# The GPars User Guide

The Whole GPars Team <gpars-developers@googlegroups.com>

Version 1.2.1, 2015-12-14

# Table of Contents


Introducing Our User Guide . . . 2
- Enter **GPars** . . . 2
- Credits . . . 3

User Guide To Getting Started . . . 4
- A Few Assumptions . . . 4
- Ready ? . . . 4
- Download and Install . . . 5
- The **GPars** Artifact . . . 5
- Transitive Dependencies . . . 6

A Hello World Example . . . 7
- Code Conventions . . . 9
- Usage . . . 10
- Getting Set Up In An IDE . . . 11
- Applicability of Concepts . . . 11
- What’s New . . . 12
- **Java** API – Using **GPars** from **Java** . . . 13
- Actors . . . 14
- Convenience Factory Methods . . . 15
- Agents . . . 16
- Dataflow Concurrency . . . 17
- Dataflow Operators . . . 17
- Performance . . . 20
- Prerequisites . . . 20

User Guide To Data Parallelism . . . 21
- Parallel Collections . . . 21
- Meet Parallel Arrays . . . 22
- GParsPool . . . 23
- Avoid Side-Effects in Functions . . . 28
- GParsExecutorsPool . . . 29
- Usage of GParsExecutorsPool . . . 29
- Avoid Side-effects in Functions . . . 32
- Memoize . . . 32
- Examples Of Use . . . 33
- Fibonacci Example . . . 34
- Available Variants . . . 34
- Map-Reduce . . . 35
- Avoid Side-effects in Functions . . . 36
- Combine . . . 38

Parallel Arrays . . . . 40
Asynchronous Invocations . . . . 41
Composable Asynchronous Functions . . . . 43
Are We Concurrent Yet? . . . . 44
Fork-Join . . . . 53
Parallel Speculations . . . . 60
Parallel Speculations . . . . 60
Alternatives Using **Dataflow Variables** and **Streams** . . . . 62
User Guide To **CSP** . . . . 64
Communicating Sequential Processes . . . . 64
The **CSP** Model Principles . . . . 64
**CSP** with **GPars** Dataflow . . . . 65
Processes . . . . 65
Channels . . . . 66
Composition . . . . 67
Alternatives . . . . 68
Components . . . . 70
User Guide To Actors . . . . 73
Types of Actors . . . . 73
Actor Threading Model . . . . 74
Usage of Actors . . . . 75
Actors Principles . . . . 85
Stateless Actors . . . . 99
Tips and Tricks . . . . 107
Active Objects . . . . 114
Classic Examples . . . . 119
User Guide To Agents . . . . 130
Introduction . . . . 130
Concepts . . . . 131
Examples . . . . 132
Factory Methods . . . . 133
Listeners and Validators . . . . 134
Validator Gotchas . . . . 135
Grouping . . . . 136
Reading The Value . . . . 140
State Copy Strategy . . . . 140
Error Handling . . . . 140
Fair and Non-fair **Agents** . . . . 141
User Guide to Dataflow . . . . 142
Introduction . . . . 142
Implementation Detail . . . . 143

Benefits ... 144
Concepts ... 144
Principles ... 144
Dataflow Queues and Broadcasts ... 145
DataflowStream ... 147
Bind Handlers ... 152
Bind Handlers Grouping ... 152
Bind Handler Chaining ... 153
Lazy **Dataflow** Tasks and Variables ... 160
Dataflow Expressions ... 163
Bind Error Notification ... 165
Further Reading ... 165
Tasks ... 166
Deterministic Deadlocks ... 172
**Dataflows** Map ... 173
Returning Values From A Task ... 173
Joining Tasks ... 174
Selects ... 175
Guards ... 177
Priority Select ... 178
Collecting Results of Asynchronous Computations ... 180
Timeouts ... 180
Cancellations ... 181
Operators ... 182
Concepts ... 183
Constructing Operators ... 187
Holding State in Operators ... 188
Parallelize Operators ... 189
Selectors ... 194
Shutting Down Dataflow Networks ... 197
Emergency Shutdown ... 197
PoisonPill ... 199
Immediate Poison Pill ... 200
Poison With Counting ... 200
Poison Strategies ... 201
Termination Tips and Tricks ... 202
Keeping PoisonPill Inside a Given Network ... 203
Graceful Shutdown ... 204
Application Frameworks ... 206
Pipeline DSL ... 207
Overriding the Default **PGroup** ... 215

The Pipeline Builder . . . 215
Passing Construction Parameters Through the Pipeline DSL . . . 216
Implementation . . . 217
Using Plain **Java** Threads . . . 217
Synchronous Variables and Channels . . . 218
Synchronous Dataflow Broadcast . . . 219
Synchronous Dataflow Variable . . . 222
Kanban Flow . . . 222
KanbanFlow Description . . . 223
Classic Examples . . . 226
User Guide To **STM** . . . 231
*Software Transactional Memory* . . . 231
Running A Piece of Code Atomically . . . 231
Customizing the Transactional Properties . . . 232
Using the *Transaction* Object . . . 232
Data Structures . . . 233
More Information . . . 234
User Guide To **GAE** . . . 235
Google App Engine Integration . . . 235
User Guide To Remoting . . . 236
Introduction . . . 236
Remote Serialization . . . 237
Dataflows . . . 237
DataflowVariable . . . 238
DataflowBroadcast . . . 238
DataflowQueue . . . 239
Actors . . . 239
Remote Actor Names . . . 240
Agents . . . 241
General **GPars** Tips . . . 243
Grouping . . . 243
**Java** API . . . 243
Performance . . . 243
Parallel Collections . . . 244
Actors . . . 245
**Agents** . . . 246
Hosted Environments . . . 247
Compatibility . . . 248
User Guide To The Conclusion . . . 249

To download this guide as a PDF - click here

# Introducing Our User Guide

The world of mainstream computing is changing rapidly these days. If you open the case and look under the covers of your computer, you'll most likely see a dual-core processor there, or a quad-core one, if you have a high-end computer. We all now run our software on multi-processor systems.

*Why do people still create single-threaded code ?*

The code we write today and tomorrow will probably never run on a single processor system: parallel hardware has become standard. Not so with the software though, at least not yet. People still create single-threaded code, even though it will not be able to leverage the full power of current and future hardware.

The code we write today will probably never run on a single processor system !

Some developers experiment with low-level concurrency primitives, like threads, and locks or synchronized blocks. However, it has become obvious that the shared-memory multi-threading approach used at the application level causes more trouble than it solves. Low-level concurrency handling is usually hard to get right, and it's not much fun either.

With such a radical change in hardware, software inevitably has to change dramatically too. Higher-levelS OF concurrency and parallelism concepts like *map/reduce*, *fork/join*, *actors* and *dataflow* provide natural abstractions for different types of problem domains while leveraging the multi-core hardware.

## Enter GPars

Meet **GPars**, an open-source concurrency and parallelism library for **Java** and **Groovy** that gives you a number of high-level abstractions for writing concurrent and parallel code in **Groovy** (*map/reduce, fork/join, asynchronous closures, actors, agents, dataflow concurrency* and other concepts), which can make your **Java** and **Groovy** code concurrent and/or parallel with little effort.

With **GPars**, your **Java** and/or **Groovy** code can easily utilize all the available processors on the target system. You can run multiple calculations at the same time, request network resources in parallel, safely solve hierarchical divide-and-conquer problems, perform functional style map/reduce or data parallel collection processing or build your applications around the actor or dataflow model.

**Apache**

The **GPars** project is open sourced under the Apache 2 License.

If you're working on a commercial, open-source, educational or any other type of software project

in **Groovy**, download the binaries or integrate them from the **Maven** repository and get going. The door to writing highly concurrent and/or parallel **Java** and **Groovy** code is wide open. Enjoy!

## Credits

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

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

Over time, many other people have contributed their ideas, provided useful feedback or helped **GPars** in one way or another. There are many people in this group, too many to name them all, but let's list at least the most active:

- Hamlet d'Arcy
- Hans Dockter
- Guillaume Laforge
- Robert Fischer
- Johannes Link
- Graeme Rocher
- Alex Miller
- Jeff Gortatowsky
- Jiří Kropáček
- Jim Northrop

Many thanks to everyone!

# User Guide To Getting Started

## A Few Assumptions

Let's set out a few assumptions before we start :

- You know and use **Groovy** and/or **Java** : otherwise you'd not be investing your valuable time studying a concurrency and parallelism library for **Groovy** and/or **Java**.
- You definitely want to write code employing concurrency and parallelism concepts.
- If you are not using **Groovy**, you are prepared to pay the inevitable verbosity tax of using **Java**.
- You target multi-core hardware with your code.
- You appreciate that in concurrent and parallel code things can happen at any time, in any order, and, more likely, with more than one thing happening at once.

## Ready ?

With those assumptions in place, we can get started.

It's becoming more and more obvious that dealing with concurrency and parallelism at the thread/synchronized/lock level, as provided by the JVM, is far too low a level to be safe and comfortable.

Many high-level concepts, such as **actors** and **dataflow** have been around for quite some time. Parallel-chipped computers have been in use, at least in data centres if not on the desktop, long before multi-core chips hit the hardware mainstream.

So now is the time to adopt these higher-level abstractions into the mainstream software industry.

This is what **GPars** enables for the **Groovy** and **Java** languages, allowing them to use higher-level abstractions and, therefore, make development of concurrent and parallel software easier and less error prone.

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

- *Code-level helpers* - Constructs that can be applied to small parts of your code-base, such as an individual algorithms or data structure, without any major changes in the overall project architecture
  - Parallel Collections
  - Asynchronous Processing
  - Fork/Join (Divide/Conquer)
- *Architecture-level concepts* - Constructs that need to be taken into account when designing the project structure
  - Actors

- Communicating Sequential Processes (CSP)
- Dataflow
- Data Parallelism

- *Shared Mutable State Protection* - More than 95% of the current use of shared mutable states can be avoided using proper abstractions. Good abstractions are still necessary for the remaining 5% of those use cases, i.e. when shared mutable state cannot be avoided.
  - Agents
  - Software Transactional Memory (not fully implemented in **GPars** as yet)

# Download and Install

**GPars** is now distributed as part of **Groovy**. So if you have a **Groovy** installation, you should already have **GPars**. Your exact version of **GPars** will, of course, depend on which version of **Groovy** you use.

If you don't already have **GPars**, and you do have **Groovy**, then perhaps you should upgrade your **Groovy** !

---

If you need it, you can download **Groovy** from here, and **GPars** from here.

---

If you don't have a **Groovy** installation, but use **Groovy** by using dependencies or perhaps, just having the **groovy-all** artifact, then you will need to get **GPars**. Also if you want to use a different version of **GPars** to the one bundled with **Groovy**, or have an old **GPars**-free **Groovy** that you cannot upgrade, you will need to get **GPars**. The ways to download **GPars** are:

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

If you're building a **Grails** or a **Griffon** application, you can use the appropriate plugins to fetch our jar files for you.

---

# The GPars Artifact

As noted above, **GPars** is now distributed as standard with **Groovy**. If however, you have to manage this dependency manually, the **GPars** artifact is in the main **Maven** repository and was in the Codehaus main and snapshots repositories before they closed.

Release versions can be found in the **Maven** main repositories but, for now, the current development version (SNAPSHOT) was in the Codehaus snapshots repository. We're moving it to another location.

To use **GPars** from **Gradle** or Grapes, use the specification:

*A* **Gradle** *Example*

```
"org.codehaus.gpars:gpars:1.2.0"
```

You may need to add our snapshot repository manually to the search list in this latter case. Using **Maven** the dependency is:

*A Sample* **Maven** *Declaration*

```
<dependency>
    <groupId>org.codehaus.gpars</groupId>
    <artifactId>gpars</artifactId>
    <version>1.2.0</version>
</dependency>
```

---

# Transitive Dependencies

**GPars** as a library depends on **Groovy** versions later than 2.2.1. Also, the **Fork/Join** concurrency library must be available. This comes as standard with **Java** 7.

**GPars 2.0** will depend on **Java 8** and will only be usable with **Groovy 3.0** and later.

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

---

# A Hello World Example

Once you're setup, try the following **Groovy** script to confirm your setup is functioning properly.

*A* **Groovy** *Sample*

```
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 receive a message "Decrypted message: **Groovy** is parallel" on the console.

### Java API

**GPars** has been designed primarily for use with the **Groovy** programming language. Of course all **Java** and **Groovy** programs are just bytecodes running on the JVM, so **GPars** can be used with **Java** source.

Despite being aimed at **Groovy**, the solid technical foundation, plus the good performance characteristics of **GPars** makes it an excellent library for **Java** programs too. In fact much of **GPars** is written in **Java**, so there's no performance penalty for **Java** applications using **GPars**.

For details please refer to the **Java API** section.

To quick-test **GPars** with the **Java API**, compile and run the following **Java** code:

*Another* **Java** *Sample*

```
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");
    }
}
```

*Artifacts maybe needed*

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

# Code Conventions

We follow certain conventions in our code samples. Understanding these conventions may help you read and comprehend **GPars** code samples better.

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

*Using The leftShift Operator*

```
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 sending a message to 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 as a parameter.

# Usage

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's more a matter of style preferences and personal taste, but we think static imports make the code more compact and readable.

---

# Getting Set Up In An IDE

Adding the **GPars** jar files to your project or defining the appropriate dependencies in pom.xml should be enough to get you started with **GPars** in your IDE.

### GPars DSL recognition

**IntelliJ IDEA** in both the free *Community Edition* and the commercial *Ultimate Edition* will recognize the **GPars** domain specific languages, complete methods like *eachParallel()* , *reduce()* or *callAsync()* and validate them. **GPars** uses the **Groovy** DSL mechanism, which teaches IntelliJ IDEA the DSLs as soon as the **GPars** jar file is added to the project.

# Applicability of Concepts

**GPars** provides a lot of concepts to pick from. We're continuously building and updating our documents to help users choose the right level of abstraction for their tasks at hands. Please, refer to Concepts Compared for details.

To briefly summarize the suggestions, here are some basic guide-lines:

- You're looking at a collection, which needs to be **iterated** or processed using one of the many beautiful **Groovy** collection methods, like *each()* , *collect()* , *find()* etc.. Suppose that processing each element of the collection is independent of the other items, then using **GPars parallel collections** can be appropriate.
- If you have a **long-lasting calculation** , which may safely run in the background, use the **asynchronous invocation support** in **GPars**. Since **GPars** asynchronous functions can be composed, you can quickly parallelize tyhese complex functional calculations without having to mark independent calculations explicitly.
- Say you need to **parallelize** an algorithm. You can identify a set of **tasks** with their mutual dependencies. The tasks typically do not need to share data, but instead some tasks may need to wait for other tasks to finish before starting. Now you're ready to express these dependencies explicitly in code. With **GPars dataflow tasks**, you create internally sequential tasks, each of which can run concurrently with the others. **Dataflow** variables and channels provide the tasks with the capability to declare their dependencies and to exchange data safely.
- Perhaps you can't avoid using **shared mutable state** in your logic. Multiple threads will be accessing shared data and (some of them) modifying it. A traditional locking and synchronized approach feels too risky or unfamiliar? Then go for **agents** to wrap your data and serialize all access to it.
- You're building a system with high concurrency demands. Tweaking a data structure here or task there won't cut it. You need to build the architecture from the ground up with concurrency in mind. **Message-passing** might be the way to go. Your choices could include :

- **Groovy CSP** to give you highly deterministic and composable models for concurrent processes. A model is organized around the concept of **calculations** or **processes**, which run concurrently and communicate through synchronous channels.
- If you're trying to solve a complex data-processing problem, consider **GPars dataflow operators** to build a data flow network. The concept is organized around event-driven transformations wired into pipelines using asynchronous channels.
- **Actors** and **Active Objects** will shine if you need to build a general-purpose, highly concurrent and scalable architecture following the object-oriented paradigm.

Now you may have a better idea of what concepts to use on your current project. Go check out more details on them in our **User Guide**.

---

# What's New

The next **GPars 1.3.0** release introduces several enhancements and improvements on top of the previous release, mainly in the dataflow area.

Check out the JIRA release notes.

---

### Project Changes

> **Breaking Changes**
>
> See the Breaking Changes listing for the lists of bug fixes and improvements.

### Asynchronous Functions

**TBD**

### Parallel Collections

**TBD**

### Fork / Join

**TBD**

### Actors

- Remote actors
- Exception propagation from active objects

### Dataflow

- Remote dataflow variables and channels
- Dataflow operators accepting variable number arguments
- Select made @CompileStatic compatible

### Agent

- Remote agents

### STM

**TBD**

### Other

- Raised the JDK dependency to version 1.7
- Raised the **Groovy** dependency to version 2.2
- Replaced the **jsr-177y fork-join** pool implementation with the one from JDK 1.7
- Removed the dependency on **jsr-166y**

---

# Java API – Using GPars from Java

Using **GPars** is very addictive, I guarantee. Once you get hooked, you won't be able to code without it. If the world forces you to write code in **Java**, you'll still be able to benefit from many of the **GPars** features.

## Java API specifics

Some parts of **GPars** are irrelevant in **Java** and it's better to use the underlying **Java** libraries directly:

- Parallel Collection – use *jsr-166y* library's **Parallel Array** directly until **GPars 1.3.0** becomes available
- Fork/Join – use *jsr-166y* library's **Fork/Join** support directly until **GPars 1.3.0** becomes available
- Asynchronous functions – use **Java** executor services directly

The other parts of **GPars** can be used from **Java** just as from **Groovy**, although most will miss the **Groovy** DSL capabilities.

## GPars Closures in Java API

To overcome the lack of closures as a language element in **Java** and to avoid forcing users to use **Groovy** closures directly through the **Java** API, a few handy wrapper classes have been provided to help you define callbacks, **actor** 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 places where the **GPars API** expects a **Groovy** closure.

---

# Actors

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

*A* **DynamicDispatchActor** *Sample*

```
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");
    }
}
```

There are few differences between **Groovy** and **Java** for **GPars** use, but notice the callbacks

instantiating the *MessagingRunnable* class in place of a **Groovy** closure.

*A* **MessagingRunnable** *Sample*

```
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 expect them.

*Factory Samples*

```
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

*Agent Samples*

```
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 avoid the handy overloaded operators and go straight to the methods, like *bind* , *whenBound*, *getVal* and other.

You may also continue to use **dataflow** tasks passing them instances of *Runnable* or *Callable* just like groovy closures.

*Dataflow Samples*

```
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** APIs for

dataflow operators.

- Use the convenience factory methods when accepting lists of channels to create operators or selectors
- Use *DataflowMessagingRunnable* to specify the operator body
- Call *getOwningProcessor()* to get hold of the operator from within the body in order to e.g. bind output values

```
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 DataflowMessagingRunnable(1) {
            @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 DataflowMessagingRunnable(2) {
            @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** and it tends to be very low anyway. **GPars actors**, for example, can compete head-to-head with other JVM **actor** options, like **Scala actors**.

Since **Groovy** code, in general, runs a little slower than **Java** code, due to dynamic method invocations, you might consider writing your code in **Java** to improve performance.

Typically numeric operations or frequent fine-grained method calls within a task or **actor** body may benefit from a rewrite into **Java**.

# Prerequisites

All the **GPars** integration rules apply equally to **Java** projects and **Groovy** projects. You only need to include the **Groovy** distribution jar file in your project and your are good-to-go.

You may also want to check out our sample **Java-Maven** project for tips on how to integrate **GPars** into a **Maven**-based pure **Java** application – **Java** Sample **Maven** Project

---

# User Guide To Data Parallelism

Focusing on data instead of processes helps us create robust concurrent programs. As a programmer, you define your data together with functions that should be applied to it and then let the underlying machinery process the data. Typically, a set of concurrent tasks will be created and submitted to a thread pool for processing.

In **GPars**, the **GParsPool** and **GParsExecutorsPool** classes give you access to low-level data parallelism techniques. The **GParsPool** class relies on the **Fork/Join** implementation introduced in **JDK** 7 and offers excellent functionality and performance. The **GParsExecutorsPool** is provided for those who still need to use the older Java executors.

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

- Processing collections concurrently
- Running functions (closures) asynchronously
- Performing **Fork/Join** (Divide/Conquer) algorithms

---

The API described here is based on using **GPars** with **JDK7**. It can be used with later JDKs, but **JDK8** introduced the **Streams** framework which can be used directly from **Groovy** and, in essence, replaces the **GPars** features covered here. Work is underway to provide the API described here based on the **JDK8 Streams** framework for use with **JDK8** and later to provide a simple upgrade path.

---

## Parallel Collections

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

Take, for example, the *min* function, which is supposed to return the smallest element of a collection. When you call the *min* method on a collection of numbers, a variable (*minVal* say) is created to store the smallest value seen so far, initialized to some reasonable value for the given type, so for example for integers and floating points, this may well be zero. The elements of the collection are then iterated through as each is compared to the stored value. Should a value be less than the one currently held in *minVal* then *minVal* is changed to store the newly seen smaller value.

Once all elements have been processed, the minimum value in the collection is stored in the *minVal*.

However simple, this solution is **totally wrong** on multi-core and multi-processor hardware. Running the *min* function on a dual-core chip can leverage **at most 50%** of the computing power of the chip. On a quad-core, it would be only 25%. So in this latter case, this algorithm effectively wastes 75% of the computing power of the chip.

Tree-like structures prove to be more appropriate for parallel processing.

The *min* function in our example doesn't need to iterate through all the elements in row and compare their values with the *minVal* variable. What it can do, instead, is rely on the multi-core/multi-processor nature of our hardware.

A *parallel_min* function can, for example, compare pairs (or tuples of certain size) of neighboring values in the collection and promote the smallest value from the tuple into a next round of comparisons. Searching for `the minimum` in different tuples can safely happen in parallel, so tuples in the same round can be processed by different cores at the same time without races or contention among threads.

---

## Meet Parallel Arrays

Although not part of **JDK7**, the **extra166y** library brings a very convenient abstraction called Parallel Arrays, and **GPars** has harnessed this mechanism to provide a very **Groovy API**.

> **What is `extra166y` ?**
>
> **extra166y** is an implementation of **Java** collections supporting parallel operations using **Fork-Join** concurrent framework provided by **JSR-166**. It was never made part of the JDK — unlike the **jsr166y** library. In fact, **extra166y** was made redundant from **JDK8** onwards by the **Streams** framework. Therefore, to continue to support **JDK7**, **GPars** includes a copy of **extra166y** in it so there is no external dependency.

As noted earlier, work is underway to rewrite the **GPars** API in terms of **Streams** for users of **JDK8** onwards. Of course people using **JDK8** onwards can simply use **Streams** directly from **Groovy**.

### How ?

**GPars** leverages the `Parallel Arrays` implementation in several ways. The **GParsPool** and **GParsExecutorsPool** classes provide parallel variants of the common **Groovy** iteration methods like *each* , *collect* , *findAll*, etc.

*A Parallel Sample*

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

It also allows for a more functional style **map/reduce** style of collections processing.

*A Map/Reduce Sample*

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

# GParsPool

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

## Usage

The **GParsPool** class provides (from **JSR-166y**), a **ParallelArray**-based concurrency DSL for collections and objects.

## Examples of use:

*Some Parallel Samples*

```
// 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 then be freely used inside the closure.

*A* **ForkJoinPool** *Sample*

```
// 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 created pool plus an `unhandled exceptions` handler.

*An Exception Handler Sample With Number of Threads Required*

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

### Pool Reuse

The *GParsPool.withExistingPool* takes an already existing **ForkJoinPool** instance to reuse. The DSL is valid only within the associated block of code and only for the thread that has called the *withPool* or *withExistingPool* methods. The *withPool* method returns only after all the worker threads have finished their tasks and the pool has been destroyed, returning the resulting value of the associated block of code. The *withExistingPool* method doesn't wait for the pool threads to finish.

Alternatively, the **GParsPool** class can be statically imported as *import static groovyx.gpars.GParsPool*, so we can omit the **GParsPool** class name.

*A Pool Sample*

```
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 classes or

individual instances with the parallel methods.

*An Enhanced Sample*

```
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 an a')
```

When using the **ParallelEnhancer** class, you're not restricted to a *withPool* block when using the **GParsPool** DSLs. The enhanced classes or instances remain enhanced till they are garbage collected.

## Exception Handling

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

### Exception Handling

The exception handling mechanism of **GParsPool** builds on the one provided in the **Fork/Join** framework. Since **Fork/Join** algorithms are by nature hierarchical, once any part of the algorithm fails, there's usually little benefit continuing the computation, since some branches of the algorithm will never return a result.

Bear in mind that the **GParsPool** implementation doesn't give any guarantees about its 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 consistent with 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 original iteration methods.

For example, you may be passing a collection into a library method, which will process your collection in a sequential way, let's say, by using the *collect* method. Then by changing the semantics of the *collect* method on your collection, you can effectively parallelize this library sequential code.

*A makeConcurrent() Sample*

```
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.

*A Sequential Sample*

```
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 allows us to specify code blocks, where the collection

maintains concurrent semantics.

*An asConcurrent() Sample*

```
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()
```

## Code Samples

Transparent parallelism, including the *makeConcurrent()* , *makeSequential()* and *asConcurrent()* methods, is also available in combination with our *ParallelEnhancer* .

*A ParallelEnhancer Sample*

```
/**
 * 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())
```

*Another ParallelEnhancer Sample*

```
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 *eachParallel* or *collectParallel()* may be run in parallel, you have to make sure that each of the closures is written

in a thread-safe manner. The closures must hold no internal state, share data nor have side-effects beyond the boundaries of the single element that they've been invoked on. Violations of these rules will open the door for race conditions and deadlocks, the most severe enemies of a modern multi-core programmer.

**Don't do this !**

*Concurrently Accessing a non-Thread-Safe Collection*

```
def thumbnails = []
images.eachParallel {thumbnails << it.thumbnail}  //Concurrently accessing a not-thread-safe collection of thumbnails? Don't do this!
```

At least, you've been warned.

**It May Not Execute The Way You Expect**

Because **GParsPool** uses a **Fork/Join** pool (with work stealing), threads may not be applied to a waiting processing task even though they may appear idle.

With a work-stealing algorithm, worker threads that run out of things to do can steal tasks from other threads that are still busy.

If you use **GParsExecutorsPool** (which doesn't use **Fork/Join**), you'll get the thread allocation behavior that you would naively expect.

# GParsExecutorsPool

Use of **GParsExecutorsPool** - the `Java Executors`-based concurrent collection processor -

# Usage of GParsExecutorsPool

The **GParsPool** classes enable a `Java Executors`-based concurrency DSL for collections and objects.

The **GParsExecutorsPool** class can be used as a pure-JDK-based `collections parallel processor`. Unlike the **GParsPool** class, **GParsExecutorsPool** doesn't require **fork/join** thread pools but, instead, leverages the standard JDK executor services to parallelize closures to process a collection or an object iteratively.

It needs to be stated, however, that **GParsPool** typically performs much better than **GParsExecutorsPool** does.

**GParsPool** typically performs much better than **GParsExecutorsPool**

## Examples of Use

*A* **GParsExecutorsPool** *Example*

```
//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 an **ExecutorService** as a parameter, which can be then used freely inside the closure.

*Another* **GParsExecutorsPool** *Example*

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

The *GParsExecutorsPool.withPool()* method takes an optional parameter declaring the number of threads in the created pool and a thread factory.

*An Example Declaring Required Thread Count*

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

The *GParsExecutorsPool.withExistingPool()* takes an already existing `executor service instance` to reuse. The DSL is only valid within the associated block of code and only for the thread that has called the *withPool()* or *withExistingPool()* method.

Did you know the *withExistingPool()* method doesn't wait for `executor service threads` to finish ?

The *withPool()* method returns control only after all the worker threads have finished their tasks and the executor service has been destroyed, returning the resulting value of the associated block of code.

Statically import the **GParsExecutorsPool** class as import static groovyx.gpars.GParsExecutorsPool.* to omit the **GParsExecutorsPool** class name.

*A FindParallel Example*

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

The following methods are currently supported on all objects that support iterations in **Groovy** :

- 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 for any classes or individual instances having 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 a')
```

When using the *GParsExecutorsPoolEnhancer* class, you're not restricted to a *withPool()* block with the use of the `GParsExecutorsPool DSLs`. The enhanced classes or instances remain enhanced until they are garbage collected.

### Exception Handling

Exceptions can be thrown while processing any of the passed-in closures. An instance of the *AsyncException* method will wrap any/all of the original exceptions re-thrown from the xxxParallel methods.

## Avoid Side-effects in Functions

Once again we need to warn you about using closures with side-effects. Please avoid logic that affects objects beyond the scope of the single, currently processed element. Please avoid logic or closures that keep state. Don't do that! It's dangerous to pass them to any of the *xxxParallel()* methods.

---

# Memoize

The *memoize* function enables caching of a function's return values. Repeated calls to the memoized function with the same argument values will, instead of invoking the calculation encoded in the original function, retrieve the resulting value from an internal, transparent cache.

Provided the calculation is considerably slower than retrieving a cached value from the cache, developers can trade-off memory for performance.

Checkout out the example, where we attempt to scan multiple websites for particular content:

The **memoize** functionality of **GPars** was donated to **Groovy** for version 1.8 and if you run on **Groovy** 1.8 or later, we recommend you use the **Groovy** functionality.

**Memoize**, in **GPars**, is almost identical, except that it searches the memoized caches concurrently

using the surrounding thread pool. This may give performance benefits in some scenarios.

**Memoize Me Up, Scotty**

The **GPars memoize** functionality has been renamed to avoid future conflicts with the **memoize** functionality in **Groovy**.

**GPars** now calls these methods with a preceding letter *g* , such as *gmemoize()*.

# Examples Of Use

*A* **GParsPool** *Example With gmemoize()*

```
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('CONCURRENCY')}
    println 'GPars sites today: ' + urls.findAllParallel {url -> cachingDownload(url)
.contains('GPARS')}
}
```

Notice how closures are enhanced inside the *GParsPool.withPool()* blocks with a *memoize()* function. This returns a new closure wrapping the original closure as a cache entry.

In the previous example, we're calling the *cachingDownload* function in several places in the code, however, each unique url is downloaded only once - the first time it's needed. The values are then cached and available for subsequent calls. Additionally, these values are also available to all threads, no matter which thread originally came first with a download request  for that particular url and had to handle the actual calculation/download.

So, to wrap up, a **memoize** call shields a function by using a cache of past return values.

However, *memoize* can do even more! In some algorithms, adding a little memory may have a dramatic impact on the computational complexity of the calculation. Let's look at a classical example of `Fibonacci` numbers.

# Fibonacci Example

A purely functional, recursive implementation that follows the definition of Fibonacci numbers is exponentially complex:

*A Fibonacci Example*

```
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 an added **memoize** cache, the algorithm magically turns into a linearly complex one:

*A Better Version of the Fibonacci Example*

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

The extra memory we added has now cut off all but one recursive branch of the calculation. And all subsequent calls to the same *fib* function will also benefit from the cached values.

Look below to see how the *memoizeAtMost* variant can reduce memory consumption in our example, yet preserve the linear complexity of the algorithm.

---

# Available Variants

## Memoize

The basic variant keeps values in the internal cache for the whole lifetime of the memoized function. It provides the best performance characteristics of all the variants.

## memoizeAtMost

Allows us to set a hard limit on number of items cached. Once the limit has been reached, all subsequently added values will eliminate the oldest value from the cache using the **LRU** (`Last Recently Used`) strategy.

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

*A Cached Fibonacci Example*

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

Setting an upper limit on the cache size serves two purposes:

- Keeps the memory footprint of the cache within defined boundaries
- Preserves desired performance characteristics of the function. Too large a cache increases the time to retrieve a cached value, compared to the time 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 and evict a `SoftReferences` entry (used by our implementation) from the memory.

The single parameter to the *memoizeAtLeast()* method indicates the minimum number of cached items that should be protected from gc eviction. The cache will never shrink below the specified number of entries. The cache ensures it only protects the most recently used items from eviction using the LRU (`Last Recently Used`) strategy.

### memoizeBetween

Combines the **memoizeAtLeast** and **memoizeAtMost** methods to allow the cache to grow and shrink in the range between the two parameter values depending on available memory and the gc activity.

The cache size will never exceed the upper size limit to preserve desired performance characteristics of the cache.

---

# Map-Reduce

The `Parallel Collection Map/Reduce` DSL gives **GPars** a more functional flavor. In general, the `Map/Reduce DSL` may be used for the same purpose as the *xxxParallel()* family of methods and has very similar semantics. On the other hand, **Map/Reduce** can perform considerably faster, if you need to chain multiple methods together to process a single collection in multiple steps:

*A* **Map-Reduce** *Example*

```
    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 must follow the same contract as their non-parallel peers. So a *collectParallel()* method must return a legal collection of items, which you can treat as a **Groovy** collection.

Internally, the *parallel collect method* builds an efficient parallel structure, called a `parallel array`. It then performs the required operation concurrently. Before returning, it destroys the *Parallel Array* as it builds a collection of results to return to you. A potential call to, for example, *findAllParallel()* on the resulting collection would repeat the whole process of construction and destruction of a `Parallel Array` instance under the covers.

With **Map/Reduce**, you turn your collection into a `Parallel Array` and back again only a single time. The **Map/Reduce** family of methods do not return **Groovy** collections, but can freely pass along the internal `Parallel Arrays` directly.

Invoking the *parallel* property of a collection will build a `Parallel Array` for the collection and then return a thin wrapper around the `Parallel Array` instance. Then you can chain any of these methods together to get an answer :

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

Returning a plain **Groovy** collection instance is always just a matter of retrieving the *collection* property.

*A* **Map-Reduce** *Example*

```
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 any closures you pass to the **Map/Reduce** functions, stateless and clean from side-effects.

To avoid nasty surprises keep your closures stateless

## Availability

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

## Classical Example

A classical example, inspired by thevery, counts occurrences of words in a string:

*A Telling Example*

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

def words = "This is just 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 can be implemented with the more general *combine* operation:

*A Combine Example*

```
def words = "This is just 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 an input list of tuples (two-element lists), often considered to be key-value pairs (such as [ [key1, value1], [key2, value2], [key1, value3], [key3, value4] … ] ). These might have potentially repeating keys.

When invoked, the *combine* method merges the values of identical keys using the provided accumulator function. This produces a map of the original (unique) keys and their (now) accumulated values.

E.g.  will be combined into [a : b+e, c : d+f]. Some logic like the '+' operation for the values will need to be provided as the accumulation closure logic.

The *accumulation function* argument needs to specify a function to use when combining (accumulating) values belonging to the same key.  An *initial accumulator value* needs to be provided as well.

Since the *combine* method processes items in parallel, the *initial accumulator value* will be reused multiple times. Thus the provided value must allow for reuse.

It should either be a **cloneable** (or **immutable**) value or a **closure** returning a fresh initial accumulator each time it's requested.  Good combinations of accumulator functions and reusable initial values include:

*Some Examples of a Combining-Accumulator Function and Reusable Initial Value*

```
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 an initial value of zero will combine into ['he' : 4, 'she' : 7, 'me' : 1]

### Comparion Logic

The keys will be mutually compared using their **equals** and **hashCode** methods. Consider using *@Canonical* or *@EqualsAndHashCode* annotations to annotate objects you use as keys.

As with 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 have a complete class to use as a key for common use cases and to apply different keys for uncommon cases.

*A Complex Example*

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

import static groovyx.gpars.GParsPool.withPool

// declare a complete class to use in combination processing
@TupleConstructor @ToString
class PricedCar implements Cloneable {  // either Clonable or Immutable
    String model
    String color
    Double price

    // declare a way to resolve comparison logic
    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()
    }
}

// some data
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 (using 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
}
```

## Parallel Arrays

As an alternative, the efficient tree-based data structures defined in **JSR-166y** - Java Concurrency can be used directly. The *parallelArray* property on any collection or object will return a *ParallelArray* instance holding the elements of the original collection. These then can be manipulated through the **jsr166y** API.

Please refer to **jsr166y** documentation for API details.

*A Parallel Array Example*

```
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.Reducer, 0)       //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, "")
```

# Asynchronous Invocations

Long running background tasks happen a lot in most systems.

Typically, a main thread of execution wants to initialize a few calculations, start downloads, do searches, etc. even when the results may not be needed immediately.

**GPars** gives the developers the tools to schedule asynchronous activities for background processing and collect the results later, when they're needed.

## Usage of GParsPool and GParsExecutorsPool Asynchronous Processing Facilities

Both **GParsPool** and **GParsExecutorsPool** methods provide nearly identical services while leveraging different underlying machinery.

### Closures Enhancements

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

- async() - To create an asynchronous variant of the supplied closure which, when invoked, returns a **future** object for the potential return value
- callAsync() - Calls a closure in a separate thread while supplying the given arguments, returning a **future** object for the potential return value,

*An* **async()** *Example*

```
GParsPool.withPool() {
    Closure longLastingCalculation = {calculate()}
    Closure fastCalculation = longLastingCalculation.async()  //create a new closure,
which starts the original closure on a thread pool

    Future result=fastCalculation()                           //returns almost
immediately

    //do stuff while calculation performs ...
    println result.get()
}
```

*A* **callAsync()** *Example*

```
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, provides a timer mechanism.

*A Timed Example*

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

To allow cancellation, our asynchronously running code must keep checking the *interrupted* flag of it's own thread and stop calculating when/if the flag is set to true.

## Executor Service Enhancements

The **ExecutorService** and **ForkJoinPool** classes are enhanced with the **'<<'** (leftShift) operator to submit tasks to the pool and return a *Future* for the result.

*A Convenient Example Using [red]'<<'*

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

### Running Functions (closures) in Parallel

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

Example:

*An Example*

```
GParsPool.withPool {
    assert [10, 20] == GParsPool.executeAsyncAndWait({calculateA()}, {calculateB()}
//waits for results
    assert [10, 20] == GParsPool.executeAsync({calculateA()}, {calculateB()})*.get()
//returns Futures instead and doesn't wait for results to be calculated
}
```

---

## Composable Asynchronous Functions

Functions are to be composed. In fact, composing side-effect-free functions is very easy. Much easier and more reliable than composing objects, for example.

Given the same input, functions always return the same result, they never change 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 a value. Provided you don't let your closures touch anything outside their scope, your closures are well-behaved, just like pure functions. Functions that you can combine for a higher good.

*A Higher Good Example*

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

For this example, by combining a function adding two numbers `{a,b}` with the *inject* function, which iterates through the whole collection, you can quickly summarize all items. Then, replacing the *adding* function with a *comparison* function immediately gives you a combined function to calculate maximums.

*Find The Maximums*

```
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 using the full power of your expensive hardware. These functions are just plain sequential! No parallelism is used! All but one processor core is doing nothing, they're idle, totally wasted!

### A Generic Way to Use Asynchronous Functions

Those paying attention might decide to use the *Parallel Collection* techniques described earlier and they would certainly be correct.

For our scenario described here, where we process a collection, using those *parallel* methods would be the best choice. However, we're now looking for a generic way to create and combine asynchronous functions. This would help us, not only for collections processing, but mostly in other, more generic, cases like the one right below.

All but one processor core is doing nothing! They're idle! Totally wasted!

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

*An Example*

```
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.html')))
println "The result of comparison: " + result
```

We need to download the page, load up the file, calculate hashes of both and finally compare the resulting numbers. Each of the functions is responsible for one particular job. One function downloads the content, a second loads the file, and a third calculates the hashes and finally the fourth one will do the comparison.

Combining the functions is as simple as nesting their calls.

---

## Making It All Asynchronous

The downside of our code is that we haven't leveraged the independence of the *download()* and the *loadFile()* functions. Neither have we allowed the two hashes to be run concurrently. They could well run in parallel, but our approach to combine functions restricts parallelism.

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

So we can only take parallelism so far, while blocking parallelism of others. Seems like a challenging task.

## Things Are Bright in the Functional World

Luckily, the dependencies between functions are already expressed implicitly in the code. There's no need to duplicate that dependency information. If one functions takes parameters and the parameters need to be calculated first by another function, we implicitly have a dependency here.

The *hash()* function depends on *loadFile()* as well as on the *download()* functions in our example.

The *inject* function in our earlier example depended on the results of the *addition* functions gradually invoked on all elements of the collection.

### Our task is, in fact, very simple !

However difficult it may seem at first, our task is, in fact, very simple. We only need to teach our functions to return a *promise* of their future results. And we need to teach the other functions to accept those *promises* as parameters so that they will wait for the real values before they start their work.

And if we convince the functions to release the threads they hold, while waiting for the values, we get directly to where the magic can happen.

In the best traditions of **GPars**, we've made it very straightforward for you to convince any function to believe in the **promises** of other functions. Call the *asyncFun()* function on a closure and you're asynchronous !

*Promises, Promises*

```
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 returned from the *addition* function, maybe it's a little surprised each call to the *addition* function returns so fast, but doesn't moan much, keeps iterating and finally returns the overall result we expect.

Now is the time you should stand behind what you say and do what you want others to do. Don't frown at the result and just accept that you got back just a **promise**. A **promise** to get the answer delivered as soon as the calculation is complete. The extra heat from your laptop is an indication that the calculation exploits natural parallelism in your functions and makes its best effort to deliver the result to you quickly.

### A Promise Is A Promise

The *promise* is a good old *DataflowVariable*, so you can query its status, register some notification hooks or even make it an input to a **Dataflow** algorithm !

*An Promising Example*

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

    println "Are we done yet? " + sumPromise.bound

    sumPromise.whenBound {sum -> println sum}
}
```

**Do You Need A Timeout ?**

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

### Can Things Go Wrong?

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

*An Exceptional Example*

```
try {
    sumPromise.get()

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

---

### This Is All Fine, But What Functions Can Really Be Combined?

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

Review our initial example comparing the content of a file with a web page. We simply make all the functions asynchronous 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.html')))

    println 'Allowed to do something else now'
    println "The result of comparison: " + result.get()
```

### Calling Asynchronous Functions from Within Asynchronous Functions

Another very valuable attribute of asynchronous functions is that `promises` can be combined.

Promises can be combined !

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

  withPool {
      Closure plus = {Integer a, Integer b ->
          sleep 3000
          println 'Adding numbers'
          a + b
      }.asyncFun(); // ok, here's one func

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

      Closure measureTime = {->
          sleep 3000
          4
      }.asyncFun(); // and another

      // declare a function within a function
      Closure distance = {Integer initialDistance, Integer velocity, Integer time ->
          plus(initialDistance, multiply(velocity, time))
      }.asyncFun(); // and another


      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(); // and declare (but not run) a final asynch.function

      // fine, now let's execute those previous asynch. functions
      println "Distance = " + distance(100, 20, measureTime()).get() + ' m'
      println "ChattyDistance = " + chattyDistance(100, 20, measureTime()).get() + '
m'
  }
```

If an asynchronous function (e.g. like the *distance* function in this example) in its body calls another asynchronous function (e.g. *plus* ) and returns the the promise of the invoked function, the inner function's ( *plus* ) resulting promise will combine with the outer function's ( *distance* ) results promise.

The inner function ( *plus* ) will now bind its result to the outer function's ( *distance* ) promise, once the inner function (plus) finishes its calculation. This ability of promises to combine logic allows functions to cease their calculation without blocking a thread. This happens not only when waiting

for parameters, but also whenever they call another asynchronous function anywhere in their code body.

---

### Methods as Asynchronous Functions

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

*An Example*

```
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()   // notice the .& syntax
    Closure scanFor = d.&scanFor.asyncFun()     // and here
    Closure lower = d.&lower.asyncFun()         // and here

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

### Using Annotations to Create Asynchronous Functions

Instead of calling the *asyncFun()* function, the *@AsyncFun* annotation can be used to annotate Closure-typed fields. The fields have to be initialized in-place and the containing class needs to be instantiated within a *withPool* block.

*An Annotation Example*

```
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` block. You may, however, specify the type of pool explicitly:

*A Explicit Example*

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

**Blocking Functions Through Annotations**

The *AsyncFun* method also allows us to specify, whether the resulting function should allow blocking (true) or non-blocking (false - default) semantics.

*An Example of Blocking 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 asynchronous function, you have two additional ways to assign a thread pool to the function.

1. The thread pool to be used by the function can be specified explicitly as an additional argument at creation time
2. The implicit thread pool can be obtained from the surrounding scope at invocation-time rather at creation time

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

*To Specify Thread Pools Explicitly*

```
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 a delayed pool assignment, only the function invocation must be surrounded with a *withPool()* block:

*A Delayed Pool Assignment Example*

```
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()
}
```

For us, this is a very interesting domain to explore. So any comments, questions or suggestions are welcome on combining asynchronous functions or hints about its limits.

---

# 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 general tree navigation problems.

- **Fork/Join** algorithms essentially split a problem into several smaller sub-problems and then recursively applies 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 helps solve its' own grand-parent problem.

**A Picture Is Worth A Thousand Words**

Check out the fancy Interactive **Fork/Join** visualization demo. It shows you how threads co-operate to solve a common divide-and-conquer algorithm.

The mighty **JSR-166y** library co-ordinates **Fork/Join** orchestration rather nicely, but leaves a few rough edges, which can hurt you, if you don’t pay enough attention. You must still deal with threads, pools and/or synchronization barriers.

## The GPars Abstraction Convenience Layer

**GPars** can hide the complexities of dealing with threads, pools and recursive tasks from you, yet let you leverage the 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
        }

    }""".toString();
}
```

The *runForkJoin()* factory method uses the supplied recursive code together with the provided values to build a hierarchical **Fork/Join** calculation. The number of values passed to the *runForkJoin()* method must match the number of expected parameters of the closure. This must equal the same number of arguments passed to the *forkOffChild()* or *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
    }
}
```

**It's Asynchronous, Mate !**

The important piece of the puzzle to note here is that *forkOffChild()* doesn't wait for the child to run. It merely schedules it for execution at a future time. If a child task throws an exception, don't expect the exception to be fired from the *forkOffChild()* method itself. The exception will have happened long after the parent has called *forkOffChild()*.

It's the *getChildrenResults()* method that will re-throw any child sub-task exceptions back to the parent.

### Alternative Approach

Alternatively, the underlying mechanism of nested **Fork/Join** worker tasks can be used directly. Custom-tailored workers can eliminate the performance overhead associated with parameter spreading imposed when using the generic workers.

Also, custom workers can be implemented in **Java** for further increases in performance.

*A Custom Worker*

```
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 own result
    }
}

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 can be written in both **Java** and **Groovy**. Either choicr lets you optimize for execution speed, if low performance of the worker becomes a bottleneck.

## Fork / Join Saves Your Resources

**Fork/Join** operations can safely be run with small numbers of threads thanks to internal use of the **TaskBarrier** class to synchronize the threads.

While a thread is blocked inside an algorithm waiting for its sub-problems to be calculated, the thread is silently returned to it's pool to take on any other available sub-problems from the task queue and process them. Although the algorithm creates as many tasks as there are sub-directories and tasks wait for the sub-directory tasks to complete, often as few as a single thread is enough to keep the computation going and eventually calculate a valid result.

## Mergesort Example

*Come on Punk, Merge my day !*

```
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, 4, 3]

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 store own result
            }
        }
    }"""
}
```

## Mergesort Example Using A Custom-tailored Worker Class

*An Example*

```
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 child task
                return merge(* childrenResults)      //use results of children tasks
to calculate and store own 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, 4, 3]

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 — called the *runChildDirectly* method. This method will run the child task directly and immediately within the current thread instead of scheduling the child task for asynchronous processing on the thread pool. Typically you'd call *forkOffChild* on every sub-task but the last, which you invoke directly without the scheduling overhead.

*A Fork-In-Time-Saves-Nine*

```
Closure fib = {number ->
    if (number <= 2) {
        return 1
    }

    forkOffChild(number - 1)  // This task will run asynchronously, probably in a
different thread
    final def result = runChildDirectly(number - 2)     //  This task is run directly
within the current thread
    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** , but not **GParsExecutorsPool** .

---

# Parallel Speculations

With processor cores having become plentiful, some algorithms might benefit from brutal-force parallel duplication. Instead of deciding up-front about how to solve a problem, what algorithm to use or which location to connect to, you 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 file, database or internet. Luckily, you know several good ways (e.g. functions or urls) to reach your goal. However, all are not equal.

Although they return the same (as far as your needs are concerned) result, the elapsed time of each will differ and some may even fail (e.g. network issues). What's worse, no-one's going to tell you which choice gives you the single best solution nor which paths might lead to no solution at all.

1. Shall I run *quick sort* or *merge sort* on my list?
2. Which url will work best?
3. Is this service available at its primary location or should I use the backup one?

**GPars Speculations** give you the option to try all the available alternatives in parallel and receive

the result from the fastest functional path, silently ignoring the slow or broken ones.

This is what the *speculate* methods on **GParsPool** and **GParsExecutorsPool** can do for you.

*A Sort Example*

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

So we're performing both a *quick sort* and a *merge sort* at the same time (concurrently), while getting the result of the faster one.

Given the parallel resources available these days on mainstream hardware, running the two functions in parallel will not have a dramatic impact on speed of calculation of either one, and thus we get the results of both in about the same time as if we ran only ran the faster of the two calculations. And also, the result arrives sooner than when running the slower one. Yet we didn't have to know up-front, which of the two sorting algorithms would perform better on our data. Thus we speculated (guessed).

Similarly, downloading a document from several sources with different speeds and/or reliability might look like this:

*A Document DownLoad Example*

```
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')
}
```

### Thread Starvation

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

## Alternatives Using Dataflow Variables and Streams

In some use cases, we can ignore failing alternatives, so **Dataflow** variables or **Streams** may be used to obtain the results of the winning speculation.

### See this User Guide's topic on Dataflow Concurrency

Please refer to the **Dataflow Concurrency** section of this **User Guide** for details on **Dataflow Variables** and streams.

*An Example*

```
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')
```

# User Guide To CSP

## Communicating Sequential Processes

The **CSP** (Communicating Sequential Processes) abstraction builds on independent composable processes, which exchange messages in a synchronous manner. GPars leverages the **JCSP** library developed at the University of Kent, UK.

*Jon Kerridge*, the author of the **CSP** implementation in **GPars**, provides exhaustive examples on of **GroovyCSP** use at www.soc.napier.ac.uk or here on our local mirror page.

### Purpose

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

If the LGPL license is not adequate for your use, you might consider checking out the **Dataflow Concurrency** chapter of this **User Guide** to learn about *tasks* , *selectors* and *operators* , which may help you resolve concurrency issues in ways similar to the **CSP** approach. In fact, the dataflow and **CSP** concepts, as implemented in **GPars**, are very close to each other.

### Apache 2 License

By default, without actively adding an explicit dependency on **JCSP** in your build file or downloading and including the **JCSP** jar file in your project, the standard commercial-software-friendly *Apache 2 License* terms apply to your project. **GPars** directly only 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 communicate through channels using synchronous (i.e. rendezvous) message passing. Unlike actors or dataflow operators, which revolve around the event-processing pattern, **CSP** processes place the focus of their activities (aka sequences of steps) around the use of communications to remain mutually in sync along the way.

Since the addressing is indirect through channels, the processes do not need to know about one another. They typically consist of a set of input and output channels and a body. Once a **CSP** process is started, it obtains a thread from a thread pool and starts processing its body, pausing only when reading from a channel or writing into a channel. Some implementations (e.g. **GoLang**) can also detach the thread from the **CSP** process when blocked on a channel.

**CSP** programs are deterministic. The same data on the program's input will always generate the

same output, irrespective of the actual thread-scheduling scheme used. This helps a lot when debugging **CSP** programs as well as analyzing deadlocks.

Determinism combined with indirect addressing results in a great level of composability of **CSP** processes. You can combine small **CSP** processes into bigger ones just by connecting their input and output channels and then wrapping them by another, bigger containing process.

The **CSP** model introduces non-determinism using *Alternatives*. A process can attempt to read a value from multiple channels at the same time through a construct called *Alternative* or *Select*. The first value that becomes available in any of the channels involved in the *Select* will be read and consumed by the process. Since the order of messages received through a *Select* depends on unpredictable conditions during program run-time, the value that will be read is 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 *Callable* to hold the actual implementation of the process
- **CSP Channels** should be modelled with *SyncDataflowQueue* and *SyncDataflowBroadcast* classes
- **CSP Alternative** is provided through the *Select* class with its *select* and _ prioritySelect_ methods

---

# Processes

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

*Start A Process*

```
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’s advisable to use resizeable thread pools as in the example above.

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

*A Runnable Sample*

```
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 values to be returned through the *get()* method:

*A Callable Sample*

```
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 their companion processes as well as with the outside world:

*A Channel Sample*

```
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
```

### Which Delivery Technique To Use for Messages ?

The **CSP** model uses synchronous messaging, however, in **GPars** you may consider using asynchronous channels as well as synchronous ones.

You can also combine these two types of channels within the same process.

# Composition

Grouping processes simply becomes a matter of connecting them with channels:

*A Grouping Sample*

```
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* methods:

*A Select Sample*

```groovy
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 highest precedence to 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** processes (aka *Runnable*/*Callable* classes), you can compose them into a larger process:

*A Larger Sample*

```
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
        }
    }
}
```

*Another Building Block*

```
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()
        }
    }
}
```

*A Sample*

```
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.task it}
    }
}

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

sleep 10000
```

# User Guide To Actors

**Actors** offer a message passing-based concurrency model: programs are collections of independent active objects that exchange messages and have no mutable shared state.

**Actors** can help us avoid issues such as deadlock, live-lock and starvation, which are common problems for shared memory based approaches.

**Actors** are a way of leveraging the multi-core nature of today's hardware without all the problems traditionally associated with shared-memory multi-threading, which is why programming languages such as **Erlang** and **Scala** have taken up this model.

The actor support in **GPars** was originally inspired by the Actors library in Scala, but has since gone well beyond what Scala offers as standard.

A nice article summarizing the key concepts behind actors has been written by *Ruben Vermeersch*.

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

Ideally actor's code should **never be invoked** directly from outside so all the code of the actor class can only be executed by the thread handling the last received message and hence all the actor's code is **implicitly thread-safe** .

If any of the actor's methods are allowed to be called by other objects directly, the thread-safety guarantee for the actor's code and state are **no longer valid** .

# Types of Actors

In general, you can find two types of actors in the wild — ones that hold **implicit state** and ones that don't.

**GPars** gives you both options.

**Stateless** actors, represented in **GPars** by the *DynamicDispatchActor* and the *ReactiveActor* classes, keep no track of what messages have arrived previously. You may think of these as flat message handlers, which process messages as they come. Any state-based behavior has to be implemented by the user.

The **stateful** actors, represented in **GPars** by the *DefaultActor* class (and previously also by the *AbstractPooledActor* class), allow us to handle implicit state directly. After receiving a message, the actor moves into a new state with different ways to handle future messages.

To give you an example, a freshly started actor may only accept some types of messages, e.g. encrypted messages for decryption, only after it has received the encryption keys. The stateful actors allow to encode such dependencies directly in the structure of the message-handling code. Implicit state management, however, comes at a slight performance cost, mainly due to the lack of continuations support on JVM.

# Actor Threading Model

Since actors are detached from the system threads, a large number of actors can share a relatively small thread pool.

This can go as far as having many concurrent actors share a single pooled thread while avoiding some of the threading limitations of the JVM.

In general, while the JVM can only give you a limited number of threads (typically around a couple of thousands), the number of actors is only limited by the available memory. If an actor has no work to do, it doesn't consume any threads.

Actor code is processed in chunks separated by quiet periods of waiting for new events (messages). This can be naturally modeled through *continuations*.

As JVM doesn't support continuations directly, they have to be simulated in the actors frameworks, which has slight impact on organization of the actors' code. However, the benefits in most cases outweigh the difficulties.

---

*An Actors Sample*

```
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 the main thread to wait until both actors have terminated.
[master, player]*.join()
```

example by *Jordi Campos i Miralles, Departament de Matemàtica Aplicada i Anàlisi, MAiA Facultat de Matemàtiques, Universitat de Barcelona*

---

# Usage of Actors

**GPars** provides consistent Actor APIs and DSLs. Actors, in principal, perform three specific operations — send messages, receive messages and create new actors. Although not specifically enforced by **GPars**, messages should be immutable or at least follow the **hands-off** policy when the sender never touches the messages after the message has been sent off.

## Sending Messages

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

*A Sample*

```
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* methods is available to block the caller until a reply from the actor is available. The *reply* is returned from the *sendAndWait* method as a return value. The *sendAndWait* methods may also return after a timeout expires or in case of termination of the called actor.

*A Sample*

```
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 closure is waiting for a reply from the actor.

*A Sample*

```
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 the caller to continue its processing while the actor is handling the submitted message.

*A Sample*

```
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 non-active actor.

---

## Receiving Messages

### Non-blocking Message Retrieval

Calling the *react* method, optionally with a timeout parameter, from within the actor's code will consume the next message from the actor's inbox, potentially waiting, if there is no message to be processed immediately.

*A Sample*

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

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

To permit detaching actors from threads, the *react* method requires code to be written in a special **continuation style**.

*A react Sample*

```
Actors.actor {
    loop {
        println 'Waiting for a gift'
        react {gift ->
            if (mySpouse.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 no messages are available in their mailbox. Essentially, *react* schedules the supplied code (closure) to be executed upon next message arrival and returns. The closure supplied to the *react* methods is the code where the computation should resume. This is a **continuation style**.

Since actors have to preserve the guarantee that at most one thread is active within the actor's body, the next message cannot be handled before the current message processing finishes. Typically, there shouldn't be a need to put code after calls to *react*. Some actor implementations even enforce this. However, **GPars** does not - for performance reasons. The *loop* method allows iterations within the actor body. Unlike typical looping constructs, like *for* or *while* loops, *loop* cooperates with nested *react* blocks and will ensure looping across subsequent message retrievals.

## Sending Replies

The *reply* and *replyIfExists* methods are not only defined on the actors themselves, but for *AbstractPooledActor* (not available in *DefaultActor* , *DynamicDispatchActor* nor *ReactiveActor* classes) also on the processed messages themselves upon their reception, which is particularly handy when handling multiple messages in a single call. In such cases, *reply()* invoked on the actor will send a reply to authors of all the currently processed messages (the last one), whereas *reply()* called on messages sends a reply to the author of that particular message only.

See DemoMultiMessage.groovy in our sample demos here

### The Sender Property

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

*A Sample*

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

---

#### Forwarding

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

*A Sample*

```
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 to it
    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 to **react** to messages sent to them. The threads are returned to the pool once a message has been processed and the actor is idle waiting for some more messages to arrive.

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

*Actor Sample*

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

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

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

*A Sample*

```
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.

*A Sample*

```
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 taken for undelivered messages, at actor termination, all unprocessed messages from its queue have their *onDeliveryError()* method called. The *onDeliveryError()* method or closure defined on the message can, for example, send a notification back to the 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 be added both dynamically

*A Dynamic Sample*

```
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:

*A Static Sample*

```
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 accepting a timeout is also available. The **Groovy** *spread-dot* ( *. ) operator comes in handy when joining multiple actors at a time.

*A Sample to Join Actors*

```
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, optionally accompanied with a 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 of the received messages.

*A Sample*

```
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.

*A Sample*

```
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 an actor's *react{}* but not *loop()* .

### Fair Vs Non-fair Actor Behavior

*DefaultActor* can be set to behave in a fair or non-fair (default) manner. Depending on the strategy chosen, the actor either makes the thread available to other actors sharing the same parallel group (fair), or keeps the thread for itself until the message queue becomes empty (non-fair). 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 scheduler, use the appropriate constructor parameter when creating a parallel group (**PGroup** class). The supplied scheduler will orchestrate threads in the group's thread pool.

Please also see the numerous Actor sample demo programs.

# Actors Principles

Actors share a **pool** of threads, which are dynamically assigned to actors when the actors need to

**react** to messages sent to them. The threads are returned back to the pool once a message has been processed and the actor is idle waiting for some more messages to arrive. Actors become detached from the underlying threads and so a relatively small thread pool can serve potentially unlimited number of actors. Virtually unlimited scalability in number of actors is the main advantage of *event-based actors* , which are detached from the underlying physical threads.

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

*A Sample*

```
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 the first message.

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

*A Sample*

```
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

*A Sample*

```
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 function
```

---

## Creating An Asynchronous Service

*A Sample*

```
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 closure parameter. Inside the actor's body, you can use *loop()* to iterate, *react()* to receive messages and *reply()* to send a message to the actor, which has sent the currently processed message. The sender of the current message is also available through the actor's *sender* property. When the decryptor actor doesn't find a message in its message queue at the time when *react()* is called, the *react()* method gives up the thread and returns it back to the thread pool for other actors to pick it up.

Only after a new message arrives to the actor's message queue, the closure of the *react()* method is scheduled for processing with the pool. Event-based actors internally simulate continuations - actor's work - is split into sequentially run chunks, which are invoked once a message is available in the inbox. Each chunk for a single actor can be performed by a different thread from the thread pool.

**Groovy**'s flexible syntax with closures allows our library to offer multiple ways to define actors. For

instance, here's an example of an actor that waits for up to 30 seconds to receive a reply to its message. Actors allow time DSL, defined by org.codehaus.groovy.runtime.TimeCategory class, to be used for timeout specification to the *react()* method, provided the user wraps the call within a *TimeCategory* use block.

*A Sample*

```
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 instead. Also the *onTimeout()* handler is invoked, if present on the actor:

*An Actor Sample*

```
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 an actor's lifecycle notification methods (e.g. *onTimeout()* ) dynamically. Obviously, the lifecycle methods can be defined the usual way when you decide to define a new class for your actor.

*A Sample*

```
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. Under the covers the memory is synchronized each time a thread is assigned to an actor. Therefore, the actor's state **can be safely modified** by code in the body without any other extra (synchronization or locking) effort.

*A Sample*

```
class MyCounterActor extends DefaultActor {
    private Integer counter = 0

    protected void act() {
        loop {
            react {
                counter++
            }
        }
    }
}
```

Ideally, an actor's code should never be invoked directly from outside so all code of the actor class can only be executed by the thread handling the last received message. Therefore, all the actor's code is **implicitly thread-safe**. If any of the actor's methods are allowed to be called by other objects directly, the thread-safety guarantee for the actor's code and state is no longer valid.

### Simple Calculator

Here is a little bit more realistic example of an event-driven actor that receives two numeric messages, sums them up and sends the result to the console actor.

*A Calculator*

```
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_driven actors* need to split the code into independent chunks assignable to different threads sequentially and **continuations** are not natively supported on JVM, the chunks are created artificially. The *react()* method creates the next message handler. As soon as the current message handler finishes, the next message handler (continuation) is scheduled.

---

#### Concurrent Merge Sort Example

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

*A Sort Sample*

```
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 thread pool, no matter how many actors are created along the way.

---

### Actor Lifecycle Methods

Each Actor can define lifecycle observing methods, which will be called whenever a certain lifecycle event occurs.

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

You can either define the methods statically in your Actor class or add them dynamically to the actor's metaclass:

*An Actor Sample*

```
class MyActor extends DefaultActor {
    public void afterStart() {
        ...
    }
    public void onTimeout() {
        ...
    }

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

*Another Sample*

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

...
}
```

### Performance Tips

To help performance, you may consider using the *silentStart()* method instead of *start()* when starting a *DynamicDispatchActor* or a *ReactiveActor*. Calling *silentStart()* will by-pass some of the start-up machinery and as a result will also avoid calling the *afterStart()* method. Due to its 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 actor group available. And, just like the *Actors* abstract factory, can be used to create actors in the default group. Custom groups can be used as abstract factories to create new actors instances belonging to these groups.

*A Group Sample*

```
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. By default, it points to the default actor group, which is *Actors.defaultActorPGroup* , and can only be changed before the actor is started.

*A Sample*

```
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 pool, by default, contains n + 1 threads, where **n** stands for the number of **CPUs** detected by the JVM. The **pool size** can be set explicitly, either by setting the *gpars.poolsize* system property or, individually, for each actor group. This is by specifying the appropriate constructor parameter.

*A Sample*

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

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

*A Sample*

```
... (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 *NonDaemonPGroup* class can be used when non-daemon threads are required.

*A Sample*

```
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 groups*, you can split your actors to leverage multiple thread pools of different sizes and so assign resources to different components of your 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 actors, to preserve system resources.

---

#### The Default Actor Group

Actors that didn't have their *parallelGroup* property changed or that were created through any of the factory methods on the *Actors* class can share a common group *Actors.defaultActorPGroup* . This group uses a **resizeable thread pool** with an upper limit of **1000 threads** . This gives you the comfort of having the pool automatically adjust to the demand of the actors. On the other hand, with a growing number of actors, the pool may become too big an inefficient. It's advisable to group your actors into your own PGroups with fixed size thread pools for all but trivial applications.

#### 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 thread finishes. Calling *actor.join()* on any, some or all of your actors would block the main thread until the actor terminates and thus keep all your actors running.

Alternatively, use instances of *NonDaemonPGroup* and assign some of your actors to these groups.

*A Sample*

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

alternatively .A Sample

```
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 blocking actors. Blocking actors hold a single pooled thread for their whole life-time including the time when waiting for messages. They avoid some of the thread management overhead, since they never fight for threads after start, and also let you write straight code without the necessity of continuation style. Since they only do blocking, message reads are via the *receive* method. Obviously, the number of blocking actors running concurrently, is limited by the number of threads available in the shared pool. On the other hand, blocking actors typically provide better performance compared to continuation-style actors, especially when the actor's message queue rarely is empty.

*A Sample*

```
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 may, in particular, be good candidates for high-traffic positions in your actor network.

# Stateless Actors

## Dynamic Dispatch Actor

The *DynamicDispatchActor* class is an actor allowing for an alternative structure of the message handling code.

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

*A Dynamic Dispatch Actor Sample*

```
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 preserved for the actor, the dynamic dispatch code structure may be more intuitive than the traditional one using nested *loop* and *react* statements.

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

*A DynamicDispatchActor Sample*

```
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:

*A Combined Sample*

```
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* will take precedence over the static *onMessage* handlers.

### Fair or non-fair Behavior of DynamicDispatchActors

*DynamicDispatchActor* can be set to behave in a fair on non-fair (default) manner. Depending on the strategy chosen, the actor either makes the thread available to other actors sharing the same parallel group (fair), or keeps the thread fot itself until the message queue gets empty (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()* method.

*A Fair Sample*

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

---

## Static Dispatch Actor

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

*A Sample*

```
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 actor's declared type 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 helper factory methods:

*A Sample*

```
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()
```

When compared to the *DynamicDispatchActor*, the *StaticDispatchActor* class is limited to a single handler method.

This simplified creation without any **when** handlers, plus the considerable performance benefits, should make *StaticDispatchActor* your default choice for straightforward message handlers. Use this wnmb hen dispatching based on message run-time type is not necessary.

For example, *StaticDispatchActors* make dataflow operators four times faster than the *DynamicDispatchActor* .

---

## Reactive Actor

The *ReactiveActor* class, constructed typically by calling *Actors.reactor()* or *DefaultPGroup.reactor()*, allows ar more event-driven approach.

When a reactive actor receives a message, the supplied block of code, which makes up the reactive actor's body, is run with the message as a parameter. The result returned from the code is sent in reply.

*A Sample*

```
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 that submits a batch of numbers to a *ReactiveActor* for processing and then prints the results gradually as they arrive.

*A Sample*

```
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 messages in a loop, process them and send back the result. This is schematically how the reactive actor looks inside:

*A Sample*

```
public class ReactiveActor extends DefaultActor {
    Closure body

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

**Fair or Non-fair Behavior of ReactiveActors**

*ReactiveActor* can be set to behave in a fair or unfair (default) manner.

Depending on the strategy chosen, the actor either makes the thread available to other actors sharing the same parallel group (fair), or keeps the thread for itself until the message queue is empty (non-fair). Generally, non-fair actors perform 2–3 times better than fair ones.

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

*A Fair Sample*

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

# Tips and Tricks

## Structuring Actor's Code

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

*A Sample*

```
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 supplied message handlers to run as continuations of the current calculation in reaction to the next message arriving.

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

*A Sample*

```
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, please kepp in mind that calls to *react()* and *loop()* methods have slightly different semantics.

This becomes a bit of a challenge once you try to implement any types of loops in your actors. On the other hand, if you leverage the fact that *react()* only schedules a continuation and returns, you may call methods recursively without fear of stack overflow. Look at the examples below that use these three described techniques for structuring actor's code.

**A Subclass Of *DefaultActor***

*A Sample*

```
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

*A Sample*

```
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

*A Groovy Sample*

```
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 idea of using the actor's *loop()* method to create a loop that runs until the actor 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()
```

# Active Objects

Active objects provide an **OO** facade on top of actors. This allows you to avoid dealing directly with the actor machinery, having to match messages, wait for results and send replies. Ouch !

## Actors With a Friendly Facade

*A Sample*

```
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 instance is created for each instance of your class. Now you can mark methods with the *@ActiveMethod* annotation indicating that you want the method to be invoked asynchronously by the target object's internal actor. An optional boolean *blocking* parameter to the *@ActiveMethod* annotation specifies, whether the caller should block until a result is available or whether instead the caller should only receive a *promise* for a future result in a form of a *DataflowVariable* and so the caller is not blocked waiting.

### Blocking or Not ?

By default, all active methods are set to be **non-blocking** . However, methods that declare their return type explicitly, must be configured as blocking, otherwise the compiler will report an error. Only *def*, *void* and *DataflowVariable* are permissible return types for non-blocking methods.

Under the covers, **GPars** will translate your method call to **a message being sent to the internal actor**. The actor will eventually handle that message by invoking the desired method on behalf of the caller and once finished a reply will be 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 result, just like when doing normal plain method invocation. All we've achieved is being thread-safe inside the Active object from concurrent access. Something the *synchronized* keyword could give you as well. So it's the **non-blocking** methods that should drive your decision towards using active objects. Blocking methods will then provide the usual synchronous semantics yet give the consistency guarantees across concurrent method invocations. The blocking methods are then still very useful when used in combination with non-blocking ones.

*A Sample*

```
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**

Calling the non-blocking active method will return as soon as the actor has been sent a message. The caller is now allowed to do whatever it likes, while the actor is taking care of the calculation.

The state of the calculation can be polled using the *bound* property on the promise. Calling the *get()* method on the returned promise will block the caller until a value is available. The call to *get()* will eventually return a value or throw an exception, depending on the outcome of the actual calculation.

The *get()* method has a variant with a timeout parameter, to avoid the risk of waiting indefinitely.

#### Annotation Rules

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

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

### Inheritance

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

*An Annotated Sample*

```
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 *@ActiveObject* since it holds the *@ActiveMethod* annotation on method *fooC()* , while class *B* does not need the annotation, since none of its methods is active.

### Groups

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

*A Group Sample*

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

The *value* parameter to the *@ActiveObject* annotation specifies a name of parallel group to bind the internal actor to. Only threads from the specified group will be used to run internal actors of instances of the class.

The groups, however, need to be created and registered prior to creation of any of the active object instances belonging to that group. If not specified explicitly, an active object will use the default actor group - *Actors.defaultActorPGroup* .

*A Sample*

```
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 internal actor field. May you need to change the default name *internalActiveObjectActor* , use the *actorName* parameter to the *@ActiveObject* annotation.

*A Named Sample*

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

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

# Classic Examples

## A Few Examples of Actors Usage

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

---

## The Sieve of Eratosthenes

Problem description

*A Sample*

```
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) -> (filter by mod 7) -> (filter by mod 11) -> (caution! Primes
falling out here)
 * The chain is built (grows) on the fly, whenever a new prime is found.
 */

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 chain.
     * 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 and chain
     * 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 and stop after a reply
        } else {  //I am the last in the chain
            stop()
            reply 'Done'
        }
    }
}
```

## Sleeping Barber

Problem description

*A Sample*

```
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

*A Sample*

```
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 creates a given number of *WordSortActors* , splits among them the files to sort words in and collects the results.

Inspired by Scala Concurrency blog post by Michael Galpin

*A Sample, Sort-Of*

```
//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 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.

*A Load Balancer Sample*

```
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()
```

# User Guide To Agents

The **Agent** class is a thread-safe non-blocking shared mutable state wrapper implementation inspired by **Agents** in **Clojure**.

### Shared Mutable State can't be avoided

A lot of the concurrency problems disappear when you eliminate the need for *Shared Mutable State* with your architecture. Indeed, concepts like **actors**, **CSP** or **dataflow concurrency** avoid or isolate mutable state completely.

In some cases, however, sharing mutable data is either inevitable or makes the design more natural and understandable. For example, think of a shopping cart in a typical e-commerce application, when multiple AJAX requests may hit the 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 protect mutable data that need to be shared across threads. **Agents** hide the data and protect it from direct access. Clients can only send commands (functions) to the **agent**. The commands will be serialized and processed against the data one-by-one in turn.

With the commands being executed serially, the commands do not need to care about concurrency and can assume the data is all theirs when run. Although implemented differently, **GPars Agents**, called *Agent* , fundamentally behave like **actors**. They accept messages and process them asynchronously. The messages, however, must be commands (functions or **Groovy** closures) and will be executed inside the **agent**. After reception, the received function is run against the internal state of the **Agent** and the return value of the function is considered to be the new internal state of the **Agent**.

Essentially, **agents** safe-guard mutable values by allowing only a single *agent-managed thread* to make modifications to them. The mutable values are **not directly accessible** from outside, but instead *requests have to be sent to the agent* and the **agent** is guaranteed to process the requests sequentially on behalf of the callers. **Agents** guarantee 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 state is a more complex object we need more support.

---

# Concepts

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

An **Agent** wraps a reference to mutable state, held inside a single field, and accepts code (closures or commands) as messages, which can be sent to the **Agent** just like to any other actor using the **'<<'** operator, the *send()* methods or the *implicit call()* method.

At some point after reception of a closure / command, the closure is invoked against the internal mutable field and can make changes to it. The closure is guaranteed to be run without intervention from other threads and so may freely alter the internal state of the **Agent** held in the internal *data* field.

The whole update process is of the `fire-and-forget` type since, once the message (closure) is sent to the Agent, the caller thread can go off to do other things and come back later to check the current value with **Agent.val** or **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 *agent8's state.
- Replacing the state object with a new one is also possible and is done using the **updateValue()** method.
- 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 internal reference field.
- The *val* property of an **Agent** will wait until all preceding commands in the agent's queue are consumed and then 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 an **Agent** instance will 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 club. To add a new member, a message (command to add a member) has to be sent to the *clubMembers* Agent.

*A Sample* -

```
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 clubMembers = new Agent<List<String>>(['Me'])  //add Me

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

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

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

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

clubMembers.await()
```

## Shared Conference Counting Number of Registrations

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

*A Conference Sample -*

```
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.

*A Sample to Make an Agent Instance*

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

---

# Listeners and Validators

Agents allow the user to add listeners and validators. While listeners are notified each time the internal state changes, validators get a chance to reject or veto a coming change by throwing an exception.

*A Concrete Example -*

```
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 $newValue"}

counter.addValidator {oldValue, newValue -> if (oldValue > newValue) throw new IllegalArgumentException('Things can only go up in Groovy')}
counter.addValidator {agent, oldValue, newValue -> if (oldValue == newValue) throw new IllegalArgumentException('Things never stay the 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 thrown from the validators will be logged inside the **agent** and can be tested using the *hasErrors()* method or retrieved through the *errors* property.

*Testing for Errors Sample*

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

---

# Validator Gotchas

**Groovy** is not very strict on variable data types and immutability, so **agent** users should be aware of potential bumps on the road.

If the submitted code modifies the state directly, validators will not be able to un-do the change in case of a validation rule violation. There are two possible solutions available:

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

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 follows :

*A Validator Sample -*

```
//Create an agent storing names, rejecting 'Joe'
final Closure rejectJoeValidator = {oldValue, newValue -> if ('Joe' in newValue) throw
new IllegalArgumentException('Joe is not 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 state
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, which created them, and will share a thread pool. To create an **Agent** instance belonging to a group, call the *agent()* factory method on the group. This way you can organize and tune performance of agents.

*Create Groups Around a Thread Pools*

```
final def group = new NonDaemonPGroup(5)  //create a group around a thread pool
def clubMembers = group.agent(['Me'])  //add Me
```

### Custom Thread Pools for Agents

The default thread pool for **agents** contains daemon threads. Make sure that your custom 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*.

Alterntively, in case your thread pools use non-daemon threads, such as when using the **NonDaemonPGroup** group class, make sure you shutdown the group or the thread pool explicitly by calling its `shutdown()` method, otherwise your applications will [red]never exit.

## Direct Pool Replacement

Alternatively, by calling the *attachToThreadPool()* method on an **Agent** instance, a custom thread pool can be specified for it.

**attachToThreadPool()** *Example*

```
def clubMembers = new Agent<List<String>>(['Me'])  //add Me

final ExecutorService pool = Executors.newFixedThreadPool(10)
clubMembers.attachToThreadPool(new DefaultPool(pool))
```

Remember, like **actors**, a single **Agent** instance (aka agent) can never use more than one thread at a time

---

## The Shopping Cart Example

*A Sample -*

```
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 executing the same functionality directly

*And so Typically Sequential Code Like This*

```
public void addItem(String product, int quantity) {
    cartState[product]=quantity
}
```

*Becomes*

```
public void addItem(String product, int quantity) {
    cartState << {it[product] = quantity}
}
```

1. Public methods may send references to internal private methods or closures, which hold the desired functionality to perform the deed.

*A Public-to-Private Sample*

```
public void clearItems() {
    cartState << performClear
}

private Closure performClear = { it.clear() }
```

**Currying might be necessary**, if the closure takes other arguments besides the current internal state instance. See the *increaseQuantity* method.

---

## The Printer Service Example

Another example - suppose a not thread-safe printer service is shared by multiple threads. The printer needs to have the document and quality properties set before printing. Obviously we have a potential for race conditions if not guarded properly. Callers don't want to block until the printer is available, which the `fire-and-forget` nature of **actors** solves very elegantly.

*A Sample Printer Service*

```
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 the latest updates, see the respective Demos

## Reading The Value

To follow the **Clojure** philosophy closely, the **Agent** class gives reads higher priority than to writes. By using the *instantVal* property, your read request will bypass the incoming message queue of the **Agent** and returns the current snapshot of the internal state. The *val* property will wait in the message queue for processing, just like the non-blocking variant *valAsync(Clojure cl)* , which will invoke the provided closure with the internal state as a parameter.

You have to bear in mind that the *instantVal* property might return although correct, but randomly looking results, since the internal state of the **Agent** at the time of *instantVal* execution is non-deterministic and depends on the messages that have been processed before the thread scheduler executes the body of *instantVal* .

The *await()* method lets you wait for the processing of all the messages submitted to the **Agent** before and so may block the calling thread.

## State Copy Strategy

To avoid leaking the internal state, the **Agent** class can specify a `copy strategy` as the second constructor argument. With the `copy strategy` specified, the internal state is processed by the `copy strategy` closure and the output value of the `copy strategy` value is returned to the caller instead of the actual internal state. This applies to *instantVal*, *val* as well as to *valAsync()* .

## Error Handling

Exceptions thrown from within the submitted commands are stored inside the **agent** and can be obtained from the *errors* property. The property is cleared once read.

*A Sample of Error Handling*

```
def clubMembers = new Agent<List>()
assert clubMembers.errors.empty

    clubMembers.send {throw new IllegalStateException('test1')}
    clubMembers.send {throw new IllegalArgumentException('test2')}
    clubMembers.await()

    List errors = clubMembers.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 clubMembers.errors.empty
```

## Fair and Non-fair Agents

**Agents** can be either fair or non-fair. Fair **agents** give up the thread after processing each message, unfair **agents** keep a thread until their message queue is empty. As a result, non-fair **agents** tend to perform better than fair ones.

The default setting for all **Agent** instances is to be **non-fair**, however by calling its *makeFair()* method the instance can be made fair.

*A Sample To Make It Fair*

```
def clubMembers = new Agent<List>(['Me'])  //add Me
clubMembers.makeFair()
```

# User Guide to Dataflow

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

## Introduction

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

*A Simple Sample*

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

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

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

task {
    x << 10
}

task {
    y << 5
}

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

The same algorithm rewritten using the *Dataflows* class looks like this :

*A Dataflows Sample*

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

final def df = new Dataflows()

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

task {
    df.x = 10
}

task {
    df.y = 5
}

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

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

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

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

---

# Implementation Detail

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

### Re-binding Is Possible

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

# Benefits

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

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

This doesn't sound bad, does it?

---

# Concepts

## Dataflow Programming

### Quoting Wikipedia

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

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

# Principles

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

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

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

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

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

# Dataflow Queues and Broadcasts

Before you check our samples of **Dataflow Variables**, **Tasks** and **Operators**, you should learn a bit about streams and queues to have a full picture of `Dataflow Concurrency`. Except for `dataflow variables`, there are also the concepts of *DataflowQueues* and *DataflowBroadcast* that you can leverage in your code.

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

*A Producer-Consumer Demo*

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

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

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

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

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

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

### DataflowChannels Combine Two Interfaces

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

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

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

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

---

## Point-to-point Communication

The *DataflowQueue* class can be viewed as a point-to-point (1 to 1, many to 1) communication channel. It allows one or more producers send messages to one reader. If multiple readers read from the same *DataflowQueue* , they will each consume different messages.

Or to put it a different way, each message is consumed by exactly one reader. You can easily imagine a simple load-balancing scheme built around a shared *DataflowQueue* with readers being added dynamically when the consumer part of your algorithm needs to scale up. This is also a useful default choice when connecting tasks or operators.

## Publish-subscribe Communication

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

*A Pub-Sub Sample*

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

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

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

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

---

# DataflowStream

The *DataflowStream* class represents a deterministic dataflow channel. It's built around the concept of a functional queue and so provides a lock-free thread-safe implementation for message passing.

Essentially, you may think of *DataflowStream* mechanisms as a 1-to-many communication channel, since when a reader consumes a messages, other readers will still be able to read the same message. Also, all messages arrive to all readers in the same order.

Since the *DataflowStream* is implemented as a functional queue, its API requires users to traverse the values in the stream themselves. On the other hand, *DataflowStream* offers handy methods for value filtering or transformation together with interesting performance characteristics.

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

*A Sample of DataflowStream Usage*

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

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

/**
 * We need a resizeable thread pool, since tasks consume threads while waiting,
```

```
blocked for values from the DataflowQueue.val
 */
group = new DefaultPGroup(new ResizeablePool(true))

final int requestedPrimeNumberCount = 100

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

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

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

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

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

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

## DataflowStream Adapters

The *DataflowStream* API as well as the semantics of its use are very different from the one defined

by *Dataflow(Read/Write)Channel*. Adapters have to be used in order to allow *DataflowStreams* to work with other dataflow elements. The *DataflowStreamReadAdapter* class will wrap a *DataflowStream* with the necessary methods to read values, while the *DataflowStreamWriteAdapter* class provides write methods around the wrapped *DataflowStream* method.

## Thread Safety

It's important to mention that the *DataflowStreamWriteAdapter* is thread safe. It allows multiple threads to add values to the wrapped *DataflowStream* through the adapter. On the other hand, the *DataflowStreamReadAdapter* is designed to be used by a single thread.

The *DataflowStreamWriteAdapter* is thread safe

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

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

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

*DataflowStreamAdapters Sample*

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

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

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

def result = new DataflowQueue()

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

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

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

*A DataflowStream Sample*

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

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

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

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

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

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

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

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

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

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

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

# Bind Handlers

*What A Bind*

```
def a = new DataflowVariable()

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

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

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

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

*A DataflowQueue().wheneverBound Sample*

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

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

*A `wheneverBound` Sample*

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

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

### Parallel Speculations Anyone ?

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

# Bind Handlers Grouping

When you need to wait for multiple `DataflowVariables Promises` to be bound, we can benefit from

calling the *whenAllBound()* function. It's available on the *Dataflow* class as well as on *PGroup* instances.

*whenAllBound() Sample*

```
    final group = new NonDaemonPGroup()

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

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

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

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

*whenAllBound() Sample*

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

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

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

# Bind Handler Chaining

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

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

*A Pointless Sample - No Need To Join The Dots !*

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

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

*This could be nicely combined with Asynchronous functions*

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

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

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

Thread.start {variable << 4}

assert 9 == result.val
```

*or ActiveObjects*

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

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

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

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

assert 9 == result.val
```

### Motivation for Chaining Promises

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

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

*An Unproductive Sample*

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

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

*An Unnecessarily Complex Nested Sample*

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

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

*A whenBound() Sample Plus a then() Example*

```
final DataflowVariable variable = new DataflowVariable()

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

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

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

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

*Chaining Promises solves both of these issues elegantly:*

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

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

*A Chaining Sample*

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

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

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

Thread.start {variable << 4}

assert 9 == result.val
```

## Error Handling for Promise Chaining

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

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

*A Sample of Error Handling*

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

initial << 0

println result.get()
```

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

*Re-throwing Potential Exceptions*

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

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

## Where Do You Want This Exception ?

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

*A Exceptional Sample*

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

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

### Handling Concrete Exception Types

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

*A Specific Exception Handling Example*

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

### Customer-site Exception Handling

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

*A Delayed Exception Handling Example*

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

## Putting It All Together

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

*A Large Asynchronous Sample*

```
withPool {

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

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

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

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

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

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

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

### Implementing the Fork/join Pattern With Promises

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

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

Let's see how this fits into the picture:

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

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

*A Sample of Multiple Activities*

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

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

*A thenForkAndJoin() Sample*

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

---

# Lazy Dataflow Tasks and Variables

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

*A Lazy Sample*

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

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

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

---

## A Bigger Example

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

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

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

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

*A Diamond Sample*

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

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

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

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

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

## Making Tasks Lazy

The *lazyTask()* method is available alongside the *task()* method to give the us a task-oriented abstraction for delayed activities. A **Lazy Task** returns an instance of a *LazyDataflowVariable* (like a **Promise** ) with the initializer set by the provided closure. As soon as someone asks for the value, the task will start asynchronously and eventually deliver a value into the *LazyDataflowVariable* .

*A Lazy Sample*

```
import groovyx.gpars.dataflow.Dataflow

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

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

# Dataflow Expressions

Look at the magic below:

*A Dataflow Sample*

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

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

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

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

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

*A* **DataflowVariable** *Sample*

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

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

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

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

task {
    searchPhrase << '2nd'
}

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

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

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

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

task {
    searchPhrase << '2nd'
}

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

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

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

## Bind Error Notification

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

*Reporting Bind Operation FailED*

```
        final DataflowVariable variable = new DataflowVariable()

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

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

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

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

        })
```

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

## Further Reading

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

# Tasks

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

Check out the examples.

---

## A Simple Mashup Example

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

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


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

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

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

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

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

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

## Grouping Tasks

Dataflow tasks can be organized into groups for performance fine-tuning. Groups provide a handy *task()* factory method to create tasks attached to these groups. Using groups allows us to organize tasks or operators around different thread pools (wrapped inside the group). While the

Dataflow.task() command schedules the task on a default thread pool `(java.util.concurrent.Executor, fixed size=#cpu+1, daemon threads)`, we might prefer defining our own thread pool(s) to run these tasks.

*A Personal Thread Pool Sample*

```
import groovyx.gpars.group.DefaultPGroup

def group = new DefaultPGroup()

group.with {
    task {
        ...
    }

    task {
        ...
    }
}
```

### Custom Thread Pools for Dataflow

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

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

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

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

*A Sample*

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

You can always override the default group by being specific:

*A Sample*

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

---

## A Mashup Variant With Methods

To avoid giving you wrong impression about structuring the Dataflow code, here's a rewrite of the mashup example, with a *downloadPage()* method performing the actual download in a separate task. It returns a *DataflowVariable* instance, so that the main application thread could eventually get hold of the downloaded content.

Dataflow variables can obviously be passed around as parameters or return values.

*A Sample*

```
package groovyx.gpars.samples.dataflow

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


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

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

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

## A Physical Calculation Example

Dataflow programs naturally scale with the number of processors. Up to a certain level, the more processors you have the faster the program runs. Check out, for example, the following script, which calculates parameters of a simple physical experiment and prints out the results.

Each task performs its part of the calculation and may depend on values calculated by some other tasks and its' results might be needed by some of the other tasks. With `Dataflow Concurrency` you can split the work between tasks or reorder the tasks themselves as you like and the dataflow mechanics will ensure the calculation is accomplished correctly.

*A DataflowVariable Sample*

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

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

def t = task {
    println """

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

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

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

    System.exit 0
}

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

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

task {
    radius << 2.5
    density <<  998.2071  //water
    acceleration << 9.80665 //free fall
```

```
    decelerationForce << 900
}

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

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

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

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

t.join()
```

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

## Deterministic Deadlocks

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

*A Deterministic Deadlock Sample*

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

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

# Dataflows Map

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

*A Convenience Example*

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

assert df.x == 'value1'

Dataflow.task {df.y = 'value2}

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

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

---

### Mixing *Dataflows* and Groovy *with* blocks

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

*A with block Makes Coding Easier*

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

    Dataflow.task {y = 'value2}

    assert y == 'value2'
}
```

---

# Returning Values From A Task

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

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

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

def results = [t1, t2]*.val

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

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

*A Sample*

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

# Joining Tasks

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

*A Bloacking Sample*

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

# Selects

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

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

*Selecting A Value From Multiple Channels*

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

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

task {
    sleep 3000
    a << 10
}

task {
    sleep 1000
    b << 20
}

task {
    sleep 5000
    c << 30
}

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

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

### A *select()* Method Returns What ?

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

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

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

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

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

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

*Random And Priority Seletions*

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

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

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

*Another Way ?*

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

*A Sample*

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

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

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

---

# Guards

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

*A Useful Filter Tool*

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

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

*Guards Enable/Disable Channels During Value Selection*

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

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

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

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

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

t.join()
```

# Priority Select

When certain channels should have precedence over others when selecting, the **prioritySelect** methods should be used instead.

*A* **prioritySelect** *Sample*

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

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

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

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

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

println 'Starting to monitor our IT department'

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

# Collecting Results of Asynchronous Computations

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

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

*How To Pick The Fastest Result*

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

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

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

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

---

# Timeouts

The *Select.createTimeout()* method will create a **DataflowVariable** bound to a value after a declared time period. This can be leveraged in *Selects* so that they unblock(resume processing) after a desired delay, if none of the other channels delivers a value before that time. Simply pass a

**timeout channel** as another input channel to the *Select* .

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

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

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

    final timeoutChannel = Select.createTimeout(500)

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

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

## Cancellations

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

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

*A Sample To Pick The Fastest Answer And Cancel The Others*

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

import java.util.concurrent.atomic.AtomicBoolean

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

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

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

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

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

    println "Result: " + result
}
```

# Operators

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

# Concepts

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

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

*An Operator Sample*

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

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

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

        def content = cache[request.site]

        if (content) {
            println "[Cache] Found in cache"
            bindOutput 1, [site: request.site, word:request.word, content: content]
        } else {
            def downloads = pendingDownloads[request.site]
            if (downloads != null) {
                println "[Cache] Awaiting download"
                downloads << request
            } else {
                pendingDownloads[request.site] = []
                println "[Cache] Asking for download"
                bindOutput 0, request
            }
        }

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

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

        def downloads = pendingDownloads[request.site]

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

### Exception Handling Explained

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

*Exception Handling Sample*

```
def listener = new DataflowEventAdapter() {

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

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

---

## Types of Operators

There are specialized versions of operator methods for specific purposes:

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

### Wiring Operators Together

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

*Using Several Operators*

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

You may alternatively refer to output channels through operators themselves:

*A More Complex Operator Example*

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

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

## Grouping Operators

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

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

```
import groovyx.gpars.group.DefaultPGroup

def group = new DefaultPGroup()

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

### Custom Thread Pools For Dataflow

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

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

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

*A Sample*

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

You can always override the default group by being specific:

*A Sample*

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

# Constructing Operators

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

*A Sample*

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

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

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

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

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

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

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

# Holding State in Operators

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

*A Sample*

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

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

*A Sample*

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

## Parallelize Operators

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

*A Sample*

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

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

### Thread Starvation

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

*A Sample*

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

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

## Synchronizing Outputs

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

*A Sample*

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

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

*A Sample*

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

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

*A Sample*

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

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

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

To set values of all the operator’s output channels in one atomic step, you may also consider calling

either the *bindAllOutputsAtomically* method, passing in a single value to write to all output channels or the *bindAllOutputsAtomically* method, which takes a multiple values, each of which will be written to the output channel with the same position index.

*A Sample*

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

### Which Bind Do I Use ?

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

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

## Operator Lifecycle

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

*A Sample*

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

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

    /**
```

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

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

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

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

    /**
     * Invoked when all messages required to trigger the operator become available in
```

```
the input channels.
     *
     * @param processor The reporting dataflow operator/selector
     * @param messages  The incoming messages
     * @return The original list of messages or a modified/new list of messages that
should be used instead
     */
    List<Object> beforeRun(DataflowProcessor processor, List<Object> messages);

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

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

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

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

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

*A Sample*

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

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

# Selectors

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

*A Sample*

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

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

*A Sample*

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

## Priority Selector

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

*A Sample*

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

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

---

## Join Selector

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

*A Sample*

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

---

## Internal Parallelism

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

*A Sample*

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

---

## Guards

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

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

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

def instruction
def nums = []

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

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

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

### Warning

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

## Shutting Down Dataflow Networks

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

Dataflow operators and selectors can be terminated in three ways:

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

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

### Shutting down the thread pool

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

## Emergency Shutdown

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

*A Sample*

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

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

*A Sample*

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

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

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

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

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

### Stopping Operators Gently

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

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

#### Detecting shutdown

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

*A Sample*

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

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

# PoisonPill

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

*A Sample*

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

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

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

a << PoisonPill.instance  //Send the poisson

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

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

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

### Operators and Selectors Should Only Terminate Themselves

Given the potential variety of operator networks and their asynchronous nature, a good termination strategy is that operators and selectors should only ever terminate themselves.

All ways of terminating them from outside (either by calling the *terminate()* method or by sending poison down the stream) may result in messages being lost somewhere in the pipes. This happens when the reading operators terminate before they fully handle the messages waiting in their input channels.

# Immediate Poison Pill

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

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

*A Sample*

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

a << PoisonPill.immediateInstance

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

---

# Poison With Counting

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

*A Sample*

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

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

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

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

*A Sample*

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

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

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

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

*A Sample*

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

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

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

---

# Poison Strategies

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

### Load-balancing Prevents Poison Shutdown

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

## Termination Tips and Tricks

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

*A Sample*

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


def group = new NonDaemonPGroup()

final DataflowQueue q = new DataflowQueue()

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

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

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

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

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

---

## Keeping PoisonPill Inside a Given Network

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

*A Sample*

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

The *Pipeline* DSL may be also helpful here:

*A Sample*

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

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

## Graceful Shutdown

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

*A Graceful Sample*

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

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

final monitor = new GracefulShutdownMonitor(100);

def op1 = group.operator(inputs: [a, b], outputs: [c], listeners: [new
GracefulShutdownListener(monitor)]) {x, y ->
```

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

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

final shutdownPromise = monitor.shutdownNetwork()

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

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

group.shutdown()
```

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

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

The *shutdownNetwork()* method returns a **Promise** so that you can do the usual set of tricks with it - block waiting for the network to terminate using the *get()* method, register a callback using the *whenBound()* method or make it trigger a whole set of activities through the *then()* method.

### Limitations of Graceful Sshutdown

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

# Application Frameworks

`Dataflow Operators` and `Selectors1 can be successfully used to build high-level domain-specific frameworks for problems that naturally fit the flow model.

## Building Flow Frameworks on Top of GPars Dataflow

**GPars** dataflow can be viewed as bottom-line language-level infrastructure.

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

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

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

For example, a network solving data-mining problems may consist of several data sources, data cleaning nodes, categorization nodes, reporting nodes and others. Image processing networks, on the other hand, may need nodes specialized in image compression and format transformation. Similarly, networks for data encryption, mp3 encoding, work-flow management as well as many

other domains that would benefit from dataflow-based solutions. These will differ in many aspects - the type of nodes in the network, the type and frequency of events, the load-balancing scheme, potential constraints on branching, the need for visualization, debugging and logging, the way users define the networks and interact with them as well as many others.

The higher-level application-specific frameworks should put effort into providing abstractions best suited for the given domain and hide **GPars** complexities.

For example, the visual graph of the network that the user manipulates on the screen should typically not show all the channels that participate in the network. Debugging or logging channels, which rarely contribute to the core of the solution, are among the first good candidates to consider for exclusion. Also channels and lifecycle-event listeners, which orchestrate aspects such as load balancing or graceful shutdown, will probably be not exposed to the user, although they will be part of the generated and executed network.

Similarly, a single channel in the domain-specific model will in reality translate into multiple channels perhaps with one or more logging/transforming/filtering operators connecting them together. The function associated with a node will most likely be wrapped with some additional infrastructural code to form the operator's body.

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

---

# Pipeline DSL

## A DSL for Building Operators Pipelines

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

*A Sample*

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

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

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

println encrypted.val
println encrypted.val
```

This saves you from directly creating, wiring and manipulating all the channels and operators that are to form the pipeline. The *pipe* operator lets you hook an output of one

function/operator/process to the input of another one. Just like chaining system processes on the command line.

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

*A Sample*

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

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

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

println encrypted.val
println encrypted.val
```

## Combining Pipelines with Straight Operators

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

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

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

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

fork1.chainWith save  //Hook in the save operation

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

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

println decrypted.val
println decrypted.val
```

### Type of Channel Preservation

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

*A Sample*

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

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

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

---

## Joining Pipelines

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

*A Sample*

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

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

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

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

## Forking the Dataflow

When a need comes to copy the output of a pipeline/channel into more than one following pipeline/channel, the *split()* method will help you:

*A Sample*

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

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

## Tapping into a Pipeline

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

*A Sample*

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

## Merging Channels

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

*A Sample*

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

## Separation

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

*A Sample*

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

## Choices

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

*A Sample*

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

---

## Filtering

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

*A Sample*

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

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

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

---

## Null Values

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

*A Sample*

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

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

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

## Customizing Thread Pools

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

*A Sample*

```
channel | {it * 2}

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

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

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

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

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

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

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

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

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

# Overriding the Default PGroup

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

*A Sample*

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

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

# The Pipeline Builder

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

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

*A Sample*

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

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

final negate = {-it}

final Pipeline pipeline = new Pipeline(pool, queue)

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

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

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

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

pool.shutdown()
```

# Passing Construction Parameters Through the Pipeline DSL

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

*A Sample*

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

# Implementation

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

## Combining Actors and `Dataflow Concurrency`

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

*A Sample*

```
final DataflowVariable a = new DataflowVariable()

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

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

fakingDoubler << 10
```

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

---

# Using Plain Java Threads

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

*A DataflowVariable Sample*

```
import groovyx.gpars.dataflow.DataflowVariable

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

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

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

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

---

# Synchronous Variables and Channels

When using asynchronous dataflow channels, apart from the fact that readers have to wait for a value to be available for consumption, the communicating parties remain completely independent. Writers don't wait for their messages to get consumed. Readers obtain values immediately as they come and ask. Synchronous channels, on the other hand, can synchronize writers with the readers as well as multiple readers among themselves.

This is particularly useful when you need to increase the level of determinism.

The writer-to-reader partial ordering imposed by asynchronous communication is complemented with reader-to-writer partial ordering, when using synchronous communication. In other words, you are guaranteed that whatever the reader did before reading a value from a synchronous channel preceded whatever the writer did after writing the value. Also, with synchronous communication writers can never get too far ahead of readers, which simplifies reasoning about the system and reduces the need to manage data production speed in order to avoid system overload.

## Synchronous Dataflow Queue

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

Synchronous channels block both the writer and the readers until all parties are ready

*A Synchronous Channel Sample*

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

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

def group = new NonDaemonPGroup()

final SyncDataflowQueue channel = new SyncDataflowQueue()

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

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

consumer.join()

group.shutdown()
```

## Synchronous Dataflow Broadcast

The *SyncDataflowBroadcast* class should be used for publish-subscribe (1:n or n:m) communication.

Each message written to the broadcast will be consumed by all subscribed readers. Writers are blocked until their message is consumed by all readers, readers are blocked until there's a value

available for them to read and all the other subscribed readers ask for the message as well. With *SyncDataflowBroadcast*, you get all readers processing the same message at the same time and waiting for one-another before getting the next one.

*A SyncDataflowBroadcast Sample*

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

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

def group = new NonDaemonPGroup()

final SyncDataflowBroadcast channel = new SyncDataflowBroadcast()

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

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

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

[fastConsumer, slowConsumer]*.join()

group.shutdown()
```

# Synchronous Dataflow Variable

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

*A Sample*

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

final NonDaemonPGroup group = new NonDaemonPGroup()

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

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

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

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

[reader, slowReader]*.join()

group.shutdown()
```

# Kanban Flow

*Table 1. Kanban Flow API Links*

| API Link | API Link | API Link | API Link |
|---|---|---|---|
| Kanban Flow | Kanban Link | Kanban Tray | Processing Node |

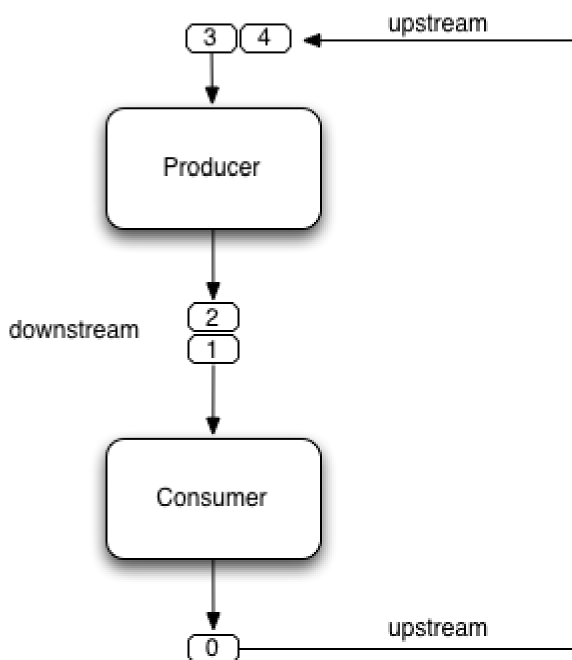
# KanbanFlow Description

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

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

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

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




---

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

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

*A Kanban Sample*

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

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

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

To put a product into a tray and send the tray downstream, one can either use the **send()** method, the << operator, or use the tray as a method object. The following lines are equivalent:

*A Sample of Equivalent Choices*

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

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

You should call **take()** only once!

If you prefer not to use an empty tray to send products downstream (as typically the case when a *ProcessingNode* acts as a filter), you must release the tray in order to keep it in play. Otherwise, the number of trays in the system decreases.

You can release a tray either by calling the **release()** method or by using the ~ operator (think "shake it off"). The following lines are equivalent:

*More Equivalent Choices*

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

**Tray Release**

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

### Various Linking Structures

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

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

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

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

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

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

---

### Composing KanbanFlows

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

*A KanbanFlow Sample*

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

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

flow = firstFlow + secondFlow

flow.start()
```

---

### Customizing Concurrency Characteristics

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

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

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

*Setting Tray Counts*

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

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

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

### Tests

- Kanban Flow Test in **Groovy**

### Kanban Code Samples in Groovy

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

# Classic Examples

## The Sieve of Eratosthenes Implementation using `Dataflow Tasks`

*A Sample Solution of The Sieve of Eratosthenes*

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

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

final int requestedPrimeNumberCount = 1000

final DataflowQueue initialChannel = new DataflowQueue()

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

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

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

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

#### The Sieve of Eratosthenes using both `Dataflow Tasks` and `Operators`

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

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

final int requestedPrimeNumberCount = 100

final DataflowQueue initialChannel = new DataflowQueue()

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

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

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

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

# User Guide To STM

## *Software Transactional Memory*

`Software Transactional Memory` (**STM**) gives developers transactional semantics for accessing in-memory data. This is similar to database concepts.

When multiple threads share data in memory, by marking blocks of code as transactional (atomic), the developer delegates the responsibility for data consistency to the **STM** engine. **GPars** leverages the Multiverse **STM** engine.

# Running A Piece of Code Atomically

When using **STM**, developers organize their code into transactions. A transaction is a piece of code, which is executed **atomically** - either **1)** all the code is run or **2)** none at all.

The data used by the transactional code remains **consistent** irrespective of whether the transaction finishes normally or abruptly. While running inside a transaction, the code is given an illusion of being **isolated** from other concurrently running transactions so that changes to data in one transaction are not visible in the other ones until the transactions commit. This gives us the **ACI** part of the **ACID** characteristics of database transactions. The **durability** transactional aspect so typical for databases, is not typically mandated for **Stm**.

**GPars** allows developers to specify transaction boundaries by using the *atomic* closures.

*Sample of* **ACI** *Transaction Boundaries*

```
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 a 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 an optimistic locking strategy and automatically rolls back and retries colliding transactions.

Developers should refrain from irreversible actions (e.g. writing to the console, sending e-mails, launching a missile, etc.) in their transactional code. To increase flexibility, the default **Multiverse** settings can be customized through custom *atomic blocks* .

## Customizing the Transactional Properties

Frequently it's desirable to specify different values for some of the transaction properties (e.g. read-only transactions, locking strategy, isolation level, etc.). The *createAtomicBlock* method will create a new *AtomicBlock* configured with the supplied values:

*Create an* **AtomicBlock** *with Custom Parameters*

```
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 using the specified settings.

*AtomicBlock* instances are thread-safe and can be freely reused among threads and transactions

## Using the *Transaction* Object

Atomic closures use the current *Transaction* as a parameter. The *Txn* object handle for a transaction can be used to manually control the transaction. This is illustrated in the example below, where we use the *retry()* method to block the current transaction until the counter reaches the desired value:

*A Sample*

```
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 previous examples that we use dedicated data structures to hold values. The fact is that normal **Java** classes do not support transactions and thus cannot be used directly, since **Multiverse** would not be able to share them safely among concurrent transactions, commit them nor roll them back.

normal **Java** classes do not support transactions

We need to use data that knows 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 was already available on the **Multiverse** website.

Unfortunately with the closure of Codehaus, that website is longer available. You may try to gather more information from the Multiverse source code.

As we are unclear about the future of the **Multiverse** project, we will consider using a different **STM** implementation in a future **GPars 2.0**.

# User Guide To GAE

## Google App Engine Integration

**GPars** can be run on the Google App Engine **(GAE)**. It can be made part of **Groovy** and **Java** *GAE applications* as well as a plugged into **Gaelyk**.

The Google App Engine is known as **GAE**

The small GPars App Engine integration library provides all the necessary infrastructure to hook **GAE** services into **GPars**. Although you'll be running on **GAE** threads and leveraging **GAE** timer services, the high-level abstractions remain the same. With a few restrictions you can still use **GPars** *actors, dataflow, agents, parallel collections* and other handy concepts.

Please refer to the GPars App Engine library for details on how to proceed with **GPars** on **GAE**.

# User Guide To Remoting

Concepts like *Actors*, *Dataflows* and *Agents* are not restricted just to a single VM. They provide an abstraction layer for concurrent programming that allows us to separate logic from low level synchronization code. These concepts can be easly extend to multiple nodes in a network.

The following notes describe **Remoting** in **GPars**.

Remoting for **GPars** was a *Google Summer of Code 2014* project.

## Introduction

To use *Actors*, *Dataflows* or *Agent* remotely, a new remote proxy object was introduced with the *Remote* prefix.

The proxy object usually has an identical interface to it's local counterpart. This allows us to use it in place of local counterpart. Under the covers, a proxy object just sends messages over the wire to an original instance.

To transport messages across the network, the Netty library was used.

To create a proxy-object, the instance serialization mechanism was used (more in **remote-serialization** below).

The general approach to using remotes is as follows (details below):

At *host A*:

1. Create remoting context and start a server to handle incoming requests.
2. Publish an instance under a specified *name*

At *host B*:

1. Create remoting context
2. Ask for an instance with specified *name* from *hostA:port*. A promise object is returned.
3. Get a proxy object from the promise.

At this point, a new connection is created for each request

## Remote Serialization

The following mechanism was used to create proxy objects:

*object* ←(serialization)→ *handle* ---- [network] ---- *handle* ←(serialization)→ *proxy-object*

One of the main advantages of this mechanism is that sending proxy-object references back is deserialized back to the original instance.

As all messages are seralized before sending over a wire, they must implement the *Serializable* interface.

This is a consequence of using built-in **Java** serialization mechanism and **Netty** *ObjectDecoder/ObjectEncoder*. On the other hand, it gives us the flexibility to send any custom object as a message to an **Actor** or to use **DataflowVariable**(s) of any type.

## Dataflows

In order to use remoting for *Dataflows*, a context (*RemoteDataflows* class) has to be created. Within this context, *Dataflows* can be published and retrieved from remote hosts.

*A Sample*

```
def remoteDataflows = RemoteDataflows.create()
```

In all subsections we assume that a context has already been created as shown above.

After creating a context, if we want to allow other hosts to retrieve published *Dataflows*, we need to start a server. We need to provide an address and port to listen on (say, like: *localhost*:11222, or 10.0.0.123:11333).

*Start A Remote Server*

```
remoteDataflows.startServer HOST PORT
```

To stop the server, we have a *stopServer()* method. Note that both *start* and *stop* methods are asynchronous, and they don't block; the server is started/stopped in the background.

Multiple execution of these methods or executing them in wrong order will raise an exception.

To only retrieve instances from remote hosts, starting a server is not necessary.

# DataflowVariable

The **DataflowVariable** is a core part of *Dataflows* subsystem that gains remoting abilities. Other structures(?) and subsystems depend on it.

Publishing a variable within context is done simply by:

*Publishing a Context*

```
def variable = new DataflowVariable()
remoteDataflows.publish variable "my-first-variable"
```

This registers the variable under a given name, so when a request for a variable with name *my-first-variable* arrives, the variable can be sent to the remote host.

It's important to remember that publishing another variable under the same name, will override the provious one and subsequent requests will send the newly published one.

Variable retrieval is done by:

*Variable Retrieval*

```
def remoteVariablePromise = remoteDataflows.getVariable HOST, PORT, "my-first-
variable"
def remoteVariable = remoteVariablePromise.get()
```

The *getVariable* method is non-blocking and returns a promise object that will eventually hold a proxy object to that variable. This proxy has the same interface as a **DataflowVariable** and can be used seemlessly as a regular variable.

To explore a full example see our: *groovyx.gpars.samples.remote.dataflow.variable* code

---

# DataflowBroadcast

It's possible to subscribe to a **DataflowBroadcast** on a remote host. To do this, we had to have published it first (assuming the context already exists):

*A DataflowBroadcast Sample*

```
def stream = new DataflowBroadcast()
remoteDataflows.publish stream "my-first-broadcast"
```

Then on the other host, it can be retrieved:

*A Retrieval Sample*

```
def readChannelPromise = remoteDataflows.getReadChannel HOST, PORT, "my-first-
broadcast"
def readChannel = readChannelPromise.get()
```

The proxy object has the same interface as a **ReadChannel** and can be used in the same fashion as a **ReadChannel** of a regular **DataflowBroadcast**.

To explore a full example, please see: *groovyx.gpars.samples.remote.dataflow.broadcast*

# DataflowQueue

The **DataflowQueue** feature received similar functionality, and is published like this :

*A Publish Sample*

```
def queue = new DataflowQueue()
remoteDataflows.publish queue, "my-first-queue"
```

and in similar way, we can retrieved it from the remote host:

*Retrieval from Remote Sources*

```
def queuePromise = remoteDataflows.getQueue HOST, PORT, "my-first-queue"
def queue = queuePromise.get()
```

New items can be pushed into the queue of the remote proxy. Such elements are sent over-the-wire to the original instance and pushed into it.

Retrieval commands send a request for an element to the **original** instance.

Conceptually, the remote proxy is an *interface* - it just sends requests to an original instance.

To explore a full example see: *groovyx.gpars.samples.remote.dataflow.queue* or *groovyx.gpars.samples.remote.dataflow.queuebalancer*

# Actors

The `Remote Actors` subsystem is designed in a similar way.

To start a *RemoteActors* class, a context must be created. Then within this context, an *Actors* instance can be published or retrieved from a remote host.

*Remote Creation*

```
def remoteActors = RemoteActors.create()
```

*Publishing :*

```
def actor = ...
remoteActors.publish actor, "actor-name"
```

*Retrieval :*

```
def actorPromise = remoteActors.get HOST, PORT, "actor-name"
def remoteActor = actorPromise.get()
```

It’s possible to join a remote **Actor**, but this will block until the original **Actor** ends its work. Sending replies and the *sendAndWait* method are supported as well.

One can send any object as a message to an **Actor**, but keep in mind it has to be **Serializable**.

See example: *groovyx.gpars.samples.remote.actor*

---

# Remote Actor Names

A *RemoteActors* class context may be identified by a name. To create one with a name use:

*Create A Named Context*

```
def remoteActors = RemoteActors.create "test-group-1"
```

*Actors* published within this context may be accessed by providing a special **Actor** URL.

For example: publishing an **actor** under the name  of actor within this context makes it accessible under the URL "test-group-1/actor".

*A Named Sample*

```
def actor = remoteActors.get "test-group-1/actor"
```

The host and port of an instance holding this actor is determined automatically.

Invoking the *get* method will send a broadcast query to *255.255.255.255* with a search for an actor within a context with that specific name. A matching instance responds to that query with necessary information like host and port.

### Allowed Actor and Context Names

As the URL can contain "\\" (backslash) as a separator between context and actor name, we cannot use backslashes in an actor's name, but a context name can contain any UTF characters.

---

# Agents

A `Remote Agent` system is designed in similar fashion.

First, a *RemoteAgents* class context must be created. Within this context, *Agents* can be published or retrieved from remote hosts.

*A Remote Create Sample*

```
def remoteAgents = RemoteAgents.create()
```

*Publishing :*

```
def agent = ...
remoteAgents.publish agent, "agent-name"
```

*Retrieval :*

```
def agentPromise = remoteAgents.get HOST, PORT, "agent-name"
def remoteAgent = agentPromise.get()
```

There are two ways to execute closures used to update the state of a remote *Agent* instance:

- *remote* - closure is serialized and sent to original instance and executed in that context
- *local* - current state is retrieved and closure is executed where the update originated, then updated value is sent to original instance. Concurrent changes to *Agent* wait until this process ends.

By default, a remote *Agent* uses a *remote* execution policy, but we can change it if necessary :

*Changing Policy to LOCAL*

```
def agentPromise = remoteAgents.get HOST, PORT, "agent"
def remoteAgent =  agentPromise.get()
remoteAgent.executionPolicy = AgentClosureExecutionPolicy.LOCAL
```

---

# General GPars Tips

## Grouping

High-level concurrency concepts, like **Agents, Actors** or **Dataflow** tasks and operators can be grouped around shared thread pools. The *PGroup* class and its sub-classes represent convenient **GPars** wrappers around thread pools. Objects created using the group's factory methods will share the group's thread pool.

*A* **xxxPGroup** *Sample*

```
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)
}
```

### Groups For Thread Pools

When customizing the thread pools for groups, consider using the existing **GPars** implementations - the *DefaultPool* or *ResizeablePool* classes. Or you may wish to create your own implementation of the *groovyx.gpars.scheduler.Pool* interface to pass to the *DefaultPGroup* or *NonDaemonPGroup* constructors.

## Java API

Much of **GPars** functionality can be used from **Java** just as well as from **Groovy**. Checkout the `Java API - Using GPars from Java` section of this `User Guide`. Then experiment with the Maven-based stand-alone Java demo applications.

Take **GPars** with you wherever you go!

## Performance

Your code in **Groovy** can be just as fast as code written in **Java**, **Scala** or any other programing

language. This should not be surprising, since **GPars** is technically a solid tasty Java-made cake with a Groovy DSL frosting on it.

Unlike **Java**, however, with **GPars**, as well as with other DSL-friendly languages, you are very likely to experience a useful code speed-up for free. This speed-up comes from a better and cleaner design of your application.

Coding with a concurrency DSL will give you a smaller code-base with code using the concurrency primitives as language constructs. So it's much easier to build robust concurrent applications, identify potential bottle-necks or errors and then eliminate them.

While this whole `User Guide` is describing how to use **Groovy** and **GPars** to create beautiful and robust concurrent code, we wanted to use some of these tips to highlight a few places where some code tuning or minor design compromises could give you interesting performance gains.

---

# Parallel Collections

Methods like parallel collection processing, like *eachParallel()* , *collectParallel()* and such-like, use *Parallel Array* , an efficient tree-like data structure behind the scenes. This data structure has to be built from the original collection each time you call any of the parallel collection methods. Thus when chaining parallel method calls, you might consider using the *map/reduce* API instead or alternatively, use the *ParallelArray* API directly to avoid the *Parallel Array* creation overhead.

*A Sample of Parallel Finds*

```
import groovyx.gpars.GParsPool;
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 == 'Joe'} != null
}
```

In many scenarios, changing the pool size from the default value can give you performance benefits. Especially if your tasks perform IO operations, such as file or database access, networking, etc. So increasing the number of threads in the pool is likely to help performance.

*Bump PoolSize To Boost Performance*

```
import groovyx.gpars.GParsPool;
GParsPool.withPool(50) {
    ...
}
```

Since the closures you provide to the parallel collection processing methods are executed frequently, and 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 *DefaultActors*, since they don't have to maintain an implicit state between subsequent message arrivals. The *DefaultActors* are, in fact, on a par in performance with actors from **Scala** , which you rarely hear of as being slow.

If top performance is what you're looking for then identify patterns in your code !

If top performance is what you're looking for, a good start is to identify the following patterns in your actor code:

*A Pattern To Look For*

```
actor {
    loop {
        react {msg ->
            switch(msg) {
                case String:...
                case Integer:...
            }
        }
    }
}
```

*A Better Replacement : 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 *messageHandler* and *reactor* factory methods will also give you some more speed:

*A Dynamic Sample*

```
class MyHandler extends DynamicDispatchActor {
    public void handleMessage(String msg) {
        ...
    }

    public void handleMessage(Integer msg) {
        ...
    }
}
```

Now, convert that *MyHandler* class to Java to 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 any way you like. It's always worthwhile to experiment with the actor pool size and type.

*FJPool* usually gives better characteristics than *DefaultPool*, but seems to be more sensitive to the number of threads in the pool. Sometimes, using a *ResizeablePool* or *ResizeableFJPool* can help performance by automatic eliminating unneeded threads.

*A Sample*

```
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 **agents** wisely around thread pools, and then tune the pool sizes and types, applies to **agents** as well as **actors**. With **agents**, you may also 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 know how you use **GPars** and how it performs. Send us your benchmarks, performance comparisons or profiling reports to help us tune **GPars** for you. See this page for more details.

# Hosted Environments

Hosted environments, such as *Google App Engine*, can impose additional restrictions on threading. For **GPars** to better integrate with these environments, the default thread factory and timer factory can be customized.

Hosted environments like *Google App Engine* impose restrictions on threading

The **GPars_Config** class provides static initialization methods allowing third parties to register their own implementations of the *PoolFactory* and *TimerFactory* interfaces. These can then be used to create default pools and timers for **Actors**, **Dataflow** and **PGroups**.

*Some Static Methods To Initialize Objects*

```
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)

    public static void shutdown()
}
```

Custom factories should be registered immediately after application startup in order for **Actors** and **Dataflow** to be able to use them for their default groups.

## Shutdown

The *GParsConfig.shutdown()* method can be used in managed environments to properly shutdown all asynchronously running timers and free up the memory from all thread-local variables.

After a call to this method, the **GPars** library can no longer provide the declared services.

# Compatibility

Some further compatibility issues can occur when running **GPars** in a hosted environment.

The most noticeable one is probably the lack of **ForkJoinThreadPool** support in **GAE**. Mechanisms like **Fork/Join** and **GParsPool** may not be available on some services as a result. However, **GParsExecutorsPool, Dataflow, Actors, Agents** and **STM** should work normally even when using managed non-Java SE thread pools.

# User Guide To The Conclusion

This was quite a wild ride, wasn't it?

Now, after reading this **User Guide**, you're certainly ready to build fast, robust, reliable and concurrent applications.

You've seen that there are many concepts you can choose from and each has its own areas of applicability. The ability to pick the right concept to apply is key to being a successful developer.

If you feel you can do this with **GPars**, the mission of this **User Guide** has been accomplished.

Now, go ahead, use **GPars** and have fun!