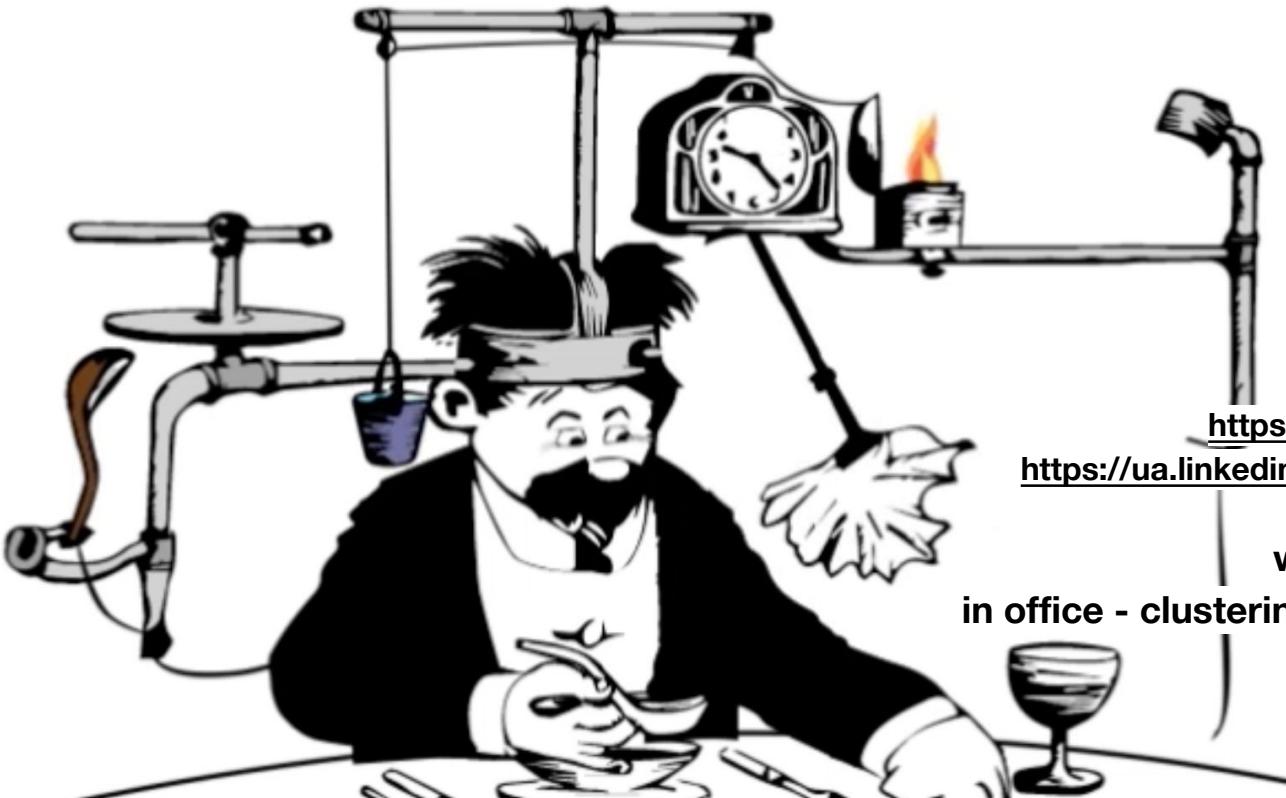


# GPar: Unsung Hero of Concurrency in Practice



# Yaroslav Yermilov

Senior Software Engineer  
EPAM Systems

<https://yermilov.github.io/>

<https://twitter.com/yermilov17>

<https://www.facebook.com/yaroslav.yermilov>

<https://ua.linkedin.com/pub/yaroslav-yermilov/58/682/506>

work for EPAM Systems since 2011

in office - clustering, Big Data and automated testing

out of office - Groovy

# Focus for this talk

obey Moore's law

how Java handles concurrent/parallel development?

GPars place in this picture

how GPars handles concurrent/parallel development?

why and when use GPars?

why and when do not use GPars?



HOW TO STAY  
FOCUSED

**MOORE'S LAW:**  
Computers will get  
exponentially faster.



**MURPHY'S LAW:**  
Anything that can go  
wrong will go wrong.



**What do you MEAN**  
***"Unexpectedly Quit"*?**



## The **Thread** class

---

802

*Thinking in Java*

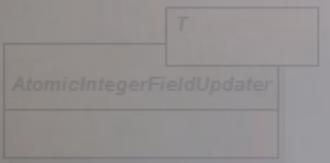
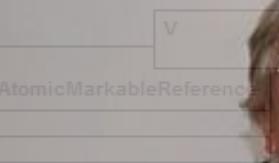
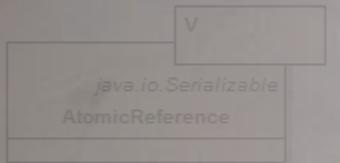
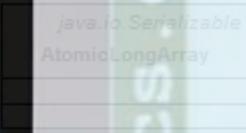
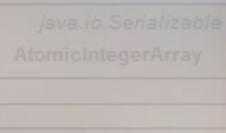
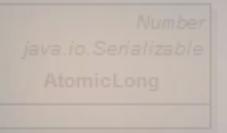
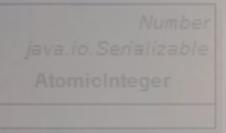
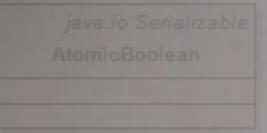
Bruce Eckel

---

The traditional way to turn a **Runnable** object into a working task is to hand it to a **Thread** constructor. This example shows how to drive a **Liftoff** object using a **Thread**:

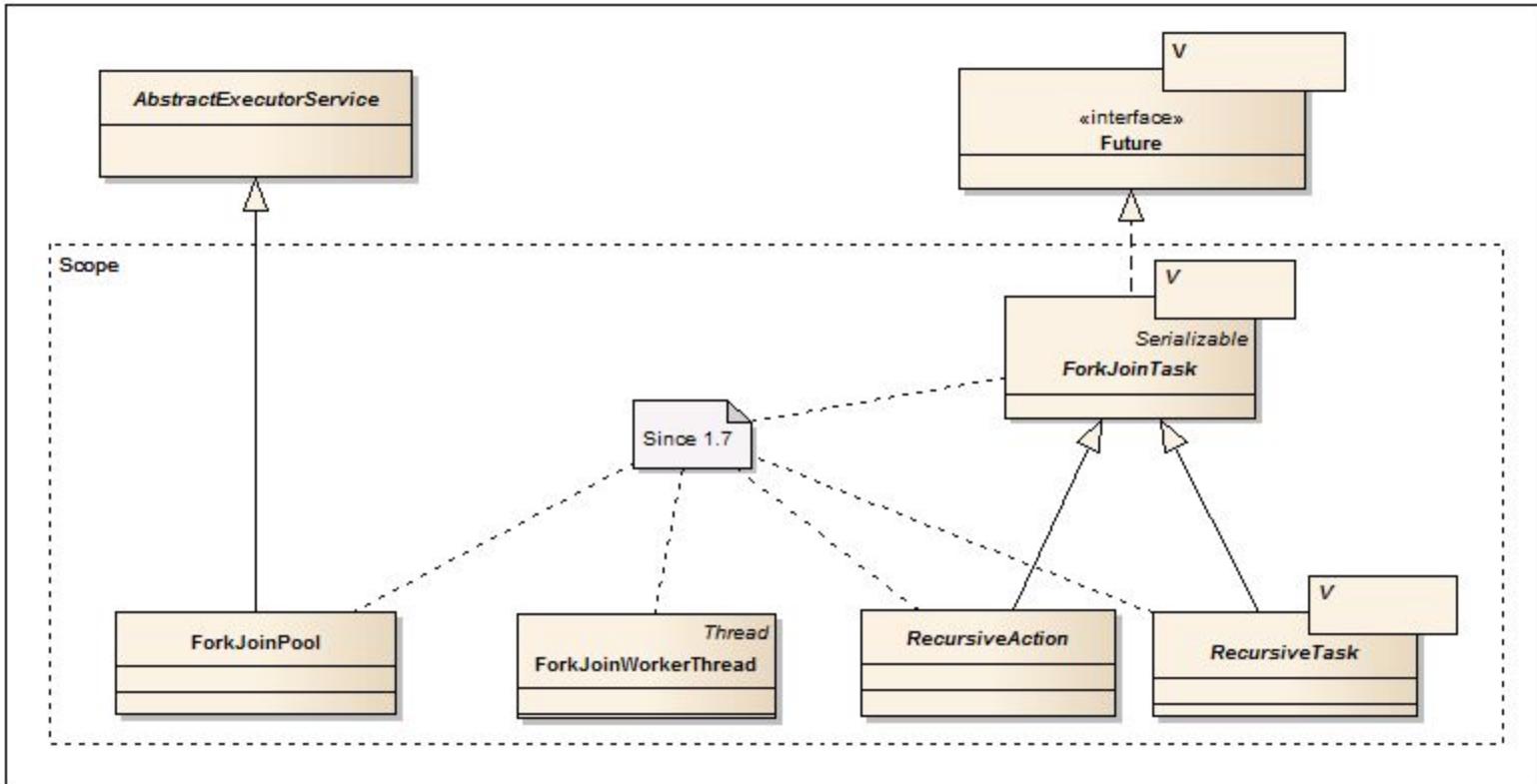
```
//: concurrency/BasicThreads.java
// The most basic use of the Thread class.

public class BasicThreads {
    public static void main(String[] args) {
        Thread t = new Thread(new Liftoff());
        t.start();
        System.out.println("Waiting for Liftoff");
    }
} /* Output: (90% match)
Waiting for Liftoff
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1),
#0(Liftoff!),
*///:-
```



protected void  
if (hi -  
All since 1.5  
sequential  
else {  
int m =  
SortTask  
.lockSupport  
x.fork  
new Sort  
r.join  
merge (a  
}  
}  
// ...  
}





```
package hello;

import java.util.concurrent.atomic.AtomicLong;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {

    private static final String template = "Hello, %s!";
    private final AtomicLong counter = new AtomicLong();

    @RequestMapping("/greeting")
    public Greeting greeting(@RequestParam(value="name", defaultValue="World") String name) {
        return new Greeting(counter.incrementAndGet(),
                            String.format(template, name));
    }
}
```

## Parallelism

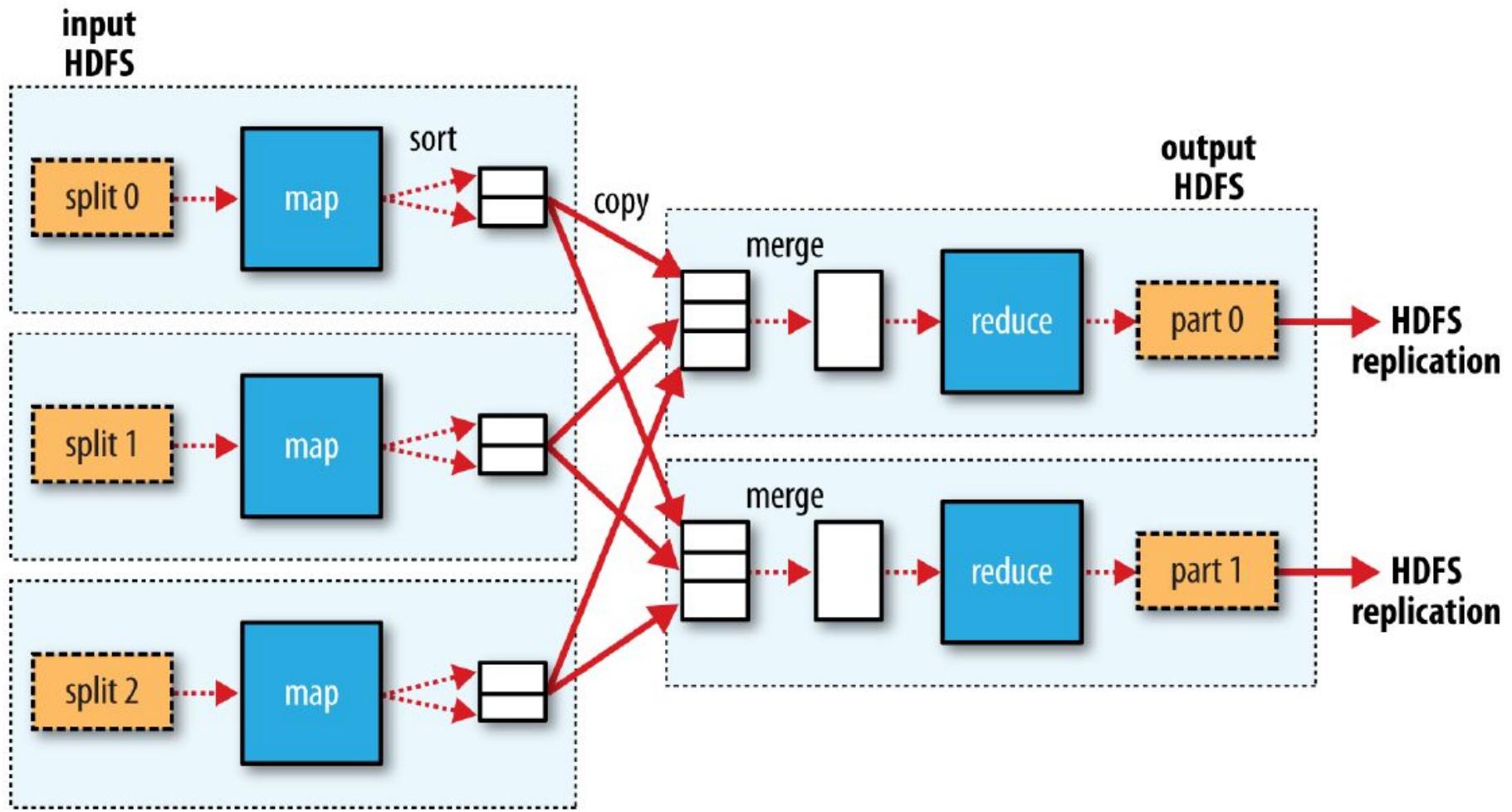
Processing elements with an explicit for-loop is inherently serial. Streams facilitate parallel execution by reframing the computation as a pipeline of aggregate operations, rather than as imperative operations on each individual element. All streams operations can execute either in serial or in parallel. The stream implementations in the JDK create serial streams unless parallelism is explicitly requested. For example, Collection has methods `Collection.stream()` and `Collection.parallelStream()`, which produce sequential and parallel streams respectively; other stream-bearing methods such as `IntStream.range(int, int)` produce sequential streams but these streams can be efficiently parallelized by invoking their `BaseStream.parallel()` method. To execute the prior "sum of weights of widgets" query in parallel, we would do:

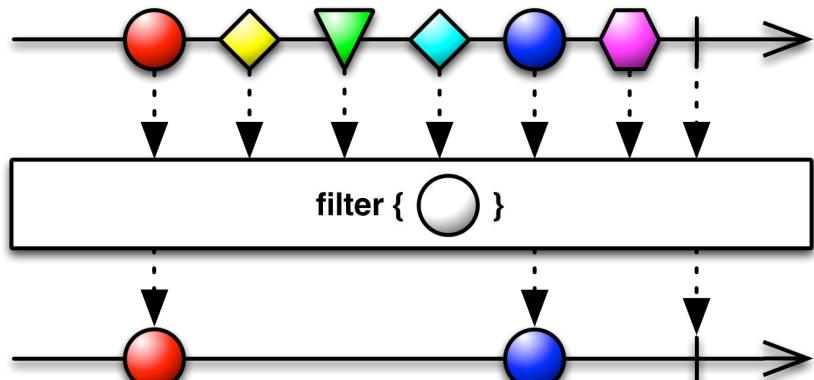
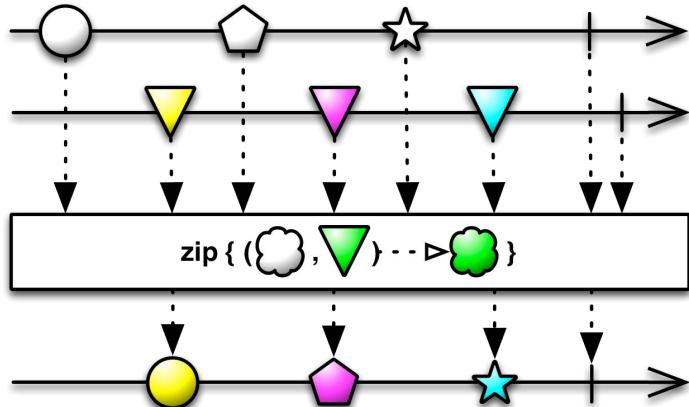
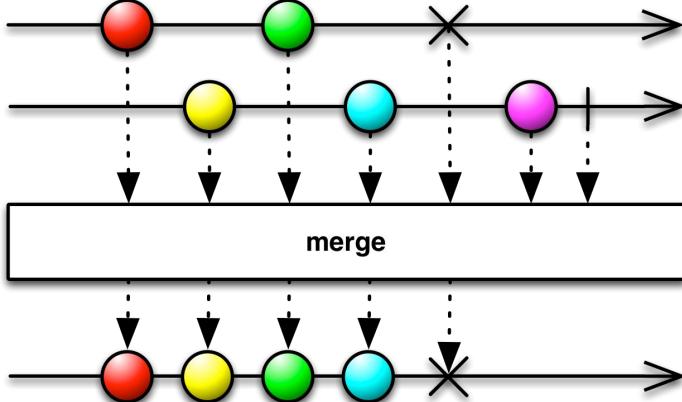
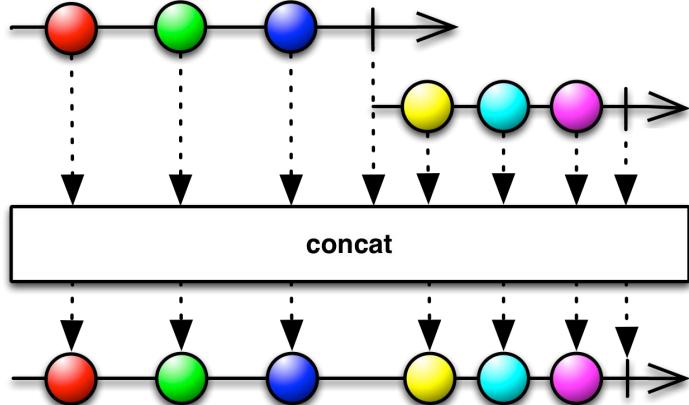
```
int sumOfWeights = widgets.parallelStream()
    .filter(b -> b.getColor() == RED)
    .mapToInt(b -> b.getWeight())
    .sum();
```

The only difference between the serial and parallel versions of this example is the creation of the initial stream, using "`parallelStream()`" instead of "`stream()`". When the terminal operation is initiated, the stream pipeline is executed sequentially or in parallel depending on the orientation of the stream on which it is invoked. Whether a stream will execute in serial or parallel can be determined with the `isParallel()` method, and the orientation of a stream can be modified with the `BaseStream.sequential()` and `BaseStream.parallel()` operations. When the terminal operation is initiated, the stream pipeline is executed sequentially or in parallel depending on the mode of the stream on which it is invoked.

Except for operations identified as explicitly nondeterministic, such as `findAny()`, whether a stream executes sequentially or in parallel should not change the result of the computation.

Most stream operations accept parameters that describe user-specified behavior, which are often lambda expressions. To preserve correct behavior, these *behavioral parameters* must be *non-interfering*, and in most cases must be *stateless*. Such parameters are always instances of a functional interface such as `Function`, and are often lambda expressions or method references.





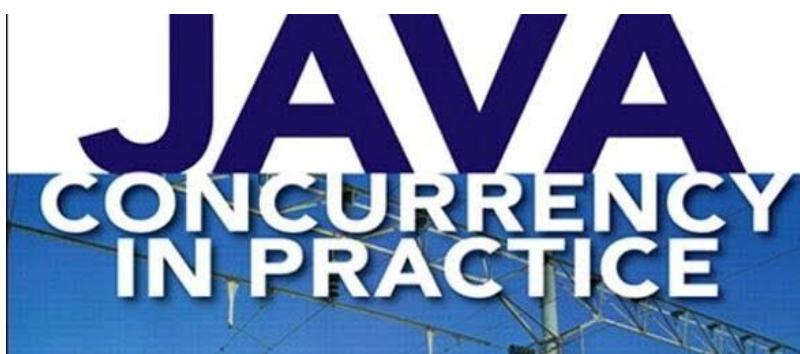


BRIAN GOETZ



WITH TIM PEIERLS, JOSHUA BLOCH,  
JOSEPH BOWBEER, DAVID HOLMES,  
AND DOUG LEA

When designing thread-safe classes, good object-oriented techniques - encapsulation, immutability, and clear specification of invariants - are your best friends.



If multiple threads access the same mutable state variable without appropriate synchronization, your program is broken. There are three ways to fix it:

- Don't share the state variable across threads;
- Make the state variable immutable; or
- Use synchronization whenever accessing the state variable.

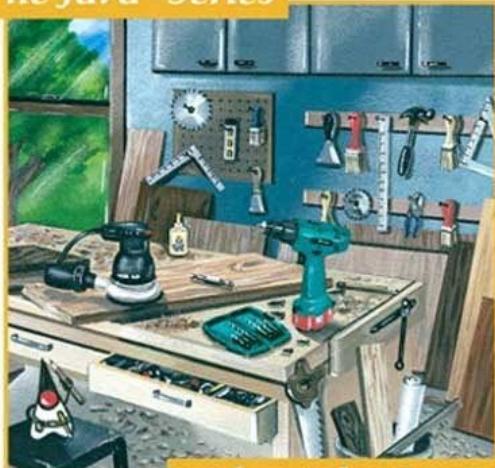


**Joshua Bloch**

Revised and  
Updated for  
Java SE 6

# Effective Java™ Second Edition

*The Java™ Series*



*...from the Source*



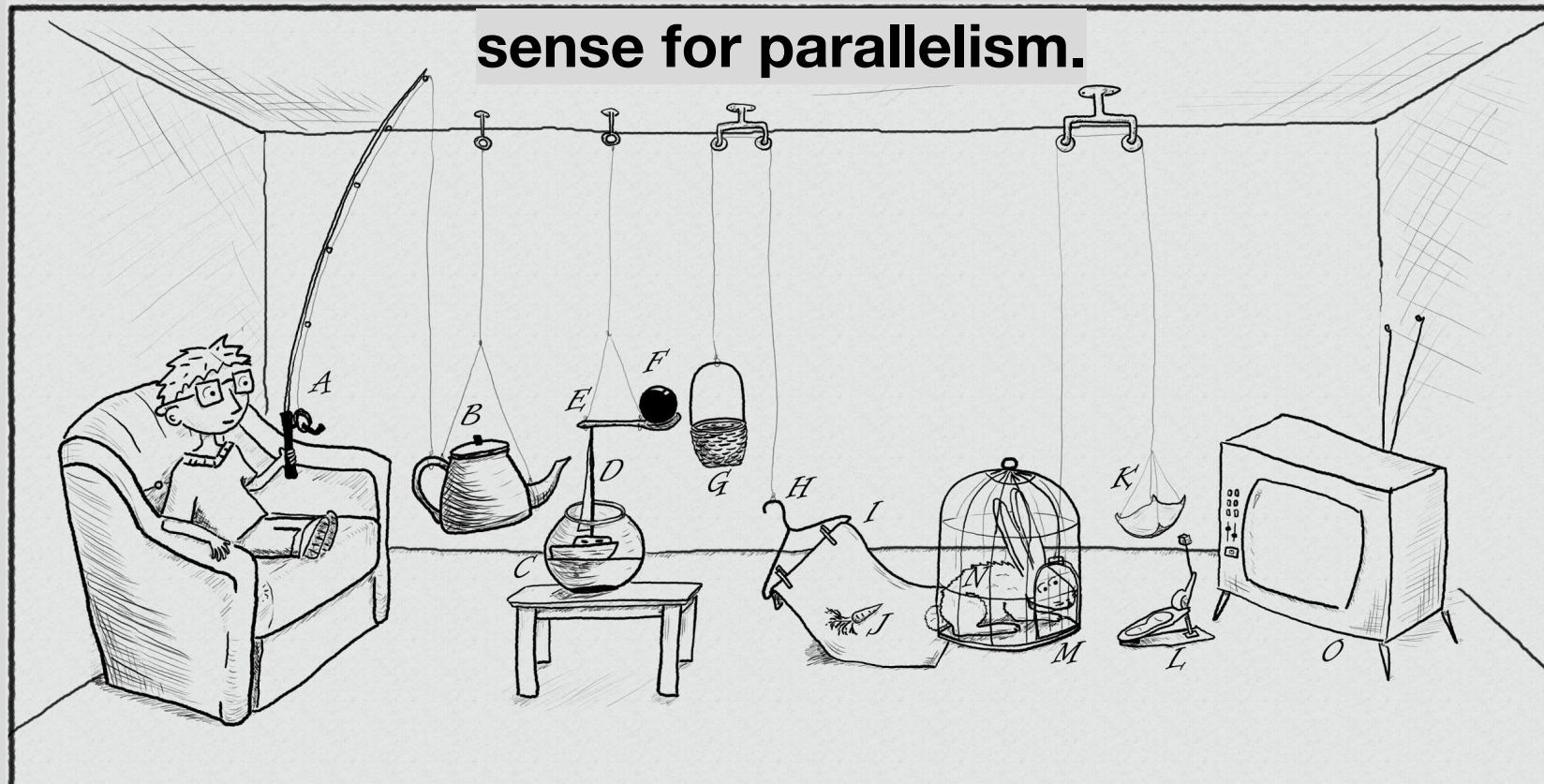
<b>10 Concurrency.....</b>	<b>259</b>
Item 66: Synchronize access to shared mutable data.....	259
Item 67: Avoid excessive synchronization .....	265
Item 68: Prefer executors and tasks to threads.....	271
Item 69: Prefer concurrency utilities to <code>wait</code> and <code>notify</code> .....	273

## CONTENTS

Item 70: Document thread safety .....	278
Item 71: Use lazy initialization judiciously .....	282
Item 72: Don't depend on the thread scheduler .....	286
Item 73: Avoid thread groups .....	288

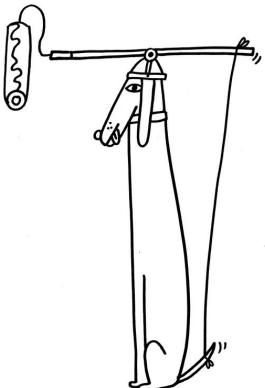


The traditional thread-based concurrency model built into Java doesn't match well with the natural human sense for parallelism.





HOW TO BRUSH  
YOUR TEETH



HOW TO UTILIZE YOUR  
DOG'S MINDLESS JOY

**GPars:**

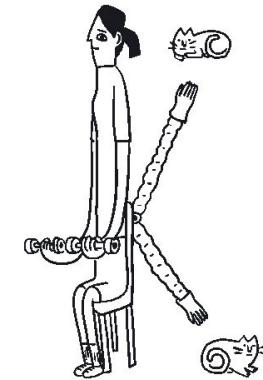
**map/reduce  
fork/join**

**asynchronous closures**

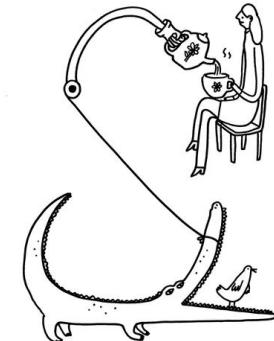
**actors**

**agents**

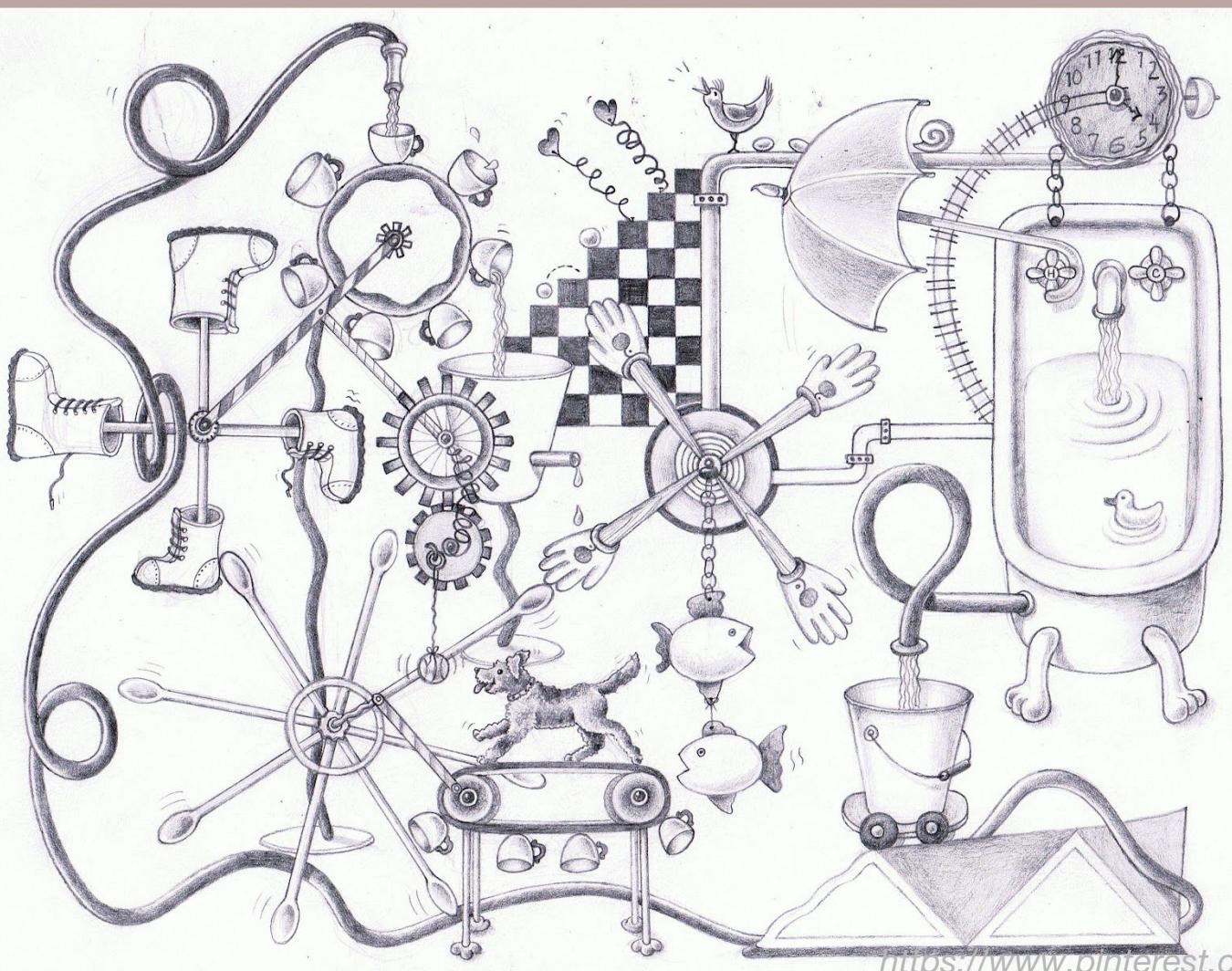
**dataflows**



HOW TO PET YOUR PETS  
WHILE DOING YOUR REPS



HOW TO SERVE TEA



**DEMO**

```
1 public final class ImmutableJavaPerson {  
2  
3     private final String name;  
4     private final Collection<String> tweets;  
5  
6     public ImmutableJavaPerson(String name, Collection<String> tweets) {  
7         this.name = name;  
8         this.tweets = new ArrayList<>(tweets);  
9     }  
10  
11    public String getName() {  
12        return name;  
13    }  
14  
15    public Collection<String> getTweets() {  
16        return unmodifiableCollection(tweets);  
17    }  
18}
```

```
19     @Override
20     public boolean equals(Object o) {
21         if (this == o) return true;
22         if (o == null || getClass() != o.getClass()) return false;
23
24         ImmutableJavaPerson that = (ImmutableJavaPerson) o;
25
26         if (name != null ? !name.equals(that.name) : that.name != null) return false;
27         return tweets != null ? tweets.equals(that.tweets) : that.tweets == null;
28     }
29
30     @Override
31     public int hashCode() {
32         int result = name != null ? name.hashCode() : 0;
33         result = 31 * result + (tweets != null ? tweets.hashCode() : 0);
34         return result;
35     }
36 }
```

```
1 @Immutable class ImmutableGroovyPerson {  
2  
3     String name  
4     Collection<String> tweets  
5 }
```

```
1 def thread1 = Thread.start {  
2     println "Hello from ${Thread.currentThread().name}"  
3 }  
4  
5 def thread2 = Thread.startDaemon {  
6     println "Hello from ${Thread.currentThread().name}"  
7 }  
8  
9 [ thread1, thread2 ]*.join()
```

---

```
1 def process =(['git', 'status']).execute([], new File('.'))  
2 def processOutput = new StringWriter()  
3 process.consumeProcessOutput processOutput, processOutput  
4 process.waitFor()  
5 println processOutput.toString().trim()
```

```
1  class SynchronizedCounter {  
2  
3      int atomicCounter  
4      int counter  
5  
6      @Synchronized  
7      int incrementAndGet() {  
8          atomicCounter = atomicCounter + 1  
9          return atomicCounter  
10     }  
11  
12     @WithReadLock  
13     int getCounter() {  
14         counter  
15     }  
16  
17     @WithWriteLock  
18     void increment() {  
19         counter = counter + 1  
20     }  
21 }
```

```
1 def findMostPopularTweet = { Map params ->
2     String topic = params.about
3     Collection tweets = params.from
4
5     tweets.findAll({ Status tweet ->
6         tweet.text.toLowerCase().contains(topic.toLowerCase())
7     }).findAll({ Status tweet ->
8         !tweet.retweet
9     }).collect ({ Status tweet ->
10        [ user: tweet.user.screenName, text: tweet.text, favourited: tweet.favoriteCount, retweeted: tweet.retweetCount ]
11    }).max({
12        it.favourited + it.retweeted
13    })
14 }
15
16 def result = findMostPopularTweet about: 'junit', from: jeeconfTweets
```

```
1 def findMostPopularTweet = { Map params ->
2     String topic = params.about
3     Collection tweets = params.from
4
5     tweets.findAll({ Status tweet ->
6         tweet.text.toLowerCase().contains(topic.toLowerCase())
7     }).findAll({ Status tweet ->
8         !tweet.retweet
9     }).collect ({ Status tweet ->
10        [ user: tweet.user.screenName, text: tweet.text, favourited: tweet.favoriteCount, retweeted: tweet.retweetCount ]
11    }).max({
12        it.favourited + it.retweeted
13    })
14 }
15
16 ParallelEnhancer.enhanceInstance(jeeconfTweets)
17 jeeconfTweets.makeConcurrent()
18
19 def result = findMostPopularTweet about: 'junit', from: jeeconfTweets
```

```
1 Optional<Status> mostPopularTweet = tweets.stream()
2
3     .parallel()
4
5         .filter(tweet -> tweet.getText().toLowerCase().contains(topic.toLowerCase()))
6         .filter(tweet -> !tweet.isRetweet())
7         .max(Comparator.comparingInt(tweet -> tweet.getFavoriteCount() + tweet.getRetweetCount()));
```

```
1 def findMostPopularTweet = { Map params ->
2     String topic = params.about
3     Collection tweets = params.from
4
5     withPool {
6         tweets.parallel.filter({ Status tweet ->
7             tweet.text.toLowerCase().contains(topic.toLowerCase())
8         }).filter({ Status tweet ->
9             !tweet.retweet
10        }).map({ Status tweet ->
11            [user: tweet.user.screenName, text: tweet.text, favourited: tweet.favoriteCount, retweeted: tweet.retweetCount]
12        }).max({
13            it.favourited + it.retweeted
14        })
15    }
16 }
17
18 def result = findMostPopularTweet about: 'junit', from: jeeconfTweets
```

```
1  withPool {
2      Closure requestFun = { ->
3          'junit'
4      }.asyncFun()
5
6      Closure countFun = { ->
7          100
8      }.asyncFun()
9
10     Closure requestTweets = { String request, int count ->
11         new Twitter().search(request, count)
12     }.asyncFun()
13
14     Promise tweets = requestTweets(requestFun(), countFun())
15
16     tweets.then { Collection result ->
17         result.size()
18     }.then { int size ->
19         println size
20     }.join()
21 }
```

```
1  withPool {  
2      Closure requestTweets = { String request ->  
3          new Twitter().search(request, 100)  
4      }  
5  
6      Closure cachedRequestTweets = requestTweets.gmemoize()  
7  
8      cachedRequestTweets('junit')  
9      cachedRequestTweets('junit')  
10 }
```

---

# **why and when use GPars?**

# **why and when do not use GPars?**

# **Further dive**