message to the Agent
    }    public void removeItem(String product) {
        cartState << {it.remove(product)}
    }    public Object listContent() {
        return cartState.val
    }    public void clearItems() {
        cartState << performClear
    }

public void increaseQuantity(String product, int quantityChange) {
        cartState << this.&changeQuantity.curry(product, quantityChange)
    }
//----------------- private methods below here --------------------------------
    private void changeQuantity(String product, int quantityChange, Map items) {
        items[product] = (items[product] ?: 0) + quantityChange
    }    private Closure performClear = { it.clear() }
}
//----------------- script code below here ------------------------------------
final ShoppingCart cart = new ShoppingCart()
cart.addItem 'Pilsner', 10
cart.addItem 'Budweisser', 5
cart.addItem 'Staropramen', 20

cart.removeItem 'Budweisser'
cart.addItem 'Budweisser', 15

println "Contents ${cart.listContent()}"

cart.increaseQuantity 'Budweisser', 3
println "Contents ${cart.listContent()}"

cart.clearItems()
println "Contents ${cart.listContent()}"
```

You might have noticed two implementation strategies in the code.

1. Public methods may internally just send the required code off to the Agent, instead of execu
   functionality directly

And so sequential code like

```
public void addItem(String product, int quantity) {
    cartState[product]=quantity
}
```

becomes

```
public void addItem(String product, int quantity) {
    cartState << {it[product] = quantity}
}
```

2. Public methods may send references to internal private methods or closures, which hold the d
perform

```
public void clearItems() {
    cartState << performClear
}

private Closure performClear = { it.clear() }
```

Currying might be necessary, if the closure takes other arguments besides the current internal
the *increaseQuantity* method.

The printer service example

Another example - a not thread-safe printer service shared by multiple threads. The printer need:
document and quality properties set before printing, so obviously a potential for race conditions if
Callers don't want to block until the printer is available, which the fire-and-forget nature of actors

```
import groovyx.gpars.agent.Agent

/**
 * A non-thread-safe service that slowly prints documents on at a time
 */
class PrinterService {
    String document
    String quality

public void printDocument() {
        println "Printing $document in $quality quality"
        Thread.sleep 5000
        println "Done printing $document"
    }
}

def printer = new Agent<PrinterService>(new PrinterService())

final Thread thread1 = Thread.start {
    for (num in (1..3)) {
        final String text = "document $num"
        printer << {printerService ->
            printerService.document = text
            printerService.quality = 'High'
            printerService.printDocument()
        }
        Thread.sleep 200
    }
    println 'Thread 1 is ready to do something else. All print tasks have been submitted'
}

final Thread thread2 = Thread.start {
    for (num in (1..4)) {
        final String text = "picture $num"
        printer << {printerService ->
            printerService.document = text
            printerService.quality = 'Medium'
            printerService.printDocument()
        }
        Thread.sleep 500
    }
    println 'Thread 2 is ready to do something else. All print tasks have been submitted'
}

[thread1, thread2]*.join()
printer.await()
```

For latest update, see the respective Demos.

Reading the value

To follow the clojure philosophy closely the Agent class gives reads higher priority than to writes. *instantVal* property your read request will bypass the incoming message queue of the Agent and snapshot of the internal state. The *val* property will wait in the message queue for processing, jus variant *valAsync(Clojure cl)* , which will invoke the provided closure with the internal state as a pa

You have to bear in mind that the *instantVal* property might return although correct, but randomly the internal state of the Agent at the time of *instantVal* execution is non-deterministic and depend that have been processed before the thread scheduler executes the body of *instantVal* .

The *await()* method allows you to wait for processing all the messages submitted to the Agent be calling thread.

State copy strategy

To avoid leaking the internal state the Agent class allows to specify a copy strategy as the secon argument. With the copy strategy specified, the internal state is processed by the copy strategy c value of the copy strategy value is returned to the caller instead of the actual internal state. This a *val* as well as to *valAsync()* .

Error handling

Exceptions thrown from within the submitted commands are stored inside the agent and can be c *errors* property. The property gets cleared once read.

```
def jugMembers = new Agent<List>()
    assert jugMembers.errors.empty

jugMembers.send {throw new IllegalStateException('test1')}
    jugMembers.send {throw new IllegalArgumentException('test2')}
    jugMembers.await()

List errors = jugMembers.errors
    assert 2 == errors.size()
    assert errors[0] instanceof IllegalStateException
    assert 'test1' == errors[0].message
    assert errors[1] instanceof IllegalArgumentException
    assert 'test2' == errors[1].message

assert jugMembers.errors.empty
```

Fair and Non-fair agents

Agents can be either fair or non-fair. Fair agents give up the thread after processing each messa keep a thread until their message queue is empty. As a result, non-fair agents tend to perform be The default setting for all Agent instances is to be **non-fair**, however by calling its *makeFair()* me be made fair.

```
def jugMembers = new Agent<List>(['Me'])  //add Me
    jugMembers.makeFair()
```

7 Dataflow

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

Introduction

Check out the small example written in Groovy using GPars, which sums results of calculations p concurrently run tasks:

```
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.

```
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 t data and they do so using **Dataflow Variables**. Think of Dataflow Variables as one-shot channel transferring data from producers to their consumers.

The Dataflow Variables have a pretty straightforward semantics. When a task needs to read a va *DataflowVariable* (through the val property), it will block until the value has been set by another ta the '<<' operator). Each *DataflowVariable* can be set **only once** in its lifetime. Notice that you do ordering and synchronizing the tasks or threads and their access to shared variables. The values transferred among tasks at the right time without your intervention. The data flow seamlessly am without your intervention or care.

Implementation detail: The three tasks in the example **do not necessarily need to be mappe threads**. Tasks represent so-called "green" or "logical" threads and can be mapped under the co physical threads. The actual mapping depends on the scheduler, but the outcome of dataflow alg depend on the actual scheduling.

> ⚠ The *bind* operation of dataflow variables silently accepts re-binding to a value, which
> an already bound value. Call *bindUnique* to reject equal values on already-bound vari

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 a defined. They run as soon as all of their inputs become valid, as opposed to when the program e Whereas a traditional program essentially consists of a series of statements saying "do this, now program is more like a series of workers on an assembly line, who will do their assigned task as s arrive. This is why dataflow languages are inherently parallel; the operations have no hidden stat the operations are all "ready" at the same time.

Principles

With Dataflow Concurrency you can safely share variables across tasks. These variables (in Gro *DataflowVariable* class) can only be assigned (using the '<<' operator) a value once in their lifetir variables, on the other hand, can be read multiple times (in Groovy through the val property), eve been assigned. In such cases the reading task is suspended until the value is set by another task write your code for each task sequentially using Dataflow Variables and the underlying mechanic 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 is not possible to change the value of a dataflow variable once it is bound.
- Dataflow variables makes it easy to create concurrent stream agents.

Dataflow Queues and Broadcasts

Before you go to check the samples of using **Dataflow Variables**, **Tasks** and **Operators**, you sh streams and queues to have a full picture of Dataflow Concurrency. Except for dataflow variables concepts of *DataflowQueues* and *DataflowBroadcast* that you can leverage in your code. You ma thread-safe buffers or queues for message transfer among concurrent tasks or threads. Check o 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 *Data* with common methods allowing users to write to them and read values from them. The ability to t identically through the *DataflowChannel* interface comes in handy once you start using them to w *selectors* together.

> The *DataflowChannel* interface combines two interfaces, each serving its purpose:
>
> - DataflowReadChannel holding all the methods necessary for reading values fron channel - getVal(), getValAsync(), whenBound(), etc.
> - DataflowWriteChannel holding all the methods necessary for writing values into a bind(), <<
>
> You may prefer using these dedicated interfaces instead of the general *DataflowChai* interface, to better express the 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 c or more producers send messages to one reader. If multiple readers read from the same *Dataflo* each consume different messages. Or to put it a different way, each message is consumed by ex can easily imagine a simple load-balancing scheme built around a shared *DataflowQueue* with re dynamically when the consumer part of your algorithm needs to scale up. This is also a useful de connecting tasks or operators.

Publish-subscribe communication

The *DataflowBroadcast* class offers a publish-subscribe (1 to many, many to many) communicati more producers write messages, while all registered readers will receive all the messages. Each consumed by all readers with a valid subscription at the moment when the message is being writ The readers subscribe by calling the *createReadChannel()* method.

```
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 hood *DataflowBroadcast* uses the *DataflowStream* class to implement the message de

DataflowStream

The *DataflowStream* class represents a deterministic dataflow channel. It is build around the con queue and so provides a lock-free thread-safe implementation for message passing. Essentially, *DataflowStream* as a 1 to many communication channel, since when a reader consumes a mess will still be able to read the message. Also, all messages arrive to all readers in the same order. S is implemented as a functional queue, its API requires that users traverse the values in the strea other hand *DataflowStream* offers handy methods for value filtering or transformation together wi performance characteristics.

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

```
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) -> (filte
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 at Dataflow(
 */
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* with other dataflow constructs, like e.g wrap it with *DataflowReadAdapter* for read access or *DataflowWriteAdapter* for write access. The is designed for single-threaded producers and consumers. If multiple threads are supposed to re the stream, their access to the stream must be serialized externally or the adapters should be us

DataflowStream Adapters

Since the *DataflowStream* API as well as the semantics of its use are very different from the one *Dataflow(Read/Write)Channel* , adapters have to be used in order to allow *DataflowStreams* to b dataflow elements. The *DataflowStreamReadAdapter* class will wrap a *DataflowStream* with nec read values, while the *DataflowStreamWriteAdapter* class will provide write methods around the *DataflowStream* .

> ⚠ It is important to mention that the *DataflowStreamWriteAdapter* is thread safe allowin threads to add values to the wrapped *DataflowStream* through the adapter. On the ot *DataflowStreamReadAdapter* is designed to be used by a single thread.
>
> To minimize the overhead and stay in-line with the *DataflowStream* semantics, the *DataflowStreamReadAdapter* class is not thread-safe and should only be used from v single thread. If multiple threads need to read from a DataflowStream, they should ea their own wrapping *DataflowStreamReadAdapter* .

Thanks to the adapters *DataflowStream* can be used for communication between operators or se *Dataflow(Read/Write)Channels* .

```
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 ad *DataflowStream* :

```
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, lik generation, filtering or mapping, you may consider using the *DataflowBroadcast* class which offers the *publish-subscribe* communication model through the *DataflowChann* interface.

Bind handlers

```
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) eithe and the *then()* or the *whenBound()* methods. They will be run once a value is bound to the variab

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

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

Obviously nothing prevents you from having more of such handlers for a single promise: They wi once the promise has a concrete value:

```
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}
```

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

Bind handlers grouping

When you need to wait for multiple DataflowVariables/Promises to be bound, you can benefit fro *whenAllBound()* function, which is available on the *Dataflow* class as well as on *PGroup* instance

```
final group = new NonDaemonPGroup()

//Calling asynchronous services and receiving back promises for the reservations
    Promise flightReservation = flightBookingService('PRG <-> BRU')
    Promise hotelReservation = hotelBookingService('BRU:Feb 24 2009 - Feb 29 2009')
    Promise taxiReservation = taxiBookingService('BRU:Feb 24 2009 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,
        "Agenda: $flight | $hotel | $taxi"
    }

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

If you cannot specify up-front the number of parameters the *whenAllBound()* handler takes, use a argument of type *List* :

```
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 handlers chaining

All dataflow channels also support the *then()* method to register a handler (a callback) that shoul value becomes available. Unlike *whenBound()* the *then()* method allows for chaining, giving you result values between functions asynchronously.

⚠ Notice that Groovy allows us to leave out some of the *dots* in the *then()* method chair

```
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
```

Chaining can save quite some code when calling other asynchronous services from w *whenBound()* handlers. Asynchronous services, such as *Asynchronous Functions* or *Methods* , return *Promises* for their results. To obtain the actual results your handlers either have to block to wait for the value to be bound, which would lock the current thr unproductive state,

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

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

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

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

```
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()* and so can be chained the sam

```
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 is important to be able to handle the effort. GPars promises can implicitly propagate exceptions from asynchronous calculations acros

1. Promises propagate result values as well as exceptions. The blocking *get()* method re-throw was bound to the Promise and so the caller can handle it.
2. For asynchronous notifications, the *whenBound()* handler closure gets the exception passed
3. The *then()* method accepts two arguments - a **value handler** and an optional **error handler** depending on whether the result is a regular value or an exception. If no errorHandler is spe re-thrown to the Promise returned by *then()* .
4. Exactly the same behavior as for *then()* holds true for the *whenAllBound()* method, which list Promises to get bound

```
Promise<Integer> initial = new DataflowVariable<Integer>()
    Promise<String> result = initial.then {it * 2} then {100 / it}                  //Will throw exception for
            .then {println "Logging the value $it as it passes by"; return it}      //Since no error handler i
will be ignored
                                                                                    //and silently re-thrown t
the chain
            .then({"The result for $num is $it"}, {"Error detected for $num: $it"}) //Here the exception is ca
    initial << 0
    println result.get()
```

ErrorHandler is a closure that accepts instances of *Throwable* as its only (optional) argument and should be bound to the result of the *then()* method call (the returned Promise). If an exception is error handler, it is bound as an error to the resulting Promise.

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

Just like with regular exception handling in Java with try-catch statements, this behavior of GPars asynchronous invocations the freedom to handle exceptions at the place where it is most conven ignore exceptions in your code and assume things just work, yet exceptions will not get accidenta

```
task {
    '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()
```

Handling concrete exception type

You may be also more specific about the handled exception type:

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

`

Customer-site exception handling

You may also leave the exception completely un-handled and let the clients (consumers) handle

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

`

Putting it together

By combining *whenAllBound()* and *then* (or >>) you can easily create large asynchronous scenar way:

```
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 f down the pipe will be invoked asynchronously by the promise even if the are synchro

Lazy dataflow variables

Sometimes you may like to combine the qualities of dataflow variables with their lazy initialization

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

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

Instances of *LazyDataflowVariable* have an initializer specified at construction time, which only g someone asks for its value, either through the blocking *get()* method or using any of the non-bloc methods, such as *then()* . Since *LazyDataflowVariables* preserve all the goodies of ordinary *Data* can again chain them easily with other *lazy* or *ordinary* dataflow variables.

Example

This deserves a more practical example. Taking inspiration from http://blog.jcoglan.com/2013/03/30/callbacks-are-imperative-promises-are-functional-nodes-bigg the following piece of code demonstrates use of *LazyDataflowVariables* to lazily and asynchrono dependent components into memory. The components (modules) will be loaded in the order of th concurrently, if possible. Each module will only be loaded once, irrespective of the number of mo it. Thanks to laziness only the modules that are transitively needed will be loaded. Our example u "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 loa once both B and C have been loaded.

```
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"
```

Dataflow Expressions

Look at the magic below:

```
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 calcula
an accelerating object. In the equation itself, however, we use the DataflowVariables directly. We
values they represent and yet we are able to do the math correctly. This shows that DataflowVari
flexible.

For example, you can call methods on them and these methods will get dispatched to the bound

```
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
instead:

```
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:

```
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), which y
value from both synchronously and asynchronously.

Bind error notification

DataflowVariables offer the ability to send notifications to the registered listeners whenever a bin
getBindErrorManager() method allows for listener to be added and removed. The listeners get nc
failed attempt to bind a value (through bind(), bindSafely(), bindUnique() or leftShift()) or an error

```
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 uniqueBi
                println "Bind failed!"
            }

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

})
```

This allows you to customize reactions to attempts to binding of already bound dataflow variables
bindSafely() you do not get bind exceptions fired to the caller, but instead a registered *BindErrorL*

Further reading

Scala Dataflow library by Jonas Bonér

JVM concurrency presentation slides by Jonas Bonér

Dataflow Concurrency library for Ruby

7.1 Tasks

The **Dataflow tasks** give you an easy-to-grasp abstraction of mutually-independent logical tasks
run concurrently and exchange data solely through Dataflow Variables, Queues, Broadcasts and
tasks with their easy-to-express mutual dependencies and inherently sequential body could also
implementation of UML *Activity Diagrams* .

Check out the examples.

A simple mashup example

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

```
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 to allow for performance fine-tuning. Groups provid factory method to create tasks attached to the groups. Using groups allows you to organize tasks different thread pools (wrapped inside the group). While the Dataflow.task() command schedules thread pool (java.util.concurrent.Executor, fixed size=#cpu+1, daemon threads), you may prefer your own thread pool(s) to run your tasks.

```
import groovyx.gpars.group.DefaultPGroup

def group = new DefaultPGroup()

group.with {
    task {
        …
    }
task {
        …
    }
}
```

> The default thread pool for dataflow tasks contains daemon threads, which means yo application will exit as soon as the main thread finishes and won't wait for all tasks to When grouping tasks, make sure that your custom thread pools either use daemon th too, which can be achieved by using DefaultPGroup or by providing your own thread a thread pool constructor, or in case your thread pools use non-daemon threads, suc using the NonDaemonPGroup group class, make sure you shutdown the group or the pool explicitly by calling its shutdown() method, otherwise your applications will not ex

You may selectively override the default group used for tasks, operators, callbacks and other dat a code block using the _Dataflow.usingGroup() method:

```
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"}).joi
}
```

You can always override the default group by being specific:

```
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.retrieveCurrent
    .then(anotherGroup, {mentionsGroovy -> println "Groovy found: $mentionsGroovy"}, {error -> println "Error:
}
```

A mashup variant with methods

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

```
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 the faster the program runs. Check out, for example, the following script, which calculates param physical experiment and prints out the results. Each task performs its part of the calculation and calculated by some other tasks as well as its result might be needed by some of the other tasks. Concurrency you can split the work between tasks or reorder the tasks themselves as you like ar mechanics will ensure the calculation will be accomplished correctly.

```
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 v
${velocity.val} m/s.

Given our ability to push the ball backwards with a force of ${decelerationForce.val} N (Newton), we can cause
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()
```

Note: I did my best to make all the physical calculations right. Feel free to change the values and
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 y
randomness allowed. That's one of the benefits of Dataflow concurrency. Irrespective of the actu
scheme, if you don't get a deadlock in tests, you won't get them in production.

```
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 have to wr
Dataflow variables.

```
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 appro
The semantics of reading a value (e.g. df.x) and binding a value (e.g. df.x = 'value') remain identi
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 Dataflo
accessed directly without the need to prefix them with the Dataflows instance identifier.

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

Dataflow.task {y = 'value2}

assert y == 'value2'
}
```

Returning a value from a task

Typically dataflow tasks communicate through dataflow variables. On top of that, tasks can also r
through a dataflow variable. When you invoke the *task()* factory method, you get back an instanc
(implemented as DataflowVariable), through which you can listen for the task's return value, just
other Promise or DataflowVariable.

```
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
```

Obviously the value can also be obtained without blocking the caller using the *whenBound()* metl

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

h2. Joining tasks

Using the *join()* operation on the result dataflow variable of a task you can block until the task fini

```
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()
```

7.2 Selects

Frequently a value needs to be obtained from one of several dataflow channels (variables, queue streams). The *Select* class is suitable for such scenarios. *Select* can scan multiple dataflow chan channel from all the input channels, which currently have a value available for read. The value fr read and returned to the caller together with the index of the originating channel. Picking the cha or based on channel priority, in which case channels with lower position index in the *Select* cons 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 th
 */
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}"
```

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

There are multiple ways to read values from a Select:

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

By default the *Select* blocks the caller until a value to read is available. The alternative *selectToF* *prioritySelectToPromise()* methods give you a way to obtain a promise for the value that will be s the future. Through the returned Promise you may register a callback to get invoked asynchronou value is selected.

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

Alternatively, *Select* allows to have the value sent to a provided *MessageStream* (e.g. an actor) v caller.

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

Guards

Guards allow the caller to omit some input channels from the selection. Guards are specified as a passed to the *select()* or *prioritySelect()* methods.

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

A typical use for guards is to make Selects flexible to adopt to the changes in the user state.

```
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 boole
 */
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 m instead.

```
/**
 * Shows a basic use of Priority Select, which monitors a set of input channels for values and makes these val
 * 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 inde
 * 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 goes 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

Asynchronous activities, no matter whether they are **dataflow tasks** , **active objects' methods** **functions** , return *Promises* . *Promises* implement the *SelectableChannel* interface and so can b selection together with other *Promises* as well as *read channels* . Similarly to Java's *CompletionS* enables you to obtain results of asynchronous activities as soon as each of them becomes availa employ *Select* to give you the first/fastest result of several computations running in parallel.

```
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 calculati
 */

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 that gets bound to a value afte
This can be leveraged in *Selects* so that they unblock after a desired delay, if none of the other c
value before that moment. Just pass the **timeout channel** as another input channel to the *Selec*

```
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 calculati
 */

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
}
```

Cancellation

In case you need to cancel the other tasks once a value has been calculated or a timeout expirec
set a flag that the tasks periodically monitor. There's intentionally no cancellation machinery built
or *Tasks* .

```
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 calculati
 * It shows a waz 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
}
```

7.3 Operators

Dataflow Operators and Selectors provide a full Dataflow implementation with all the usual cerem

Concepts

Full dataflow concurrency builds on the concept of channels connecting operators and selectors, values coming through input channels, transform them into new values and output the new value channels. While *Operators* wait for **all** input channels to have a value available for read before th *Selectors* are triggered by a value available on **any** of the input channels.

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

```
/**
 * CACHE
 *
 * Caches sites' contents. Accepts requests for url content, outputs the content. Outputs requests for downloa
 * 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)
        }
    }
}
```

The standard error handling will print out an error message to the standard error outp terminate the operator in case an uncaught exception is thrown from withing the oper body. To alter the behavior, you can register your own event listener:

```
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 ->
    …
}
See the  Operator lifecycle  section for more details.
```

Types of operators

There are specialized versions of operators serving specific purposes:

- operator - the basic general-purpose operator
- selector - operator that is triggered by a value being available in any of its input channels
- prioritySelector - a selector that prefers delivering messages from lower-indexed input chanr 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, when some operators consume output by other

```
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:

```
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
```

Grouping operators

Dataflow operators can be organized into groups to allow for performance fine-tuning. Groups pr *operator()* factory method to create tasks attached to the 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
    }
}
```

The default thread pool for dataflow operators contains daemon threads, which mean application will exit as soon as the main thread finishes and won't wait for all tasks to When grouping operators, make sure that your custom thread pools either use daem threads, too, which can be achieved by using DefaultPGroup or by providing your ow factory to a thread pool constructor, or in case your thread pools use non-daemon thr such as when using the NonDaemonPGroup group class, make sure you shutdown t or the thread pool explicitly by calling its shutdown() method, otherwise your applicati not exit.

You may selectively override the default group used for tasks, operators, callbacks and other dat a code block using the _Dataflow.usingGroup() method:

```
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:

```
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* ca the operator has been build. You may find the *groovyx.gpars.dataflow.ProcessingNode* class hel collecting channels and values into lists before you finally build an operator.

```
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
}
```

State in operators

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

```
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 definitic
object into the operator as a *stateObject* parameter at construction time:

```
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 settin
operator's body to be written in a non-thread-safe manner, once an operator becomes "hot" and
accumulate in the operator's input queues, you might consider allowing multiple threads to run th
concurrently. Bear in mind that in such a case you need to avoid or protect shared resources fror
access. To enable multiple threads to run the operator's body concurrently, pass an extra *maxFc*
creating an operator:

```
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 co
positive numbers are allowed with value 1 being the default.

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

```
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 the output

When enabling internal parallelization of an operator by setting the value for *maxForks* to a value important to remember that without explicit or implicit synchronization in the operators' body race Especially bear in mind that values written to multiple output channels are not guarantied to be w same order to all the channels

```
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 -> 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 operato state:

```
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 purp *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 call *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 same position index.

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

> Using the *bindAllOutputs* or the *bindAllOutputValues* methods will not guarantee aton writes across al the output channels when using internal parallelism. If preserving the 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 intere notifications and potential alter operator's behavior. The *DataflowEventListener* interface offers a methods:

```
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 op
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

/**
     * 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,

/**
     * 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

/**
     * 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 t exceptions, notify a supervising entity, generate an alternative output or perform any steps requir situation. If there's no listener registered or if any of the listeners returns *true* the operator will ter contract of *afterStop()* . Exceptions that occur outside the actual operator's body, i.e. at the parar phase before the body is triggered or at the clean-up and channel subscription phase, after the b lead to operator termination.

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

```
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
    else bindOutput sum
}
```

Selectors

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

```
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 i processed. This allows the selector to distinguish between values coming through different input

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

Priority Selector

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

```
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 channels with higher position index.

Join selector

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

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

Internal parallelism

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

```
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 cl
The *guards* input property can be used to set the initial mask on all input channels and the *setGu*
methods are then available in the selector's body.

```
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 boole
 */
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 guarc
    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
}
```

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

7.4 Shutting Down Dataflow Networks

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

Dataflow operators and selectors can be terminated in three ways:

1. by calling the terminate() method on all operators that need to be terminated
2. by sending a poisson message
3. by setting up a network of activity monitors that will shutdown the network after all messages

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 forget to shutdown the pool once the network is terminated. Otherwise the thread poo consume system resources and, in case of using non-daemon threads, it will prevent exit.

Emergency shutdown

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

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

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

```
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 also obvisouly shutdown the network, however, no lifecycle listeners will be invoked in such cases.

Stopping operators gently

Operators handle incoming messages repeatedly. The only safe moment for stopping an operato loosing any messages is right after the operator has finished processing messages and is just ab messages in its incoming pipes. This is exactly what the *terminateAfterNextRun()* method does. I 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 late
different operator/selector or in some other way. Using *terminateAfterNextRun()* you will not loos
This may be particularly handy when you use a group of operators/selectors to load-balance mes
channel. Once the work-load decreases, the terminateAfterNextRun() method may be used to sa
load-balancing operators.

> **Detecting shutdown**
>
> Operators and electors offer a handy *join()* method for those who need to block until t
> operator terminates.
>
> ```
> allMyProcessors*.join()
> ```
>
> This is the easies way to wait until the whole dataflow network shuts down, irrespecti
> shutdown method used.

PoisonPill

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

```
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
poisson is sent to all its output channels, so that the poisson can spread to the connected operat
operators typically wait for all inputs to have a value, in case of *PoisonPills* , the operator will tern
soon as a *PoisonPill* appears on any of its inputs. The values already obtained from the other cha
can be considered an error in the design of the network, if these messages were supposed to be
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 ch
it on the the output channels. This behavior prevents networks containing **feed-back loops invo**
being shutdown using *PoisonPill* . A selector would never receive a *PoisonPill* from the channel t
behind the selector. A different shutdown strategy should be used for such networks.

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

Immediate poison pill

Especially for selectors to shutdown immediately after receiving a poison pill, a notion of **immedi** been introduced. Since normal, non-immediate poison pills merely close the input channel leavin until at least one input channel remains open, the immediate poison pill closes the selector instar unprocessed messages from the other selector's input channels will not be handled by the select immediate poison pill.

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

```
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 specified number of them have been stopped. The *CountingPoisonPill* class serves exactly this p

```
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 l properties.

```
//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 - *ImmediateCountingPoi*

```
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 ev selectors involved in feedback loops, which normal non-immediate poison pill would r to.

Poison strategies

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

> Load-balancing architectures, which use multiple operators reading messages off a s channel (queue), will also prevent poison shutdown to work properly, since only one c reading operators will get to read the poison message. You may consider using **forke operators** instead, by setting the *maxForks* property to a value greater than 1. Anoth alternative is to manually split the message stream into multiple channels, each of wh 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 t 'terminator' operator below leverages the fact that *DataflowVariables* are implementations of the interface and thus can be consumed by operators. As soon as both tasks finish, the operator will down the *q* channel to stop the consumer as soon as it processes all data.

```
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 on the network boundaries. This can be easily achieved by putting a single-input single-output filt such channel.

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

The *Pipeline* DSL may be also helpful here:

```
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 approach will keep the network running until all the messages get handled and than gracefully sh letting you know when this happens. You have to pay a modest performance penalty, though. Th we need to keep track of what's happening inside the network.

```
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,
    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 GracefulShutdo
{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 pro instances of *GracefulShutdownListener* attached to all operators/selectors. These listeners obse processors together with their input channels and report to the shared *GracefulShutdownMonitor* *shutdownNetwork()* is called on *GracefulShutdownMonitor* , it will periodically check for reported 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 shutdo been initiated, since this may cause the network to never terminate. The shutdown pr should only be started after all data producers have ceased sending additional messa monitored network.

The *shutdownNetwork()* method returns a *Promise* so that you can do the usual set of tricks with the network to terminate using the *get()* method, register a callback using the *whenBound()* meth whole set of activities through the *then()* method.

Limitations of graceful shutdown

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

7.5 Application Frameworks

Dataflow Operators and Selectors can be successfully used to build high-level domain-specific fr 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 listeners can be very useful at language level to combine, for example, with actors or parallel coll need comes for asynchronous handling of events that come through one of more channels, a da small dataflow network could be a very good fit. Unlike tasks, operators are lightweight and relea there's no message to process. Unlike actors, operators are addressed indirectly through channe combine messages from multiple channels into one action.

Alternatively, operators can be looked at as continuous functions, which instantly and repeatedly values into output. We believe that a concurrency-friendly general-purpose programming languag type of abstraction.

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

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

The higher-level application-specific frameworks should put effort into providing abstractions bes domain and hide GPars complexities. For example, the visual graph of the network that the user screen should typically not show all the channels that participate in the network. Debugging or lo rarely contribute to the core of the solution, are among the first good candidates to consider for e channels and lifecycle-event listeners, which orchestrate aspects such as load balancing or grac probably be not exposed to the user, although they will be part of the generated and executed ne single channel in the domain-specific model will in reality translate into multiple channels perhaps logging/transforming/filtering operators connecting them together. The function associated with a 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 complete application-specific framework. This keeps GPars domain-agnostic and universal, yet useful at th level.

7.6 Pipeline DSL

A DSL for building operators pipelines

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

```
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 tha pipeline. The *pipe* operator lets you hook an output of one function/operator/process to the input like chaining system processes on the command line.

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

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

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

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 networks. Only your imagination can limit your ability to mix pipelines with channels and operator definitions.

```
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 {'###
'###'}

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.toLo
      .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
```

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

```
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:

```
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 si case).

Forking the data flow

When a need comes to copy the output of a pipeline/channel into more than one following pipelir method will help you:

```
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 the pipeline

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

```
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 fu second argument needs to accept as many arguments as there are channels being merged - eac the corresponding channel.

```
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, eacl output into an output channel with the corresponding position index.

```
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) c indicated by the return value from a closure.

```
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.

```
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
operator that no value should be passed further down the pipeline, a *NullObject.nullObject* instan

```
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 the thread pools

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

```
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 ove default Dataflow PGroup.

```
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 value specified.

The pipeline builder

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

```
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 oper listeners to attach or the value for *maxForks* . Just like when building operators directly, the Pipel accept an optional map of parameters to pass in.

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

7.7 Implementation

The Dataflow Concurrency in GPars builds on the same principles as the actor support. All of the
a thread pool and so the number threads created through *Dataflow.task()* factory method don't n
the number of physical threads required from the system. The *PGroup.task()* factory method can
created task to a group. Since each group defines its own thread pool, you can easily organize ta
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
problem at hands. You can freely you use Dataflow Variables from actors.

```
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 com
doubler actor.

Using plain java threads

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

```
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 flo
neither the actor lifecycle methods, nor the send/react functionality or thread pooling take effect i

7.8 Synchronous Variables and Channels

When using asynchronous dataflow channels, apart from the fact that readers have to wait for a v
for consumption, the communicating parties remain completely independent. Writers don't wait fo
get consumed. Readers obtain values immediately as they come and ask. Synchronous channel
can synchronize writers with the readers as well as multiple readers among themselves. This is p
you need to increase the level of determinism. The writer-to-reader partial ordering imposed by a
communication is complemented with reader-to-writer partial ordering, when using synchronous c
other words, you are guaranteed that whatever the reader did before reading a value from a sync
preceded whatever the writer did after writing the value. Also, with synchronous communication w
too far ahead of readers, which simplifies reasoning about the system and reduces the need to m
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. Eac
the queue will be consumed by exactly one reader. Writers are blocked until their message is con
blocked until there's a value available for them to read.

```
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 con
 * Unlike when using asynchronous channels, synchronous channels block both the writer and the readers until a
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) communica
written to the broadcast will be consumed by all subscribed readers. Writers are blocked until the
consumed by all readers, readers are blocked until there's a value available for them to read and
subscribed readers ask for the message as well. With *SyncDataflowBroadcast* you get all reader
message at the same time and waiting for one-another before getting the next one.

```
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 c
 * Unlike when using asynchronous channels, synchronous channels block both the writer and the readers until a
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 bou
SyncDataflowVariable class provides a one-shot data exchange mechanism that blocks the write
a specified number of waiting parties is reached.

```
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()
```

7.9 Kanban Flow

APIs: KanbanFlow | KanbanLink | KanbanTray | ProcessingNode

KanbanFlow

A *KanbanFlow* is a composed object that uses dataflow abstractions to define dependencies bet
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 Kanba
described in [The KanbanFlow Pattern] (recommended read). They use objects of type *KanbanTr*
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 numbere
0 has been used to take a product from producer to consumer, has been emptied by the consum
back to the producer's input queue. Trays 1 and 2 wait carry products waiting for consumption, tr
used by producers.

A *KanbanFlow* object links producers to consumers thus creating *KanbanLink* objects. In the cou
second link may be constructed where the producer is the same object that acted as the consum
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. Th
sends the number 1 downstream and the consumer prints this number.

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

```
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 automatic

> ⚠ You should call `take()` only once!

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

```
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 consumer
producers to a single consumer (collector) or any combination of the above that results in a direc
(DAG).

The *KanbanFlowTest* class has many examples for such structures, including scenarios where a
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 p
his own consumer that increases a product value in every cycle. The generator itself remains sta
is only stored as a product riding on a tray. Such a generator can be used for e.g. lazy sequence
"heartbeat" of a subsequent flow.

The approach of generator "loops" can equally be applied to collectors, where a collector does n
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 tha
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 ca
to form new greater flows from existing smaller ones.

```
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
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 fl

```
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* . . example will not allow much concurrency.

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

Test:
[KanbanFlowTest]
Demos:
[DemoKanbanFlow]
[DemoKanbanFlowBroadcast]
[DemoKanbanFlowCycle]
[DemoKanbanLazyPrimeSequenceLoops]

7.10 Classic Examples

The Sieve of Eratosthenes implementation using dataflow tasks

```
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 implementation using a combination of dataflow operators

```
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 operato
        */
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)
       }
```

8 STM

Software Transactional Memory (STM) gives developers transactional semantics for accessing ir multiple threads share data in memory, by marking blocks of code as transactional (atomic) the c the responsibility for data consistency to the Stm engine. GPars leverages the Multiverse Stm en details on the transactional engine at the [Multiverse site]()

Running a piece of code atomically

When using Stm, developers organize their code into transactions. A transaction is a piece of coc **atomically** - either all the code is run or none at all. The data used by the transactional code rem irrespective of whether the transaction finishes normally or abruptly. While running inside a trans given an illusion of being **isolated** from the other concurrently run transactions so that changes t transaction are not visible in the other ones until the transactions commit. This gives us the **ACI** p characteristics of database transactions. The **durability** transactional aspect so typical for datab mandated for Stm.

GPars allows developers to specify transaction boundaries by using the *atomic* closures.

```
import groovyx.gpars.stm.GParsStm
import org.multiverse.api.references.TxnInteger
import static org.multiverse.api.StmUtils.newTxnInteger

public class Account {
    private final TxnInteger amount = newTxnInteger(0);

public void transfer(final int a) {
        GParsStm.atomic {
            amount.increment(a);
        }
    }

public int getCurrentAmount() {
        GParsStm.atomicWithInt {
            amount.get();
        }
    }
}
```

There are several types of *atomic* closures, each for different type of return value:

- *atomic* - returning *Object*
- *atomicWithInt* - returning *int*
- *atomicWithLong* - returning *long*
- *atomicWithBoolean* - returning *boolean*
- *atomicWithDouble* - returning *double*
- *atomicWithVoid* - no return value

Multiverse by default uses optimistic locking strategy and automatically rolls back and retries coll Developers should thus restrain from irreversible actions (e.g. writing to the console, sending anc missile, etc.) in their transactional code. To increase flexibility, the default Multiverse settings can through custom *atomic blocks* .

Customizing the transactional properties

Frequently it may be desired to specify different values for some of the transaction properties (e.g
transactions, locking strategy, isolation level, etc.). The *createAtomicBlock* method will create a r
configured with the supplied values:

```
import groovyx.gpars.stm.GParsStm
import org.multiverse.api.AtomicBlock
import org.multiverse.api.PropagationLevel

final TxnExecutor block = GParsStm.createTxnExecutor(maxRetries: 3000, familyName: 'Custom', PropagationLevel:
PropagationLevel.Requires, interruptible: false)
assert GParsStm.atomicWithBoolean(block) {
    true
}
```

The customized *AtomicBlock* can then be used to create transactions following the specified sett
instances are thread-safe and can be freely reused among threads and transactions.

Using the *Transaction* object

The atomic closures are provided the current *Transaction* as a parameter. The *Txn* objects repre
can then be used to manually control the transaction. This is illustrated in the example below, wh
method to block the current transaction until the counter reaches the desired value:

```
import groovyx.gpars.stm.GParsStm
import org.multiverse.api.PropagationLevel
import org.multiverse.api.TxnExecutor

import static org.multiverse.api.StmUtils.newTxnInteger

final TxnExecutor block = GParsStm.createTxnExecutor(maxRetries: 3000, familyName: 'Custom', PropagationLevel:
PropagationLevel.Requires, interruptible: false)

def counter = newTxnInteger(0)
final int max = 100
Thread.start {
    while (counter.atomicGet() < max) {
        counter.atomicIncrementAndGet(1)
        sleep 10
    }
}
assert max + 1 == GParsStm.atomicWithInt(block) { tx ->
    if (counter.get() == max) return counter.get() + 1
    tx.retry()
}
```

Data structures

You might have noticed in the code examples above that we use dedicated data structures to ho
that normal Java classes do not support transactions and thus cannot be used directly, since Mu
able to share them safely among concurrent transactions, commit them nor roll them back. We n
know about transactions:

- TxnIntRef
- TxnLongRef
- TxnBooleanRef
- TxnDoubleRef
- TxnRef

You typically create these through the factory methods of the *org.multiverse.api.StmUtils* class.

More information

We decided not to duplicate the information that is already available on the Multiverse website. P
Multiverse site and use it as a reference for your further Stm adventures with GPars.

9 Google App Engine Integration

GPars can be run on the Google App Engine (GAE) . It can be made part of Groovy and Java G/ as a plugged into Gaelyk. The small GPars App Engine integration library provides all the necess hook GAE services into GPars. Although you'll be running on GAE threads and leveraging GAE t high-level abstractions remain the same. With a few restrictions you can still use GPars actors, d parallel collections and other handy concepts.

Please refer to the GPars App Engine library documentation for details on how to proceed with G

10 Tips

General GPars Tips

Grouping

High-level concurrency concepts, like Agents, Actors or Dataflow tasks and operators can be gro
thread pools. The *PGroup* class and its sub-classes represent convenient GPars wrappers arour
Objects created using the group's factory methods will share the group's thread pool.

```
def group1 = new DefaultPGroup()
def group2 = new NonDaemonPGroup()

group1.with {
    task {...}
    task {...}
    def op = operator(...) {...}
    def actor = actor{...}
    def anotherActor = group2.actor{...}  //will belong to group2
    def agent = safe(0)
}
```

When customizing the thread pools for groups, consider using the existing GPars implementations - the *DefaultPool* or *ResizeablePool* classes. Or you may create you implementation of the *groovyx.gpars.scheduler.Pool* interface to pass to the *DefaultP NonDaemonPGroup* constructors.

Java API

Most of GPars functionality can be used from Java just as well as from Groovy. Checkout the *2.6 GPars from Java* section of the User Guide and experiment with the maven-based stand-alone J Take GPars with you wherever you go!

10.1 Performance

Your code in Groovy can be just as fast as code written in Java, Scala or any other programing la not be surprising, since GPars is technically a solid tasty Java-made cake with a Groovy DSL cre

Unlike in Java, however, with GPars, as well as with other DSL-friendly languages, you are very useful kind of code speed-up for free, a speed-up coming from a better and cleaner design of you with a concurrency DSL will give you smaller code-base with code using the concurrency primitiv constructs. So it is much easier to build robust concurrent applications, identify potential bottle-ne eliminate them.

While this whole User Guide is describing how to use Groovy and GPars to create beautiful and code, let's use this chapter to highlight a few places, where some code tuning or minor design co you interesting performance gains.

Parallel Collections

Methods for parallel collection processing, like *eachParallel()* , *collectParallel()* and such use *Par*
tree-like data structure behind the scenes. This data structure has to be built from the original col
call any of the parallel collection methods. Thus when chaining parallel method calls you might co
map/reduce API instead or resort to using the *ParallelArray* API directly, to avoid the *Parallel Arra*

```
GParsPool.withPool {
    people.findAllParallel{it.isMale()}.collectParallel{it.name}.any{it == 'Joe'}
    people.parallel.filter{it.isMale()}.map{it.name}.filter{it == 'Joe'}.size() > 0
    people.parallelArray.withFilter({it.isMale()} as Predicate).withMapping({it.name} as Mapper).any{it == 'Jo
}
```

In many scenarios changing the pool size from the default value may give you performance bene
tasks perform IO operations, like file or database access, networking and such, increasing the nu
pool is likely to help performance.

```
GParsPool.withPool(50) {
    …
}
```

Since the closures you provide to the parallel collection processing methods will get executed fre
concurrently, you may further slightly benefit from turning them into Java.

Actors

GPars actors are fast. *DynamicDispatchActors* and *ReactiveActors* are about twice as fast as the
they don't have to maintain an implicit state between subsequent message arrivals. The *DefaultA*
in performance with actors in *Scala* , which you can hardly hear of as being slow.

If top performance is what you're looking for, a good start is to identify the following patterns in yo

```
actor {
    loop {
        react {msg ->
            switch(msg) {
                case String:…
                case Integer:…
            }
        }
    }
}
```

and replace them with *DynamicDispatchActor* :

```
messageHandler {
    when{String msg -> ...}
    when{Integer msg -> ...}
}
```

The *loop* and *react* methods are rather costly to call.

Defining a *DynamicDispatchActor* or *ReactiveActor* as classes instead of using the *messageHan*
factory methods will also give you some more speed:

```
class MyHandler extends DynamicDispatchActor {
    public void handleMessage(String msg) {
        …
    }

public void handleMessage(Integer msg) {
        …
    }
}
```

Now, moving the *MyHandler* class into Java will squeeze the last bit of performance from GPars.

Pool adjustment

GPars allows you to group actors around thread pools, giving you the freedom to organize actors always worthwhile to experiment with the actor pool size and type. *FJPool* usually gives better ch *DefaultPool* , but seems to be more sensitive to the number of threads in the pool. Sometimes us or *ResizeableFJPool* could help performance by automatic eliminating unneeded threads.

```
def attackerGroup = new DefaultPGroup(new ResizeableFJPool(10))
def defenderGroup = new DefaultPGroup(new DefaultPool(5))

def attacker = attackerGroup.actor {...}
def defender = defenderGroup.messageHandler {...}
...
```

Agents

GPars *Agents* are even a bit faster in processing messages than actors. The advice to group age thread pools and tune the pool sizes and types applies to agents as well as to actors. With agent benefit from submitting Java-written closures as messages.

Share your experience

The more we hear about GPars uses in the wild the better we can adapt it for the future. Let us k GPars and how it performs. Send us your benchmarks, performance comparisons or profiling rep GPars for you.

10.2 Integration into hosted environment

Hosted environments, such as Google App Engine, impose additional restrictions on threading. F with these environments better, the default thread factory and timer factory can be customized. T class provides static initialization methods allowing third parties to register their own implementat *PoolFactory* and *TimerFactory* interfaces, which will then be used to create default pools and tim Dataflow and PGroups.

```
public final class GParsConfig {
    private static volatile PoolFactory poolFactory;
    private static volatile TimerFactory timerFactory;

public static void setPoolFactory(final PoolFactory pool)

public static PoolFactory getPoolFactory()

public static Pool retrieveDefaultPool()

public static void setTimerFactory(final TimerFactory timerFactory)

public static TimerFactory getTimerFactory()

public static GeneralTimer retrieveDefaultTimer(final String name, final boolean daemon)
}
```

The custom factories should be registered immediately after the application startup in order for A be able to use them for their default groups.

Compatibility

Some further compatibility problems may occur when running GPars in a hosted environment. Th
is probably the lack of ForkJoinThreadPool (aka jsr-166y) support in GAE. Functionality such as
GParsPool may thus not be available on some services as a result. However, GParsExecutorsPo
Agents and Stm should work normally even when using managed non-Java SE thread pools.

11 Conclusion

This was quite a wild ride, wasn't it? Now, after going through the User Guide, you're certainly rea robust and reliable concurrent applications. You've seen that there are many concepts you can c has its own areas of applicability. The ability to pick the right concept to apply to a given problem the rest of the system is key to being a successful developer. If you feel you can do this with GPa User Guide has been accomplished.

Now, go ahead, use GPars and have fun!