Guide To Actors

Russell Winder

Version 1.0, 2015-10-01

Table of Contents

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are allowed to be called by other objects directly, the thread-safety guarantee for the actor's code and state are **no longer valid** .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 larg3 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 allow detaching actors from the threads the *react* method demands the code to be written in a special **continuation style**.

A 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 **continue** . Thus **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 iteration 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 sends a reply to authors of all the currently processed message (the last one), whereas *reply()* called on messages sends a reply to the author of the particular message only.

DemoMultiMessage.groovy - See 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 back 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.

A 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.

A Sample

```
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 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 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

```
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 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 *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 Demos.

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 gets scheduled for processing with the pool. Event-based actors internally simulate continuations - actor's work is split into sequentially run chunks, which get 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 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:

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')

    delegate.metaClass.onTimeout = {->
        friend.send('I see, busy as usual. Never mind.')
        stop()
    }

    //wait for answer 1sec
    react(1000) {msg ->
        if (msg != Actor.TIMEOUT) {
            //continue conversation
            println "Thank you for $msg"
        }
    }
}

me.join()
```

Notice the possibility to use Groovy meta-programming to define actor's lifecycle notification 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 and also under the covers the memory gets synchronized each time a thread gets 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** .

A Sample

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

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

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

Simple calculator

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 Sample

```
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) gets 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 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 of a 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:

A 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 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. It by default 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 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.

A Sample

```
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 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 is 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 they let you write straight code without the necessity of continuation style, since they only do blocking message reads 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 gets 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 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 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 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 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* 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 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, when 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()*, allow for more event-driven like 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, which 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 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.

A Sample

```
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 which respectively use the 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 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 possibility to use the actor's *loop()* method to create a loop that runs until the actor terminates.

A Sample

```
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, allowing you to avoid dealing directly with the actor machinery, having to match messages, wait for results and send replies.

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 is 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

Now 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 he 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.

A 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 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 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

```
//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 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()
```