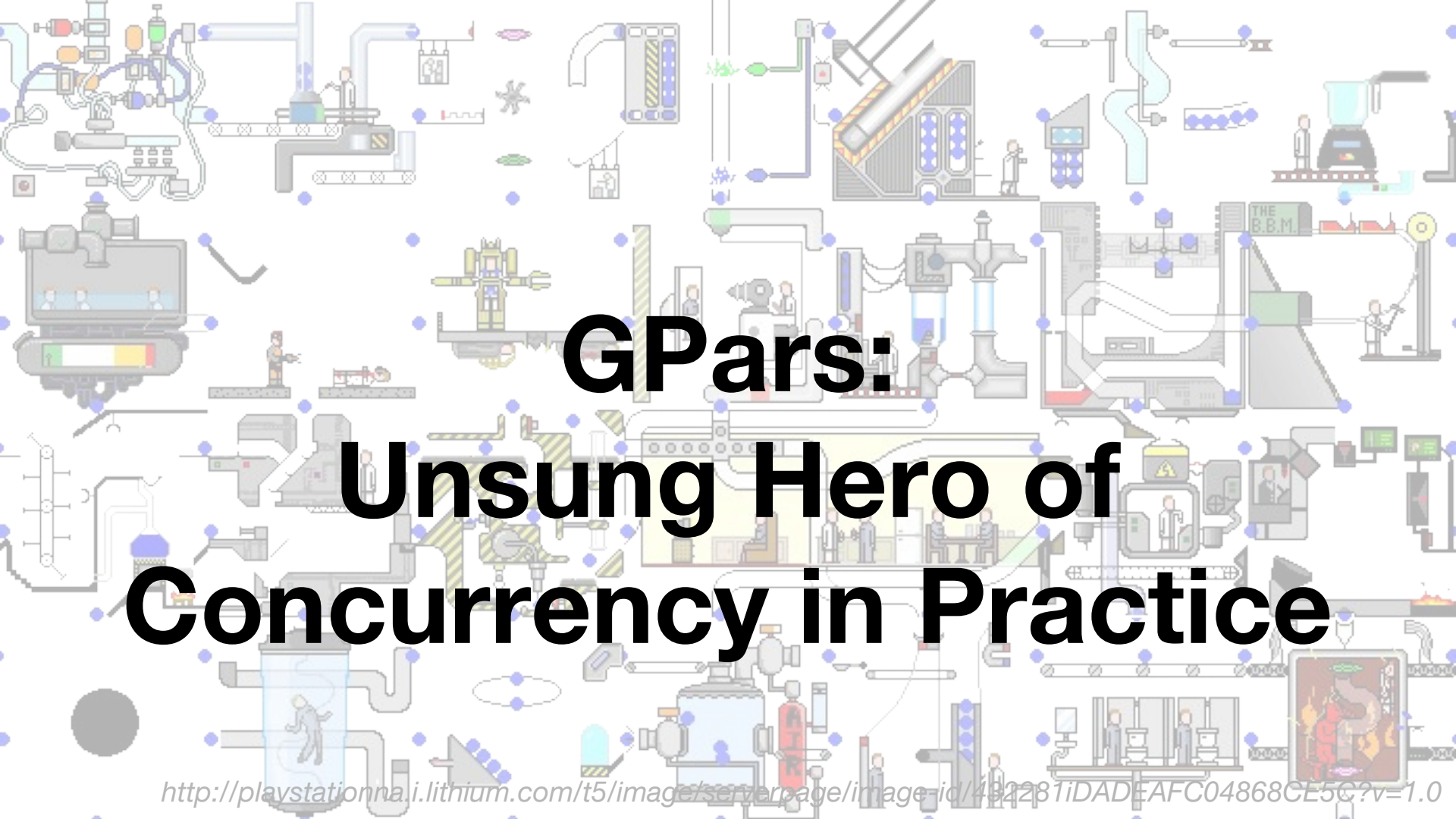
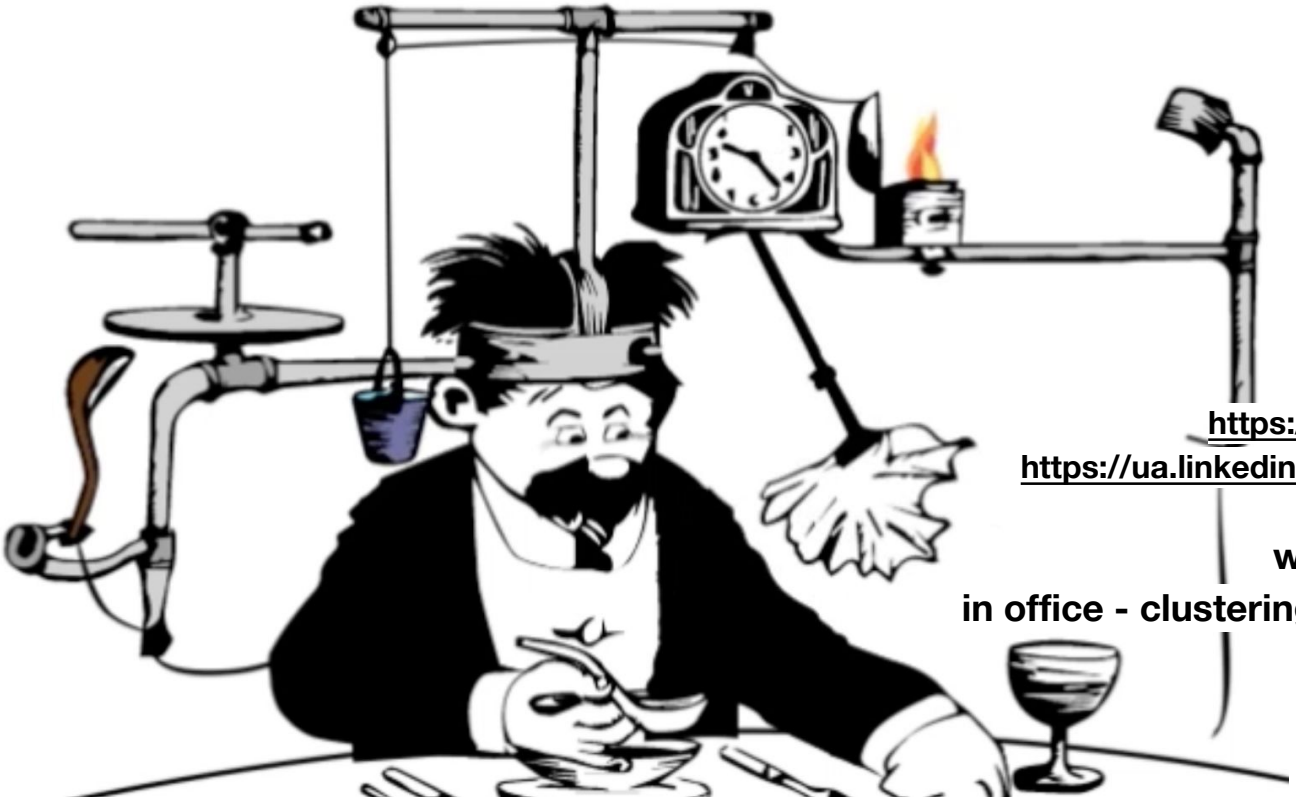


GPars: Unsung Hero of Concurrency in Practice

http://playstationna.i.lithium.com/t5/image/serverpage/image-id/432281iDADEAFC04868CE5C?v=1.0



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

https://motherboard.vice.com/en_us/article/inside-rube-goldberg-machine-youtube-video-artist-joseph-herschers-bedroom-workshop

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?

http://www.theverge.com/2015/4/22/8381963/rube-goldberg-machine-contest-history-ideas

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*"?

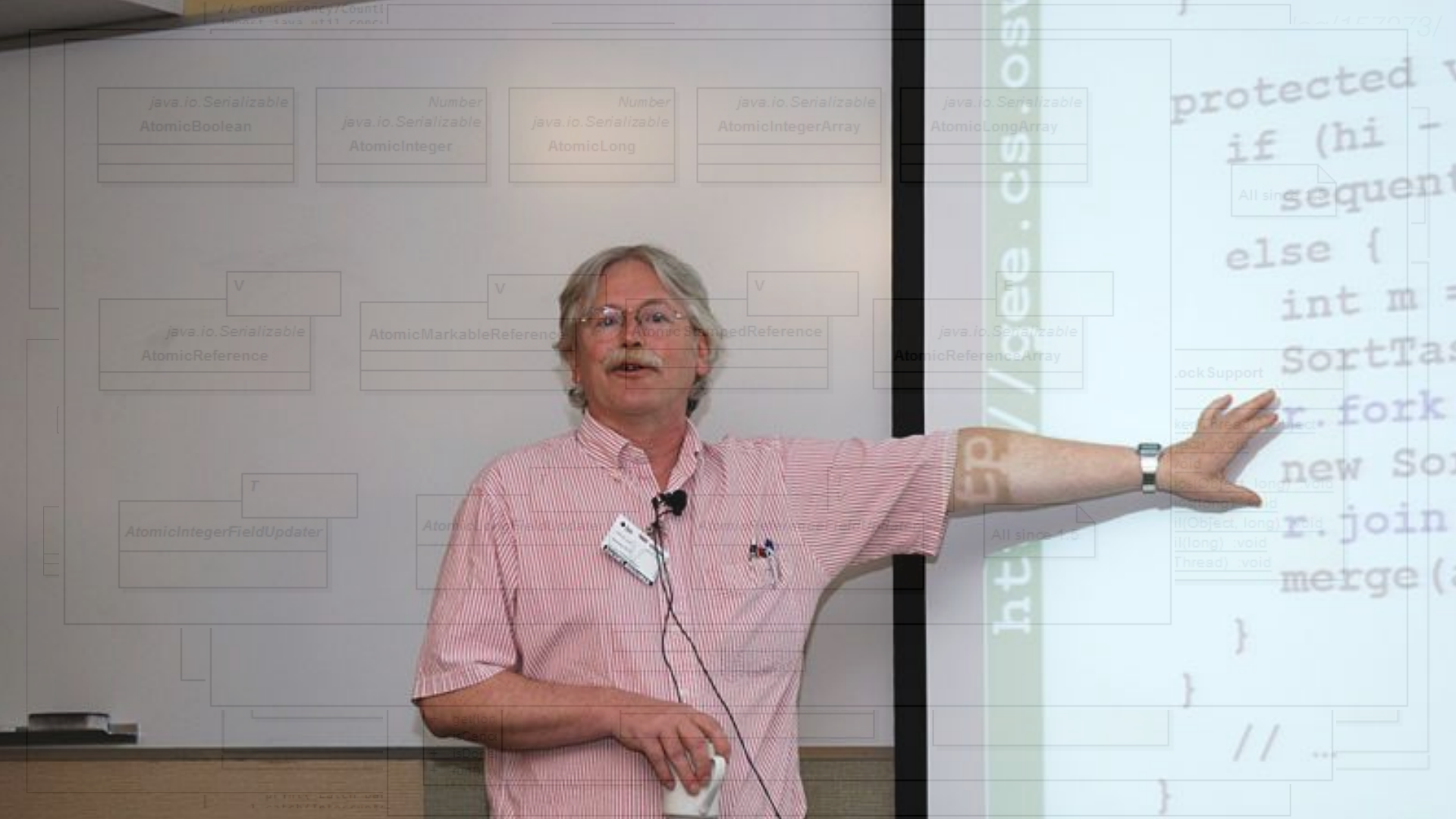
http://www.systemcomic.com/2009/04/27/the-system-197

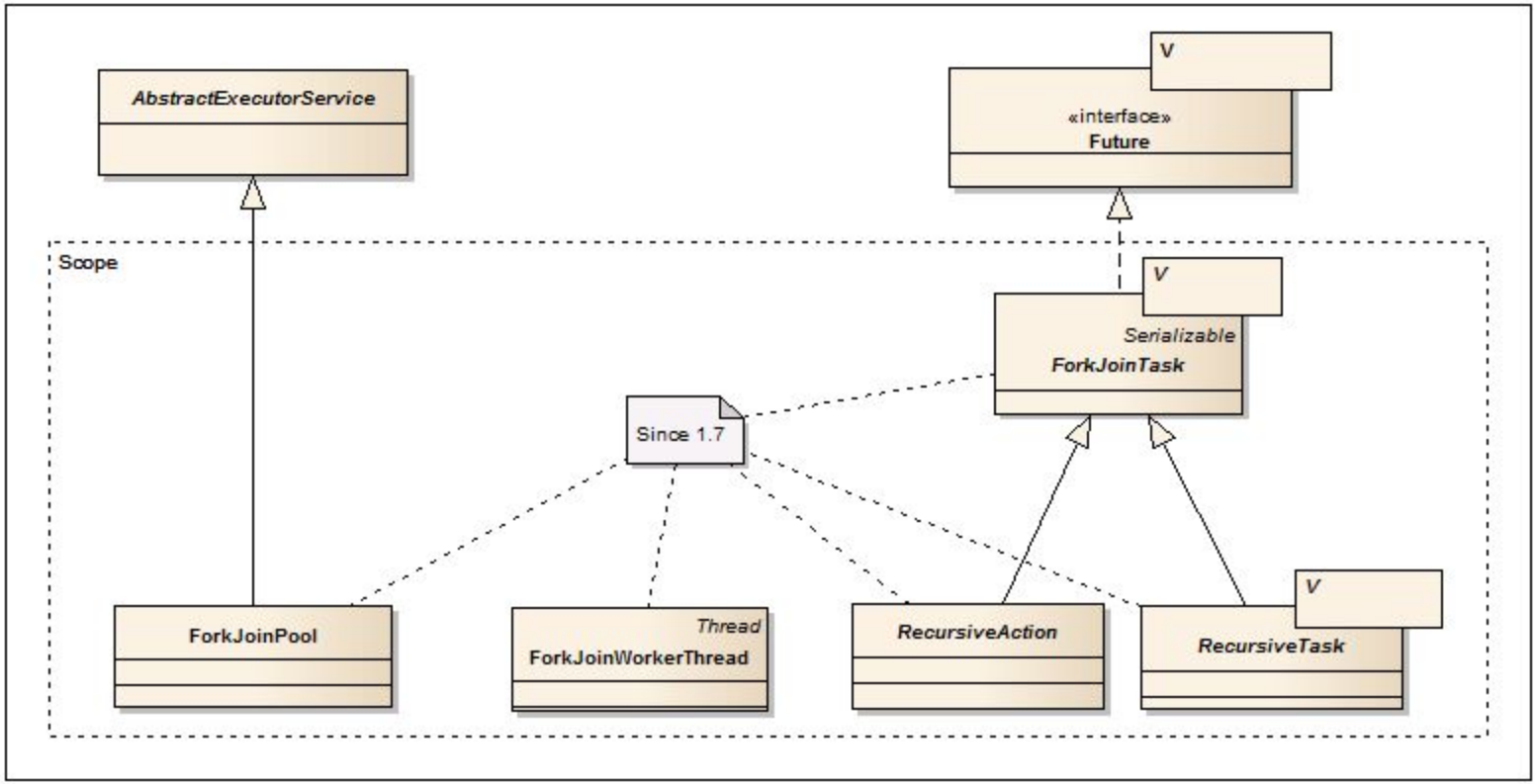
The **Thread** class

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!),
*///:~
```


java.io.Serializable
AtomicBoolean
Number
java.io.Serializable
AtomicInteger
Number
java.io.Serializable
AtomicLong
java.io.Serializable
AtomicIntegerArray
java.io.Serializable
AtomicLongArray
V
java.io.Serializable
AtomicReference
V
AtomicMarkableReference
V
java.io.Serializable
AtomicReferenceArray
T
AtomicIntegerFieldUpdater
protected
if (hi -
sequen
else {
int m
SortTas
r.fork
new So
r.join
merge(
}
}
// ...
}


AbstractExecutorService
V
«interface»
Future
Scope
V
Serializable
ForkJoinTask
Since 1.7
ForkJoinPool
Thread
ForkJoinWorkerThread
RecursiveAction
V
RecursiveTask

```java
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
        return new Greeting(counter.incrementAndGet(),
                            String.format(template, name));
    }
}
```

Parallelism

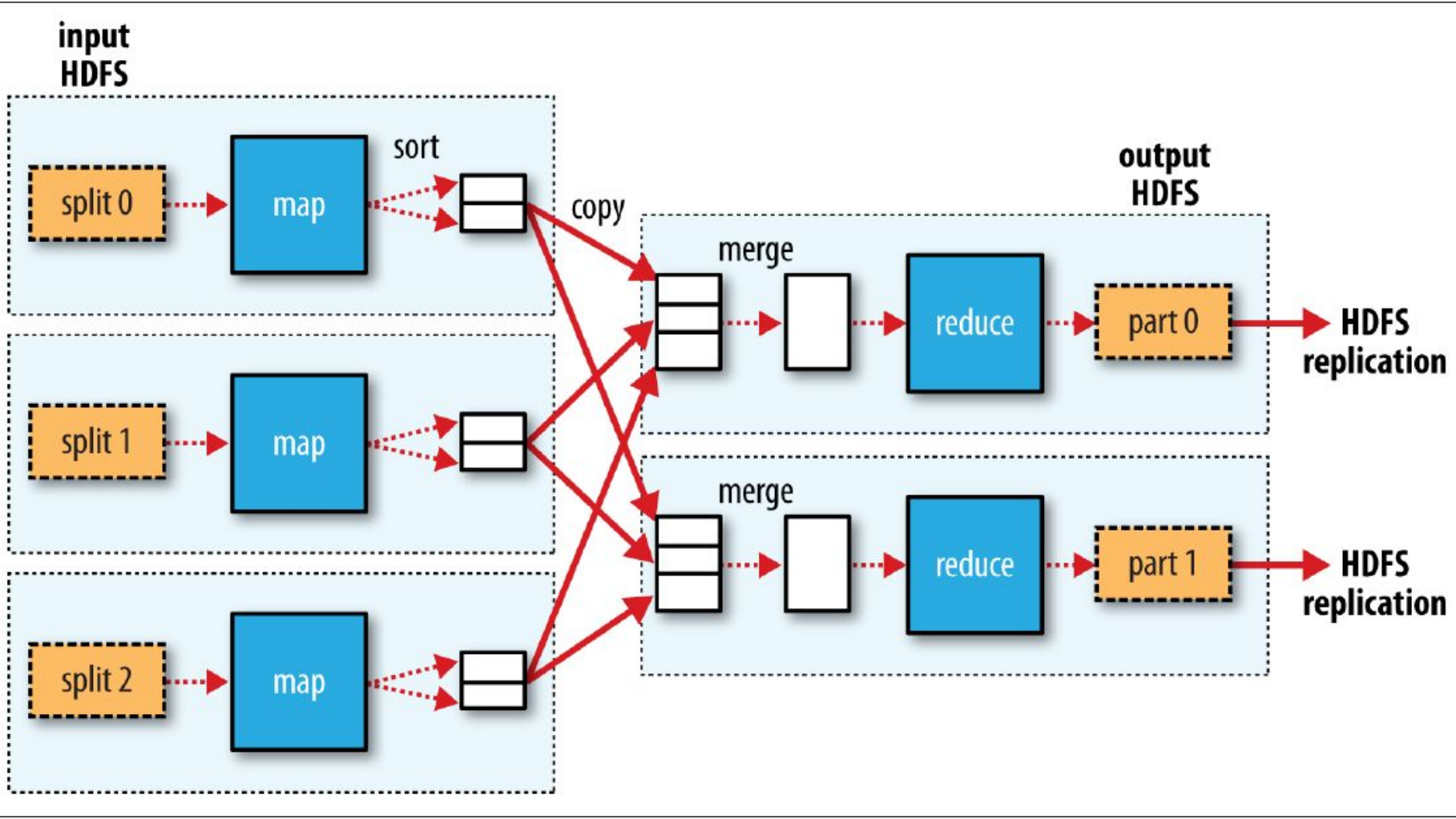
Processing elements with an explicit `for`-loop is inherently serial. Streams facilitate parallel execution by reframing the computation as a pipeline of aggregate operations, rather than as imperative operations on each individual element. All streams operations can execute either in serial or in parallel. 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:

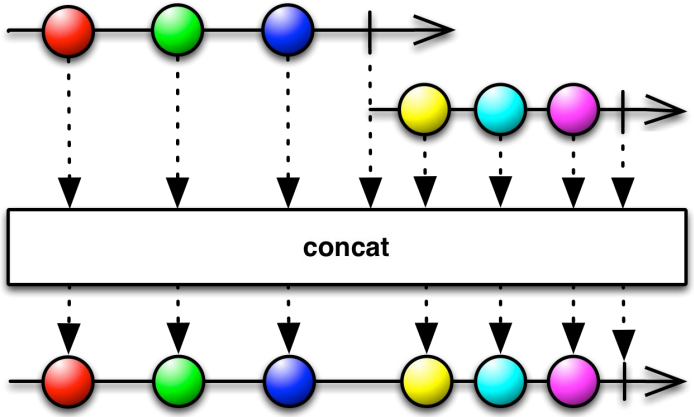
```
int sumOfWeights = widgets.parallelStream()
                          .filter(b -> b.getColor() == RED)
                          .mapToInt(b -> b.getWeight())
                          .sum();
```

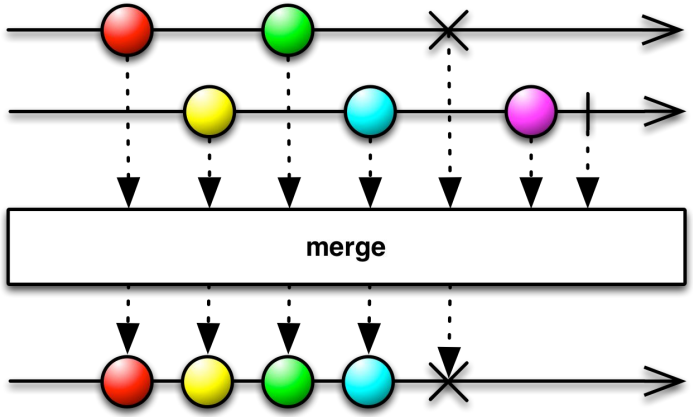
The only difference between the serial and parallel versions of this example is the creation of the initial stream, using "`parallelStream()`" instead of "`stream()`". When the terminal operation is initiated, the stream pipeline is executed sequentially or in parallel depending on the orientation of 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.

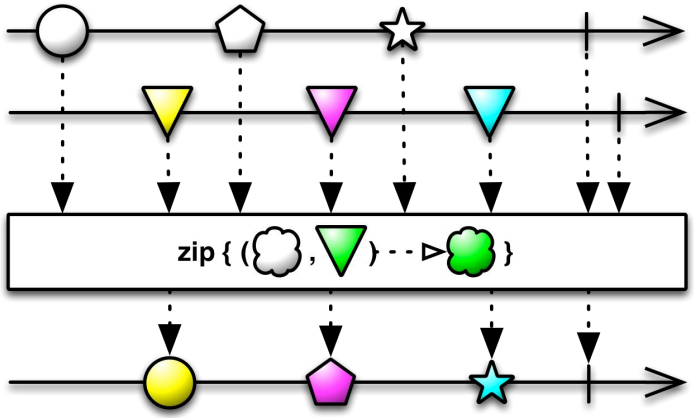
Except for operations identified as explicitly nondeterministic, such as `findAny()`, whether a stream executes sequentially or in parallel should not change the result of the computation.

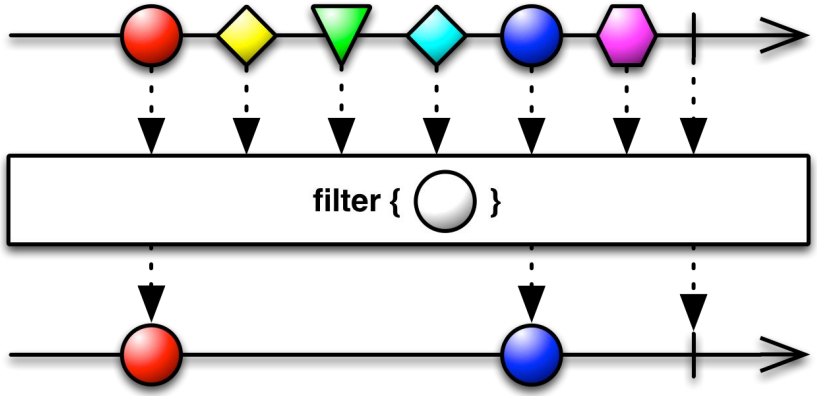
Most stream operations accept parameters that describe user-specified behavior, which are often lambda expressions. To preserve correct behavior, these *behavioral parameters* must be *non-interfering*, and in most cases must be *stateless*. Such parameters are always instances of a functional interface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


input
HDFS
split 0
map
sort
split 1
map
split 2
map
copy
merge
reduce
part 0
output
HDFS
HDFS
replication
merge
reduce
part 1
HDFS
replication









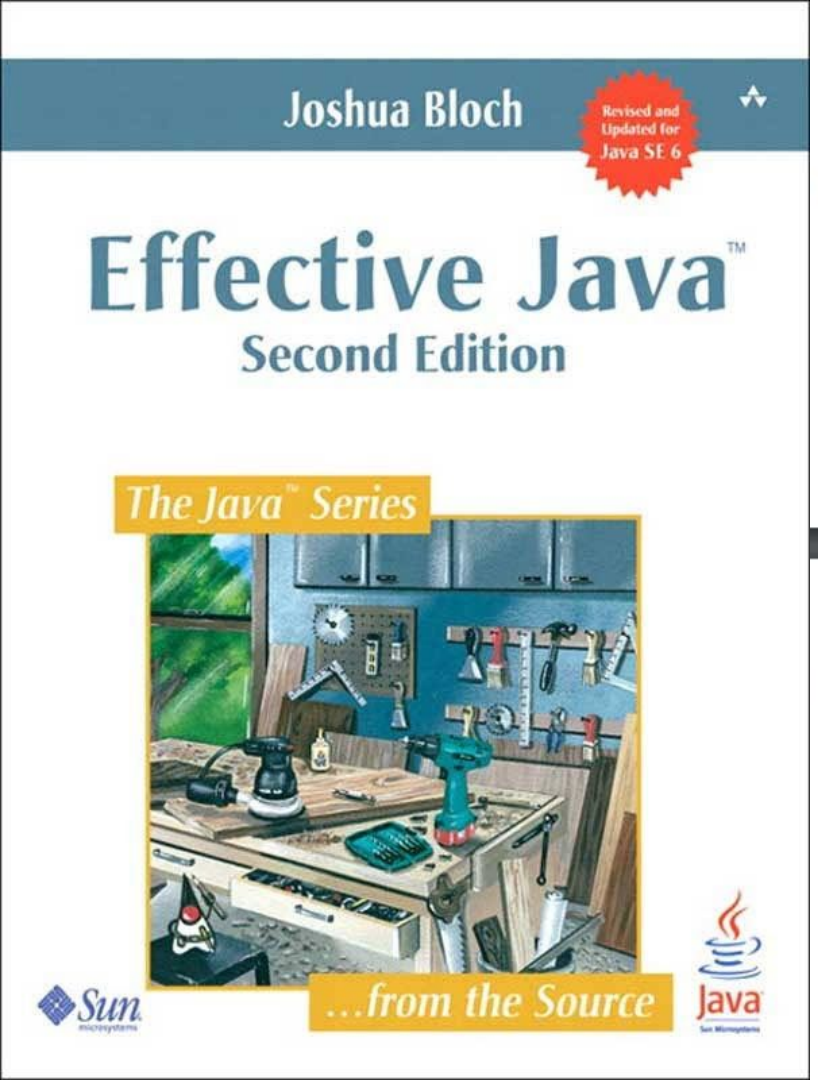
http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Flowable.html

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

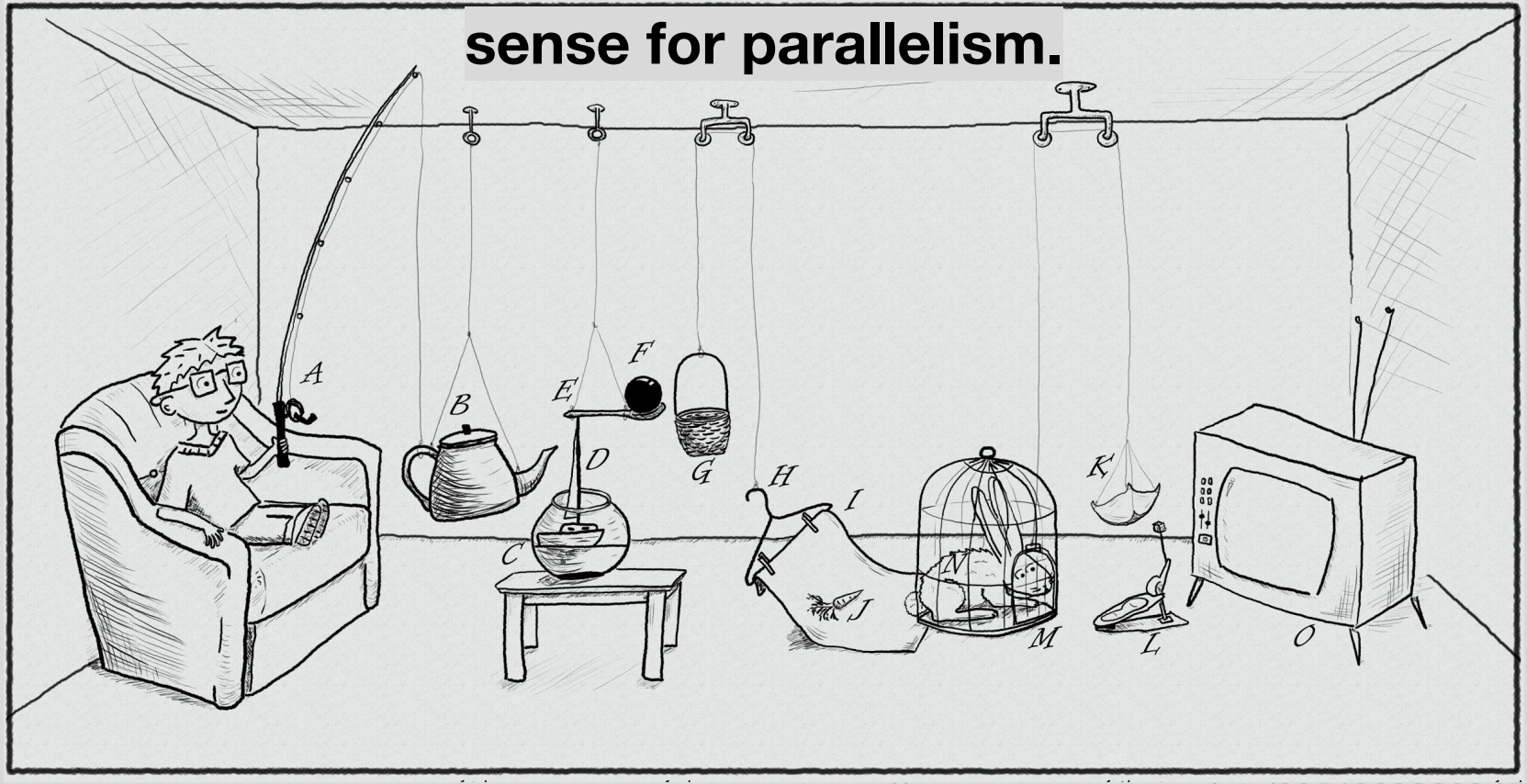
When designing thread-safe classes, good object-oriented techniques - encapsulation, immutability, and clear specification of invariants - are your best friends.

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

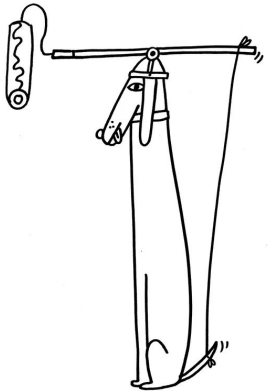
- Don't share the state variable across threads;
- Make the state variable immutable; or
- Use synchronization whenever accessing the state variable.



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



https://www.pinterest.com/pin/395190936024778319/



GPars:

map/reduce

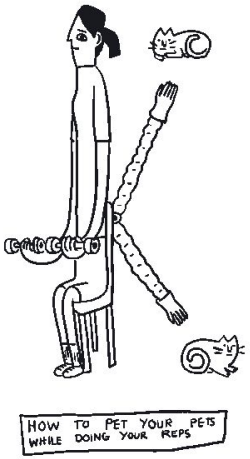
fork/join

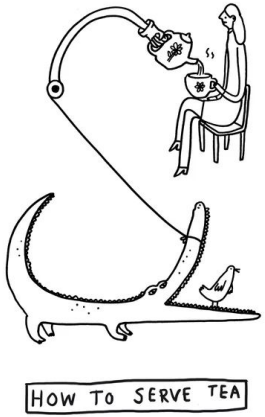
asynchronous closures

actors

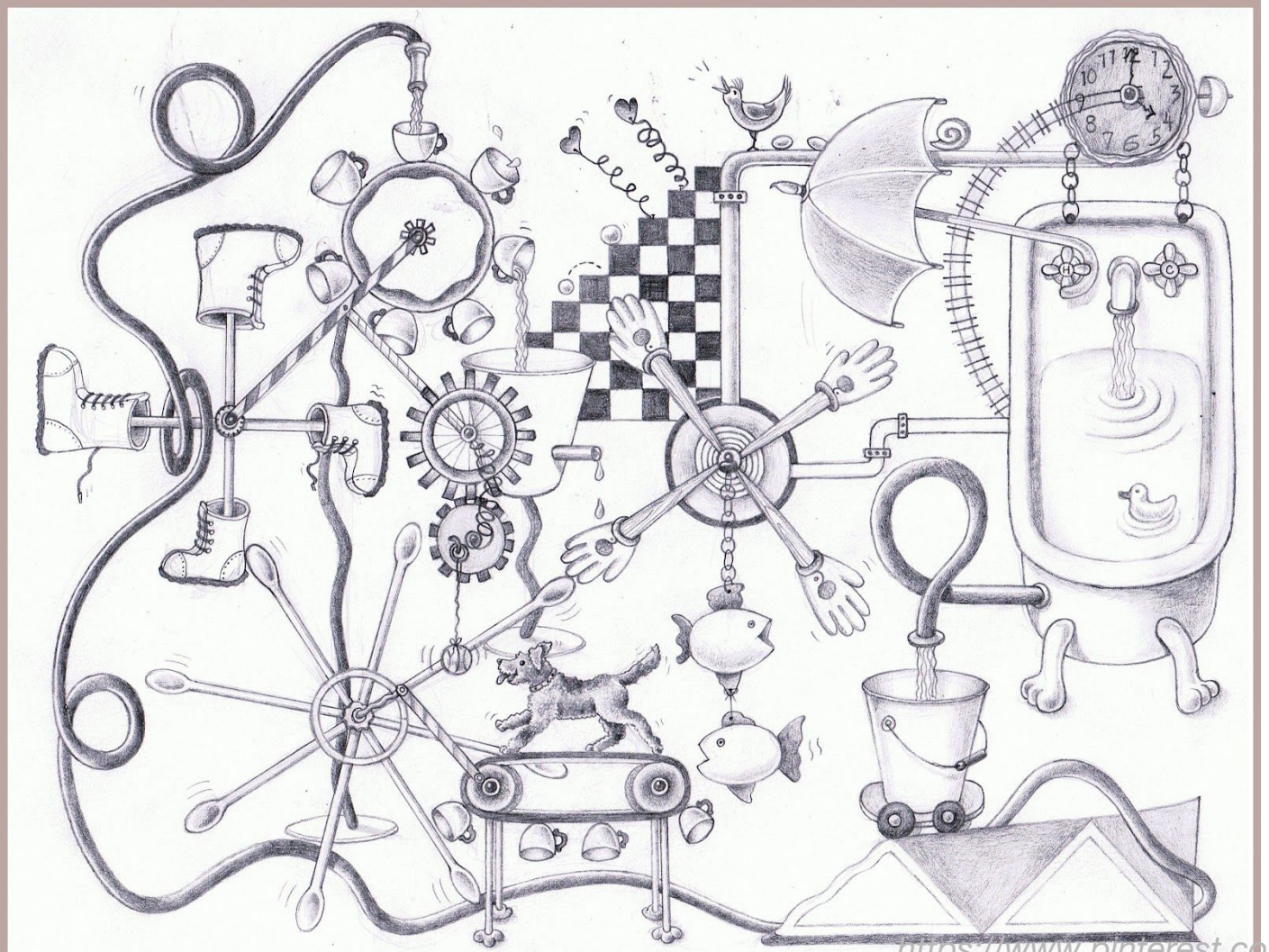
agents

dataflows





http://www.theverge.com/2015/4/22/8381963/rube-goldberg-machine-contest-history-ideas



DEMO

```java
public final class ImmutableJavaPerson {

    private final String name;
    private final Collection<String> tweets;

    public ImmutableJavaPerson(String name, Collection<String> tweets) {
        this.name = name;
        this.tweets = new ArrayList<>(tweets);
    }

    public String getName() {
        return name;
    }

    public Collection<String> getTweets() {
        return unmodifiableCollection(tweets);
    }

```

```
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  }
```

```groovy
@Immutable class ImmutableGroovyPerson {

    String name
    Collection<String> tweets
}
```

```groovy
def thread1 = Thread.start {
    println "Hello from ${Thread.currentThread().name}"
}

def thread2 = Thread.startDaemon {
    println "Hello from ${Thread.currentThread().name}"
}

[ thread1, thread2 ]*.join()
```

```groovy
def process = (['git', 'status']).execute([], new File('.'))
def processOutput = new StringWriter()
process.consumeProcessOutput processOutput, processOutput
process.waitFor()
println processOutput.toString().trim()
```

```
class SynchronizedCounter {

    int atomicCounter
    int counter

    @Synchronized
    int incrementAndGet() {
        atomicCounter = atomicCounter + 1
        return atomicCounter
    }

    @WithReadLock
    int getCounter() {
        counter
    }

    @WithWriteLock
    void increment() {
        counter = counter + 1
    }
}
```

ALL CREDIT CARD PIN CODES IN THE WORLD LEAKED

BY: A GUEST ON SEP 10TH, 2012 | SYNTAX: NONE | SIZE: 49.87 KB | VIEWS: 675,352 | EXPIRES: NEVER

DOWNLOAD | RAW | EMBED | REPORT ABUSE | PRINT | QR CODE | CLONE

USE PASTEBIN A LOT? UPGRADING TO A PRO ACCOUNT UNLOCKS MANY COOL FEATURES.

```
**** LEAK ****
ALL CREDIT CARD PIN CODES IN THE WORLD LEAKED

0000 0001 0002 0003 0004 0005 0006 0007 0008 0009
0010 0011 0012 0013 0014 0015 0016 0017 0018 0019
0020 0021 0022 0023 0024 0025 0026 0027 0028 0029
0030 0031 0032 0033 0034 0035 0036 0037 0038 0039
0040 0041 0042 0043 0044 0045 0046 0047 0048 0049
0050 0051 0052 0053 0054 0055 0056 0057 0058 0059
0060 0061 0062 0063 0064 0065 0066 0067 0068 0069
0070 0071 0072 0073 0074 0075 0076 0077 0078 0079
0080 0081 0082 0083 0084 0085 0086 0087 0088 0089
0090 0091 0092 0093 0094 0095 0096 0097 0098 0099
0100 0101 0102 0103 0104 0105 0106 0107 0108 0109
0110 0111 0112 0113 0114 0115 0116 0117 0118 0119
0120 0121 0122 0123 0124 0125 0126 0127 0128 0129
0130 0131 0132 0133 0134 0135 0136 0137 0138 0139
0140 0141 0142 0143 0144 0145 0146 0147 0148 0149
0150 0151 0152 0153 0154 0155 0156 0157 0158 0159
0160 0161 0162 0163 0164 0165 0166 0167 0168 0169
0170 0171 0172 0173 0174 0175 0176 0177 0178 0179
0180 0181 0182 0183 0184 0185 0186 0187 0188 0189
0190 0191 0192 0193 0194 0195 0196 0197 0198 0199
```

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

def creditCardNumber = "3141 5926 5358 9793"
def expirationDate = "05/17"
def leakedAllCreditCardPinCodes = (0..9999)

withPool {
    leakedAllCreditCardPinCodes.makeConcurrent().each { pinCode ->
        paymentService.pay(creditCardNumber, expirationDate, pinCode)
    }
}
```

why and when use GPars?

why and when do not use GPars?

Further dive