

GPar^s: Unsung Hero of Concurrency in Practice

<http://playstationna.i.lithium.com/t5/image/serverpage/image-id/49228TiDADEAFC04868CE5C?v=1.0>

Focus for this talk

When it comes to **concurrency and parallelism**, first things to appear in someone's mind may be “Java Concurrency in Practice” by Brian Göetz, **threads**, **java.util.concurrent**, **Fork-Join**, **parallel streams**, **reactive**, **Akka** or **MapReduce**.

When it comes to **Groovy**, first things to appear in someone's mind may be **Gradle**, **Grails**, **Spock**, **DSLs** or **scripting**.



HOW TO STAY
FOCUSED

Focus for this talk

Great injustice is that you rarely meet **GPars** in both these lists.

Framework that provides **high-level APIs and DSLs for writing concurrent and parallel code both in Java and Groovy**

and support for concepts of **map/reduce, fork/join, asynchronous code, actors, agents, dataflows (not all mentioned)** deserves a little more attention, isn't it?

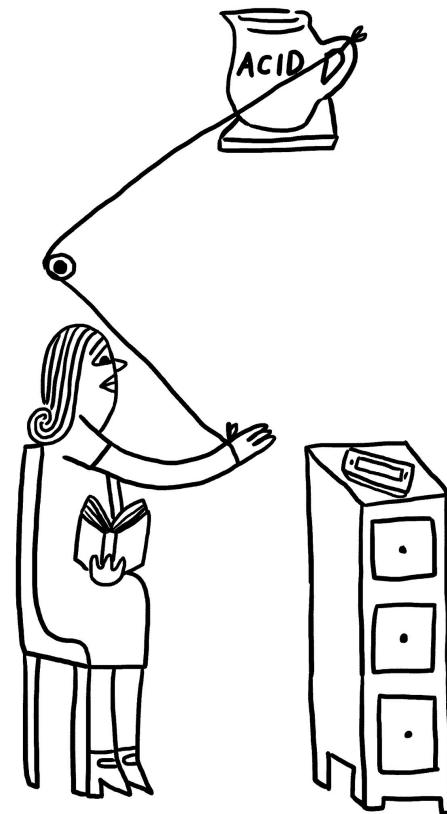


HOW TO STAY
FOCUSED

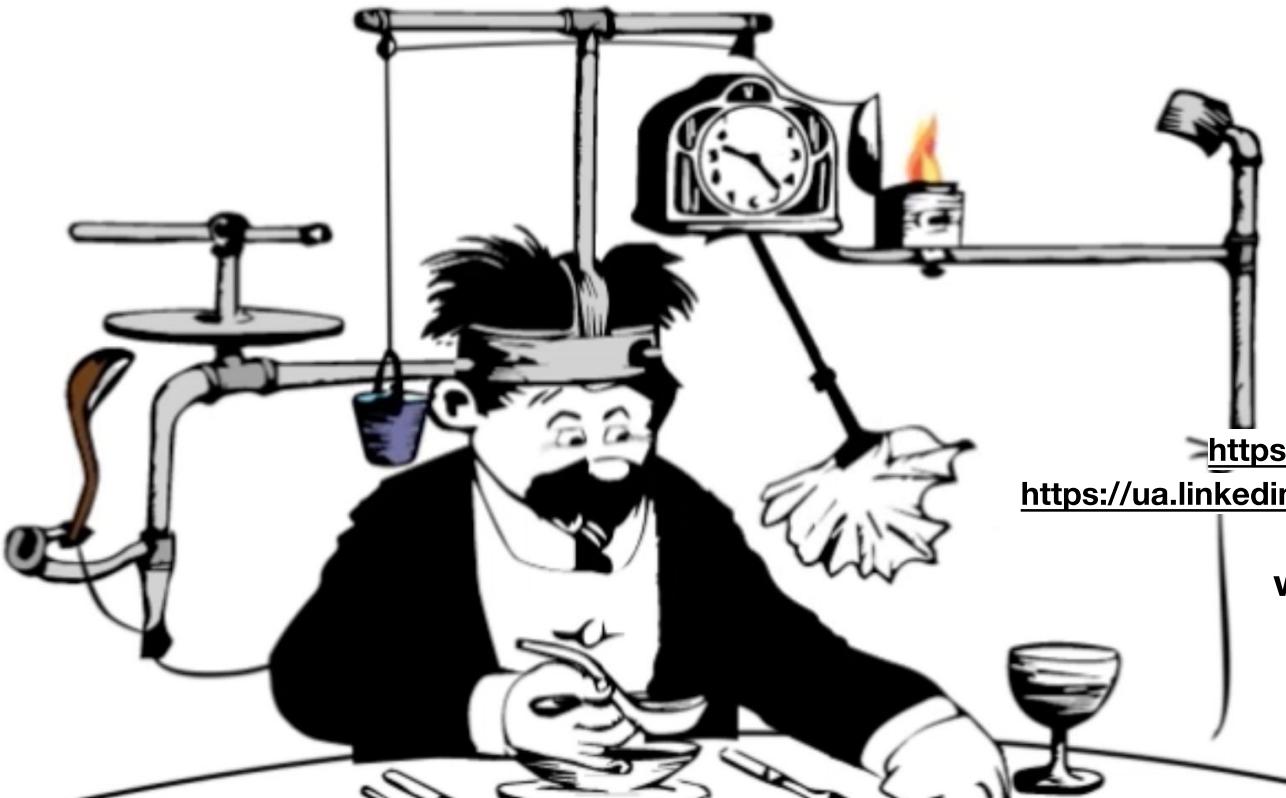
Focus for this talk

In this talk we will try to fix it. One by one, **we will explore various use cases of GPars** with all its pragmatism and conciseness.

Not forgetting neither plain Java nor Groovy adepts, we will use Groovy to empower our solutions and ensure that everything works from Java the same way.



HOW TO STAY
FOCUSED



Yaroslav Yermilov

Senior Software Engineer
EPAM Systems

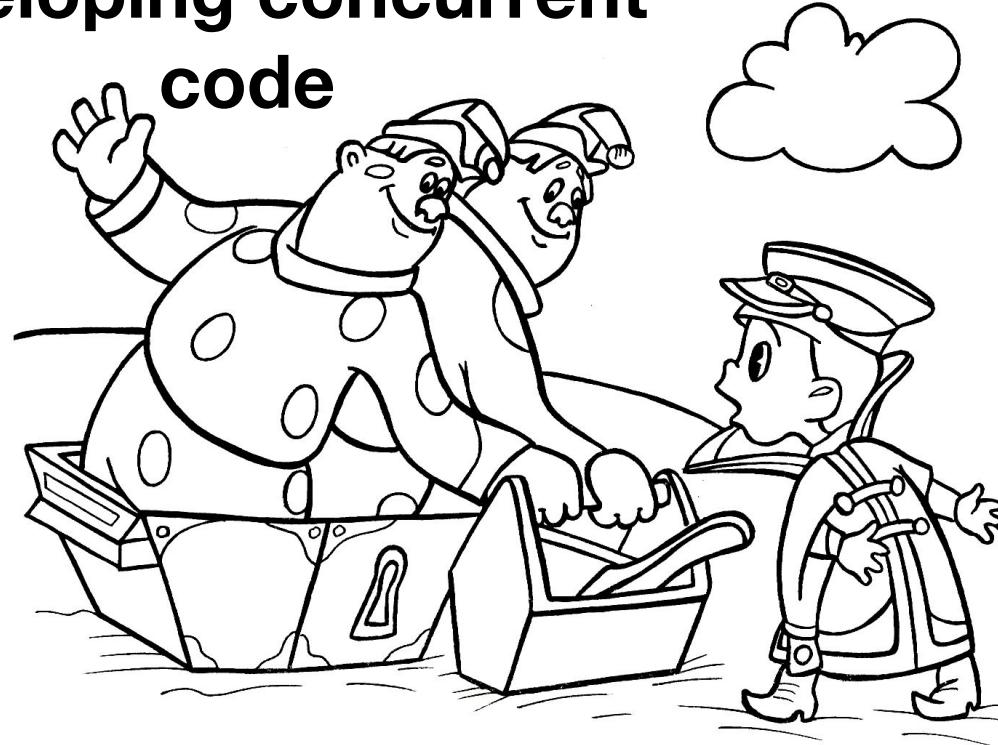
yermilov17@gmail.com
<https://yermilov.github.io/>
<https://twitter.com/yermilov17>
<https://www.facebook.com/yaroslav.yermilov>
<https://ua.linkedin.com/pub/yaroslav-yermilov/58/682/506>

work for EPAM Systems since 2011
distributed systems,
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

two guidelines for developing concurrent code



guideline #2 for developing concurrent code

MURPHY'S LAW:

Anything that can go wrong will go wrong.



```
public class Holder {  
    private int n;  
  
    public Holder(int n) { this.n = n; }  
  
    public void assertSanity() {  
        if (n != n) {  
            throw new AssertionError("Even it can go wrong!");  
        }  
    }  
  
    public class Initializer {  
        public Holder holder;  
  
        public void init() { holder = new Holder(42); }  
    }  
}
```



Holder was not
properly
published!

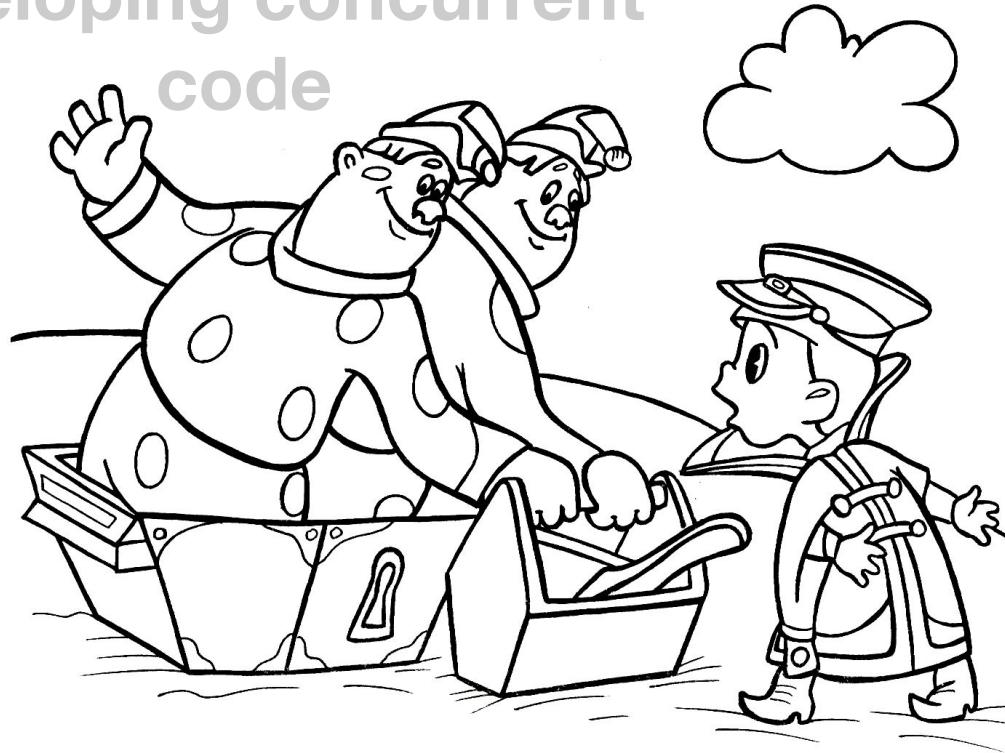
guideline #1 for developing concurrent code

MOORE'S LAW:

Computers will get exponentially faster.



two guidelines for
developing concurrent
code



Java toolbox

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!),
*///:-
```

```
    print(this + " completed");
}
public String toString() {
    return String.format("%12s", name);
}

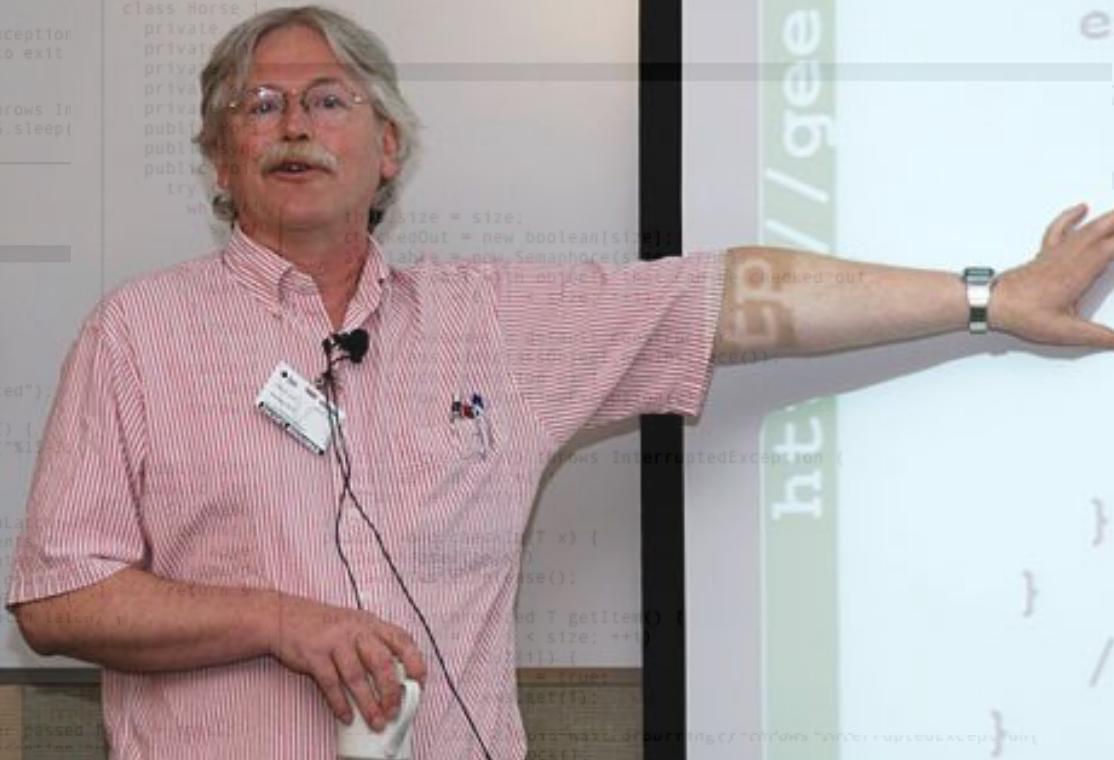
// Waits on the CountDownLatch
class WaitingTask implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final CountDownLatch latch;
    WaitingTask(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            latch.await();
            print("Latch barrier passed " + id);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

ht

```
//: concurrency/Pool.java
// Using a Semaphore inside a Pool, to restrict
// the number of tasks that can use a resource.
import java.util.concurrent.*;
import java.util.*;

public class Pool<T> {
    private int size;
    private List<T> items = new ArrayList<T>();
    private volatile boolean[] checkedOut;
    private Semaphore available;
    public Pool(Class<T> classObject, int size) {
```

Thinking in Java



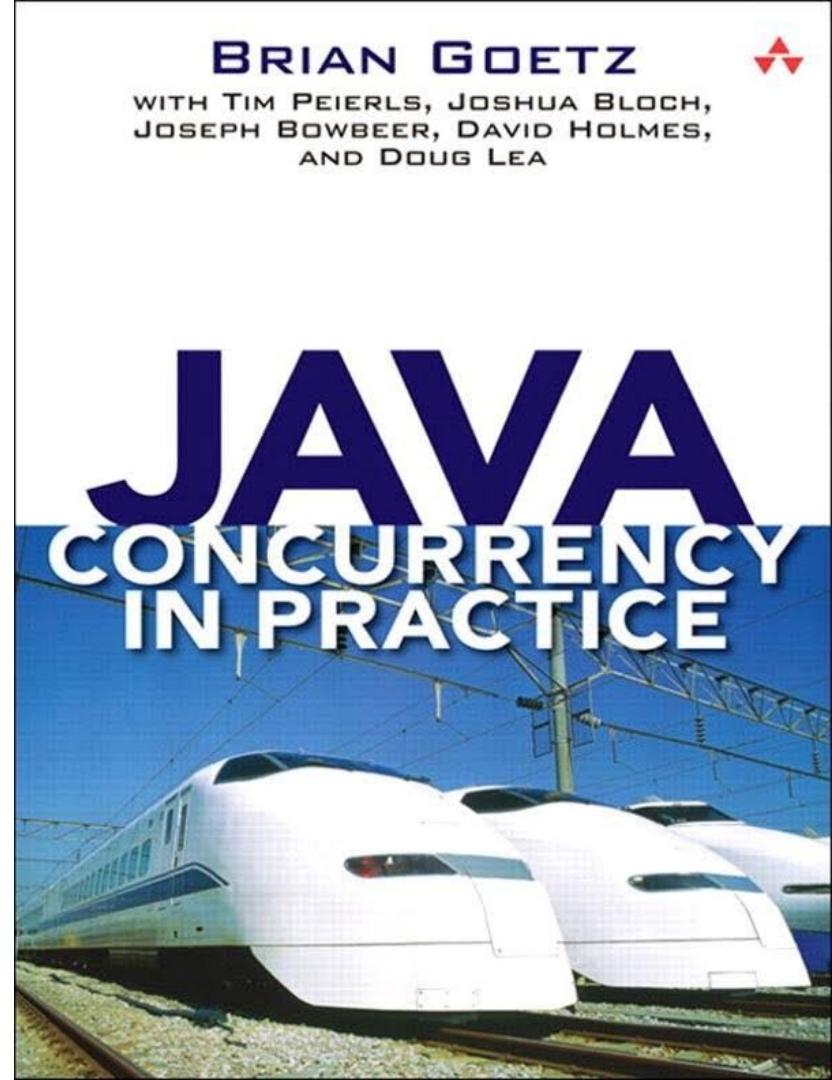
two guidelines for
developing concurrent
code



Java toolbox

If multiple threads access the same mutable state variable without appropriate synchronization, your program is broken.

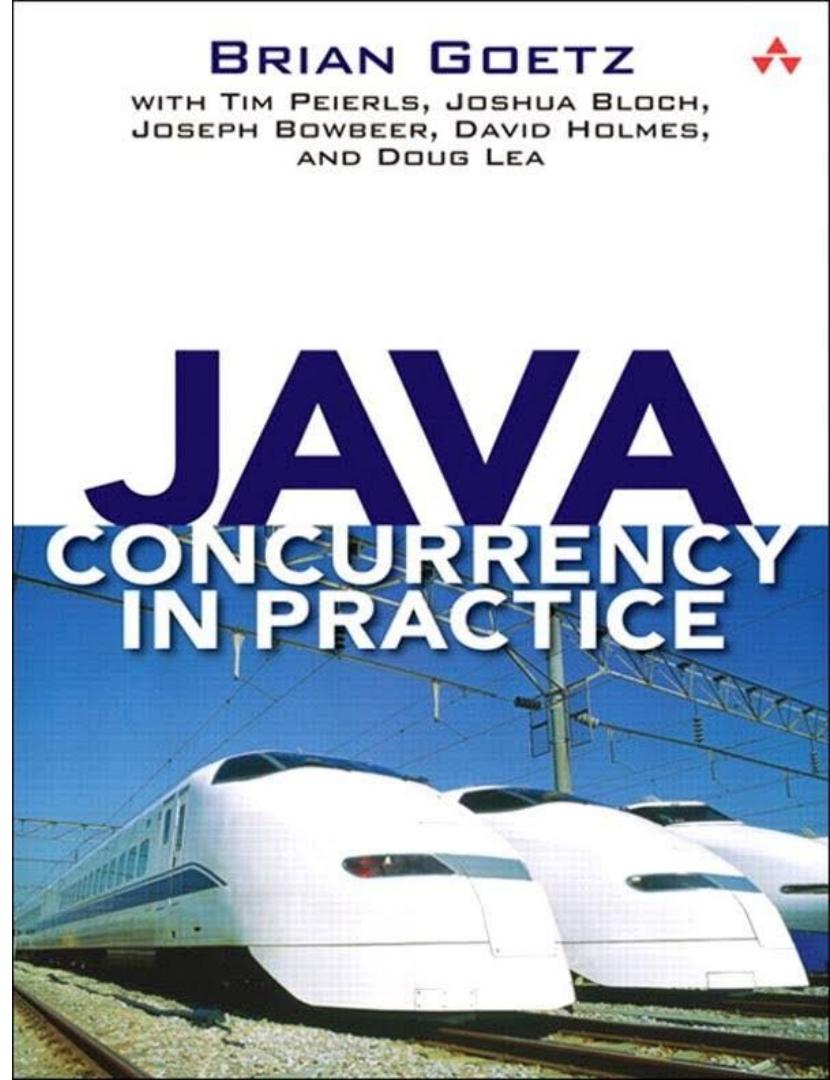
There are three ways to fix it:



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



BRIAN GOETZ



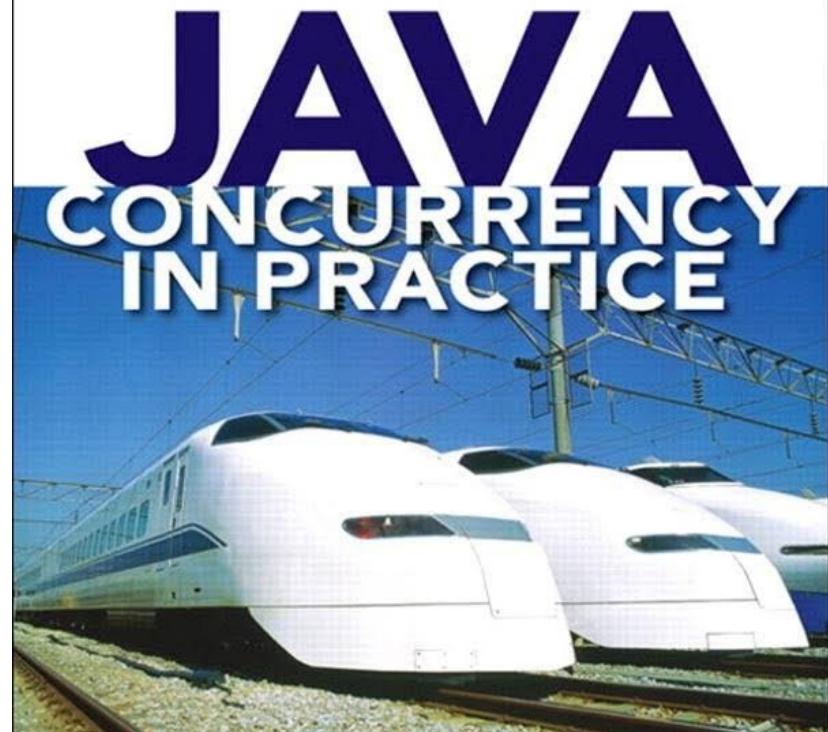
WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA

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



BRIAN GOETZ



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

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

Use synchronization whenever accessing the state variable

JAVA CONCURRENCY IN PRACTICE

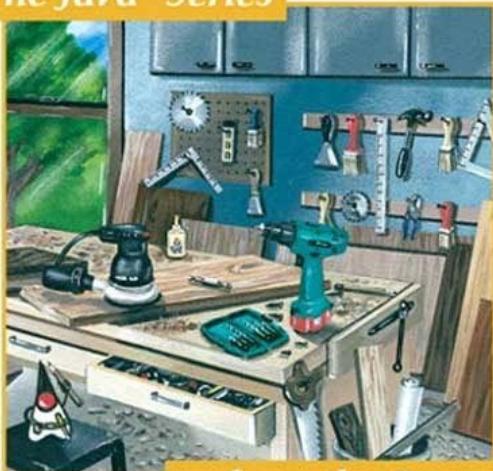


Joshua Bloch

Revised and
Updated for
Java SE 6

Effective Java™ Second Edition

The Java™ Series



...from the Source

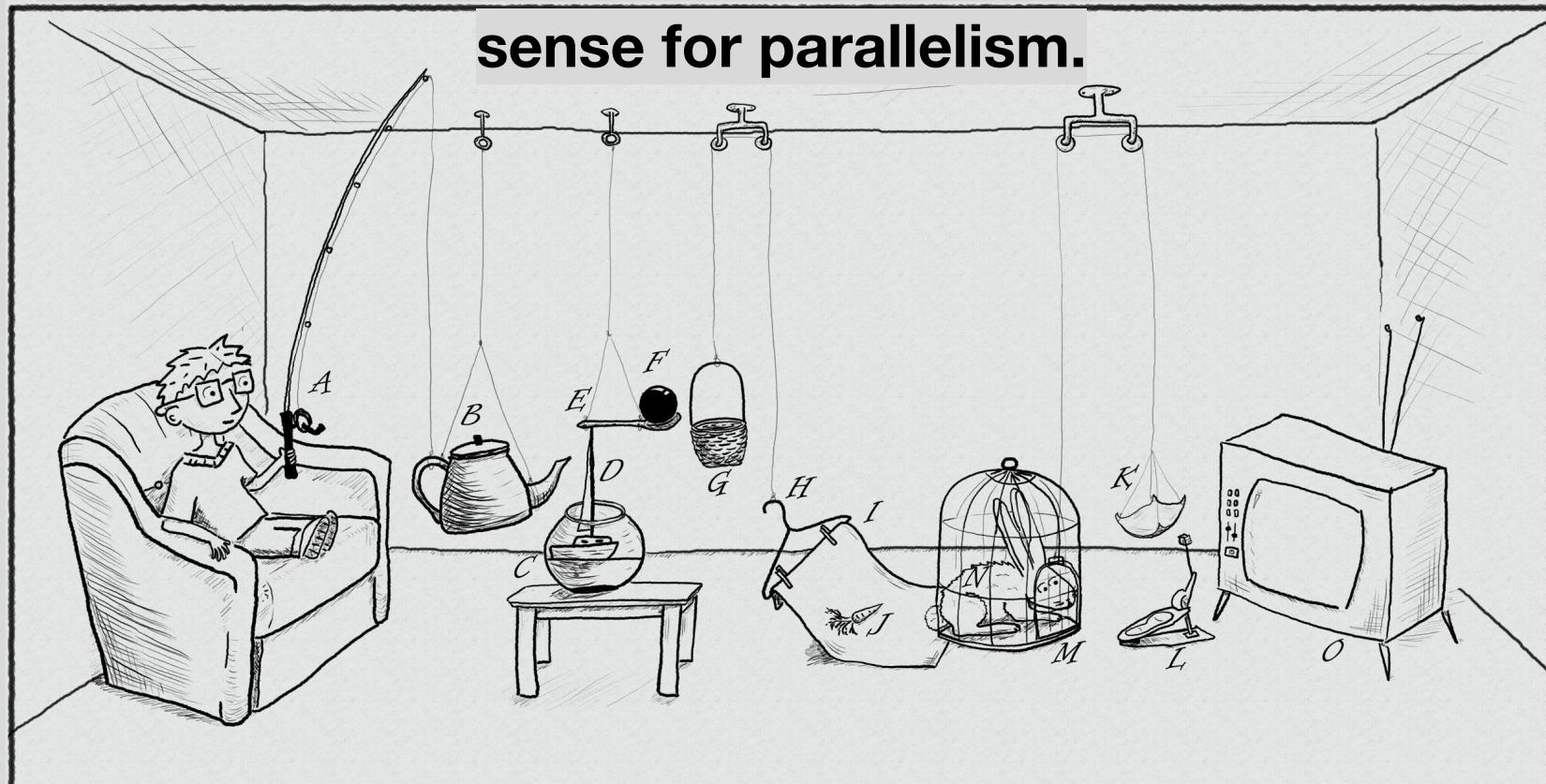


10 Concurrency.....	259
Item 66: Synchronize access to shared mutable data.....	259
Item 67: Avoid excessive synchronization	265
Item 68: Prefer executors and tasks to threads.....	271
Item 69: Prefer concurrency utilities to <code>wait</code> and <code>notify</code>	273

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.



```
public class MergeSortTask<T extends Comparable<T>> extends RecursiveTask<List<T>> {
    @Override
    protected List<T> compute() {
        if (list.size() < 2) { return list; }
        if (list.size() == 2) {
            if (list.get(0).compareTo(list.get(1)) != 1) {
                return list;
            } else {
                return asList(list.get(1), list.get(0));
            }
        }
        MergeSortTask<T> leftTask = new MergeSortTask<>(list.subList(0, list.size() / 2));
        MergeSortTask<T> rightTask = new MergeSortTask<>(list.subList(list.size() / 2,
list.size()));
        leftTask.fork(); rightTask.fork();
        List<T> left = leftTask.join();
        List<T> right = rightTask.join();
        return merge(left, right);
    }
}
```



```

public class MergeSortTask<T extends Comparable<T>> extends RecursiveTask<List<T>> {
    @Override
    protected List<T> compute() {
        if (list.size() < 2) { return list; }
        if (list.size() == 2) {
            if (list.get(0).compareTo(list.get(1)) != 1) {
                return list;
            } else {
                return asList(list.get(1), list.get(0));
            }
        }
        MergeSortTask<T> leftTask = new MergeSortTask<>(list.subList(0, list.size() / 2));
        MergeSortTask<T> rightTask = new MergeSortTask<>(list.subList(list.size() / 2,
list.size()));
        leftTask.fork(); rightTask.fork();
        List<T> left = leftTask.join();
        List<T> right = rightTask.join();
        return merge(left, right);
    }
}

```



```
public class MergeSortTask<T extends Comparable<T>> extends RecursiveTask<List<T>> {
    @Override
    protected List<T> compute() {
        if (list.size() < 2) { return list; }
        if (list.size() == 2) {
            if (list.get(0).compareTo(list.get(1)) != 1) {
                return list;
            } else {
                return asList(list.get(1), list.get(0));
            }
        }
        MergeSortTask<T> leftTask = new MergeSortTask<>(list.subList(0, list.size() / 2));
        MergeSortTask<T> rightTask = new MergeSortTask<>(list.subList(list.size() / 2,
list.size()));
        leftTask.fork(); rightTask.fork();
        List<T> left = leftTask.join();
        List<T> right = rightTask.join();
        return merge(left, right);
    }
}
```



```

public class MergeSortTask<T extends Comparable<T>> extends RecursiveTask<List<T>> {
    @Override
    protected List<T> compute() {
        if (list.size() < 2) { return list; }
        if (list.size() == 2) {
            if (list.get(0).compareTo(list.get(1)) != 1) {
                return list;
            } else {
                return asList(list.get(1), list.get(0));
            }
        }
        MergeSortTask<T> leftTask = new MergeSortTask<>(list.subList(0, list.size() / 2));
        MergeSortTask<T> rightTask = new MergeSortTask<>(list.subList(list.size() / 2,
list.size()));
        leftTask.fork(); rightTask.fork();
        List<T> left = leftTask.join();
        List<T> right = rightTask.join();
        return merge(left, right);
    }
}

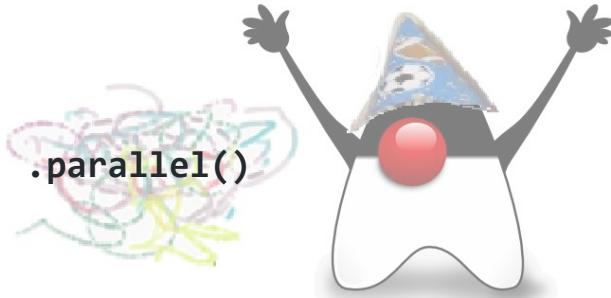
```



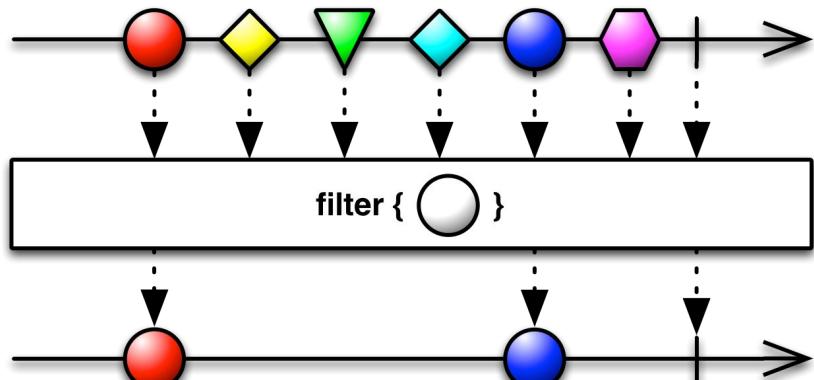
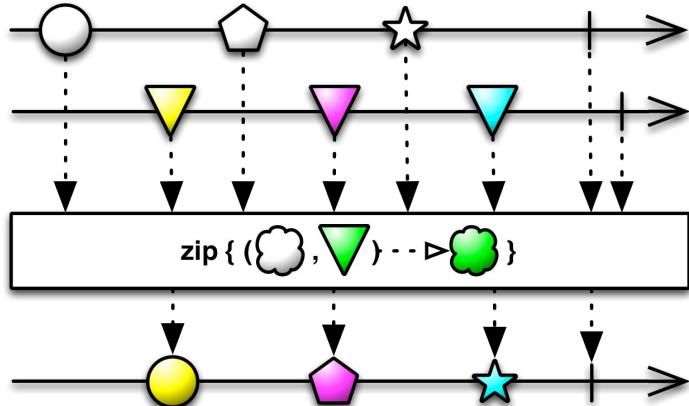
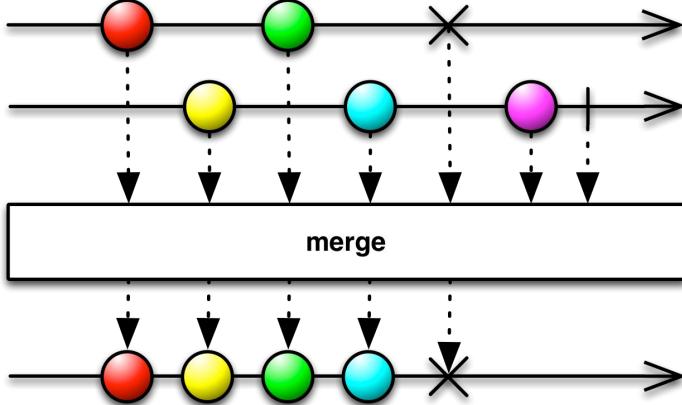
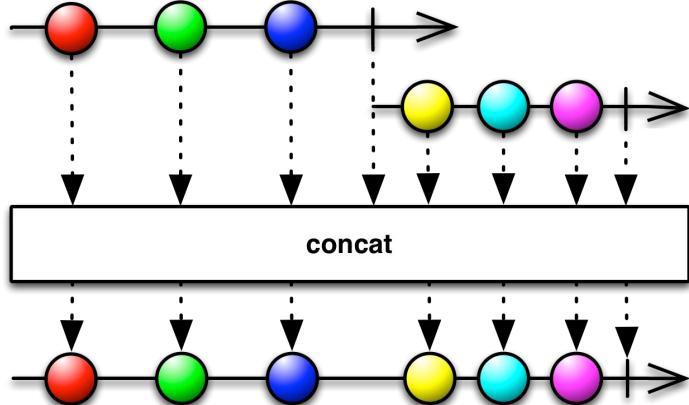
```
public class MergeSortTask<T extends Comparable<T>> extends RecursiveTask<List<T>> {
    @Override
    protected List<T> compute() {
        if (list.size() < 2) { return list; }
        if (list.size() == 2) {
            if (list.get(0).compareTo(list.get(1)) != 1) {
                return list;
            } else {
                return asList(list.get(1), list.get(0));
            }
        }
        MergeSortTask<T> leftTask = new MergeSortTask<>(list.subList(0, list.size() / 2));
        MergeSortTask<T> rightTask = new MergeSortTask<>(list.subList(list.size() / 2,
list.size()));
        leftTask.fork(); rightTask.fork();
        List<T> left = leftTask.join();
        List<T> right = rightTask.join();
        return merge(left, right);
    }
}
```



```
Optional<Status> mostPopularTweet = tweets.stream()
```



```
.parallel()  
.filter(tweet -> tweet.getText().toLowerCase().contains(topic.toLowerCase()))  
.filter(tweet -> !tweet.isRetweet())  
.max(comparingInt(tweet -> tweet.getFavoriteCount() + tweet.getRetweetCount()));
```



Next you create the resource controller that will serve these greetings.

Create a resource controller

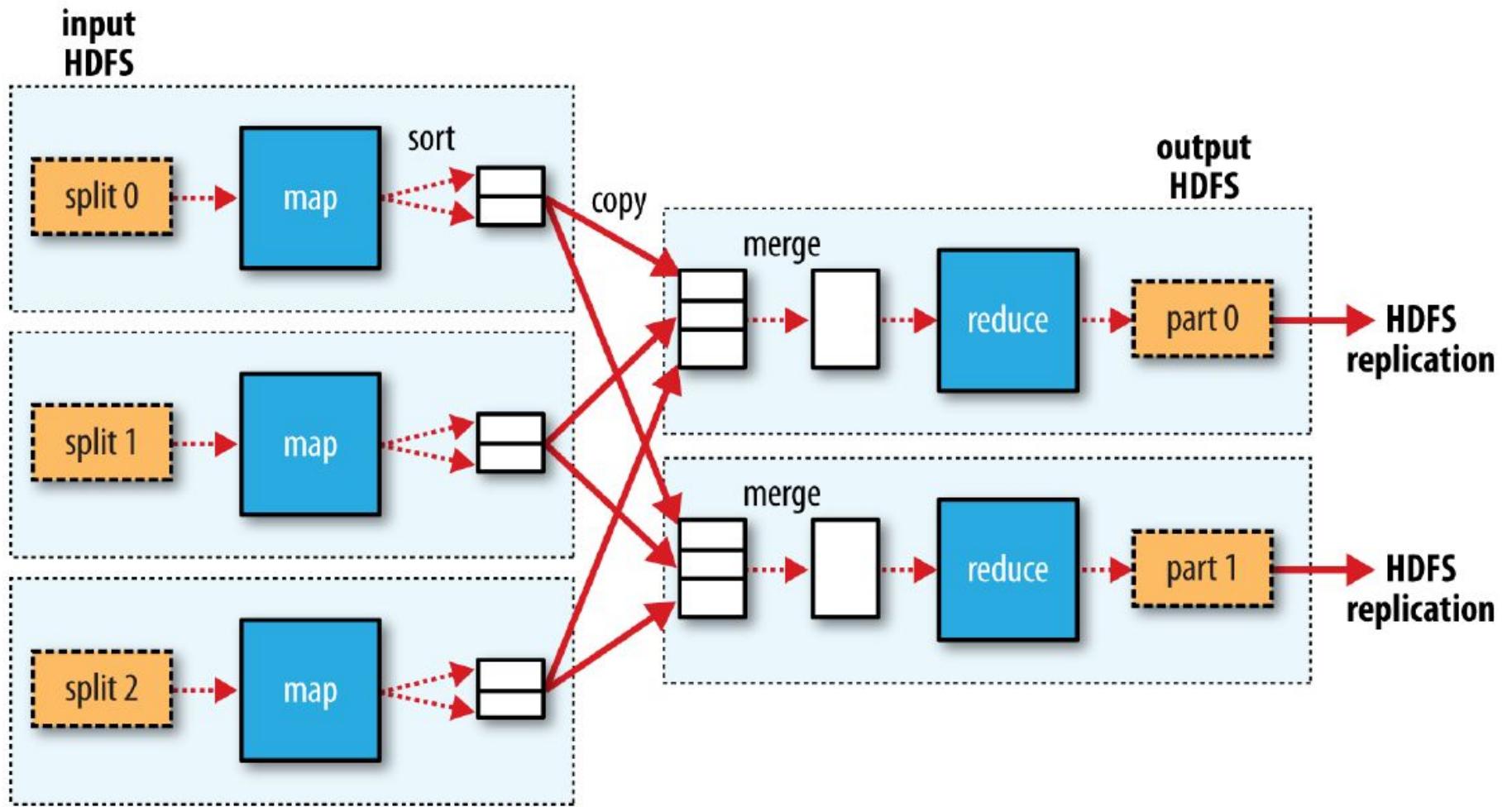
In Spring's approach to building RESTful web services, HTTP requests are handled by a controller. These components are easily identified by the `@RestController` annotation, and the `GreetingController` below handles `GET` requests for `/greeting` by returning a new instance of the `Greeting` class:

`src/main/java/hello/GreetingController.java`

```
@RestController
public class GreetingController {

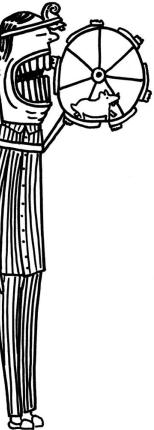
    private static final String template = "Hello, %s!";
    private final AtomicLong counter = new AtomicLong();

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





GPars



HOW TO BRUSH
YOUR TEETH



HOW TO UTILIZE YOUR
DOG'S MINDLESS JOY

data parallelism

map/reduce

fork/join

**asynchronous
execution**

actors

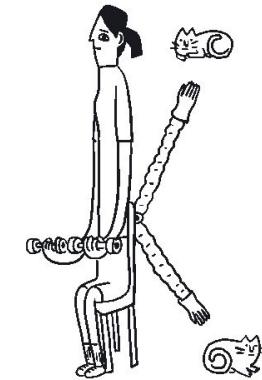
agents

dataflows

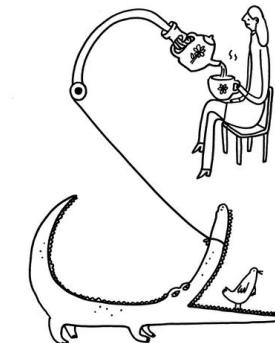
remote execution

**Communicating
Sequential
Processes**

**Software
Transactional
Memory**

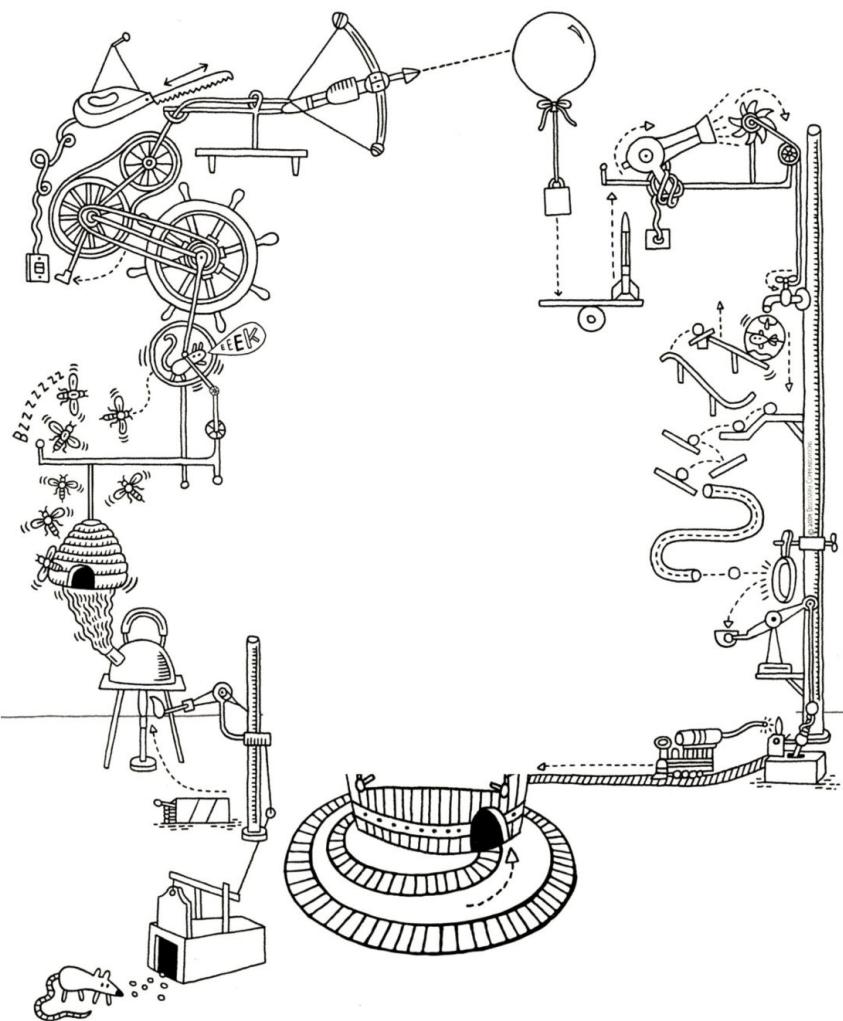


HOW TO PET YOUR PETS
WHILE DOING YOUR REPS



HOW TO SERVE TEA

JAVA OR GROOVY?



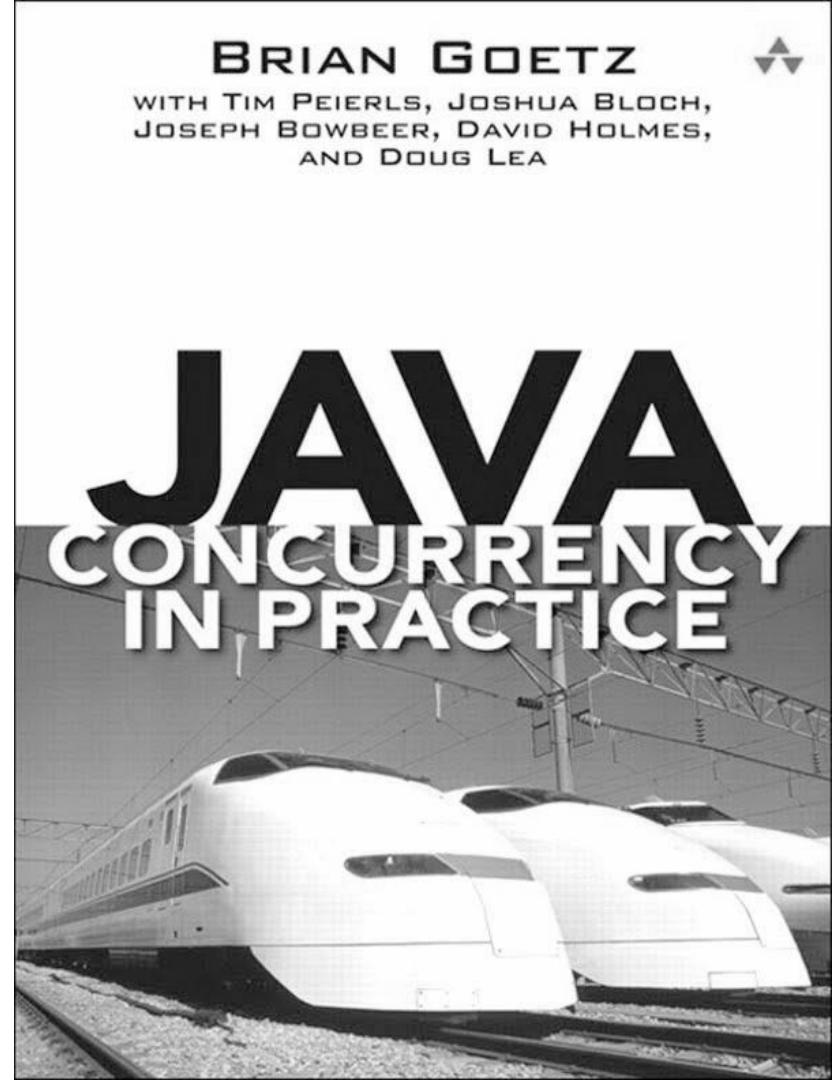
If multiple threads access the same mutable state variable without appropriate synchronization, your program is broken.

There are three ways to fix it:

Don't share the state variable across threads

Make the state variable immutable

Use synchronization whenever accessing the state variable



```
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);  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;
```

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    ImmutableJavaPerson that = (ImmutableJavaPerson) o;

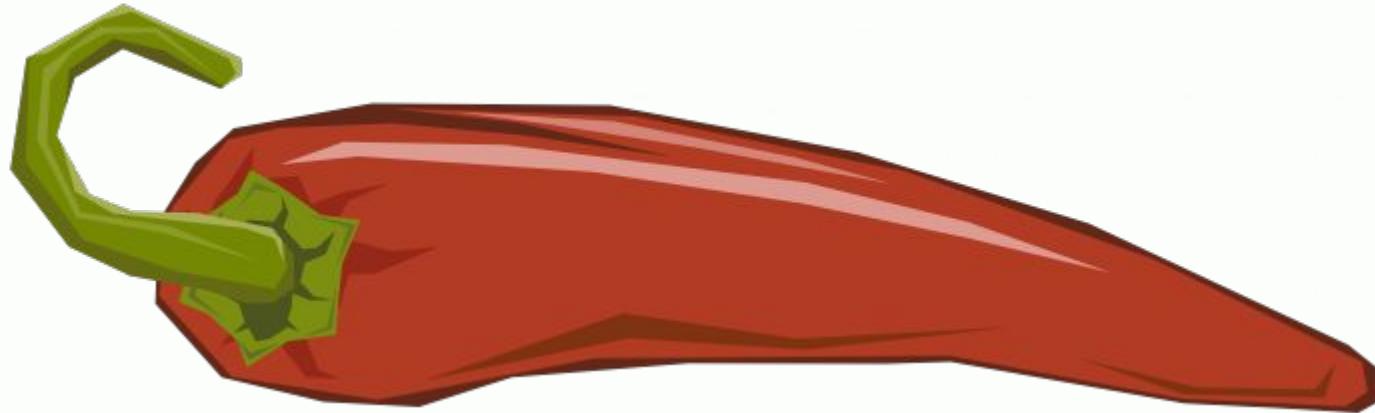
    if (name != null ? !name.equals(that.name) : that.name != null) return false;
    return tweets != null ? tweets.equals(that.tweets) : that.tweets == null;
}

@Override
public int hashCode() {
    int result = name != null ? name.hashCode() : 0;
    result = 31 * result + (tweets != null ? tweets.hashCode() : 0);
    return result;
}
```

```
@Immutable class ImmutableGroovyPerson {  
  
    String name  
    Collection<String> tweets  
}
```

CLICK TO PLACE
YOUR
AD
HERE!

```
@Immutable class ImmutableGroovyPerson {  
  
    String name  
    Collection<String> tweets  
}
```



```
class SynchronizedCounter {  
  
    int atomicCounter  
    int counter  
  
    @Synchronized  
    int incrementAndGet() {  
        atomicCounter = atomicCounter + 1  
        return atomicCounter  
    }  
  
    @WithReadLock  
    int value() {  
        counter  
    }  
  
    @WithWriteLock  
    void increment() {  
        counter = counter + 1  
    }  
}
```

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

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

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

```
def process =(['git', 'status']).execute([], new File('.'))

def processOutput = new StringWriter()

process.consumeProcessOutput processOutput, processOutput

process.waitFor()

println processOutput.toString().trim()
```

```
def config = new CompilerConfiguration()
config.addCompilationCustomizers(new ASTTransformationCustomizer(ThreadInterrupt))
def binding = new Binding(i:0)
def shell = new GroovyShell(binding,config)
def t = Thread.start {
    shell.evaluate(userCode)
}
t.join(1000)
if (t.alive) {
    t.interrupt()
}

@TimedInterrupt(value=1, unit=TimeUnit.SECONDS)
class FibCalculator {

    def fib(int n) {
        n < 2 ? n : fib(n - 1) + fib(n - 2)
    }
}

@ConditionalInterrupt({ Quotas.disallow('user') })
class UserService {

    void longRunningRequest() { ... }
}
```

```
"Hello from ${Thread.currentThread().name}"                                '''Multi-line
                                                                     strings'''  
  
def numbers = [ 1, 2, 3 ]  
def colors = [ red: '#ff0000', green: '#00ff00', blue: '#0000ff' ]  
  
carNames = car*.name  
Clousure sum = { a, b -> a + b }  
Clouse increment = sum.curry(1)  
  
name = user?.name  
displayName = user.name ?: 'unknown'  
assert new String('text') == new String('text')  
assert (2 <= 5) == -1  
  
Object str1 = 'text'  
def str2 = str1  
assert str1.length() == str2.length()  
  
def sneakyThrow() { throw new IOException("i don't care") }  
  
@IgnoreIf({ os.linux || jdk.version == 8 })
```

```
def rows = Sql.newInstance(url, user, password, driver).rows "SELECT * FROM table"

(0..<5).collect { it % 2 == 0 }
println list.find { it > 7 }
if (4 in nums) println 'yes'
println persons.max{ person -> person.age })
5.times { println 'hello!' }
println list[0..2]

new File('poem.txt').eachLine { line ->
    println line
}

def increment(int number, int delta = 0) { ... }
def connectJdbc(Map params) { ... }
connectJdbc(url: 'jdbc://', driver: 'generic', username: 'admin', password: 'admin')

def json = new JsonSlurper().parseText(jsonText)
json.clients[3].location.country.code

def xml = new XmlSlurper().parseText(xmlText)
xml.clients[3].location.country.@code
xmlBuilder.root(attribute: 28) {
    elem1('hello')
    elem2('xml')
}
```

```
@Grab(group='org.springframework', module='spring-orm', version='4.3.8.RELEASE')
import org.springframework.jdbc.core.JdbcTemplate

@TypeChecked
@CompileStatic
@Canonical
@InheritConstructors
@Lazy
@Sortable
@Builder
@Delegate
@Memoized
@Singleton
@Log

class MetaProgramming {
    def methodMissing(String methodName, def methodArgs) { ... }
}

Number.metaClass.doubleIt = { -> delegate * 2 }
ReturnStatement returnStatement = macro { return "42" }

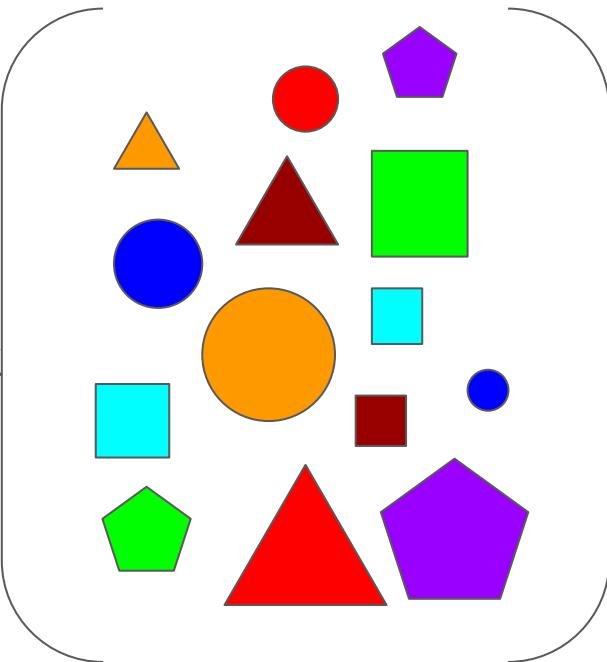
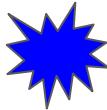
class MyNumber {
    MyNumber plus(MyNumber other) { ... }
}

trait DatabaseTest { ... }

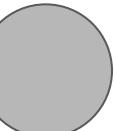
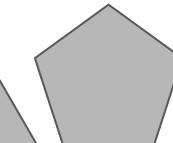
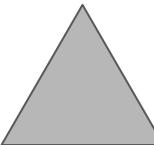
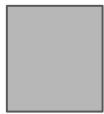
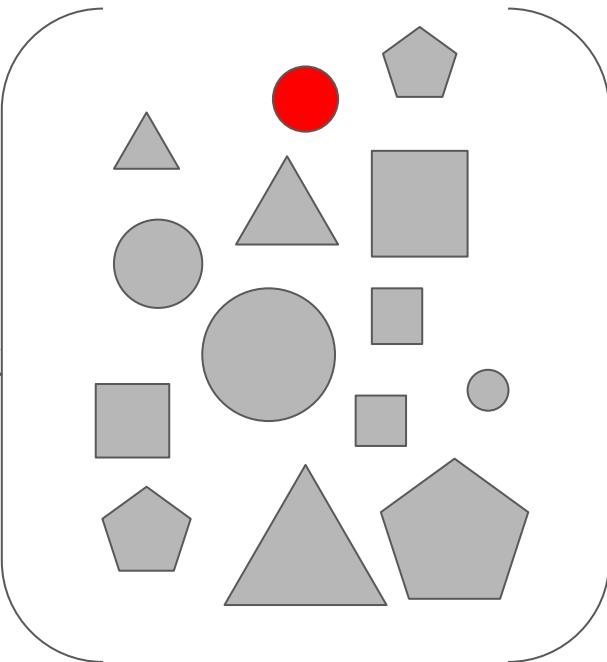
please show the square_root of 100
```

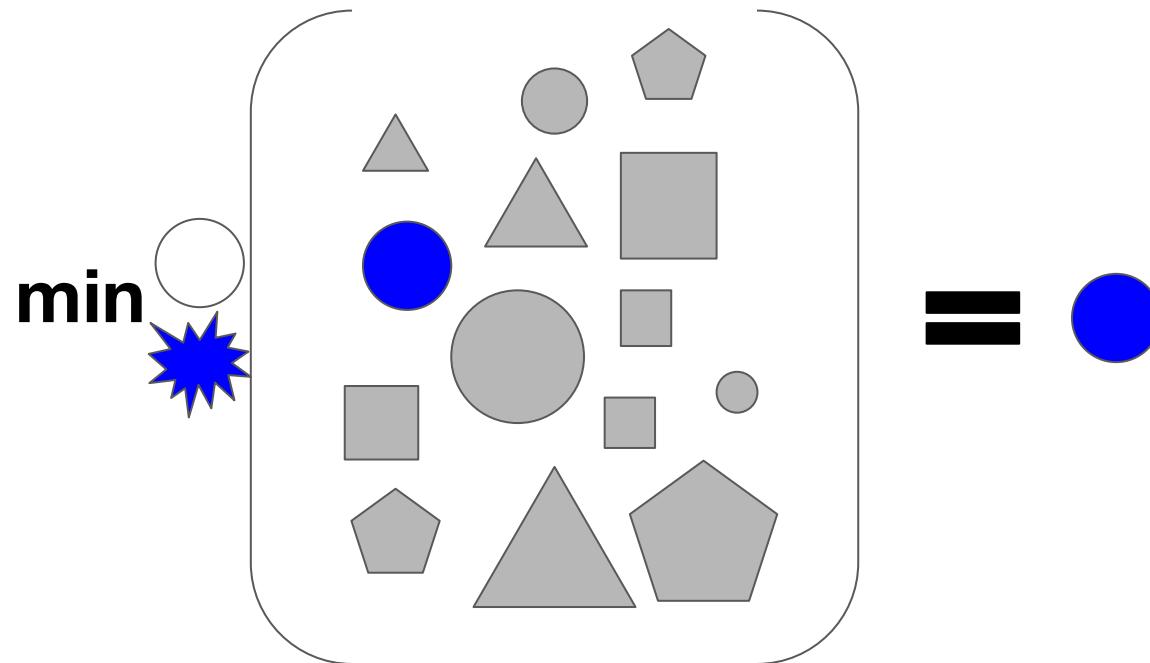
```
new GroovyShell(variables).evaluate 'println persons.sort { person -> person.name }'
```

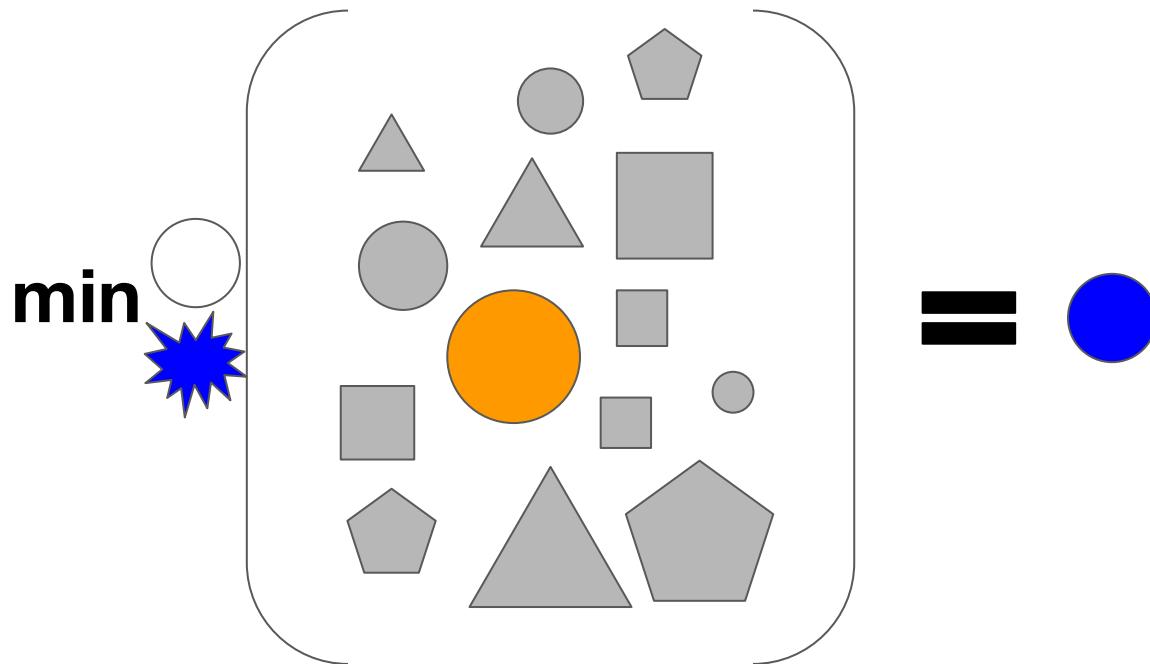
min

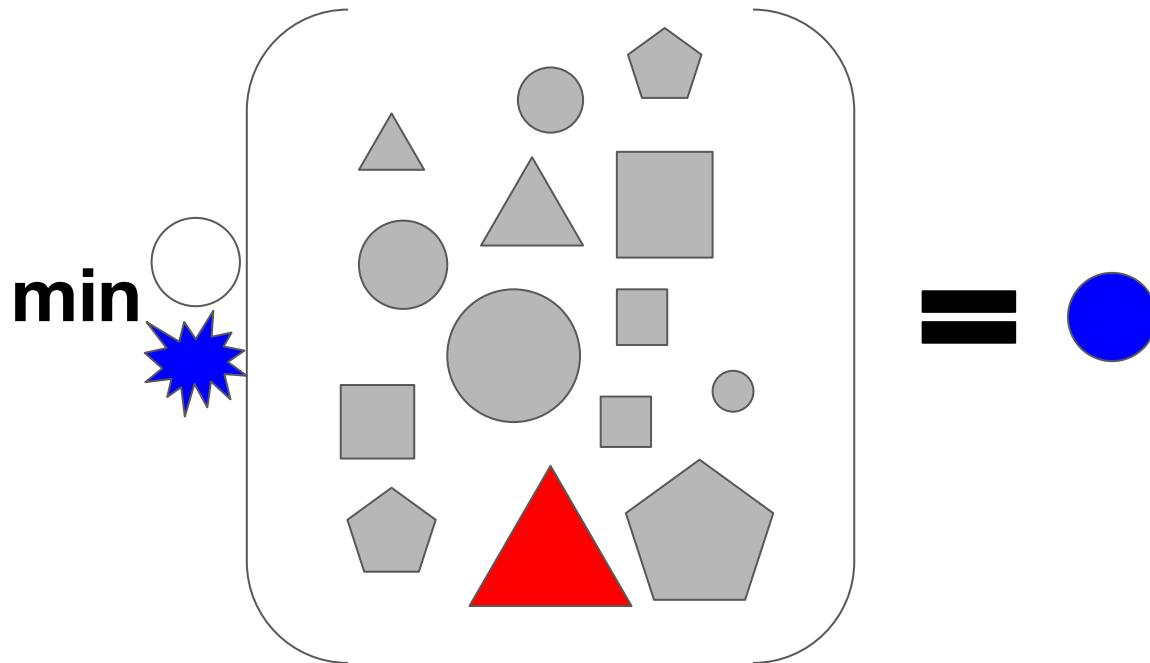


min

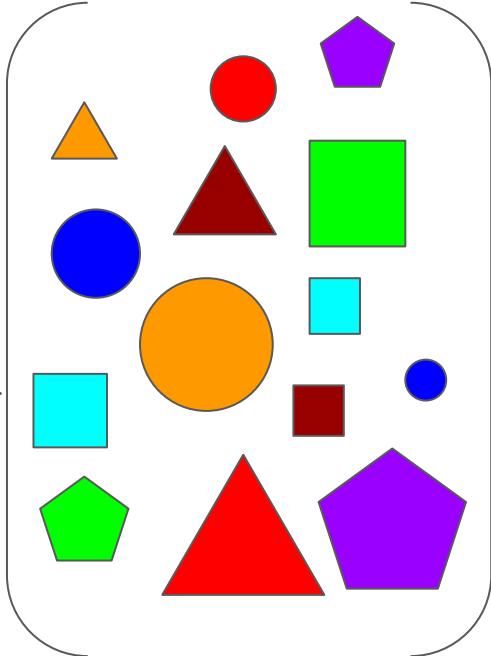
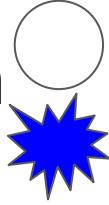




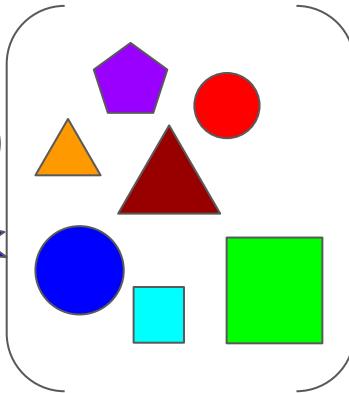




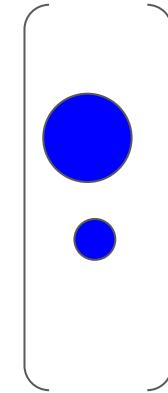
min



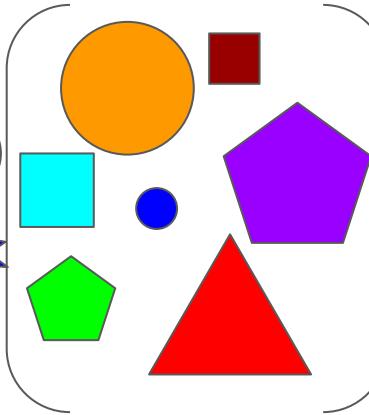
min



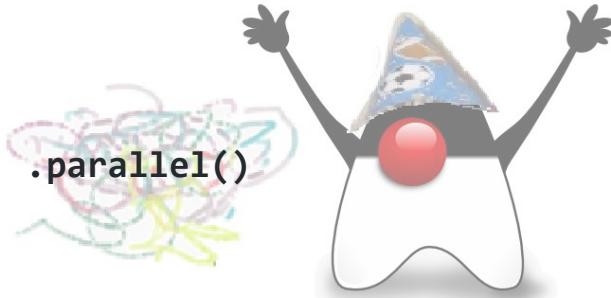
min



min



```
Optional<Status> mostPopularTweet = tweets.stream()
```



```
.parallel()  
.filter(tweet -> tweet.getText().toLowerCase().contains(topic.toLowerCase()))  
.filter(tweet -> !tweet.isRetweet())  
.max(comparingInt(tweet -> tweet.getFavoriteCount() + tweet.getRetweetCount()));
```

```
shapes.stream()
```

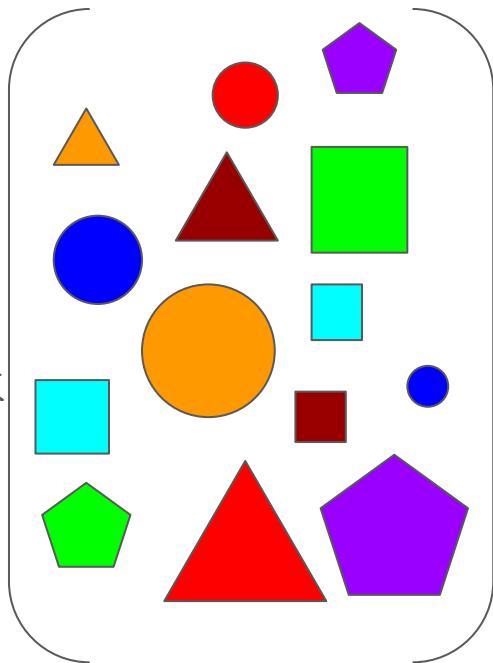
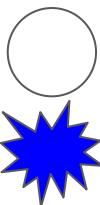
```
Optional<Shape> minBlueCircle = shapes.stream()  
    .filter(shape -> shape.getType() == CIRCLE)
```

```
Optional<Shape> minBlueCircle = shapes.stream()  
    .filter(shape -> shape.getType() == CIRCLE)  
    .filter(shape -> shape.getColor() == BLUE)
```

```
Optional<Shape> minBlueCircle = shapes.stream()  
        .filter(shape -> shape.getType() == CIRCLE)  
        .filter(shape -> shape.getColor() == BLUE)  
        .min(comparingInt(shape -> shape.getSize()));
```

```
Optional<Shape> minBlueCircle = shapes.stream()  
    .parallel()  
    .filter(shape -> shape.getType() == CIRCLE)  
    .filter(shape -> shape.getColor() == BLUE)  
    .min(comparingInt(shape -> shape.getSize()));
```

min



```
.filter(shape -> shape.getType() == CIRCLE)  
.filter(shape -> shape.getColor() == BLUE)  
.min(comparingInt(shape -> shape.getSize()));
```



shapes

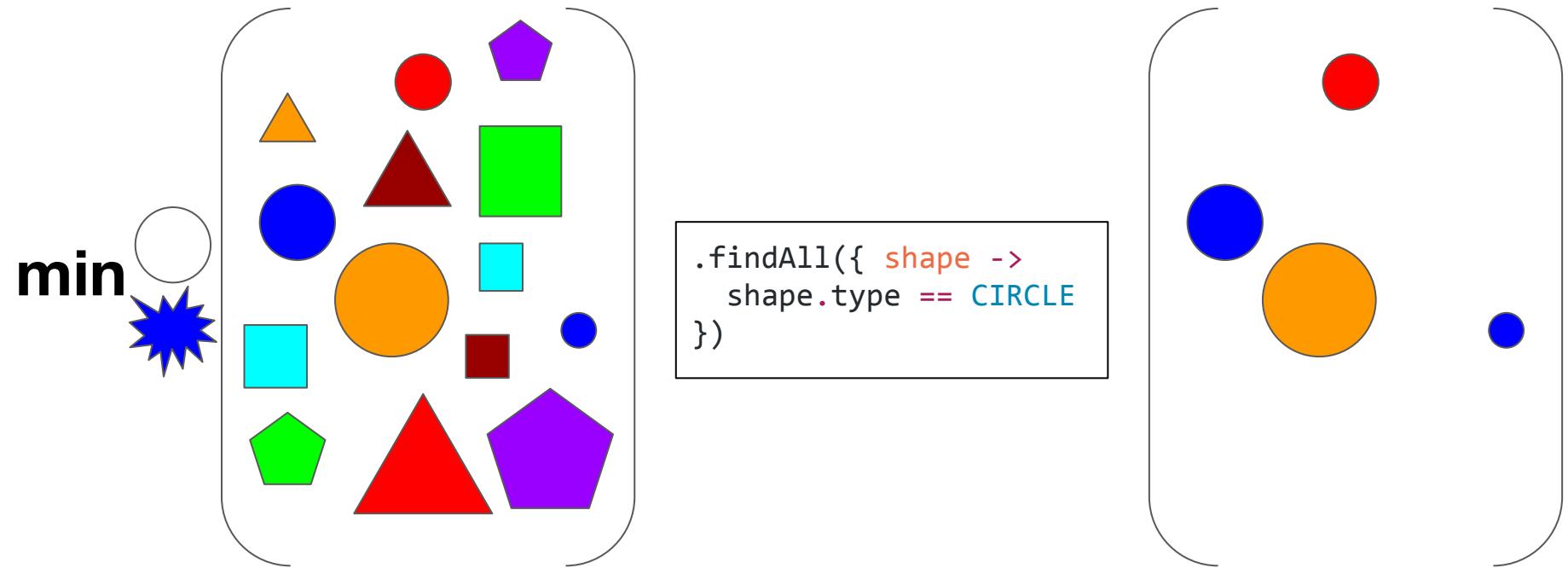
```
Shape minBlueCircle = shapes
    .findAll({ shape -> shape.type == CIRCLE })
```

```
Shape minBlueCircle = shapes
    .findAll({ shape -> shape.type == CIRCLE })
    .findAll({ shape -> shape.color == BLUE })
```

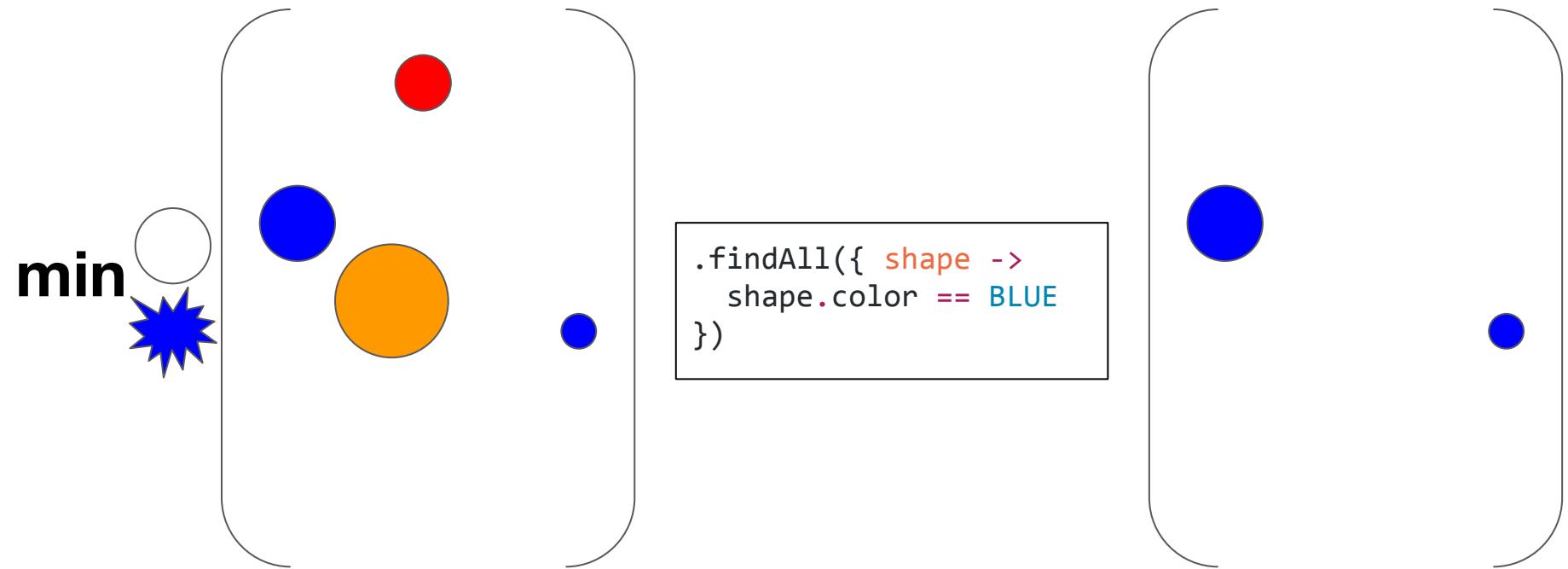
```
Shape minBlueCircle = shapes
    .findAll({ shape -> shape.type == CIRCLE })
    .findAll({ shape -> shape.color == BLUE })
    .min({ shape -> shape.size })
```

```
ParallelEnhancer.enhanceInstance shapes
Shape minBlueCircle = shapes
    .findAllParallel({ shape -> shape.type == CIRCLE })
    .findAllParallel({ shape -> shape.color == BLUE })
    .minParallel({ shape -> shape.size })
```

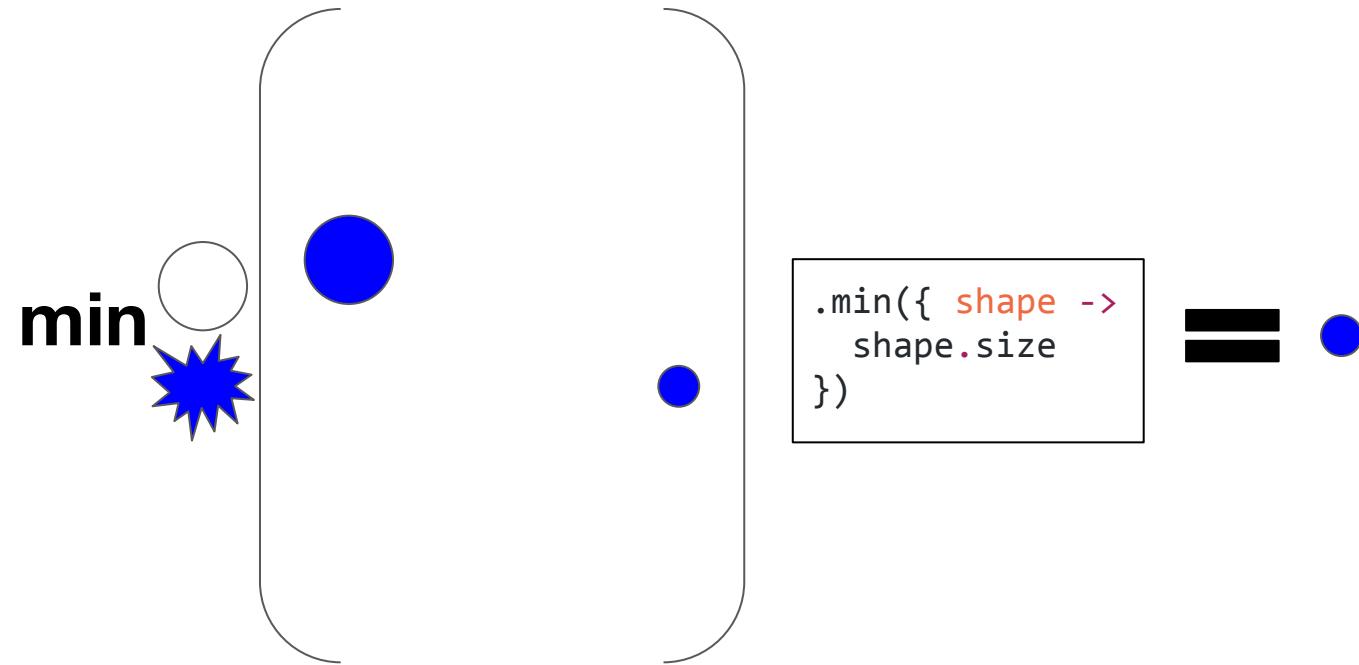
```
ParallelEnhancer.enhanceInstance shapes
shapes.makeConcurrent()
Shape minBlueCircle = shapes
    .findAll({ shape -> shape.type == CIRCLE })
    .findAll({ shape -> shape.color == BLUE })
    .min({ shape -> shape.size })
```



```
ParallelEnhancer.enhanceInstance shapes
shapes.makeConcurrent()
Shape minBlueCircle = shapes
    .findAll({ shape -> shape.type == CIRCLE })
    .findAll({ shape -> shape.color == BLUE })
    .min({ shape -> shape.size })
```



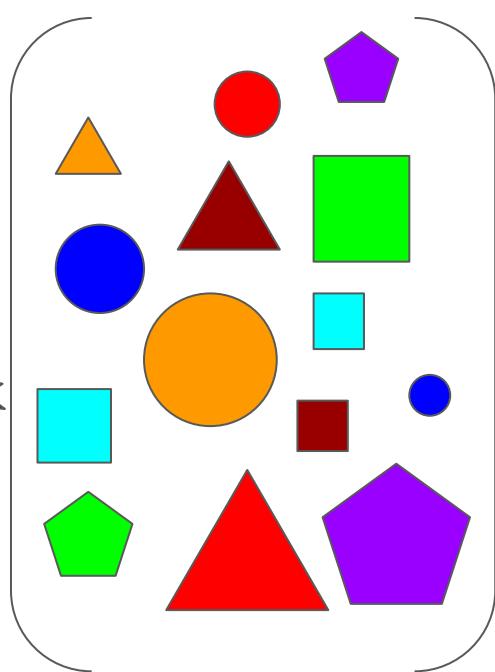
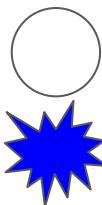
```
ParallelEnhancer.enhanceInstance shapes
shapes.makeConcurrent()
Shape minBlueCircle = shapes
    .findAll({ shape -> shape.type == CIRCLE })
    .findAll({ shape -> shape.color == BLUE })
    .min({ shape -> shape.size })
```



```
Shape minBlueCircle = shapes
    .findAll({ shape -> shape.type == CIRCLE })
    .findAll({ shape -> shape.color == BLUE })
    .min({ shape -> shape.size })
```

```
withPool {  
    Shape minBlueCircle = shapes.parallel  
        .filter({ shape -> shape.type == CIRCLE })  
        .filter({ shape -> shape.color == BLUE })  
        .min({ shape -> shape.size })  
}
```

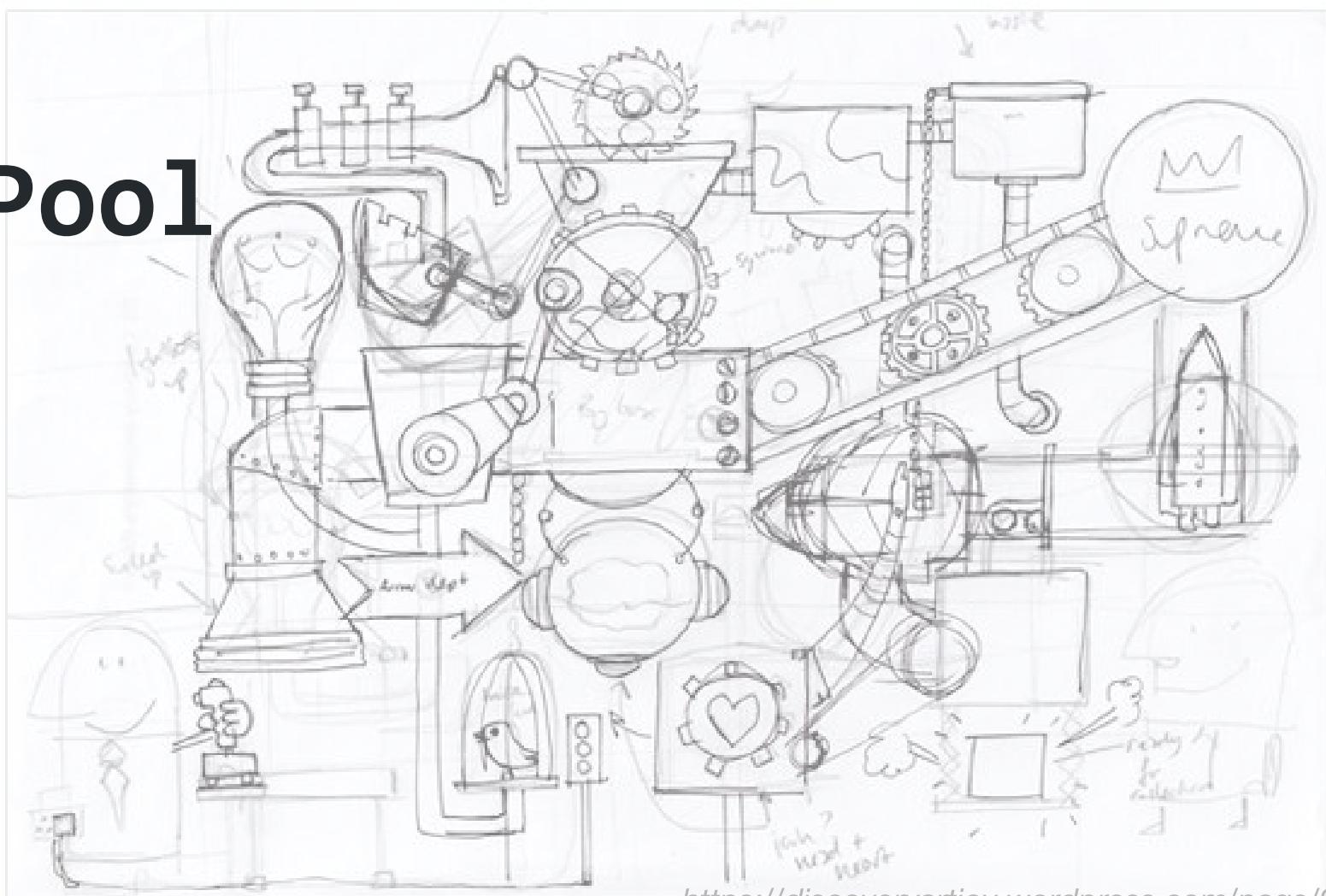
min



```
.filter({ shape -> shape.type == CIRCLE })  
.filter({ shape -> shape.color == BLUE })  
.min({ shape -> shape.size })
```



withPool



```
withPool {
```

```
}
```

```
GParsPool.withPool {
```

```
}
```

```
GParsPool.withPool { ForkJoinPool pool ->  
}  
}
```

```
GParsPool.withExistingPool(pool) {
```

```
}
```

```
GParsExecutorsPool.withPool { ExecutorService executor ->
```

```
}
```

```
GParsPool.withPool { ForkJoinPool pool ->  
}  
}
```

```
GParsPool.withPool(numberOfThreads, exceptionHandler) { ForkJoinPool pool ->
```



```
    Runtime.getRuntime().availableProcessors() + 1
```

```
}
```

```
withPool {  
    Shape minBlueCircle = shapes  
        .findAllParallel({ shape -> shape.type == CIRCLE })  
        .findAllParallel({ shape -> shape.color == BLUE })  
        .minParallel({ shape -> shape.size })  
  
    shapes.makeConcurrent()  
    Shape minBlueCircle = shapes  
        .findAll({ shape -> shape.type == CIRCLE })  
        .findAll({ shape -> shape.color == BLUE })  
        .min({ shape -> shape.size })  
  
    Shape minBlueCircle = shapes.parallel  
        .filter({ shape -> shape.type == CIRCLE })  
        .filter({ shape -> shape.color == BLUE })  
        .min({ shape -> shape.size })  
}
```

```
withPool {
```

```
}
```

```
withPool {  
  
    List latestTweets = twitter.fetchLatestTweets()  
  
    println 'Done:'  
    println latestTweets  
  
}  

```

```
withPool {  
  
    Future latestTweets = executeAsync({ twitter.fetchLatestTweets() })  
  
    println 'Loading...'  
    println latestTweets.get()  
  
}
```

```
withPool {  
  
    Future latestTweets = twitter.&fetchLatestTweets.callAsync()  
  
    println 'Loading...'  
    println latestTweets.get()  
  
}  
}
```

```
withPool {  
  
    Closure fetchLatestTweetsAsync = twitter.&fetchLatestTweets.async()  
    Future latestTweets = fetchLatestTweetsAsync()  
  
    println 'Loading...'  
    println latestTweets.get()  
  
}
```

```
withPool {  
  
    Closure fetchLatestTweetsAsync = twitter.&fetchLatestTweets.async()  
    Future latestTweets = fetchLatestTweetsAsync()  
  
    Closure extractKeywordsAsync = keywords.&extractKeywords.async()  
  
    println 'Loading tweets...'  
    Future keywords = extractKeywordsAsync(latestTweets.get())  
  
    println 'Extracting keywords...'  
    println keywords.get()  
  
}
```

```
withPool {  
  
    Closure fetchLatestTweetsAsync = twitter.&fetchLatestTweets.asyncFun()  
    Promise latestTweets = fetchLatestTweetsAsync()  
  
    Closure extractKeywordsAsync = keywords.&extractKeywords.asyncFun()  
    Promise keywords = extractKeywordsAsync(latestTweets)  
  
    println 'Loading tweets and extracting keywords...'  
    println keywords.get()  
  
}
```

```
withPool {  
  
    Closure fetchLatestTweetsAsync = twitter.&fetchLatestTweets.asyncFun()  
    Promise latestTweets = fetchLatestTweetsAsync()  
  
    Closure extractKeywordsAsync = keywords.&extractKeywords.asyncFun()  
    Promise keywords = extractKeywordsAsync(latestTweets)  
  
    keywords.whenBound {  
        println it  
    }  
  
    println 'Loading tweets and extracting keywords...'  
  
}
```

```
withPool {  
  
    Closure fetchAsync = twitter.&fetchLatestTweets.asyncFun()  
  
    Closure watsonsExtractKeywords = watsons.&extractKeywords.curry(fetchAsync())  
    Closure googleExtractKeywords = google.&extractKeywords.curry(fetchAsync())  
  
    List keywords = watsonsExtractKeywords()  
  
    println keywords  
  
}
```

```
withPool {  
  
    Closure fetchAsync = twitter.&fetchLatestTweets.asyncFun()  
  
    Closure watsonsExtractKeywords = watsons.&extractKeywords.curry(fetchAsync())  
    Closure googleExtractKeywords = google.&extractKeywords.curry(fetchAsync())  
  
    List keywords = googleExtractKeywords()  
  
    println keywords  
  
}
```

```
withPool {  
  
    Closure fetchAsync = twitter.&fetchLatestTweets.asyncFun()  
  
    Closure watsonsExtractKeywords = watsons.&extractKeywords.curry(fetchAsync())  
    Closure googleExtractKeywords = google.&extractKeywords.curry(fetchAsync())  
  
    List keywords = speculate( watsonsExtractKeywords(), googleExtractKeywords() )  
  
    println keywords  
  
}
```

```
withPool {  
  
    Closure fetchAsync = twitter.&fetchLatestTweets.asyncFun()  
  
    Closure watsonsExtractKeywords = watsons.&extractKeywords.curry(fetchAsync())  
    Closure googleExtractKeywords = google.&extractKeywords.curry(fetchAsync())  
  
    List keywords = speculate( watsonsExtractKeywords(), googleExtractKeywords() )  
  
    println keywords  
  
}
```

```
withPool {  
  
    List latestTweets = twitter.fetchLatestTweets()  
  
    Closure watsonsExtractKeywords = watsons.&extractKeywords.curry(latestTweets)  
    Closure googleExtractKeywords = google.&extractKeywords.curry(latestTweets)  
  
    List keywords = speculate( watsonsExtractKeywords(), googleExtractKeywords() )  
  
    println keywords  
  
}
```

```
withPool {  
  
    Closure fetchWithCache = twitter.&fetchLatestTweets.gmemoize()  
  
    Closure watsonsExtractKeywords = watsons.&extractKeywords.curry(fetchWithCache())  
    Closure googleExtractKeywords = google.&extractKeywords.curry(fetchWithCache())  
  
    List keywords = speculate( watsonsExtractKeywords(), googleExtractKeywords() )  
  
    println keywords  
  
}
```

```
withPool {  
    runForkJoin(list) { list ->  
        if (list.size() < 2) { return list }  
        if (list.size() == 2) {  
            if (list[0] <= list[1]) {  
                return list  
            } else {  
                return list[-1..0]  
            }  
        }  
  
        int middleIndex = list.size() / 2  
  
        forkOffChild list[0..<middleIndex]  
        forkOffChild list[middleIndex..<list.size()]  
  
        return merge(*childrenResults)  
    }  
}
```

```
withPool(lambdaToClosure(() -> {  
    ...  
});
```

```
withPool(lambdaToClosure(() -> {

    Future result = executeAsync (
        lambdaToClosure(() -> { return twitter.fetchLatestTweets(); })
    ).get(0);

    final List tweets = (List) result.get();

    List keywords = speculate (
        lambdaToClosure(() -> { return watsons.extractKeywords(tweets); }),
        lambdaToClosure(() -> { return google.extractKeywords(tweets); })
    );

    return keywords;
}));
```

```
withPool(lambdaToClosure(() -> {

    Future result = executeAsync (
        lambdaToClosure(() -> { return twitter.fetchLatestTweets(); }))
    .get(0);

    final List tweets = (List) result.get();

    List keywords = speculate (
        lambdaToClosure(() -> { return watsons.extractKeywords(tweets); }),
        lambdaToClosure(() -> { return google.extractKeywords(tweets); })
    );

    return keywords;
}));
```

```
withPool {  
  
    Shape minBlueCircle = shapes  
        .findAllParallel({ shape -> shape.type == CIRCLE })  
        .findAllParallel({ shape -> shape.color == BLUE })  
        .minParallel({ shape -> shape.size })  
  
}  

```

```
withPool {  
  
    List<Shape> blueCircles = []  
  
    shapes.findAllParallel({ shape -> shape.type == CIRCLE })  
        .findAllParallel({ shape -> shape.color == BLUE })  
        .eachParallel({ shape -> blueCircles.add(shape) })  

```

If multiple threads access the same mutable
state variable without appropriate
synchronization, your program is broken.

```
}
```

Thread #1

blueCircles.add(shape)

blueCircles.add(shape)

blueCircles.add(shape)

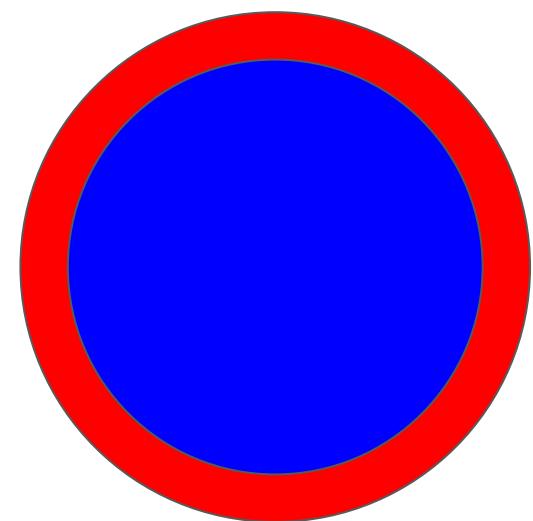
Thread #2

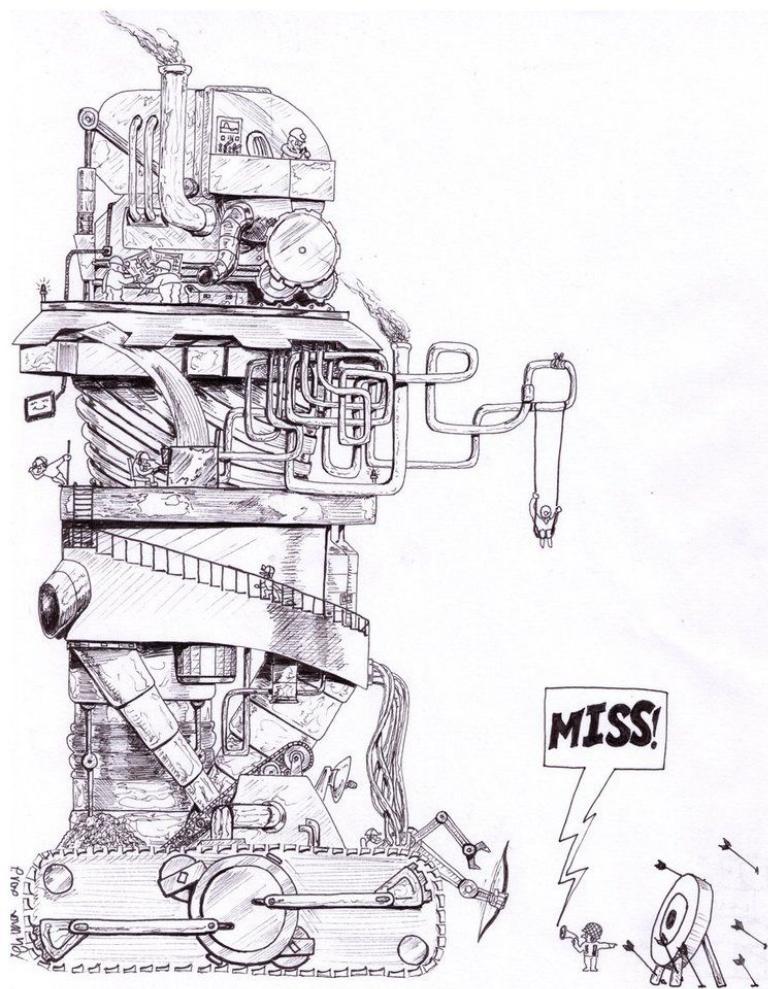
blueCircles.add(shape)

blueCircles.add(shape)

blueCircles.add(shape)

synchronized

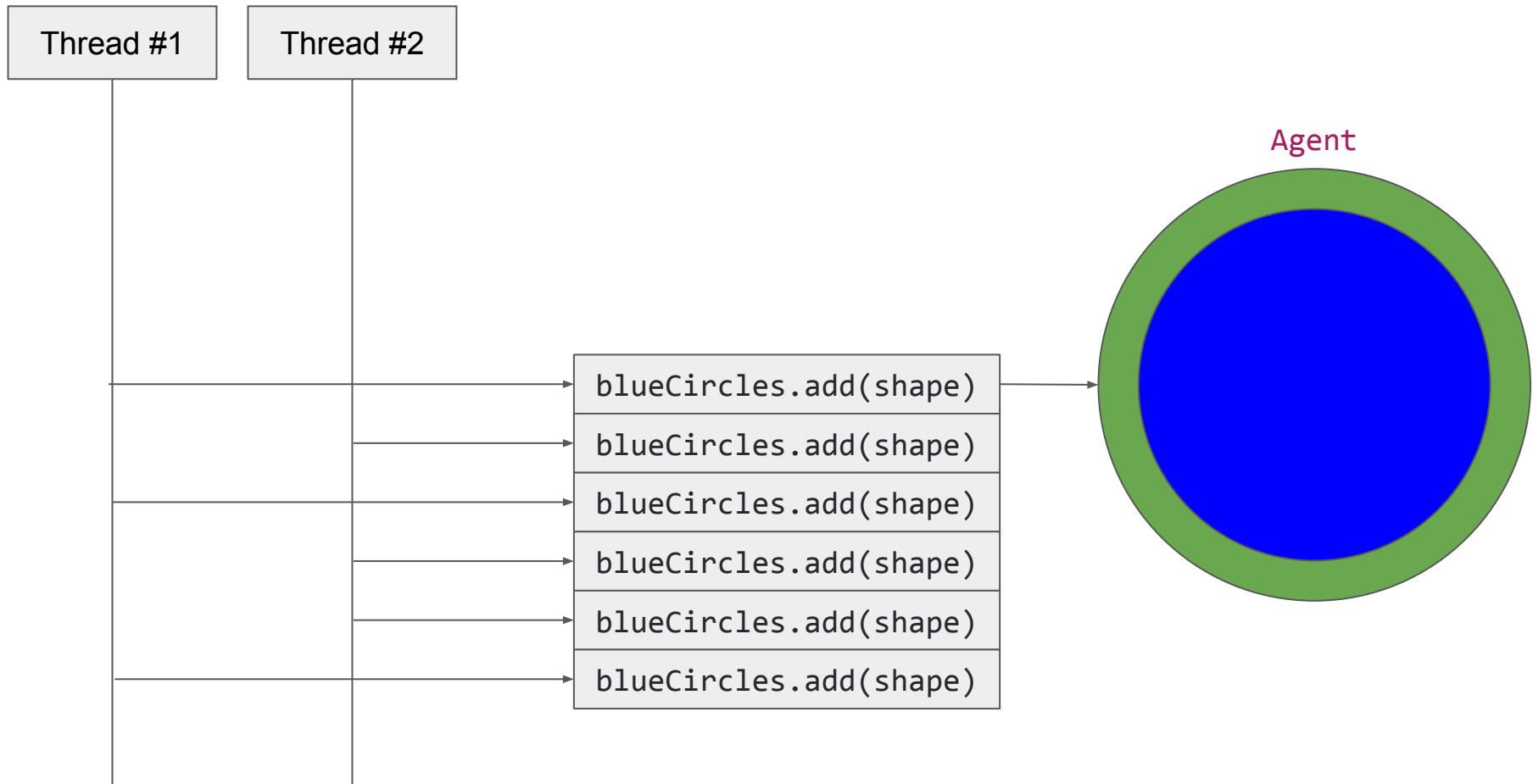




AGENTS

<http://www.keywordsuggests.com/DT5zG9YMrnEK707aqUa2lhkNs5vUwu3mV6WrrLHR0soHddyR42EFvbEkYg9kaqswD%7CyremGnEFLO%7CzJXdp0SMQ/>

```
withPool {  
  
    def blueShapesGuard = new Agent<List<Shape>>()  
    blueShapesGuard { updateValue [] }  
  
    shapes.findAllParallel({ shape -> shape.type == CIRCLE })  
        .findAllParallel({ shape -> shape.color == BLUE })  
        .eachParallel({ shape -> blueShapesGuard { it.add(shape) } })  
  
}  
}
```



```
withPool {  
  
    def blueShapesGuard = new Agent<Shape>()  
    blueShapesGuard { updateValue [] }  
  
    blueShapesGuard.addListener { oldValue, newValue -> ... }  
  
    blueShapesGuard.addValidator { oldValue, newValue -> ... }  
  
    shapes.findAllParallel({ shape -> shape.type == CIRCLE })  
        .findAllParallel({ shape -> shape.color == BLUE })  
        .eachParallel({ shape -> blueShapesGuard { it.add(shape) } })  
  
    println blueShapesGuard.val  
  
    blueShapesGuard.valAsync { println it }  
  
}
```

```
final Agent<List<Shape>> blueShapesGuard = new Agent<>();
blueShapesGuard.send(new ArrayList<>());
blueShapesGuard.send(new MessagingRunnable<List<Shape>>() {
    @Override
    protected void doRun(List<Shape> value) {
        value.add(shape);
    }
});

blueShapesGuard.valAsync(new MessagingRunnable<List<Shape>>() {
    @Override
    protected void doRun(List<Shape> value) {
        System.out.println(value);
    }
});
```

Thread #1

Thread #2

Agent

```
blueCircles.add(shape)  
blueCircles.add(shape)  
blueCircles.add(shape)  
blueCircles.add(shape)  
blueCircles.add(shape)  
blueCircles.add(shape)
```



Thread #1

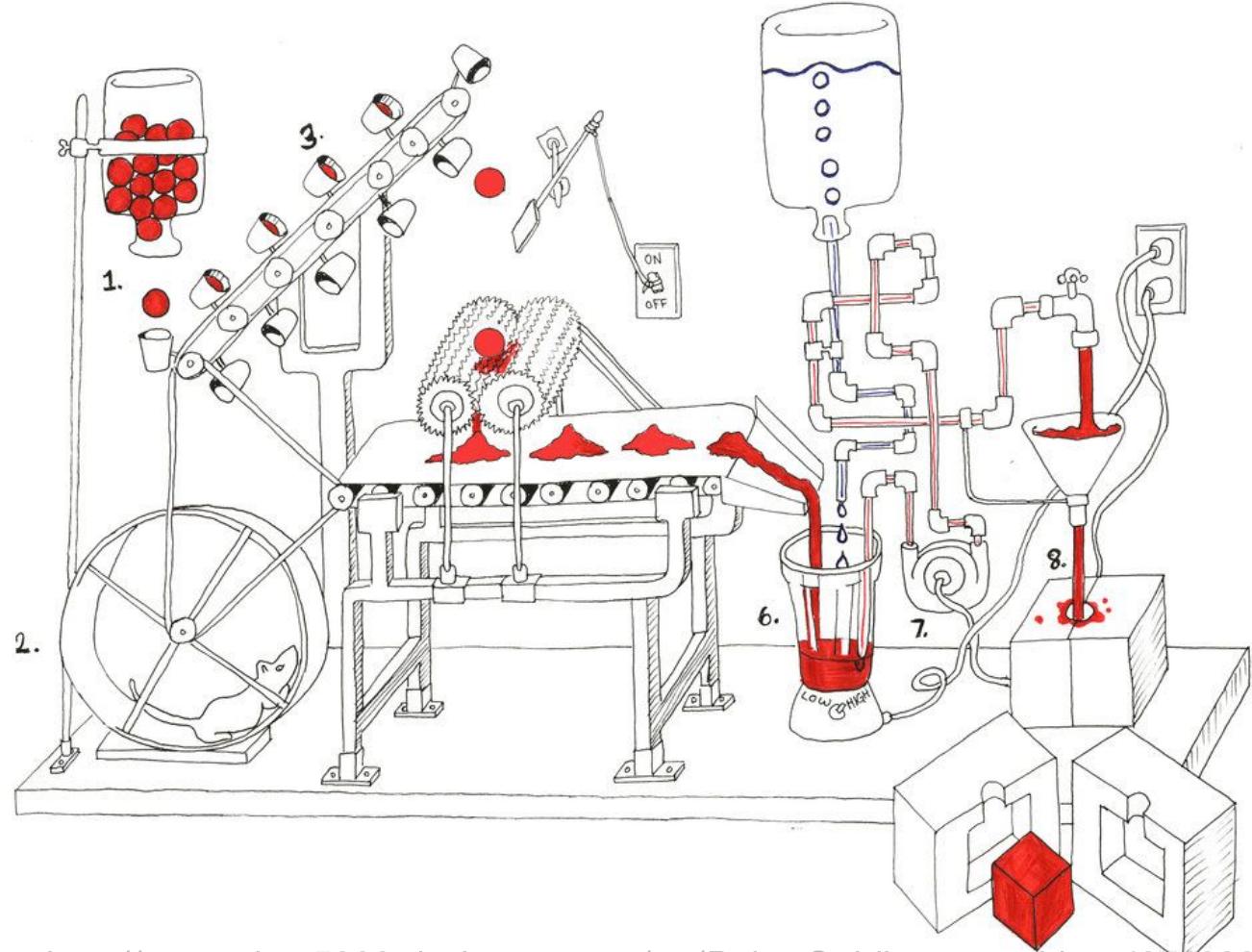
Thread #2



Actor



ACTORS



```
class ShapesGuardActor extends DefaultActor {  
    List<Shape> shapes  
  
    void afterStart() {  
        shapes = []  
    }  
  
    void act() {  
        loop {  
            react { Shape shape -> shapes.add shape }  
        }  
    }  
}
```

```
class ShapesGuardActor extends DefaultActor {  
    List<Shape> shapes  
  
    void afterStart() {  
        shapes = []  
    }  
  
    void act() {  
        loop {  
            react { Shape shape -> shapes.add shape }  
        }  
    }  
}
```

executed for received message

message

```
class ShapesGuardActor extends DefaultActor {  
    List<Shape> shapes  
  
    void afterStart() {  
        shapes = []  
    }  
  
    void act() {  
        loop { ← continuously receive messages, one thread at most  
            react { Shape shape -> shapes.add shape }  
        }  
    }  
}
```

```
class KeywordStoreActor extends DefaultActor {  
    KeywordsRepository keywordsRepository  
  
    void act() {  
        loop {  
            react { Keyword keyword -> keywordsRepository.store keyword }  
        }  
    }  
}
```

```
def keywordStoreActor = actor {
    loop {
        react { Keyword keyword ->
            keywordsRepository.store keyword
        }
    }
}
```

```
def keywordStoreActor = actor {
    loop {
        react { Keyword keyword ->
            keywordsRepository.store id, keyword
        }
    }
}
```

```
def keywordStoreActor = actor {
    loop {
        react { Keyword keyword ->
            idGenerator << keyword ← send message
        }
        process → react { Id id ->
            answer
            keywordsRepository.store id, keyword
        }
    }
}

def idGenerator = actor {
    loop {
        react { def keyword ->
            Id id = ...
            keyword.reply id
        }
    }
}
```

process → *react* { *Id id* ->
answer

idGenerator << keyword ← send message

↑ reply to sender

```
def keywordStoreActor = actor {
    loop {
        react { Keyword keyword ->
            idGenerator << keyword
            react { Id id ->
                keywordsRepository.store id, keyword
            }
        }
    }
}

def idGenerator = reactor { def keyword ->
    Id id = ...
    keyword.reply id
}
```

```
Closure keywordStoreHandler = new ReactorMessagingRunnable<Keyword, Object>() {
    @Override protected Object doRun(final Keyword keyword) {
        idGenerator.sendAndContinue(keyword, new MessagingRunnable<Id>() {
            @Override protected void doRun(final Id id) {
                keywordsRepository.store(id, keyword);
            }
        });
        return null;
    }
};
Actor keywordStoreActor = new ReactiveActor(keywordStoreHandler);

Closure idGeneratorHandler = new ReactorMessagingRunnable<Keyword, Id>() {
    @Override protected Id doRun(Keyword argument) {
        Id id = ...
        return id;
    }
};
Actor idGenerator = new ReactiveActor(idGeneratorHandler);
```

```
def keywordStoreActor = actor {
    loop {
        react { Keyword keyword ->
            idGenerator << keyword
            react { Id id ->
                keywordsRepository.store id, keyword
            }
        }
    }
}

def idGenerator = reactor { def keyword ->
    Id id = ...
    keyword.reply id
}
```

```
def keywordStoreActor = actor {
    loop {
        react { Keyword keyword ->
            idGenerator << keyword
            react { Id id ->
                keywordsRepository.store id, keyword
            }
        }
    }
}

def idGenerator = reactor { def message ->
    if (message instanceof Keyword) {
        id = ...
    }
    if (message instanceof Tweet) {
        id = ...
    }
    message.reply id
}
```

```
def keywordStoreActor = actor {
    loop {
        react { Keyword keyword ->
            idGenerator << keyword
            react { Id id ->
                keywordsRepository.store id, keyword
            }
        }
    }
}
```

```
def idGenerator = reactor { def message ->
    if (message instanceof Keyword) {
        id = ...
    }
    if (message instanceof Tweet) {
        id = ...
    }
    if (message instanceof FacebookPost) {
        id = ...
    }
}
```

```
def keywordStoreActor = actor {
    loop {
        react { Keyword keyword ->
            idGenerator << keyword
            react { Id id ->
                keywordsRepository.store id, keyword
            }
        }
    }
}

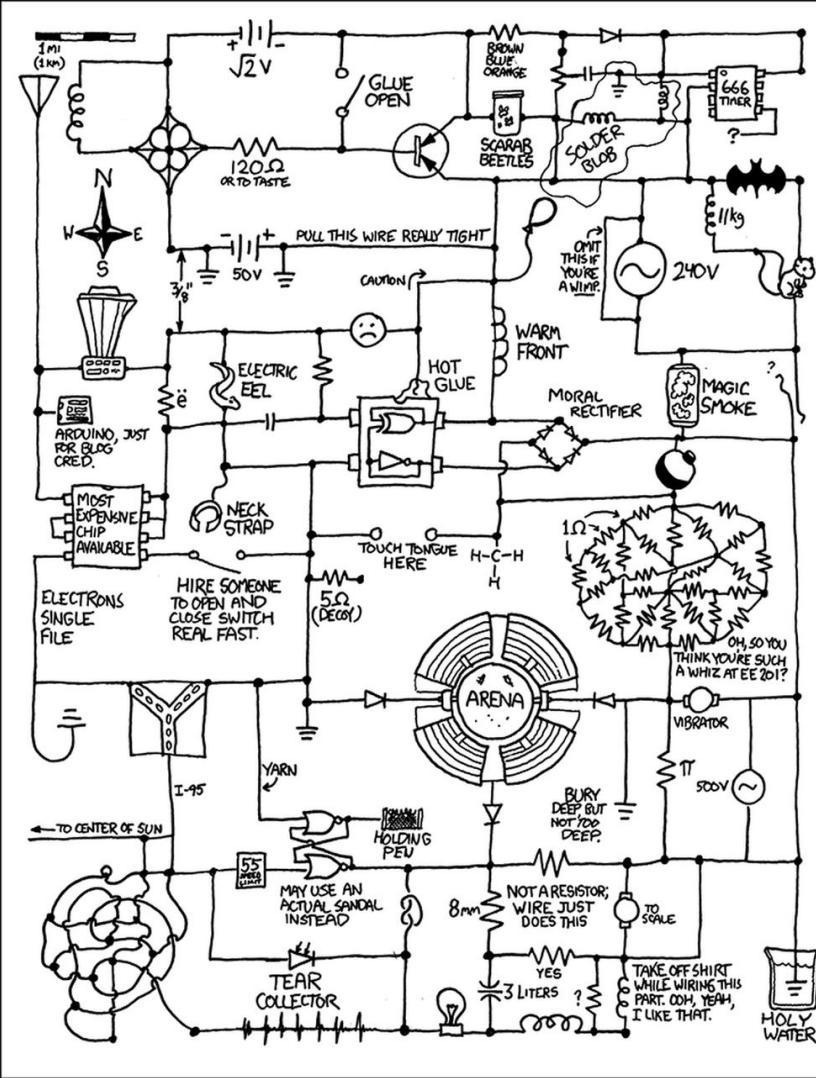
def idGenerator = messageHandler { def message ->
    when { Keyword keyword -> reply generateKeywordId(keyword) }
    when { Tweet tweet -> reply generateTweetId(tweet) }
    when { FacebookPost facebookPost -> reply generateFacebookPostId(facebookPost) }
}
```

```
Closure keywordStoreHandler = new ReactorMessagingRunnable<Keyword, Object>() {
    @Override protected Object doRun(final Keyword keyword) {
        idGenerator.sendAndContinue(keyword, new MessagingRunnable<Id>() {
            @Override protected void doRun(final Id id) {
                keywordsRepository.store(id, keyword);
            }
        });
        return null;
    }
};

Actor keywordStoreActor = new ReactiveActor(keywordStoreHandler);
Actor idGenerator = new DynamicDispatchActor() {

    public void onMessage(final Keyword keyword) {
        replyIfExists(generateKeywordId(keyword));
    }

    public void onMessage(final Tweet tweet) {
        replyIfExists(generateTweetId(tweet));
    }
}
```



DATAFLOWS

<http://www.projectroomseattle.org/programs-content/2014/6/preparing-for-the-failure-varietys-show-the-rube-goldberg-confessionals>

```
final Dataflow dataflow = new Dataflows()

task {
    dataflow.numberOfInteractions = dataflow.retweets + dataflow.facebookLikes
}

task {
    dataflow.retweets = twitter.numberOfRetweets(article)
}

task {
    dataflow.facebookLikes = facebook.numberOfLikes(article)
}

println dataflow.numberOfInteractions
```

```
final DataflowVariable<Integer> numberOfInteractions = new DataflowVariable<>();
final DataflowVariable<Integer> retweets = new DataflowVariable<>();
final DataflowVariable<Integer> facebookLikes = new DataflowVariable<>();

task((Runnable) () -> {
    try {
        numberOfInteractions.bind(retweets.getVal() + facebookLikes.getVal());
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
});

task(() -> retweets.bind(twitter.numberOfRetweets(article)));

task(() -> facebookLikes.bind(facebook.numberofLikes(article)));

System.out.println(numberOfInteractions.getVal());
```

```
def retweets = new DataflowQueue()
def facebookLikes = new DataflowQueue()
def numberOfInteractions = new DataflowQueue()

operator(
    inputs: [ retweets, facebookLikes ],
    outputs: [ numberOfInteractions ],
    { retweetsCount, facebookLikesCount ->
        numberOfInteractions << retweetsCount + facebookLikesCount
    }
)

task {
    ALL_ARTICLES.each { article -> retweets << twitter.numberOfRetweets(article) }
}

task {
    ALL_ARTICLES.each { article -> facebookLikes << facebook.numberofLikes(article) }
}
```

```
final DataflowQueue retweets = new DataflowQueue();
final DataflowQueue facebookLikes = new DataflowQueue();
final DataflowQueue numberOfInteractions = new DataflowQueue();

operator(
   .asList(retweets, facebookLikes),
   .asList(numberOfInteractions),
    new DataflowMessagingRunnable(2) {

        @Override
        protected void doRun(Object... arguments) {
            getOwningProcessor().bindOutput((Integer)arguments[0] + (Integer)arguments[1]);
        }
});

task(() -> ALL_ARTICLES.forEach(a -> retweets.bind(twitter.numberOfRetweets(a))));

task(() -> ALL_ARTICLES.forEach(a -> facebookLikes.bind(facebook.numberOfLikes(a))));
```

WHY USE GPARS?

TO IMPROVE PERFORMANCE OF SEQUENTIAL CODE

SIMPLIFY DEVELOPMENT OF CONCURRENT/PARALLEL CODE

USE CLOSE-TO-DOMAIN ABSTRACTIONS

REDUCE NUMBER OF ERRORS

INTEGRATE DIFFERENT ABSTRACTIONS EASILY

WHEN DO USE GPARS? - PARALLEL STREAMS

FURTHER DIVE

Examples from slides

Groovy and Concurrency with GPars by Paul King

Gpars: Concurrency in Java & Groovy by Ken Kousen

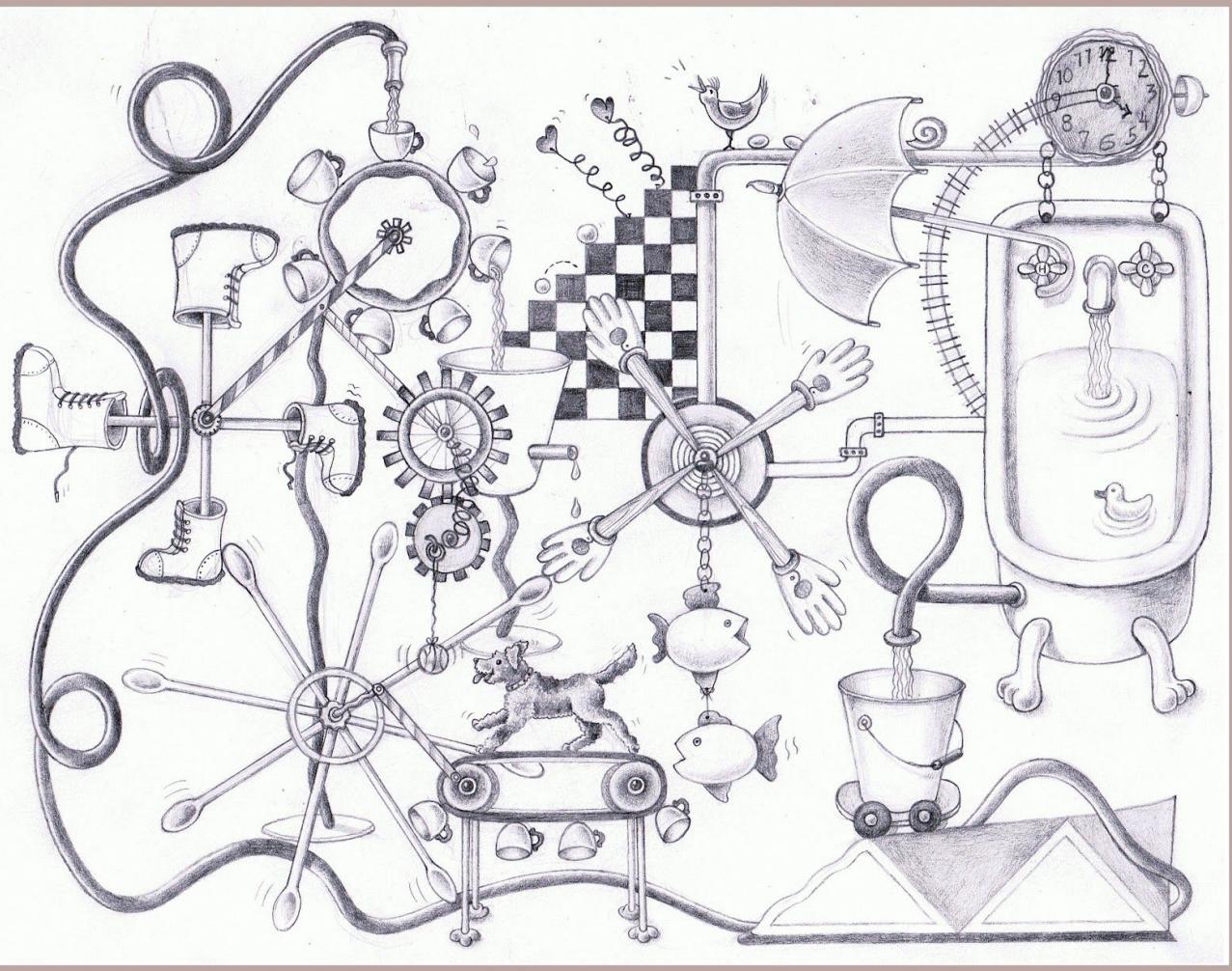
Official site

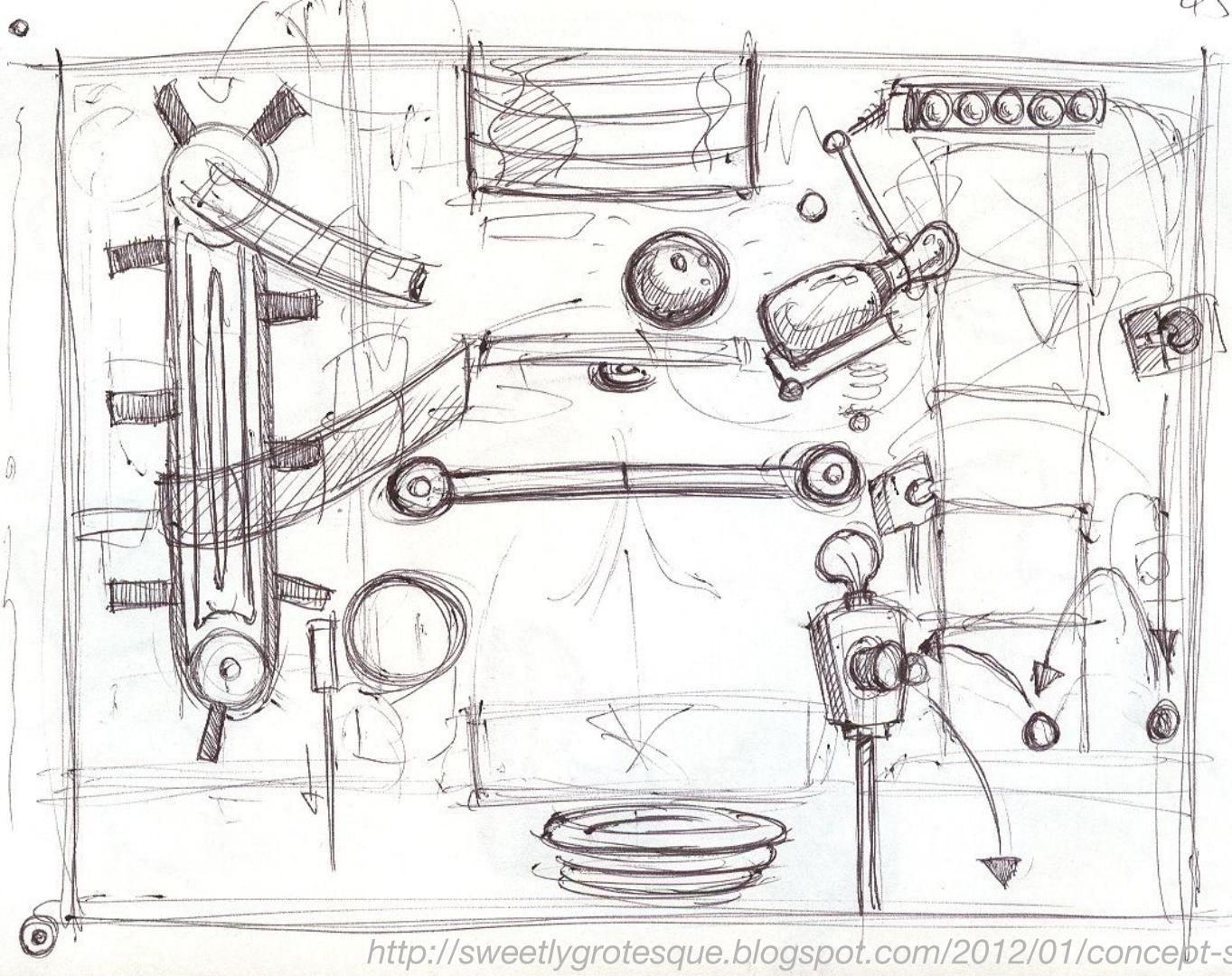
Sources

Slack channel

The GPars User Guide

'Concurrent Groovy with GPars' chapter from 'Groovy in Action' book





Q/A

**THANK
YOU!**