

GPar: Unsung Hero of Concurrency in Practice

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**.



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?



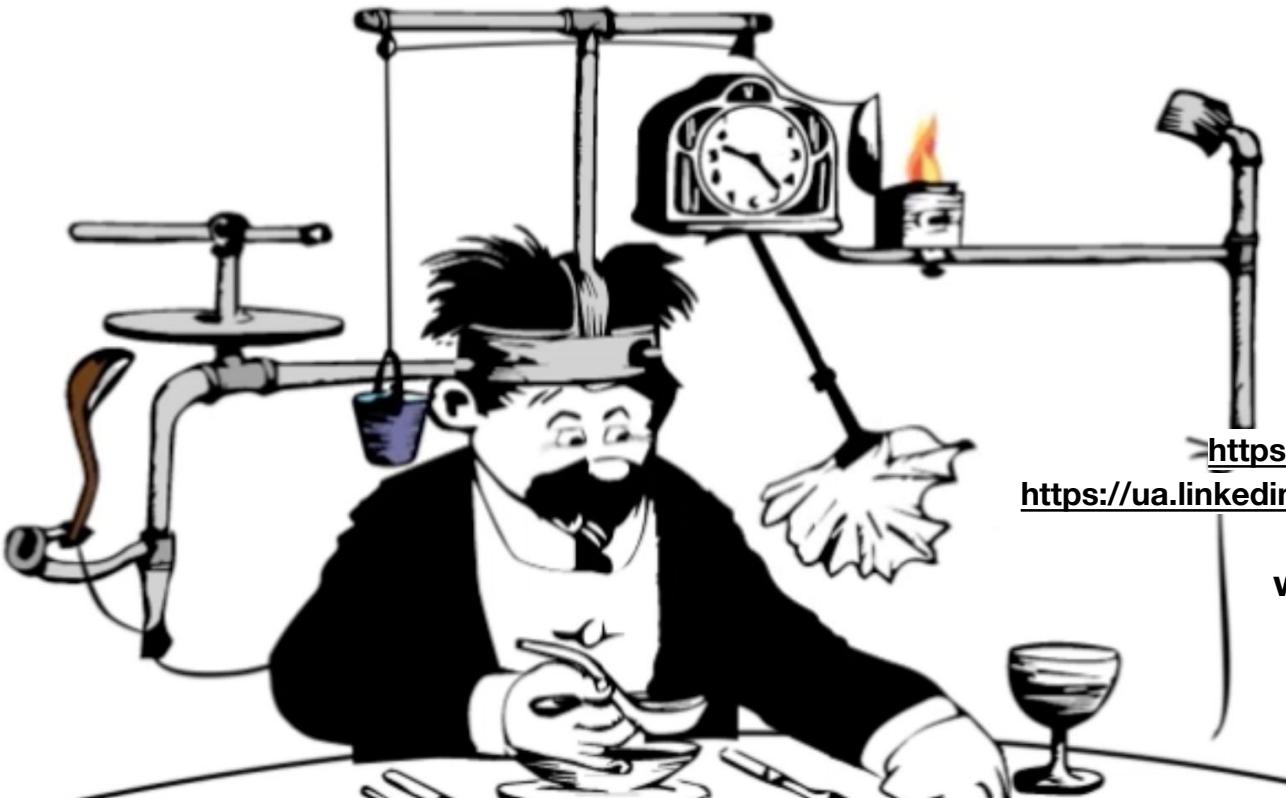
Focus for this talk

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

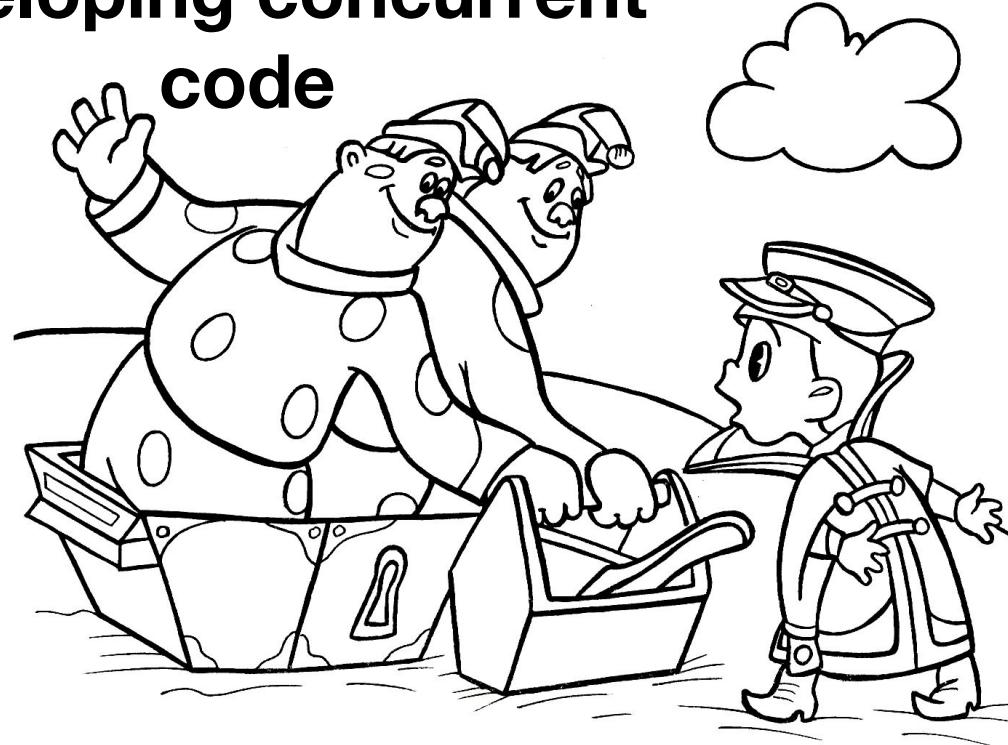
yermilov17@gmail.com
<https://yermilov.github.io/>
<https://twitter.com/yermilov17>
<https://www.facebook.com/yaroslav.yermilov>
<https://ua.linkedin.com/pub/yaroslav-yermilov/58/682/506>

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); }  
    }  
}
```

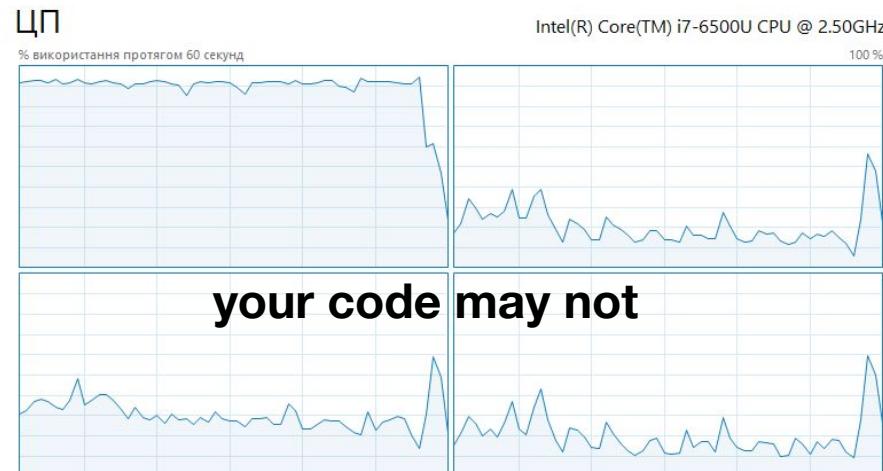


Holder was not
properly
published!

guideline #1 for developing concurrent code

MOORE'S LAW:

Computers will get exponentially faster.



Використання	Швидкість	Максимальна швидкість:	2,59 ГГц
23%	2,46 ГГц	Сокети:	1
Процеси	Потоки	Ядра:	2
165	2292	Логічних процесорів:	4
	73885	Віртуалізація:	Вимкнuto
Час роботи		Підтримка технології Hyper-V:	Так
0:01:01:26		Кеш 1 рівня:	128 КБ
		Кеш 2 рівня:	512 КБ
		Кеш 3 рівня:	4,0 МБ

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

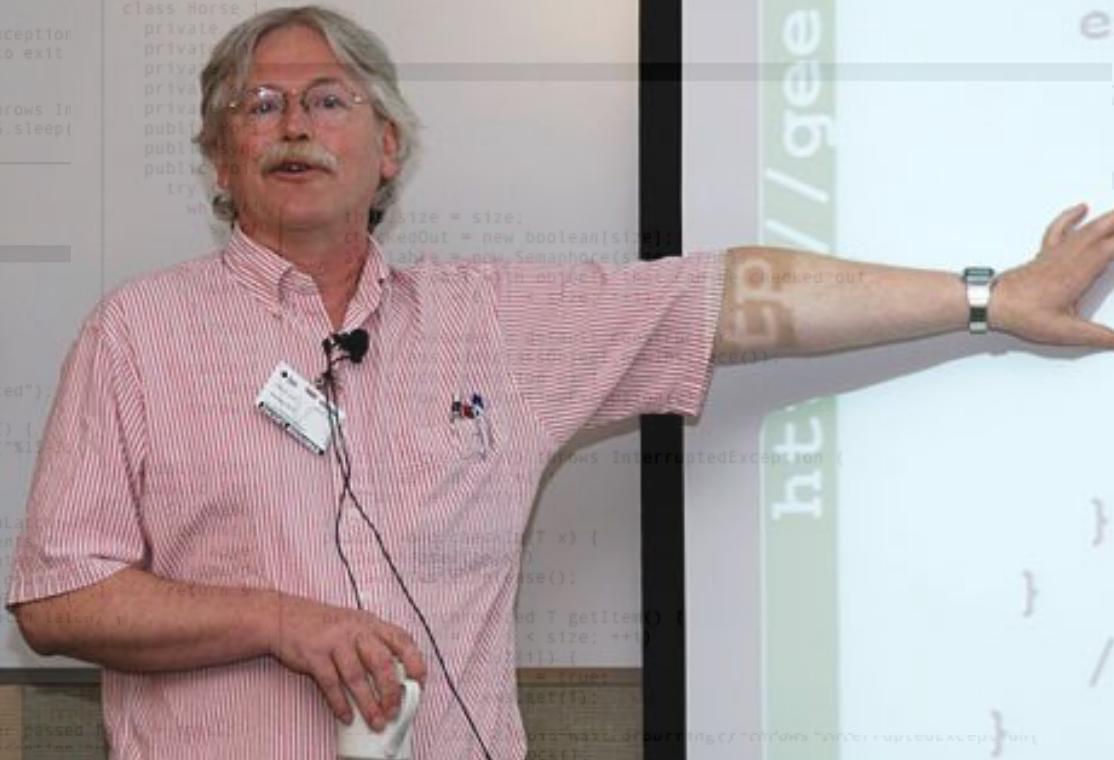
print(this + " completed");
}
public String toString() {
 return String.format("%12s", name);
}
}

// 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();
 }
 }
}

```
//: 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



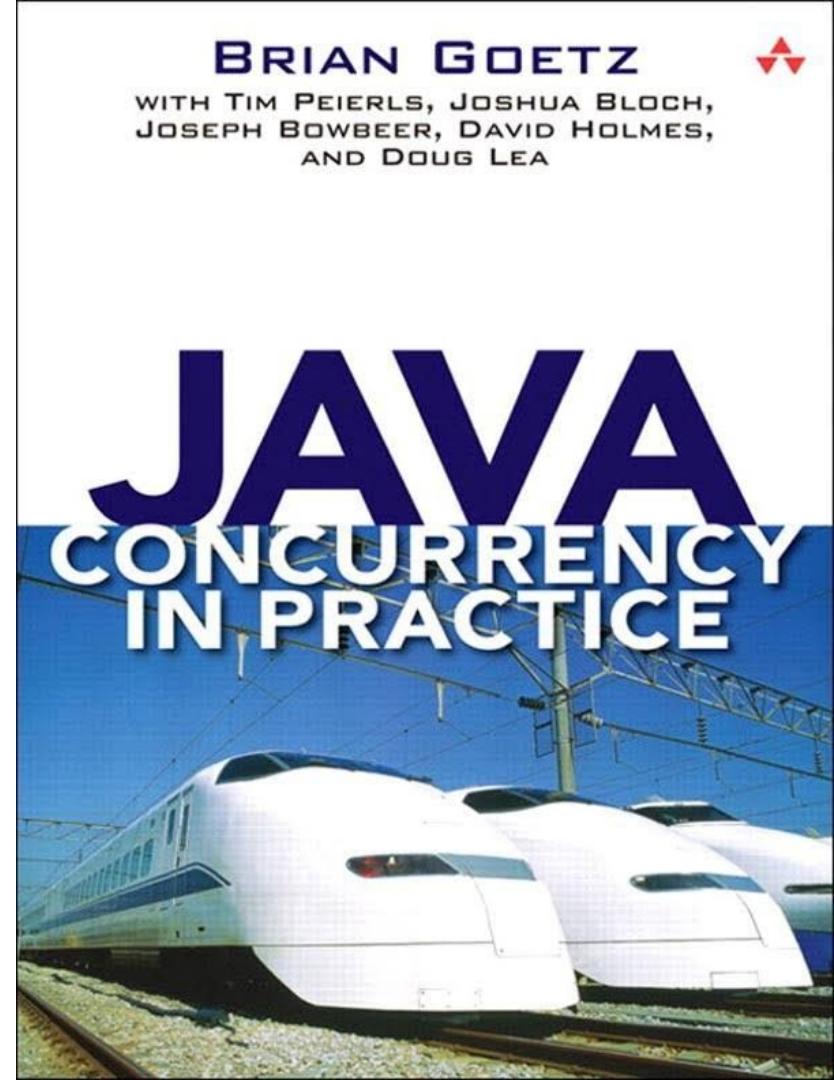
two guidelines for
developing concurrent
code



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:

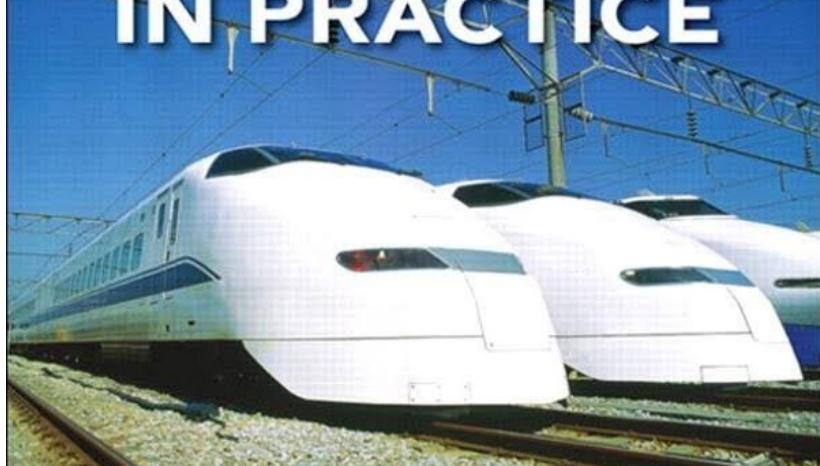
Don't share the state variable across threads

BRIAN GOETZ

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



JAVA CONCURRENCY IN PRACTICE



BRIAN GOETZ



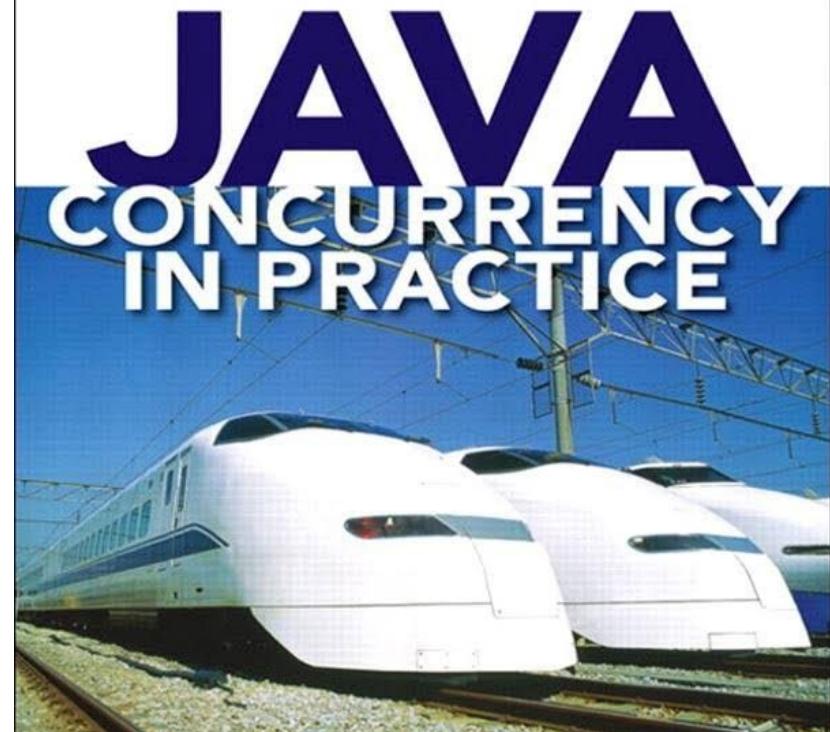
WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA

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

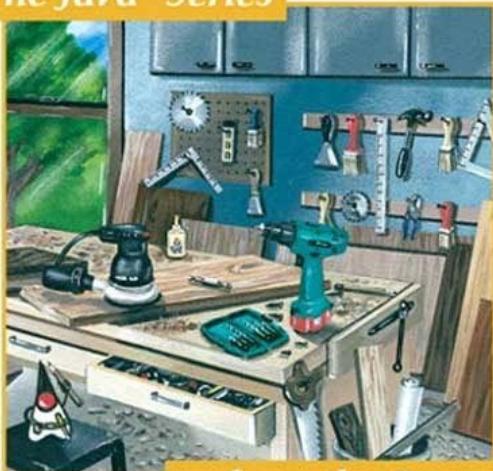


Joshua Bloch

Revised and
Updated for
Java SE 6

Effective Java™ Second Edition

The Java™ Series



...from the Source



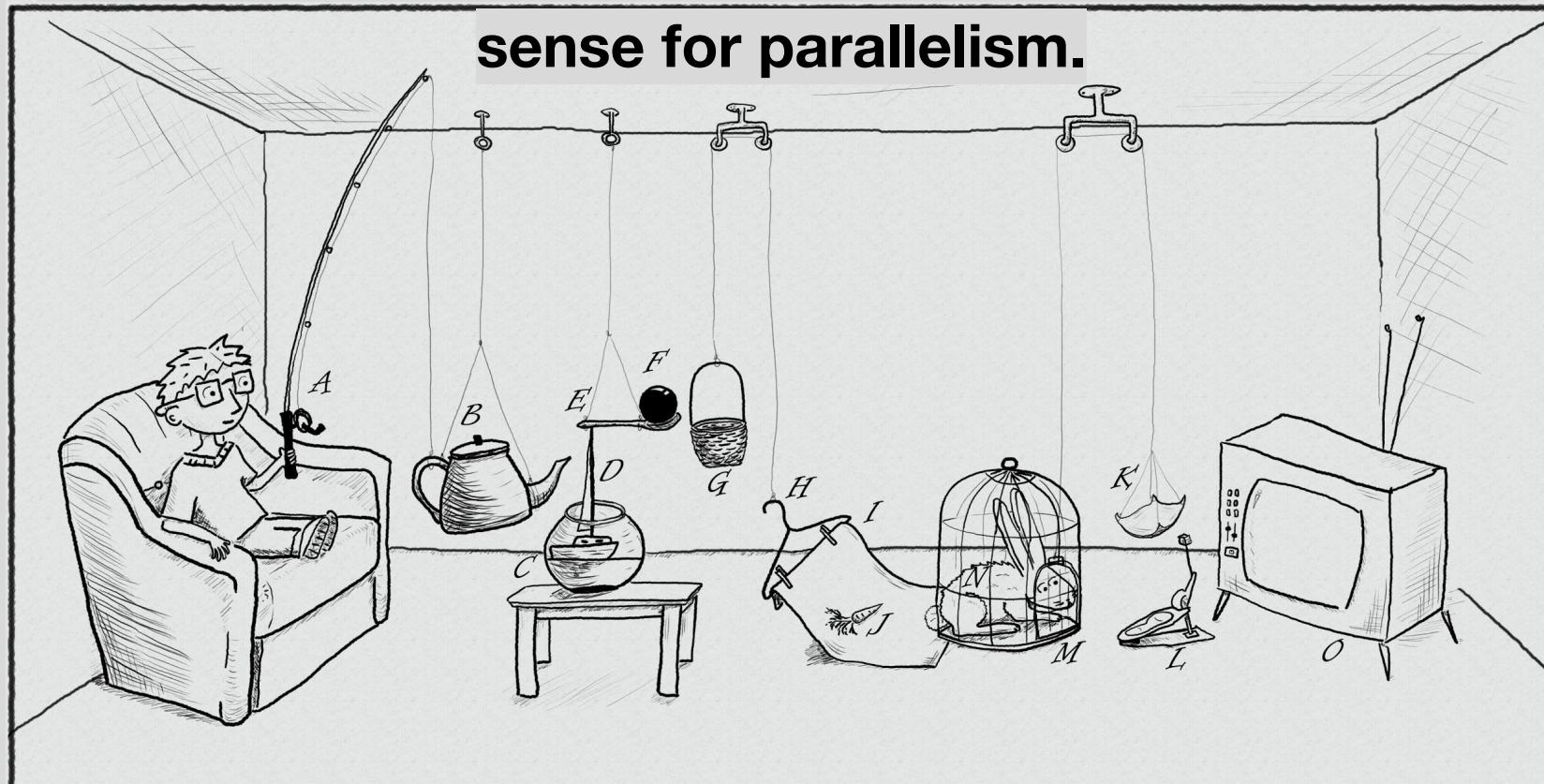
10 Concurrency.....259

- Item 66: Synchronize access to shared mutable data.....259
- Item 67: Avoid excessive synchronization265
- Item 68: Prefer executors and tasks to threads.....271
- Item 69: Prefer concurrency utilities to `wait` and `notify`.....273

CONTENTS

- Item 70: Document thread safety278
- Item 71: Use lazy initialization judiciously282
- Item 72: Don't depend on the thread scheduler286
- Item 73: Avoid thread groups288

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);
    }
}
```



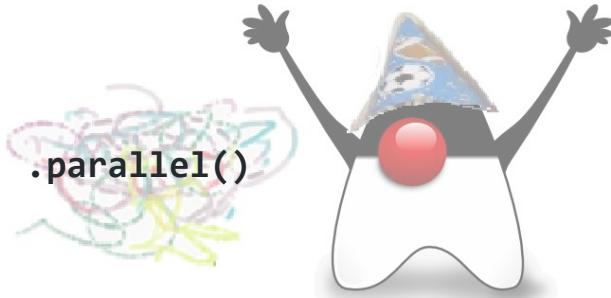
```
public class MergeSortTask<T extends Comparable<T>> extends RecursiveTask<List<T>> {
    @Override
    protected List<T> compute() {
        if (list.size() < 2) { return list; }
        if (list.size() == 2) {
            if (list.get(0).compareTo(list.get(1)) != 1) {
                return list;
            } 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);
    }
}
```



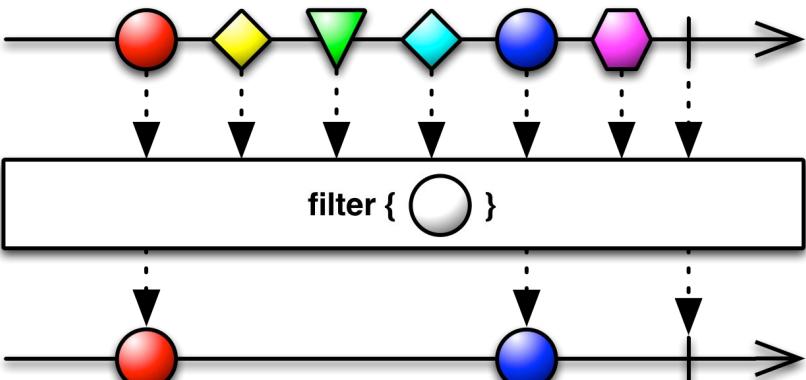
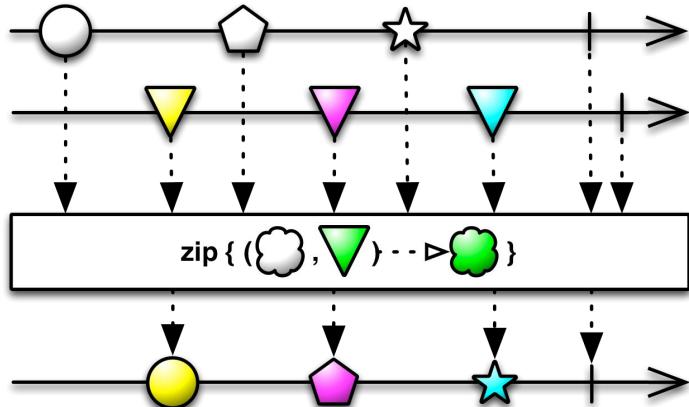
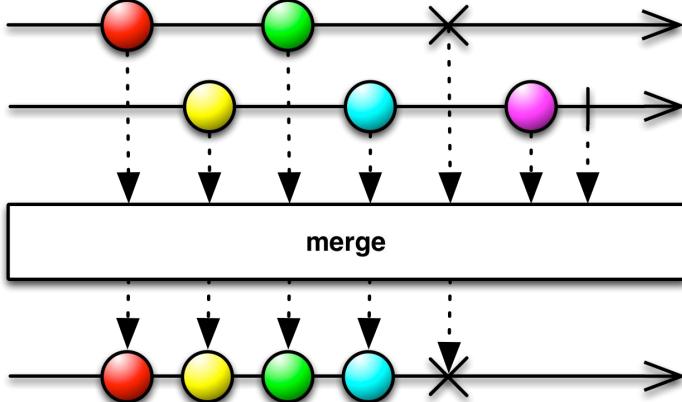
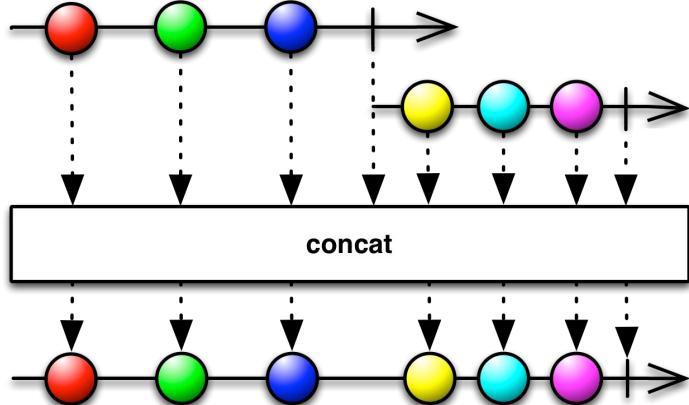
```
public class MergeSortTask<T extends Comparable<T>> extends RecursiveTask<List<T>> {
    @Override
    protected List<T> compute() {
        if (list.size() < 2) { return list; }
        if (list.size() == 2) {
            if (list.get(0).compareTo(list.get(1)) != 1) {
                return list;
            } 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

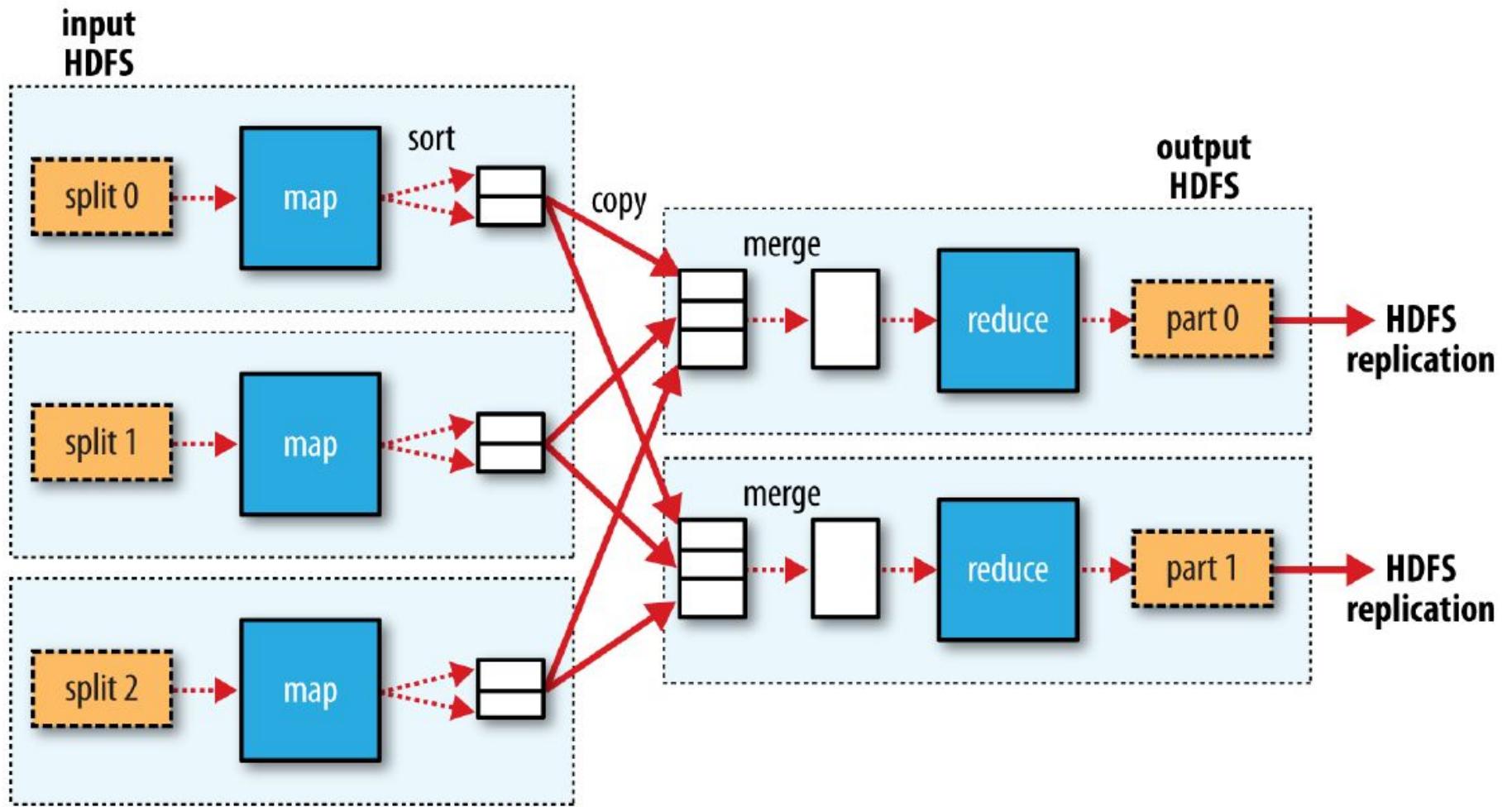
In Spring's approach to building RESTful web services, HTTP requests are handled by a controller. These components are easily identified by the `@RestController` annotation, and the `GreetingController` below handles `GET` requests for `/greeting` by returning a new instance of the `Greeting` class:

src/main/java/hello/GreetingController.java

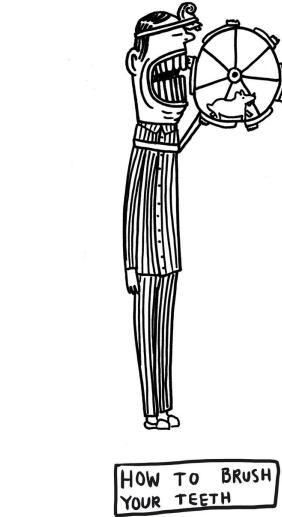
```
@RestController
public class GreetingController {

    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



data parallelism

map/reduce

fork/join

asynchronous execution

actors

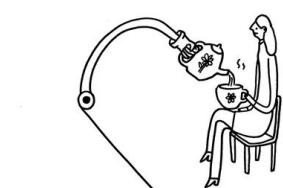
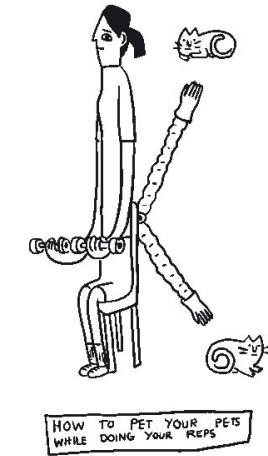
agents

dataflows

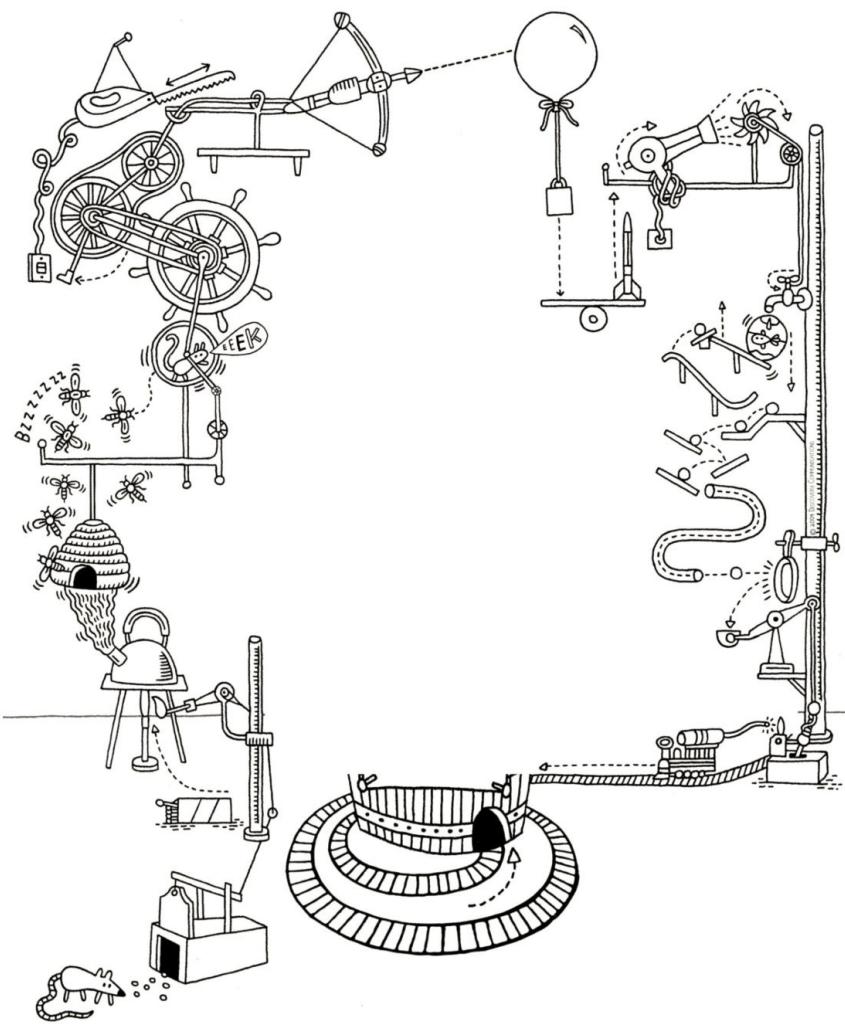
remote execution

Communicating Sequential Processes

Software Transactional Memory



JAVA OR GROOVY?



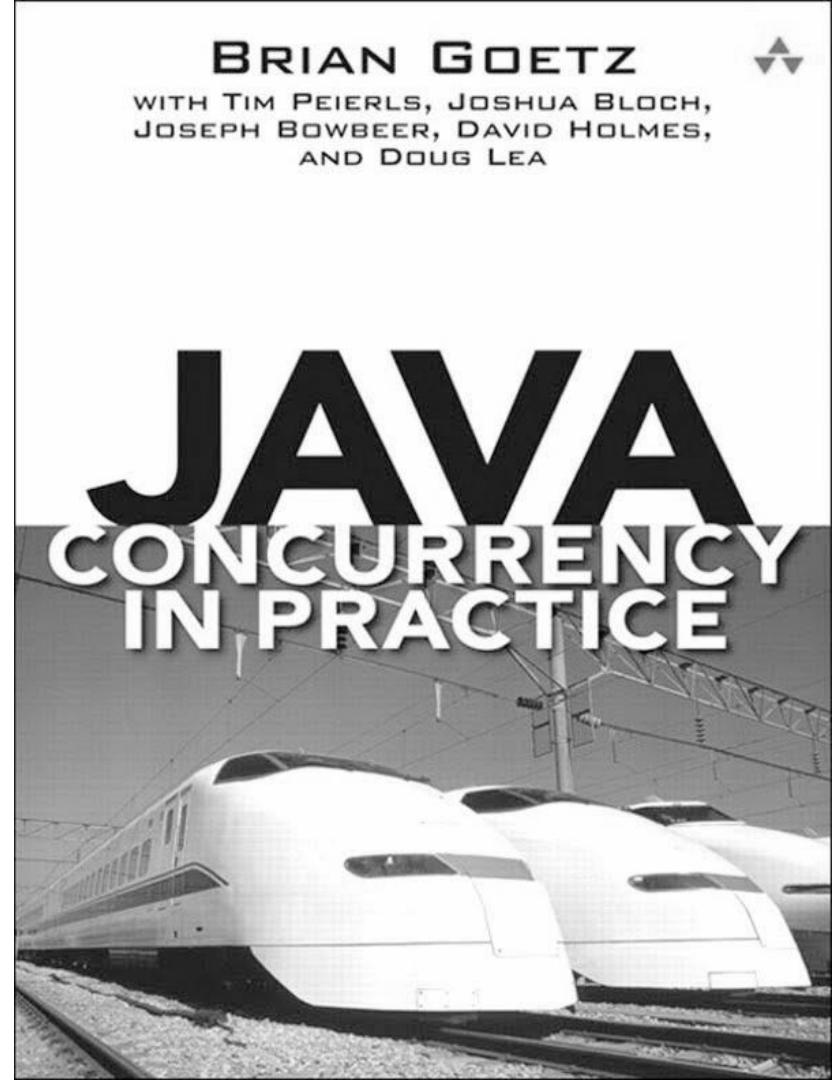
If multiple threads access the same mutable state variable without appropriate synchronization, your program is broken.

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;

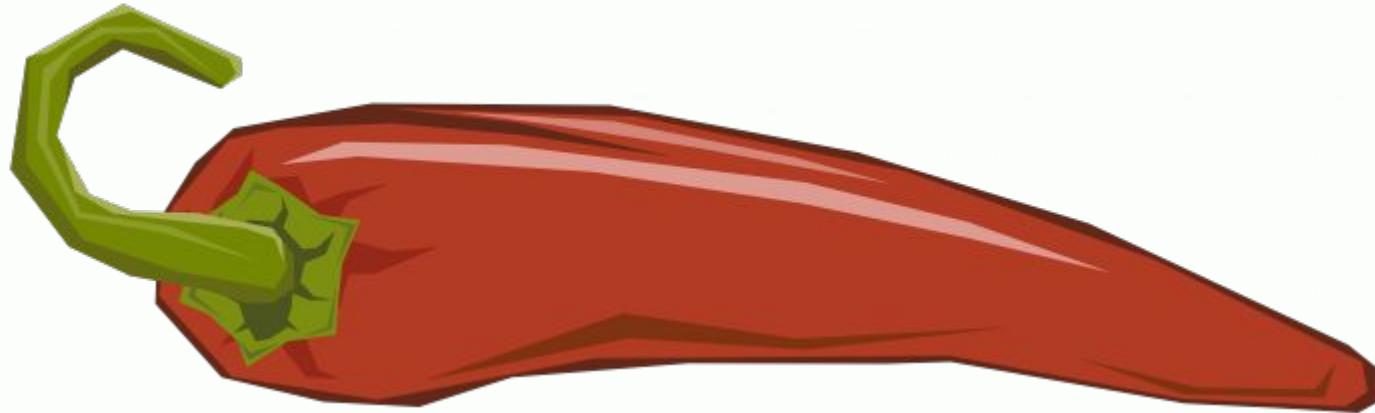
    if (name != null ? !name.equals(that.name) : that.name != null) return false;
    return tweets != null ? tweets.equals(that.tweets) : that.tweets == null;
}

@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) { ... }
}

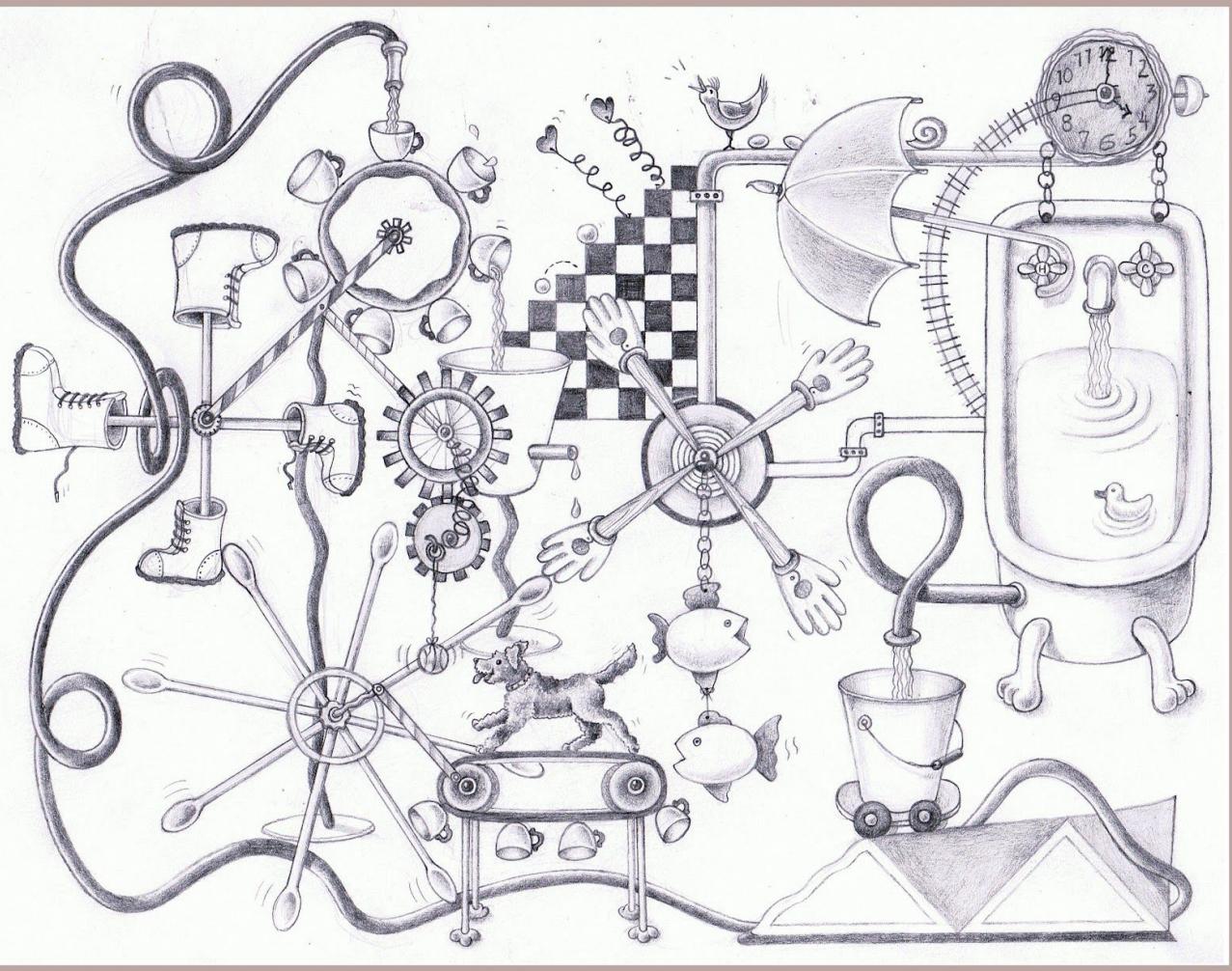
Number.metaClass.doubleIt = { -> delegate * 2 }
ReturnStatement returnStatement = macro { return "42" }

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

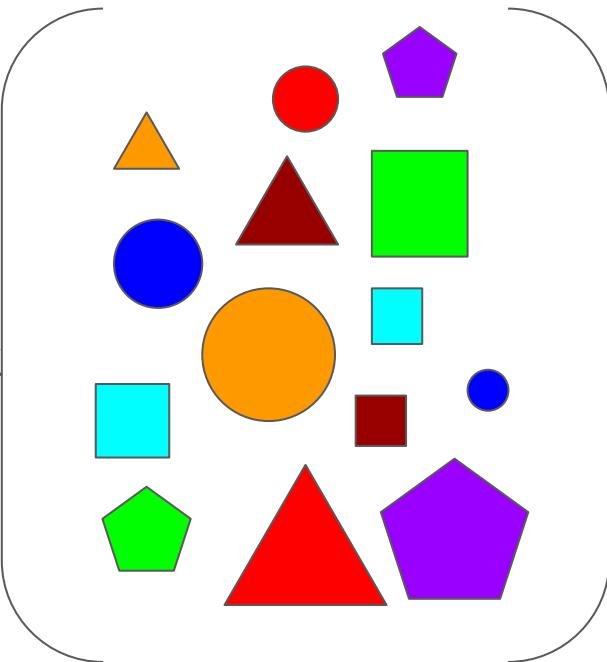
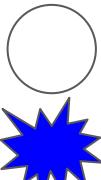
trait DatabaseTest { ... }

please show the square_root of 100
```

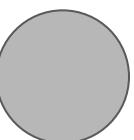
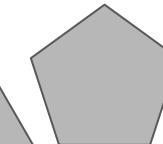
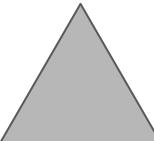
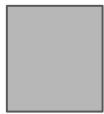
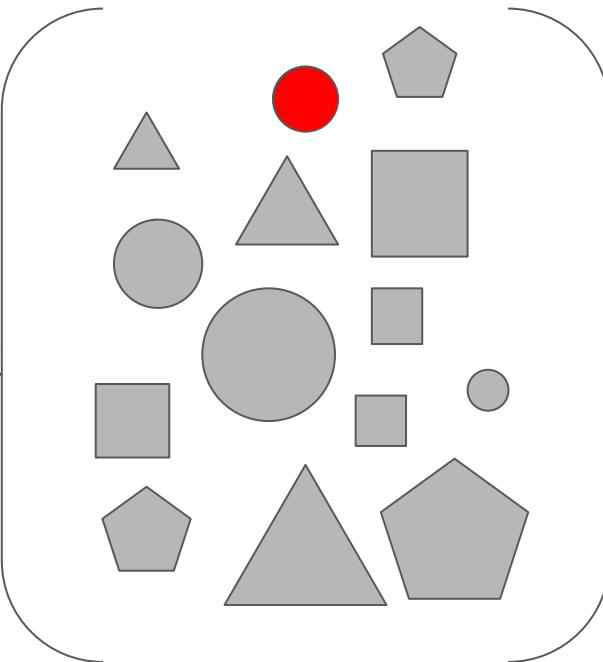
```
new GroovyShell(variables).evaluate 'println persons.sort { person -> person.name }'
```

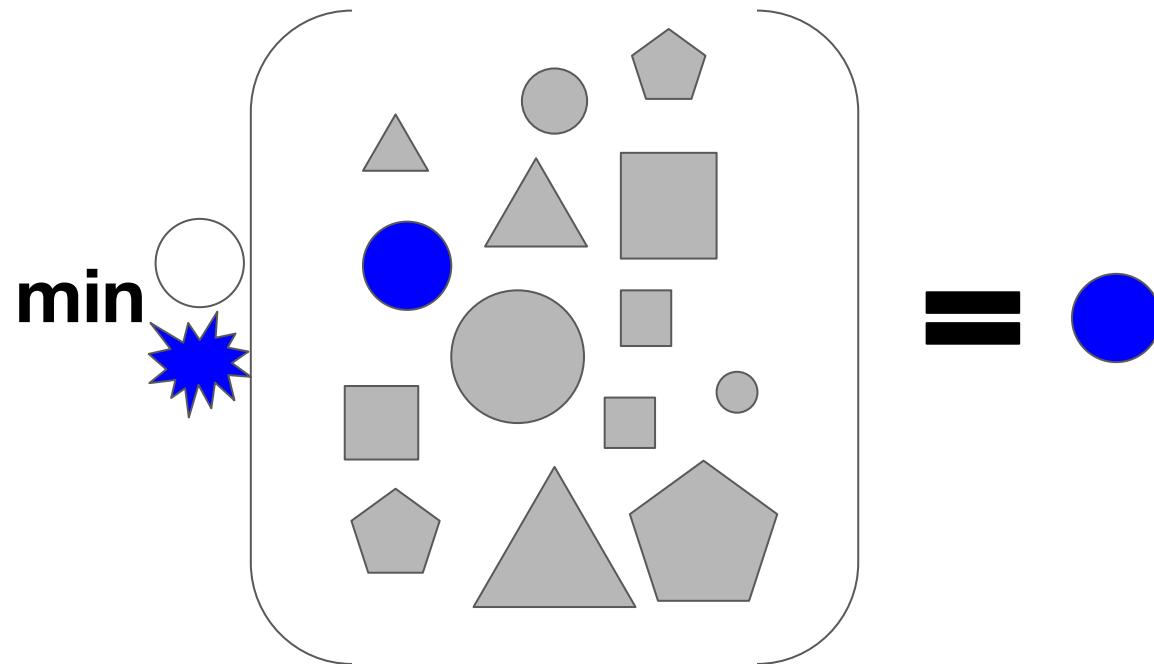


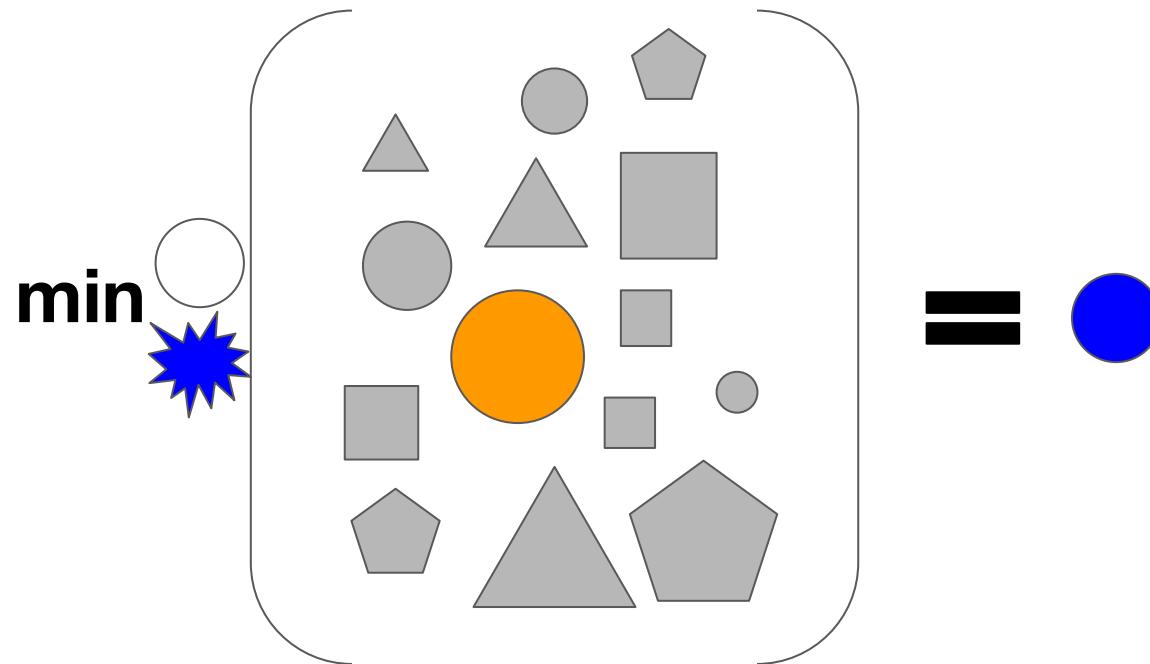
min

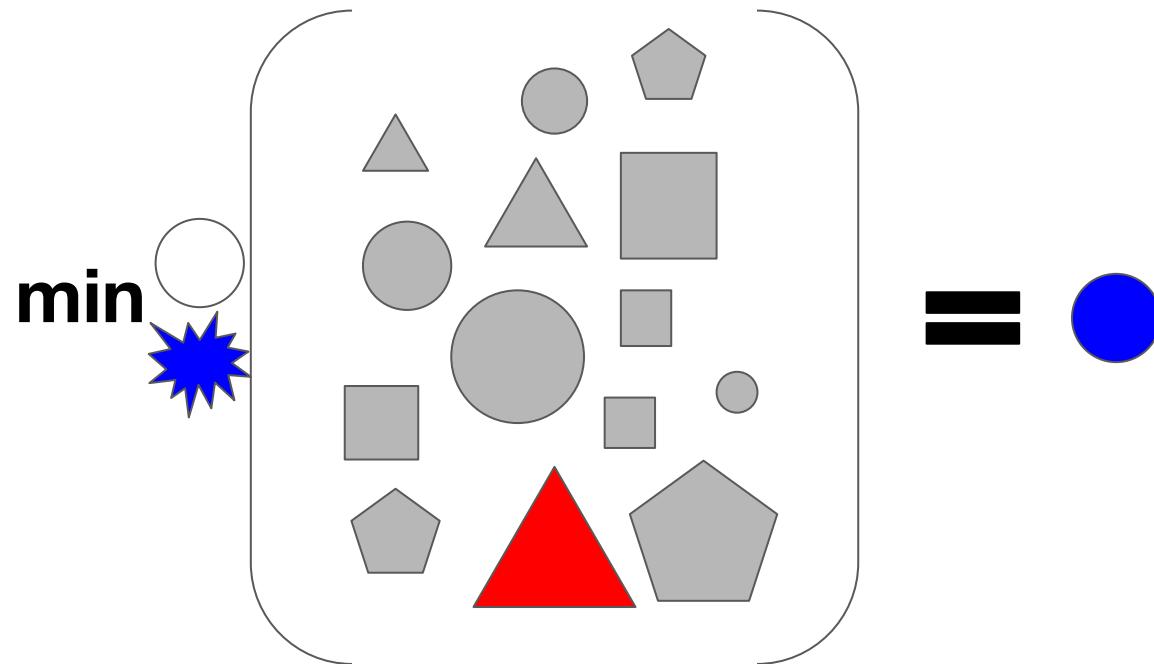


min

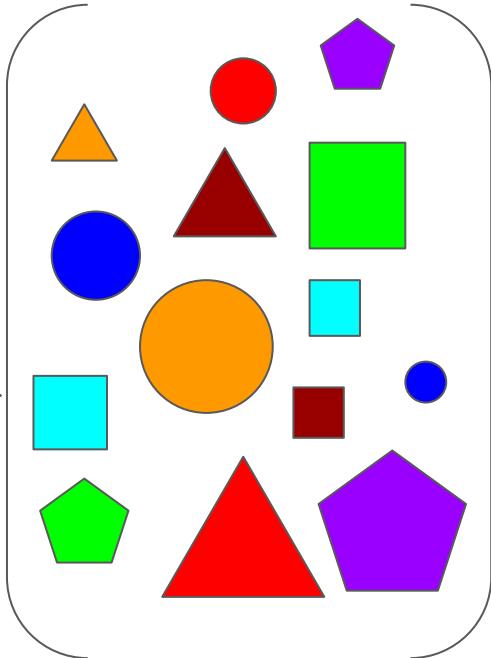
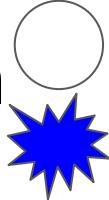




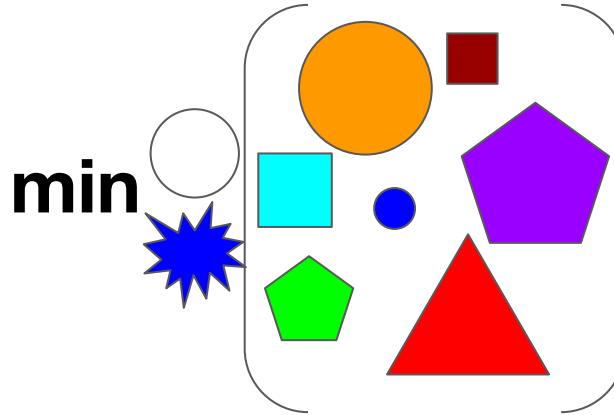




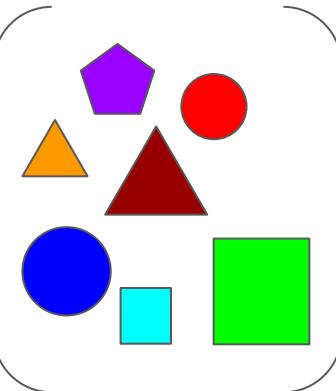
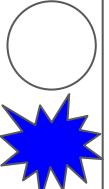
min



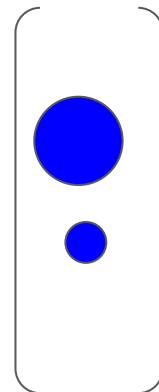
min



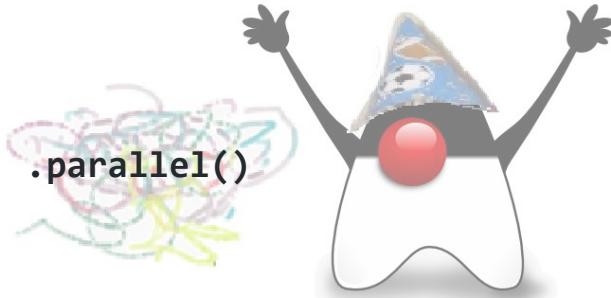
min



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()
```

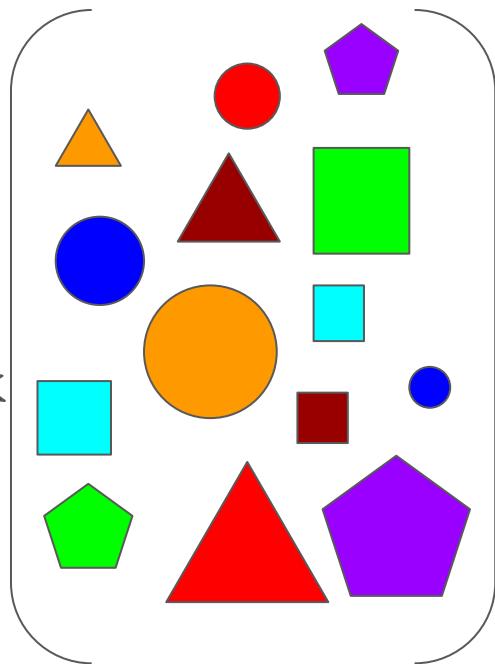
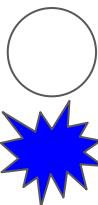
```
Optional<Shape> minBlueCircle = shapes.stream()  
    .filter(shape -> shape.getType() == CIRCLE)
```

```
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

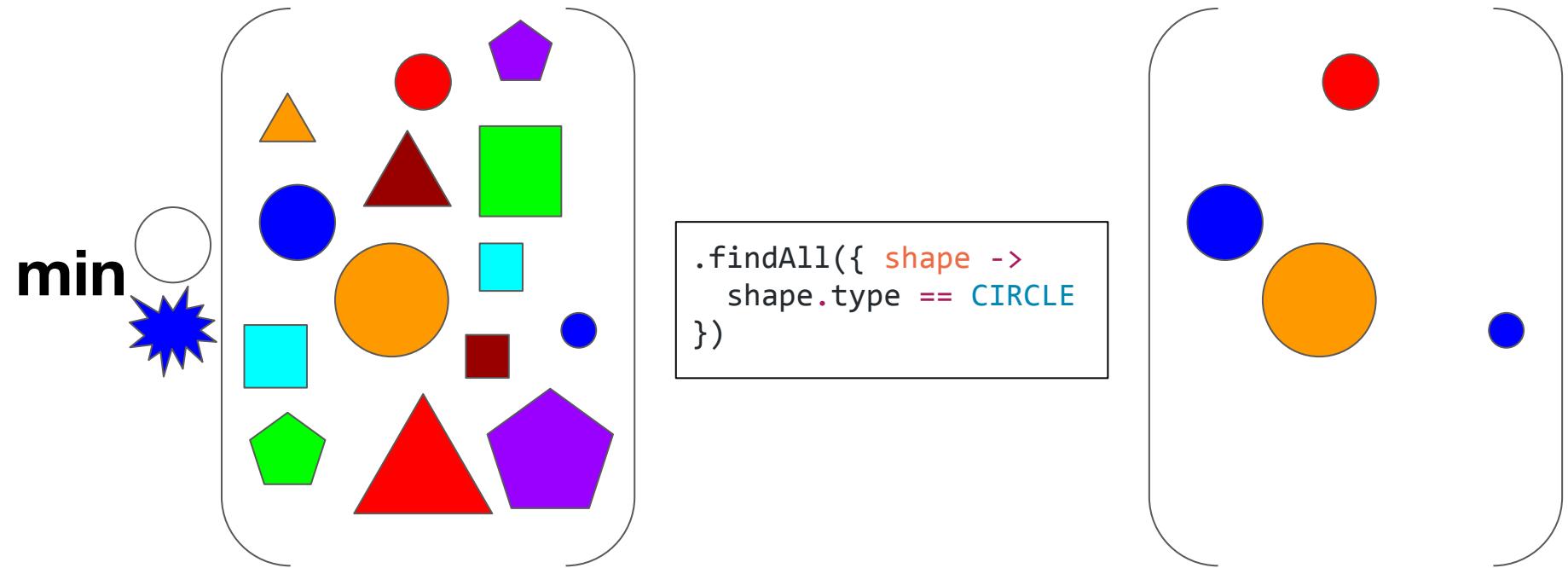
```
Shape minBlueCircle = shapes
    .findAll({ shape -> shape.type == CIRCLE })
```

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

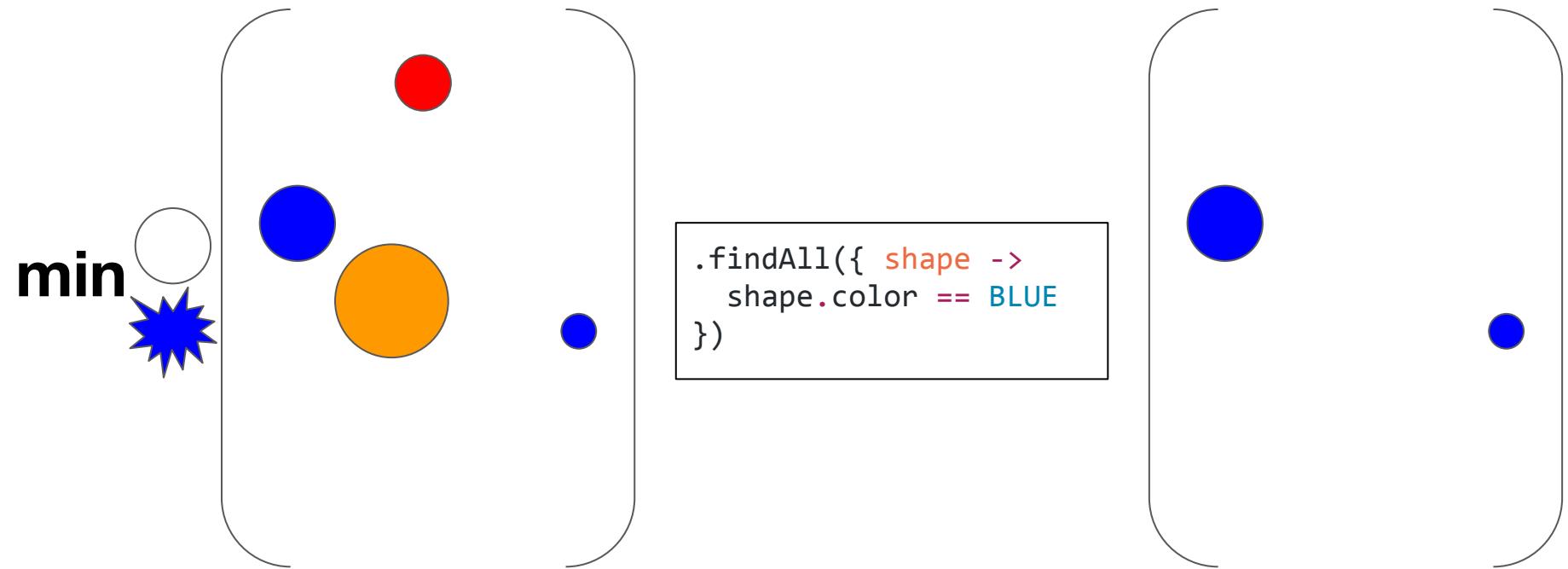
```
Shape minBlueCircle = shapes
    .findAll({ shape -> shape.type == CIRCLE })
    .findAll({ shape -> shape.color == BLUE })
    .min({ shape -> shape.size })
```

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

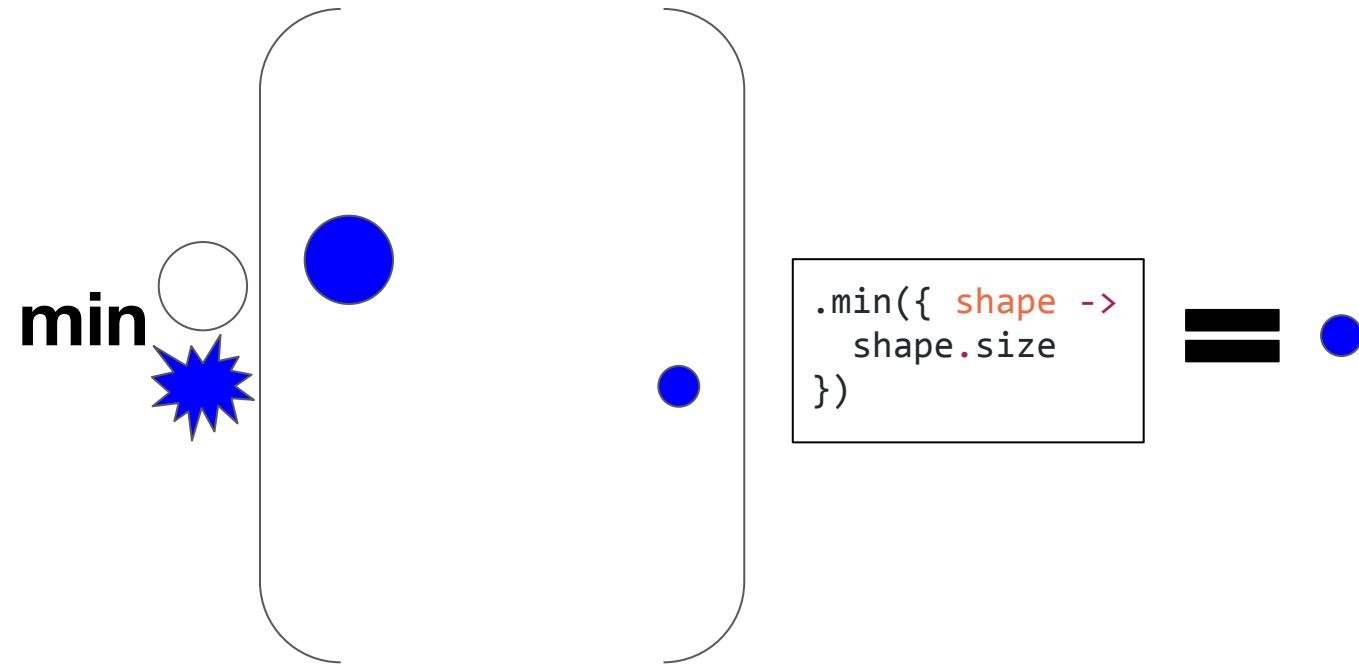
```
ParallelEnhancer.enhanceInstance shapes
shapes.makeConcurrent()
Shape minBlueCircle = shapes
    .findAll({ shape -> shape.type == CIRCLE })
    .findAll({ shape -> shape.color == BLUE })
    .min({ shape -> shape.size })
```



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



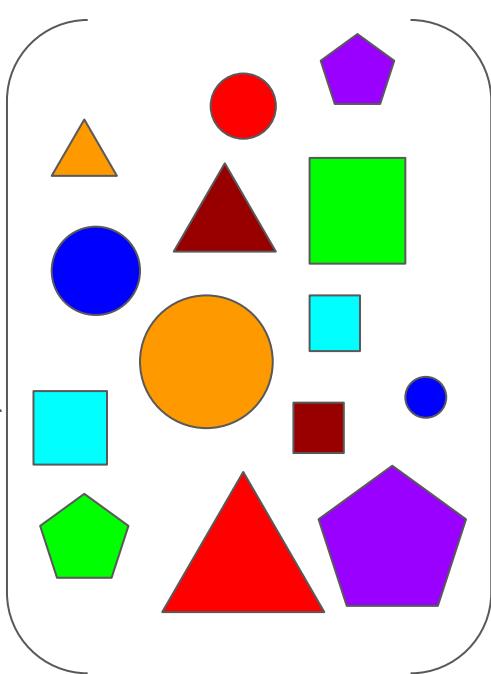
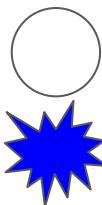
```
ParallelEnhancer.enhanceInstance shapes
shapes.makeConcurrent()
Shape minBlueCircle = shapes
    .findAll({ shape -> shape.type == CIRCLE })
    .findAll({ shape -> shape.color == BLUE })
    .min({ shape -> shape.size })
```



```
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 })  
}
```

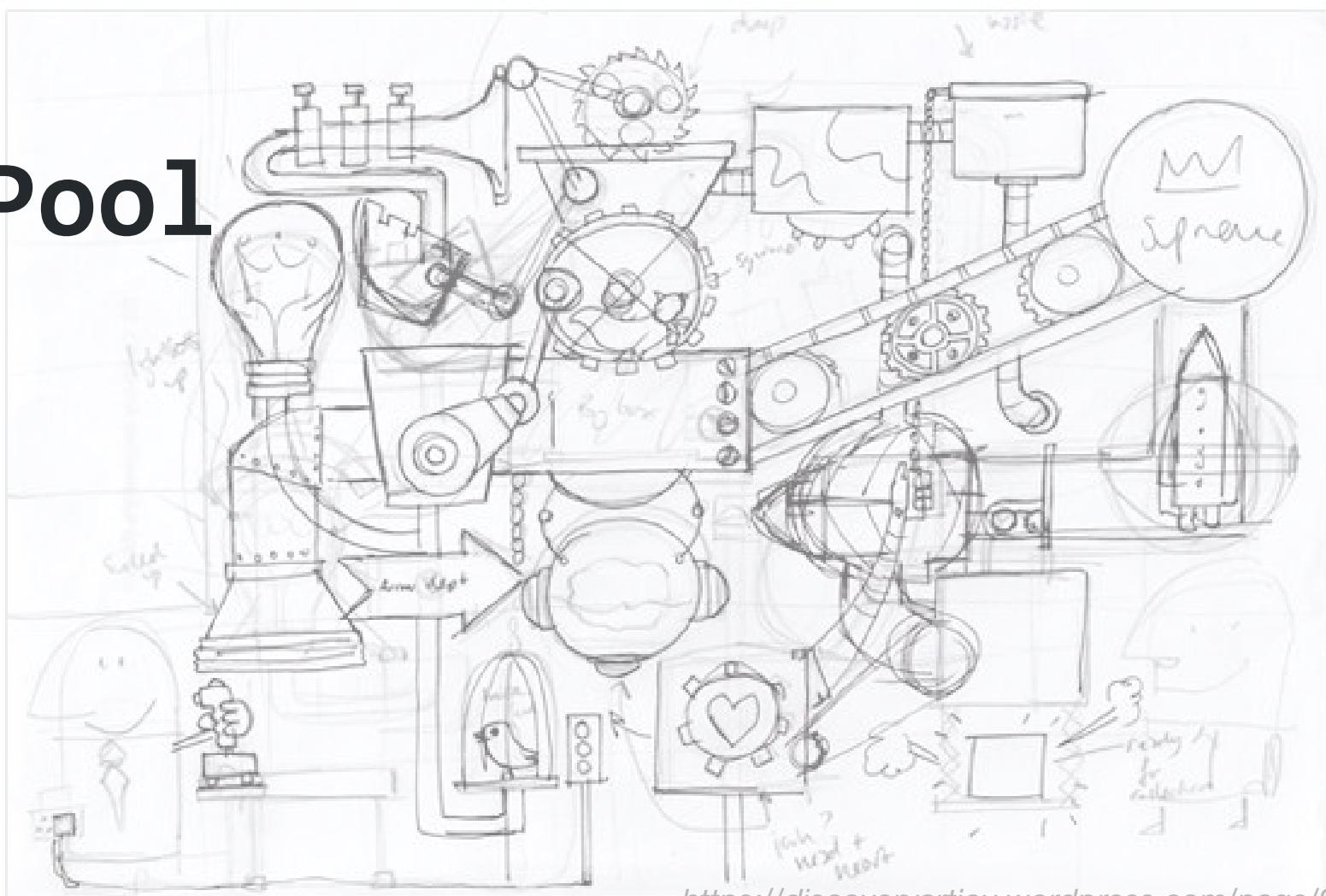
min



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



withPool



```
withPool {
```

```
}
```

```
GParsPool.withPool {
```

```
}
```

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

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

```
}
```

```
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  
  
}
```

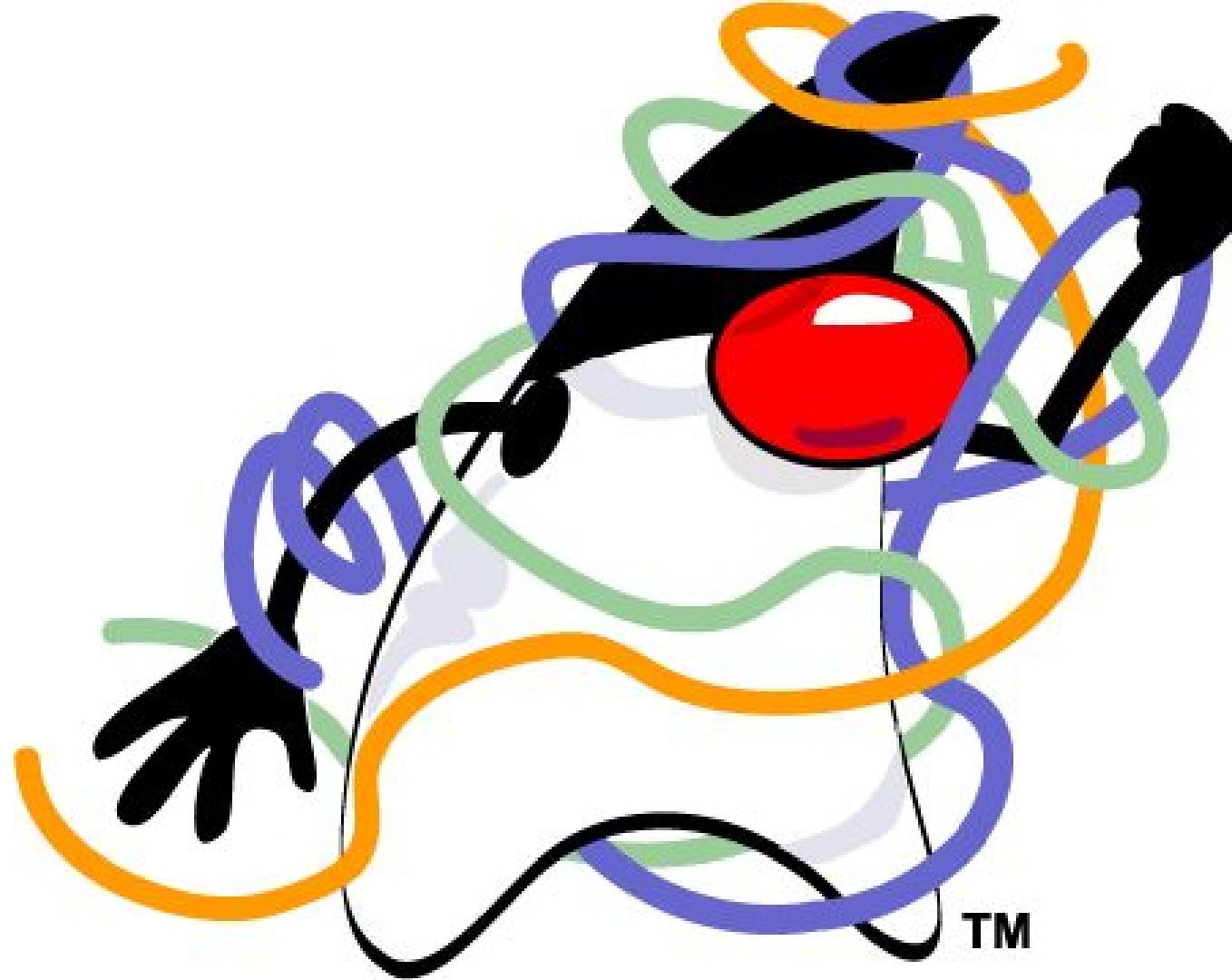
```
withPool {  
  
    Closure fetchAsync = twitter.&fetchLatestTweets.asyncFun()  
  
    Closure watsonsExtractKeywords = watsons.&extractKeywords.curry(fetchAsync())  
    Closure googleExtractKeywords = google.&extractKeywords.curry(fetchAsync())  
  
    List keywords = speculate( watsonsExtractKeywords(),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



TM

```
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)

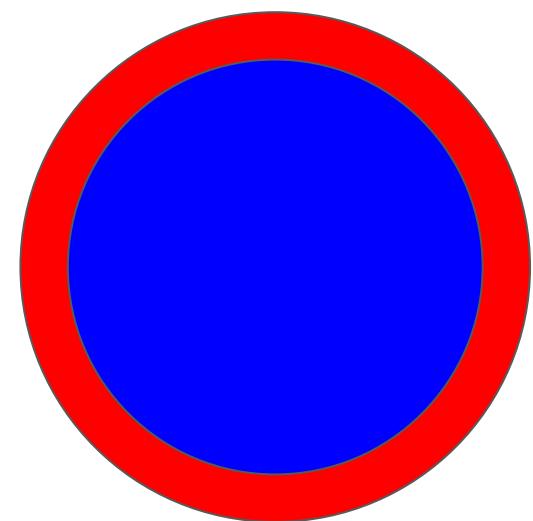
Thread #2

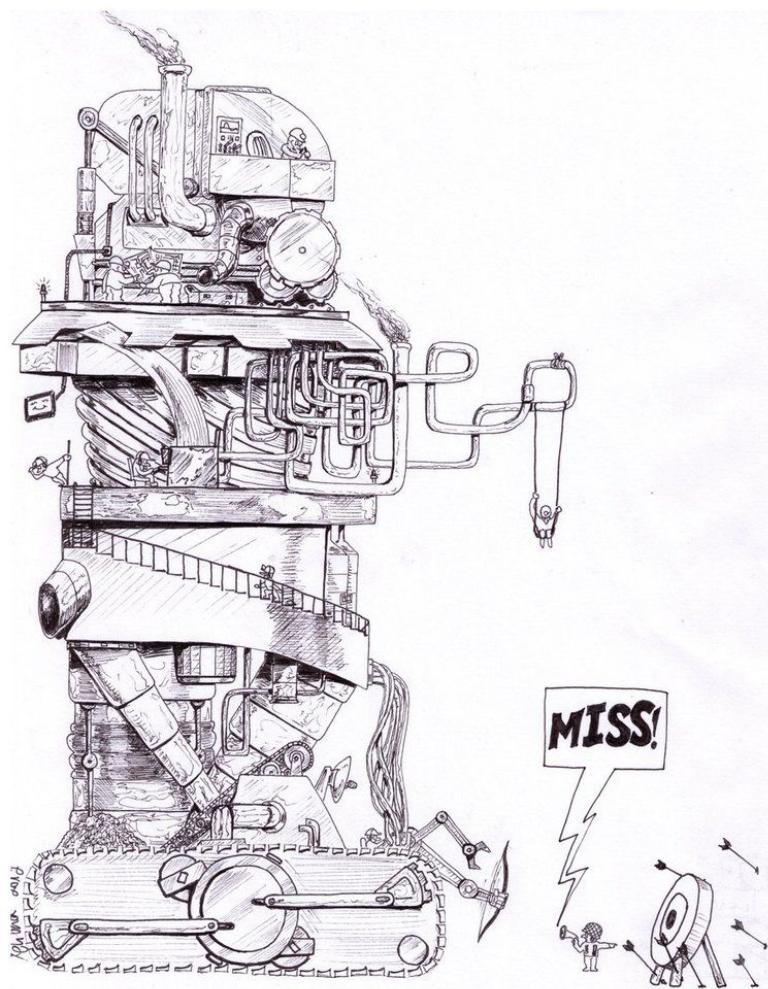
blueCircles.add(shape)

blueCircles.add(shape)

blueCircles.add(shape)

synchronized

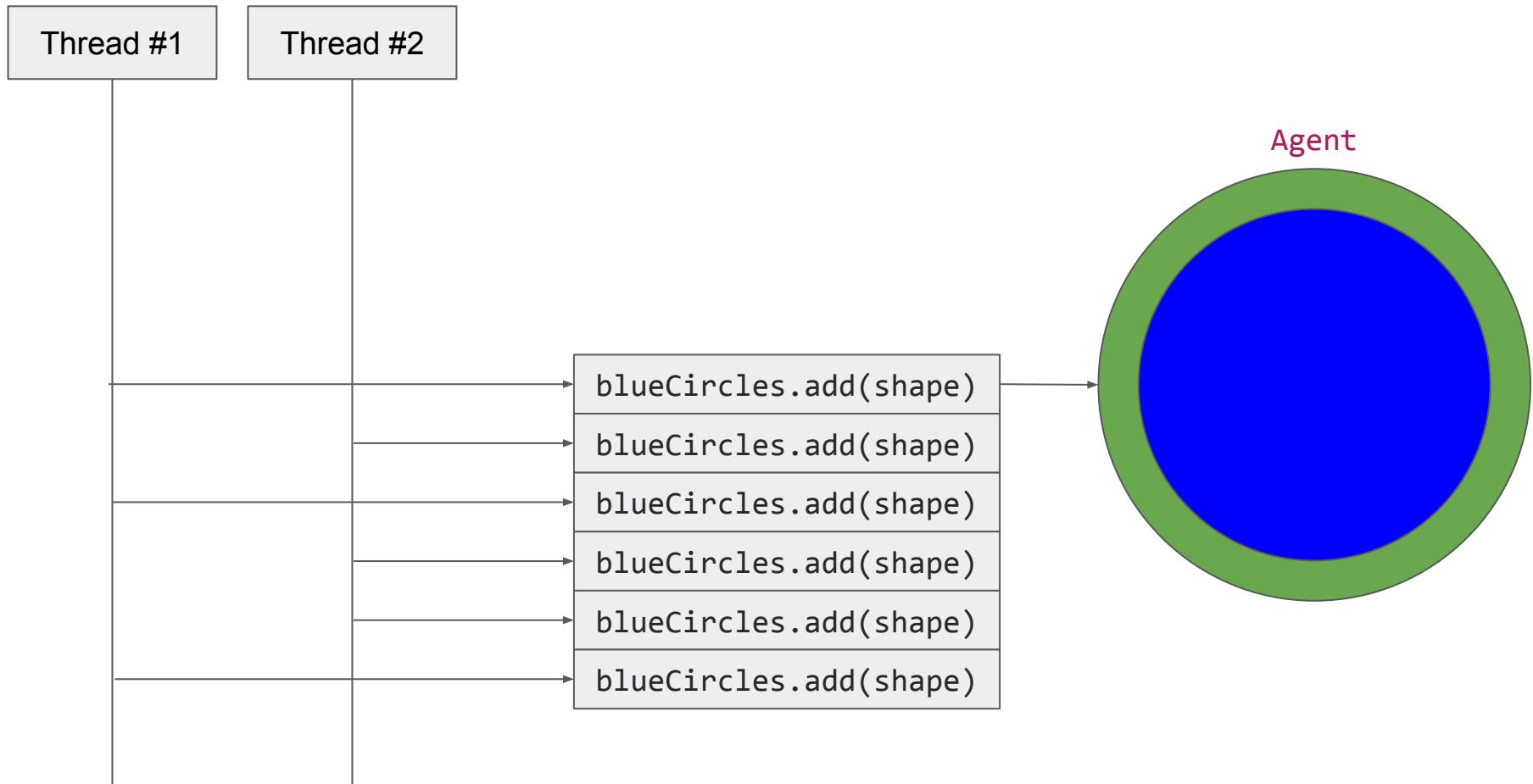




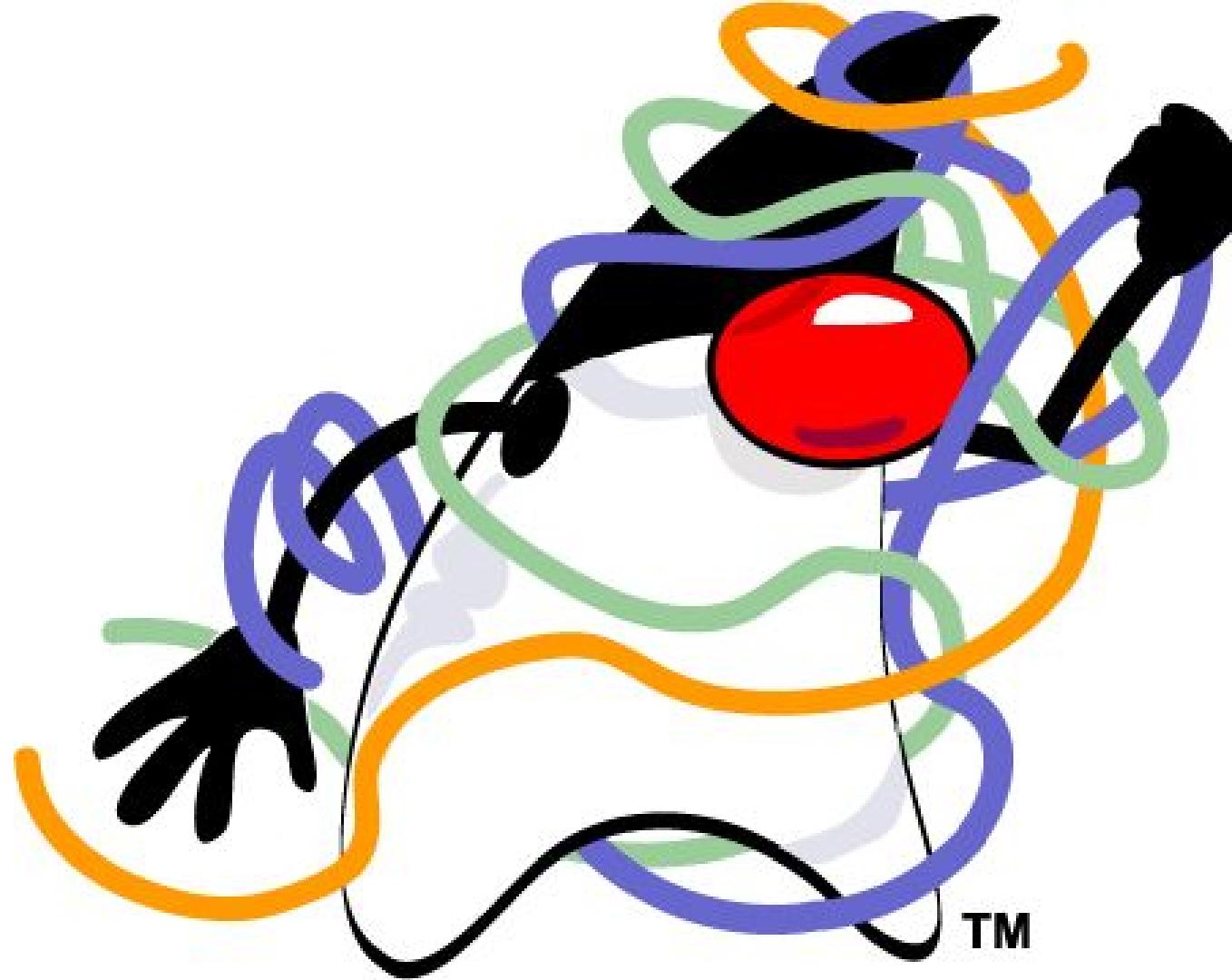
AGENTS

<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 }  
  
}
```



TM

```
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);
    }
});

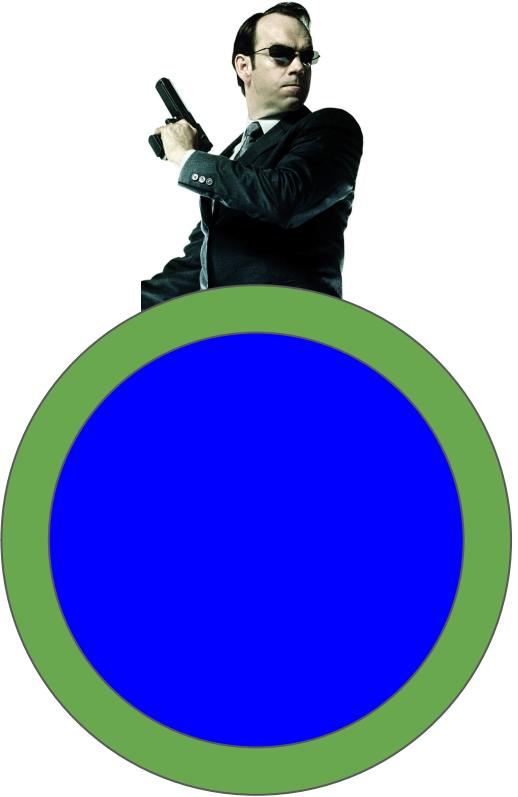
blueShapesGuard.valAsync(new MessagingRunnable<List<Shape>>() {
    @Override
    protected void doRun(List<Shape> value) {
        System.out.println(value);
    }
});
```

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

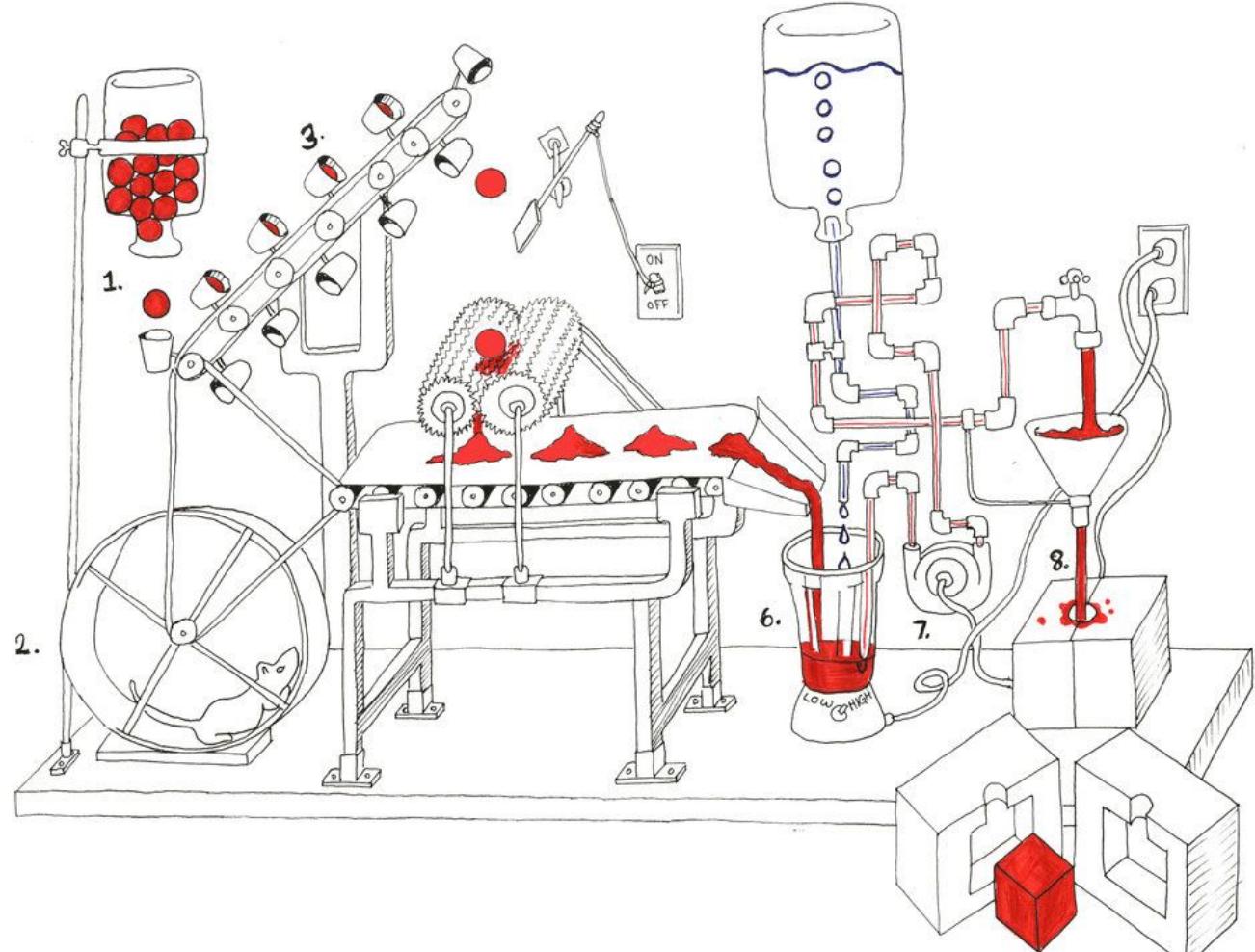
Thread #2



Actor



ACTORS



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

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

```
class KeywordStoreActor extends DefaultActor {
    KeywordsRepository keywordsRepository

    void act() {
        loop {
            react { def keyword -> keywordsRepository.store keyword }
        }
    }
}
```

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

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

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

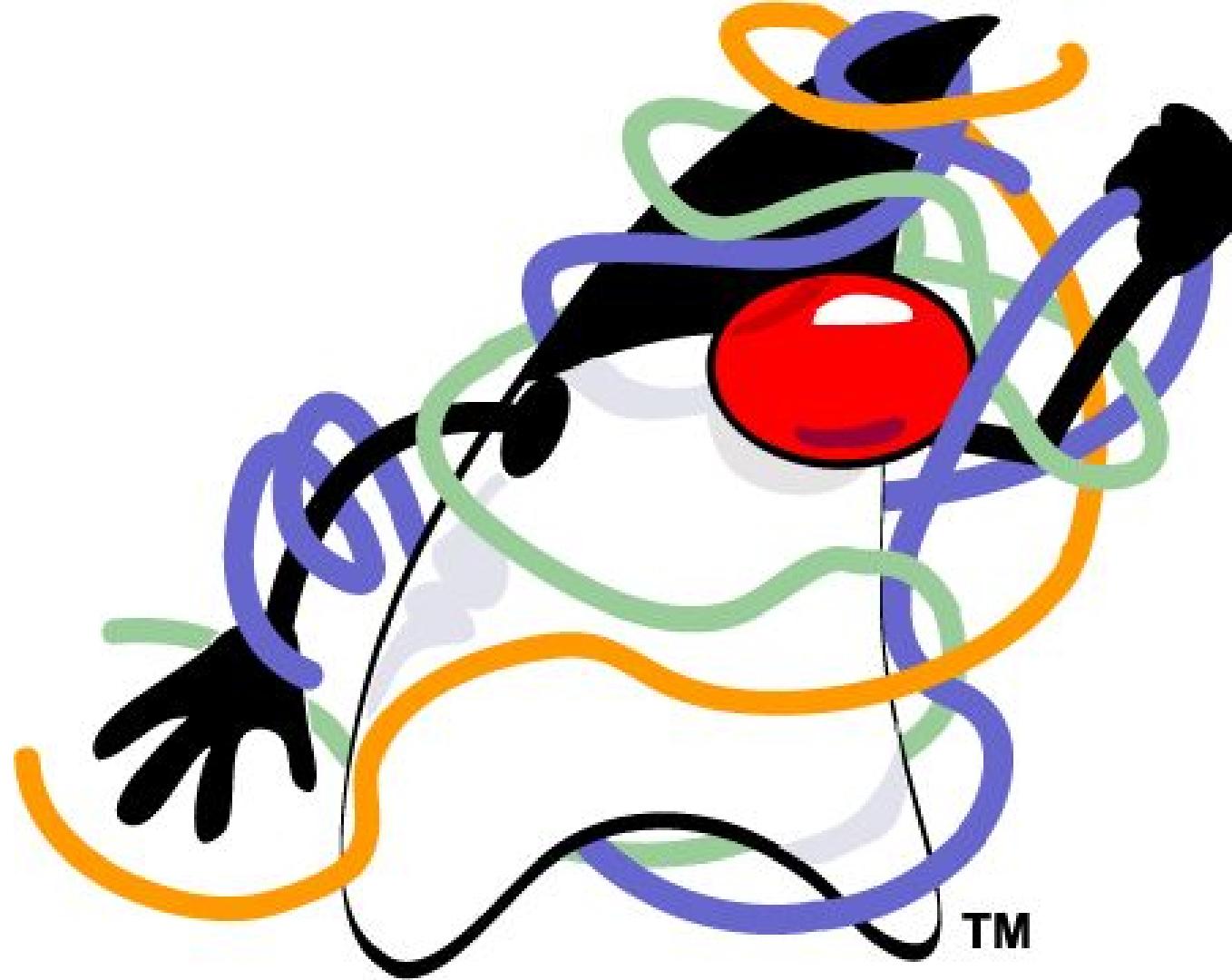
def idGenerator = actor {
    loop {
        react { def keyword ->
            Id id = ...
            keyword.reply id
        }
    }
}
```

process → *react* { *Id id* ->
answer *keywordsRepository.store id, keyword*

↑ reply to sender

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

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



TM

```
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 { def 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 { def keyword ->
            idGenerator << keyword
            react { Id id ->
                keywordsRepository.store id, keyword
            }
        }
    }
}

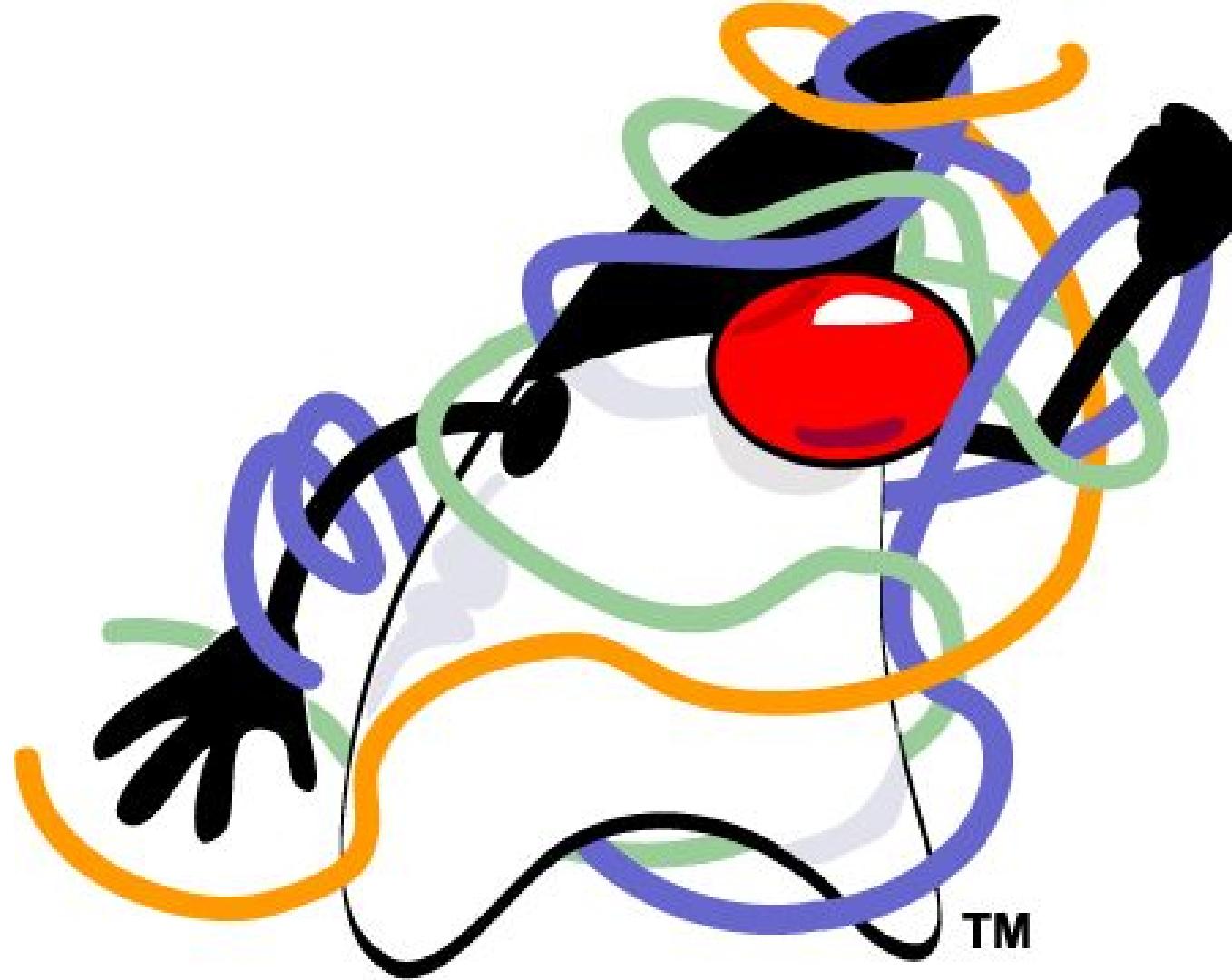
def idGenerator = reactor { def message ->
    if (message instanceof Keyword) {
        id = ...
    }
    if (message instanceof Tweet) {
        id = ...
    }
    message.reply id
}
```

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

def idGenerator = messageHandler {
    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));
    }
}
```

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

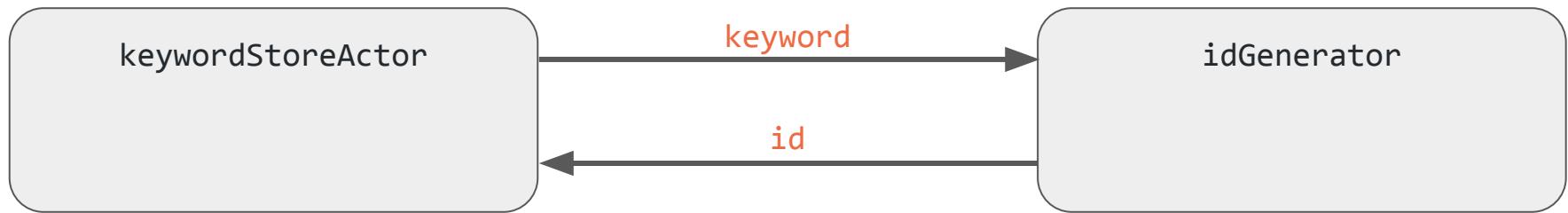
```
def idGenerator = reactor { def keyword ->
    Id id = generateId(keyword)
    keyword.reply id
}
```

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

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

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

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



keywordStoreTask

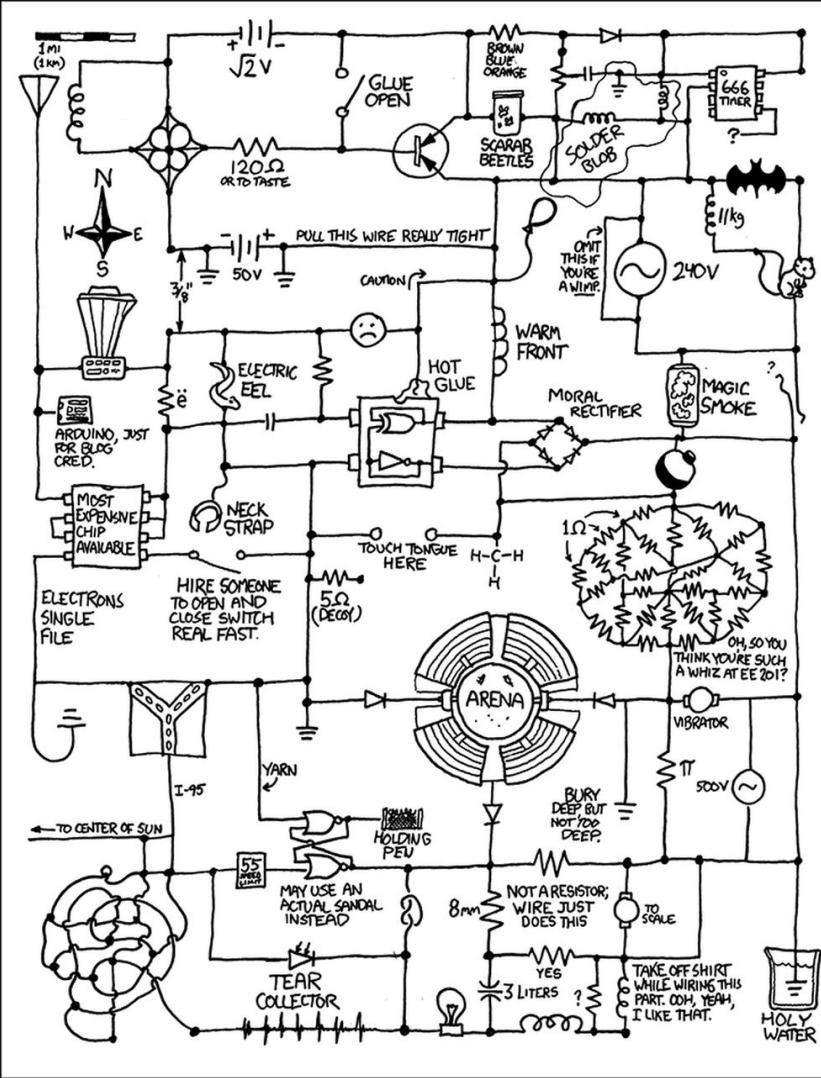
get
keyword

get
id

idGeneratorTask

get
keyword

set
id



DATAFLOWS

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

```
final def keyword = new DataflowVariable() ← created
final def id = new DataflowVariable() ← created
```



```
final def keyword = new DataflowVariable() ← created
final def id = new DataflowVariable() ← created

task {
    keywordsRepository.store id, keyword ← blocked because id and keyword are not bound
}

task {
    id = generateId(keyword) ← blocked because keyword is not bound
}
```

```
final def keyword = new DataflowVariable() ← created
final def id = new DataflowVariable() ← created

task {
    keywordsRepository.store id, keyword ← blocked because id and keyword are not bound
}

task {
    id = generateId(keyword) ← blocked because keyword is not bound
}

task {
    List keywords = keywordsService.extractKeywords()
    keyword = orderByPriority(keywords)[0] ← bind keyword
}
```

```
final def keyword = new DataflowVariable() ← bound
final def id = new DataflowVariable() ← created

task {
    keywordsRepository.store id, keyword ← blocked because id
    is not bound
}

task {
    id = generateId(keyword) ← bind id
}

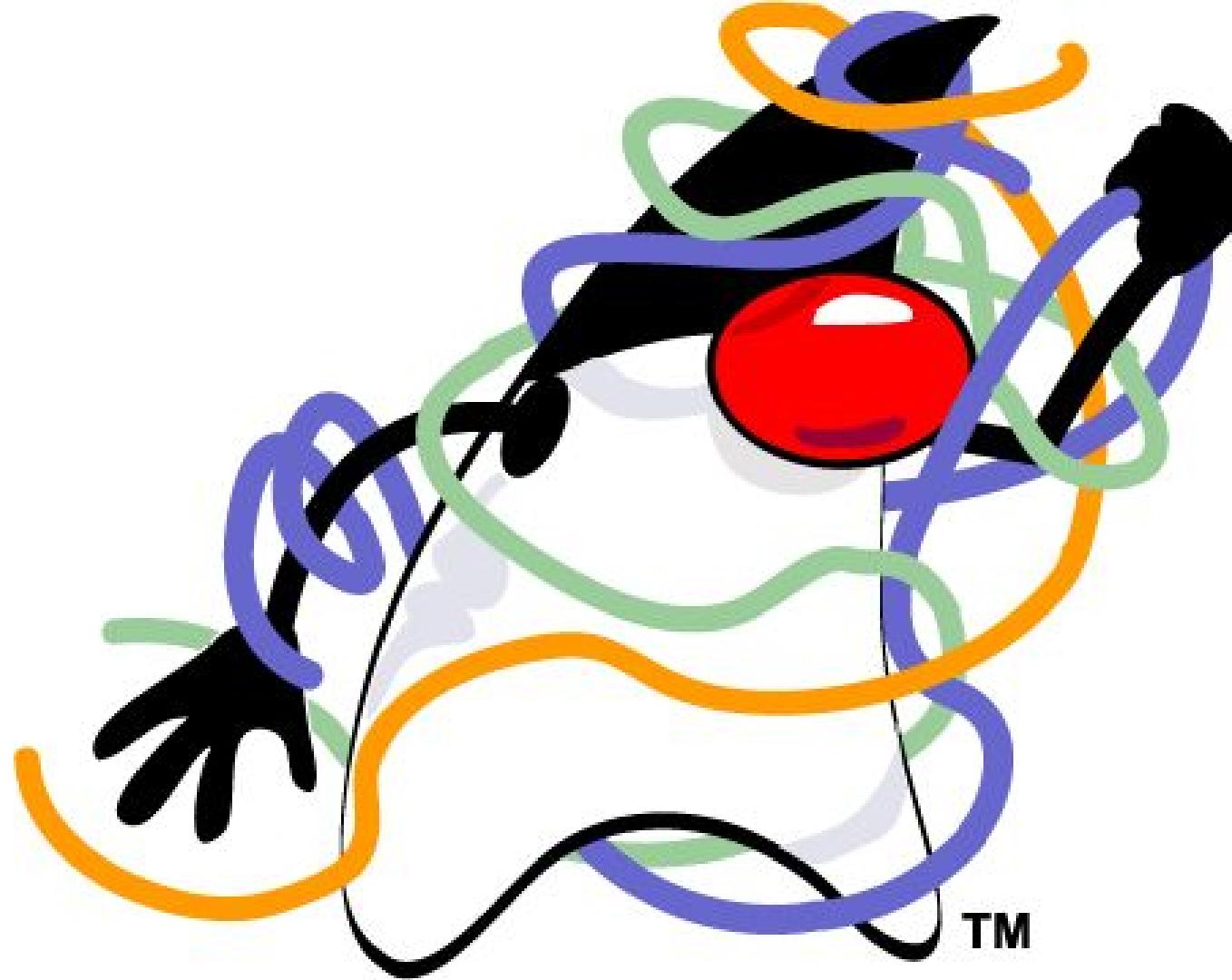
task {
    List keywords = keywordsService.extractKeywords()
    keyword = orderByPriority(keywords)[0]
}
```

```
final def keyword = new DataflowVariable() ← bound
final def id = new DataflowVariable() ← bound

task {
    keywordsRepository.store id, keyword ← done
}

task {
    id = generateId(keyword)
}

task {
    List keywords = keywordsService.extractKeywords()
    keyword = orderByPriority(keywords)[0]
}
```



TM

```
final DataflowVariable<Keyword> keyword = new DataflowVariable<>();
final DataflowVariable<Id> id = new DataflowVariable<>();

task(lambdaToClosure(() -> {
    keywordsRepository.store(id.getVal(), keyword.getVal());
}));
```

```
task(lambdaToClosure(() -> {
    id.bind(generateId(keyword.getVal()));
}));
```

```
task(lambdaToClosure(() -> {
    List<Keyword> keywords = keywordsService.extractKeywords();
    keyword.bind(orderByPriority(keywords).get(0));
}));
```

```
final def keyword = new DataflowVariable()
final def id = new DataflowVariable()

task { ← started asynchronously           blocked because id
       keywordsRepository.store id, keyword ← and keyword are
}                                         not bound

task {
    id = generateId(keyword)
}

task {
    List keywords = keywordsService.extractKeywords()
    keyword = orderByPriority(keywords)[0]
}
```

```
final def keyword = new DataflowVariable()
final def id = new DataflowVariable()

whenAllBound(keyword, id) { ← blocked until all mentioned variables are bound
    keywordsRepository.store id, keyword
}

task {
    id = generateId(keyword)
}

task {
    List keywords = keywordsService.extractKeywords()
    keyword = orderByPriority(keywords)[0]
}
```

```
final def keywords = new DataflowVariable()

task {
    def keyword = orderByPriority(keywords)[0]
    keywordsRepository.store keyword
}

task {
    keywords = keywordsService.extractKeywords()
}
```

```
final def keywords = new DataflowVariable()

keywords.whenBound {
    def keyword = orderByPriority(keywords)[0]
    keywordsRepository.store keyword
}

task {
    keywords = keywordsService.extractKeywords()
}
```

```
task {
    keywordsService.extractKeywords()
}.whenBound {
    def keyword = orderByPriority(keywords)[0]
    keywordsRepository.store keyword
}
```

```
task {
    keywordsService.extractKeywords()
}.whenBound {
    Promise orderedKeywords = orderByPriorityAsync(keywords)
    orderedKeywords.whenBound {
        keywordsRepository.store orderedKeywords[0]
    }
}
```

```
task {  
    keywordsService.extractKeywords()  
}.then orderByPriority then { it[0] } then keywordsRepository.&store
```

```
task {  
    keywordsService.extractKeywords()  
}.then priorityOrderActor.&sendAndPromise then { it[0] } then  
repositoryStoreActor.&sendAndPromise
```

```
final def keywords = new DataflowQueue()

task {
    while (true) { keywordsRepository.store keywords.val }
}

task {
    keywordsService.extractKeywords().each { keyword ->
        keywords << keyword
    }
}
```

The diagram illustrates the interaction between two tasks. The top task, which runs a loop to store values from a repository, has an arrow pointing to the expression `keywords.val`. A handwritten-style note next to this arrow reads *blocked until new value is bound*. The bottom task, which extracts keywords from a service and adds them to a queue, has an arrow pointing to the assignment expression `keywords << keyword`. Another handwritten-style note next to this arrow reads *bind new value*.

```
final def keywords = new DataflowQueue()

task {
    while (true) { keywordsRepository.store keywords.val }
}

task {
    withPool {
        List allKeywords = keywordsService.extractKeywords()
        allKeywords.findAllParallel({!spamService.isSpam(it)}).each { keyword ->
            keywords << keyword
        }
    }
}
```

```
final def keywords = new DataflowQueue()

keywords.wheneverBound { ← started each time value is bound
    keywordsRepository.store it
}

task {
    keywordsService.extractKeywords().each { keyword ->
        keywords << keyword
    }
}
```

```
final def keywords = new DataflowQueue()

task {
    while (true) { keywordsRepository.store keywords.val }
}

task {
    keywordsService.extractKeywords().each { keyword ->
        keywords << keyword
    }
}

task {
    while (true) { logService.debug keywords.val }
}
```

```
final def keywordsBroadcast = new DataflowBroadcast()
final def keywordsRepositoryChannel = keywordsBroadcast.createReadChannel()
final def keywordsLogChannel = keywordsBroadcast.createReadChannel()

task {
    while (true) { keywordsRepository.store keywordsRepositoryChannel.val }
}

task {
    keywordsService.extractKeywords().each { keyword ->
        keywordsBroadcast << keyword
    }
}

task {
    while (true) { logService.debug keywordsLogChannel.val }
}
```

```
final def keywords = new DataflowQueue()

splitter (keywords, [ keywordsForStoring, keywordsForLogging ])

operator (inputs: [ keywordsForStoring ], outputs: []) { keywordForStoring ->
    keywordsRepository.store keywordForStoring
}

task {
    keywordsService.extractKeywords().each { keyword ->
        keywords << keyword
    }
}

keywordsForLogging.wheneverBound {
    logService.debug it
}
```

```
final def keywords = new DataflowQueue()

splitter (keywords, [ keywordsForStoring, keywordsForLogging ])

operator (inputs: [ keywordsForStoring ], outputs: [], stateObject: [id: 0]) {
    keywordForStoring -> keywordsRepository.store id++, keywordForStoring
}

task {
    keywordsService.extractKeywords().each { keyword ->
        keywords << keyword
    }
}

keywordsForLogging.wheneverBound {
    logService.debug it
}
```

```
final def keywords = new DataflowQueue()

splitter (keywords, [ keywordsForId, keywordsForLogging ])

operator (inputs: [ id, keywordsWithId ], outputs: []) { id, keywordsWithId ->
    keywordsRepository.store id, keywordsWithId
}

task {
    keywordsService.extractKeywords().each { keyword ->
        keywords << keyword
    }
}

operator (inputs: [ keywordsForId ], outputs: [ id, keywordsWithId ]) { keyword ->
    bindAllOutputValues generateId(keyword), keyword
}

keywordsForLogging.wheneverBound {
    logService.debug it
}
```

```
splitter (keywords, [ keywordsForId, keywordsForLogging ])

operator (inputs: [ id, keywordsWithId ], outputs: []) { id, keywordsWithId ->
    keywordsRepository.store id, keywordsWithId
}

selector (inputs: [ fromWatsons, fromGoogle ], outputs: [ keywords ]) { fromOneOf ->
    fromOneOf.each {
        bindOutput 0, it
    }
}

operator (inputs: [ keywordsForId ], outputs: [ id, keywordsWithId ]) { keyword ->
    bindAllOutputValues generateId(keyword), keyword
}

keywordsForLogging.wheneverBound {
    logService.debug it
}
```

DataflowVariable

can be bound once

*attempt to read
unbound value
suspend execution
until value is bound*

*bound value can be
read multiple times*

DataflowQueue

*can be bound
multiple times*

*attempt to read
unbound value
suspend execution
until new value is
bound*

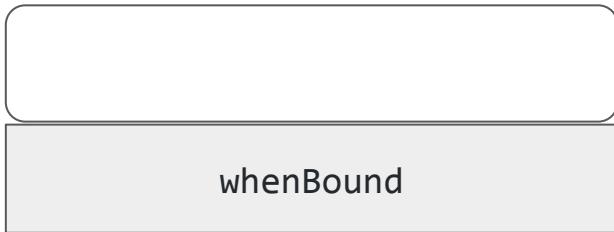
*each bound value can
be read once*

DataflowBroadcast

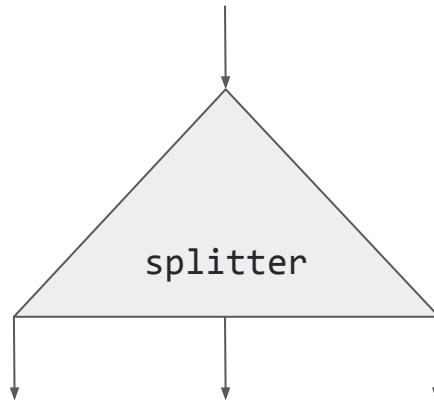
*can be bound
multiple times*

*attempt to read
unbound value
suspend execution
until new value is
bound*

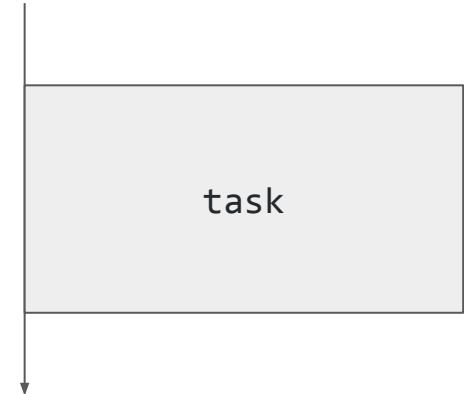
*each bound value can
be read once for
each reader*



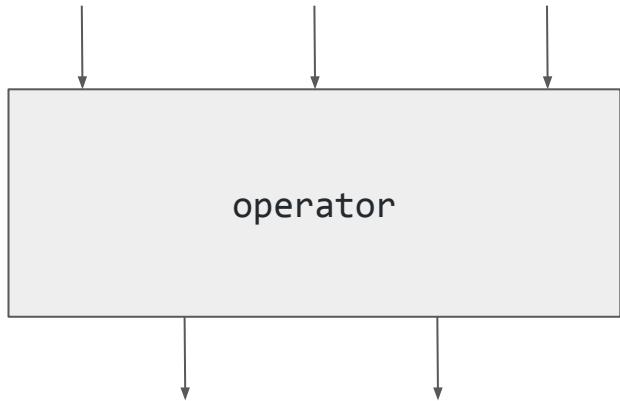
*started when
variable is bound*



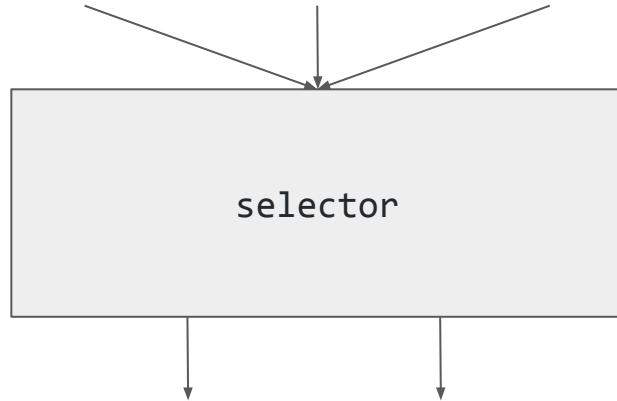
*immediately produce
several outputs
exactly the same as
input*



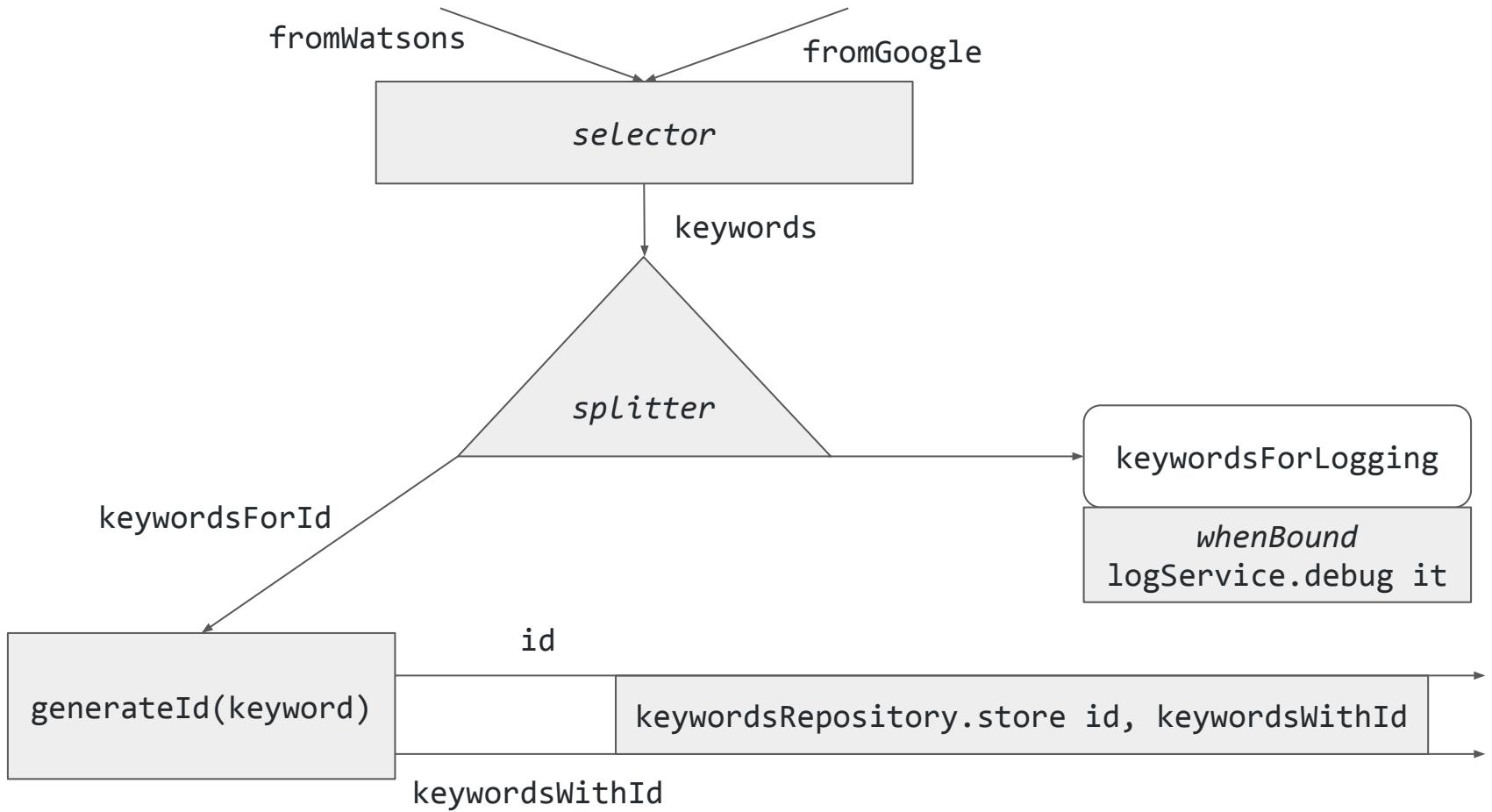
*started immediately,
suspended when reach
unbound value*



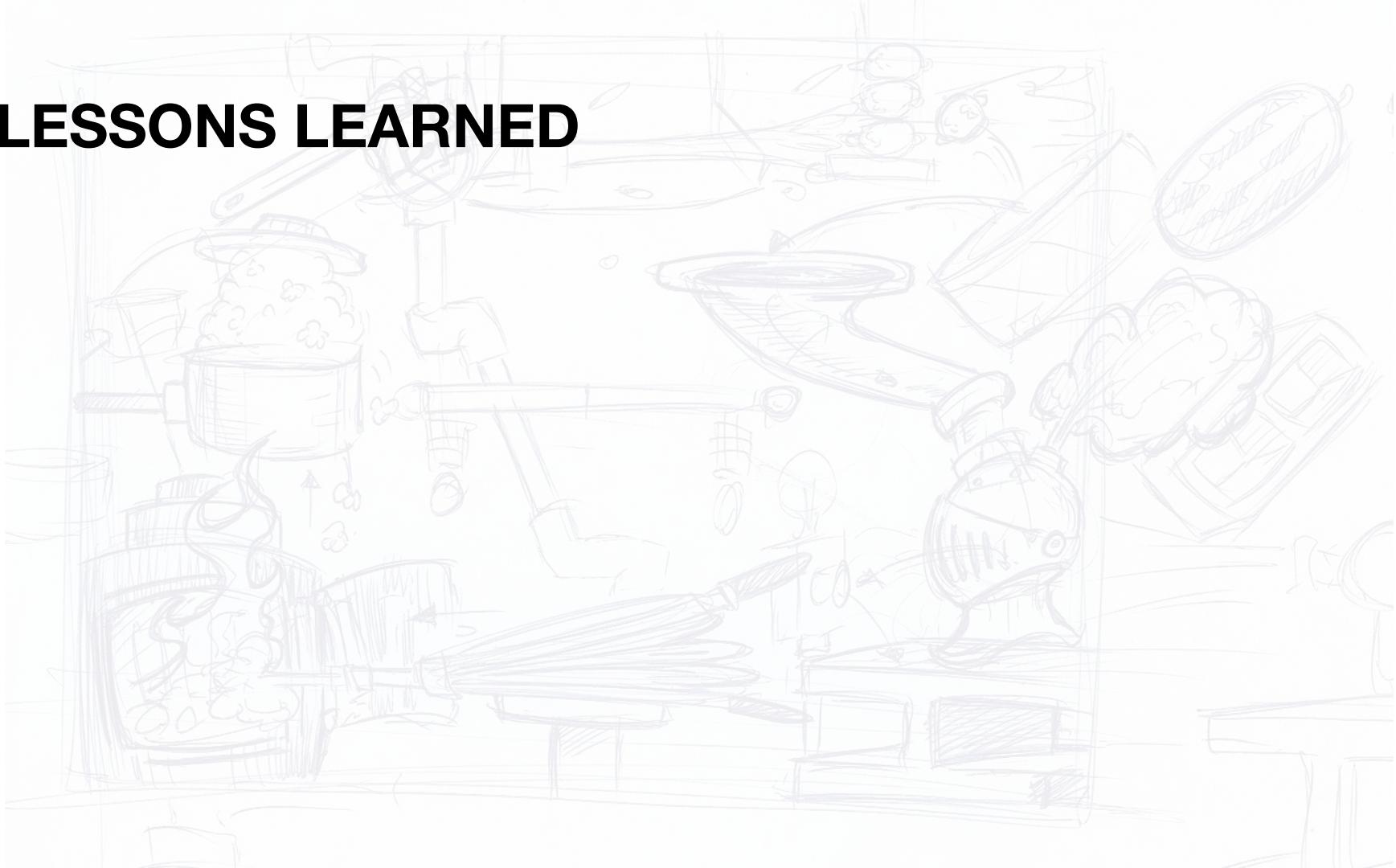
*started when all
inputs are bound,
bind outputs*



*started when at
least one input is
bound, bind outputs*



LESSONS LEARNED

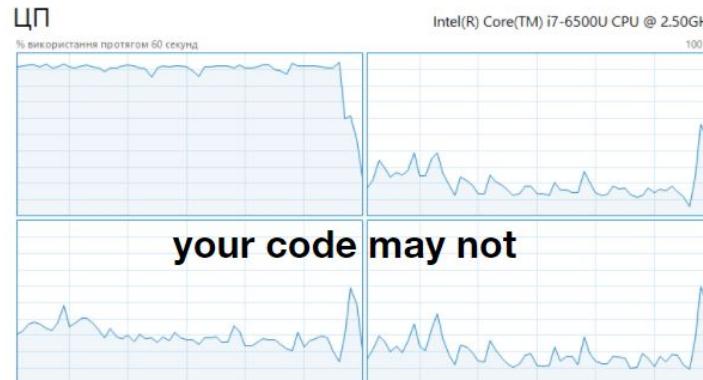


LESSONS LEARNED

guideline #1 for developing concurrent code

MOORE'S LAW:

Computers will get exponentially faster.



Використання	Швидкість	Максимальна швидкість:
23%	2,46 ГГц	2,59 ГГц
Процесор	Потоки	Сокет:
165	2292	1
		2
Час роботи	Дескриптори	Логічних процесорів:
0:01:01:26	73885	4
		Віртуалізація:
		Підтримка технології Hyper-V:
		Так
		Кеш 1 рівня:
		128 КБ
		Кеш 2 рівня:
		512 КБ
		Кеш 3 рівня:
		4,0 МБ

LESSONS LEARNED

guideline #2 for developing concurrent code

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



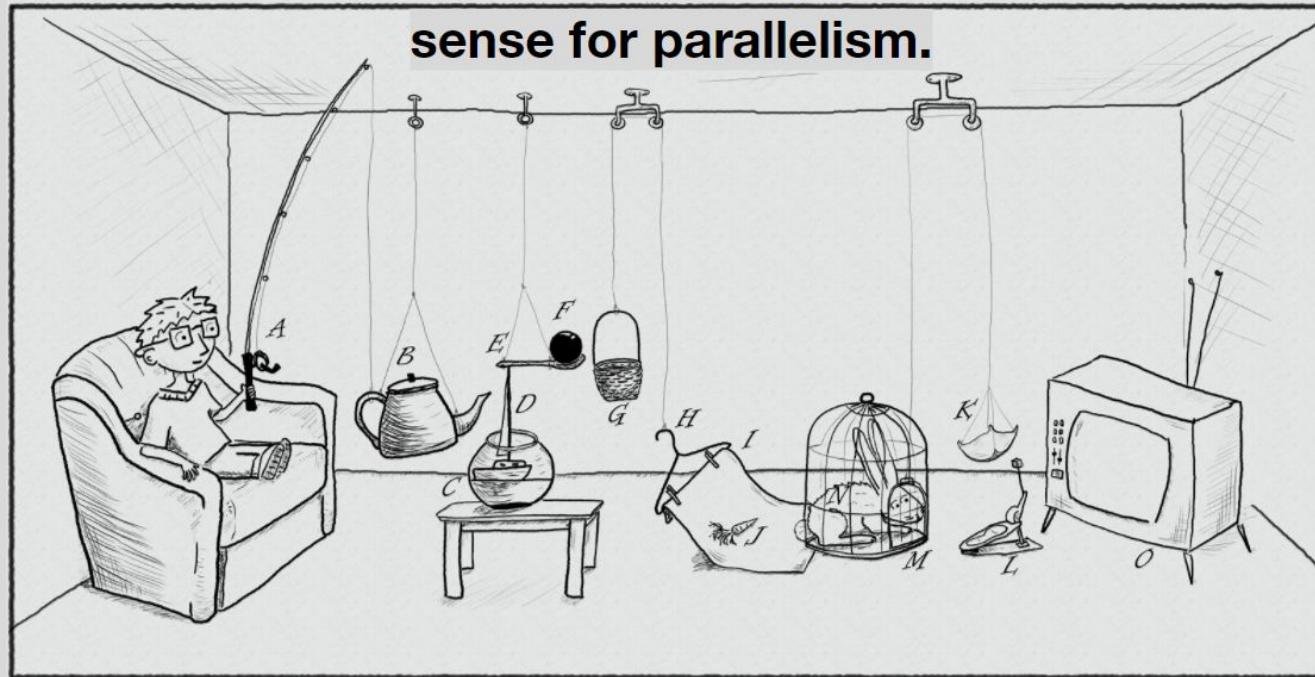
```
public class Holder {  
    private int n;  
  
    public Holder(int n) { this.n = n; }  
  
    public void assertSanity() {  
        if (n != n) {  
            throw new AssertionError("Even it can go wrong!");  
        }  
    }  
  
    public class Initializer {  
        public Holder holder;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



Holder was not
properly
published!

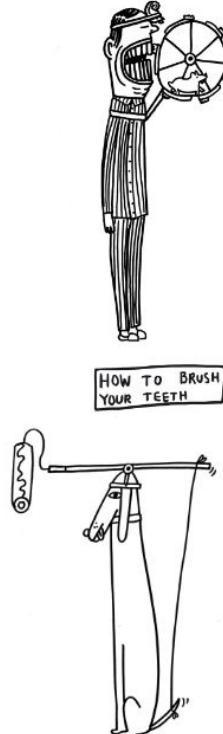
LESSONS LEARNED

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



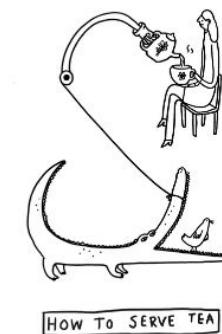
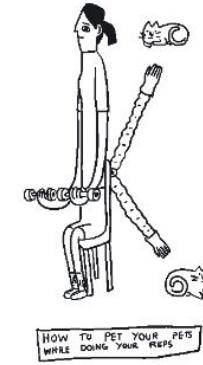
LESSONS LEARNED

GPars



data parallelism
map/reduce
fork/join
asynchronous execution
actors
agents

dataflows
remote execution
Communicating Sequential Processes
Software Transactional Memory



LESSONS LEARNED

data parallelism

map/reduce

fork/join

**asynchronous
execution**

actors

agents

GPars

dataflows

remote execution

**Communicating
Sequential
Processes**

Software

**Transactional
Memory**

Java 8 Streams API is better

HOW TO UTILIZE YOUR
DOG'S MINDLESS JOY

HOW TO SERVE TEA

LESSONS LEARNED

fork/join

**asynchronous
execution**

actors

agents

data parallelism

map/reduce

GPars

for all long-running tasks

Communicating
Sequential
Processes

Software
Transactional
Memory

HOW TO UTILIZE YOUR
DOG'S MINDLESS JOY

HOW TO PET YOUR PETS
WHILE DOING YOUR PETZ

HOW TO SERVE TEA

LESSONS LEARNED

**protect inevitably
mutable objects**

agents

asynchronous
execution

actors

GPars

dataflows

remote execution

Communicating
Sequential
Processes

Software
Transactional
Memory

HOW TO UTILIZE YOUR
DOG'S MINDLESS JOY

HOW TO SERVE TEA

LESSONS LEARNED

define tasks and explicitly coordinate them

actors

agents

Communicating Sequential Processes

Software Transactional Memory

data parallelism

map/reduce

fork/join

asynchronous execution

GPar



HOW TO UTILIZE YOUR DOG'S MINDLESS JOY

HOW TO SERVE TEA

LESSONS LEARNED

define operations on data

dataflows

remote execution

Communicating
Sequential
Processes

Software
Transactional
Memory

data parallelism

map/reduce

fork/join

asynchronous
execution

actors

agents

GPars

HOW TO UTILIZE YOUR
DOG'S MINDLESS JOY

HOW TO SERVE TEA

HOW TO FEED YOUR PETS
WHILE DOING YOUR PET'S

LESSONS LEARNED

learn more if you are interested

remote execution

Communicating
Sequential
Processes

Software
Transactional
Memory

LEARN MORE IF YOU ARE INTERESTED

[Official site](#)

[The GPars User Guide](#)

[Groovy and Concurrency with GPars by Paul King](#)

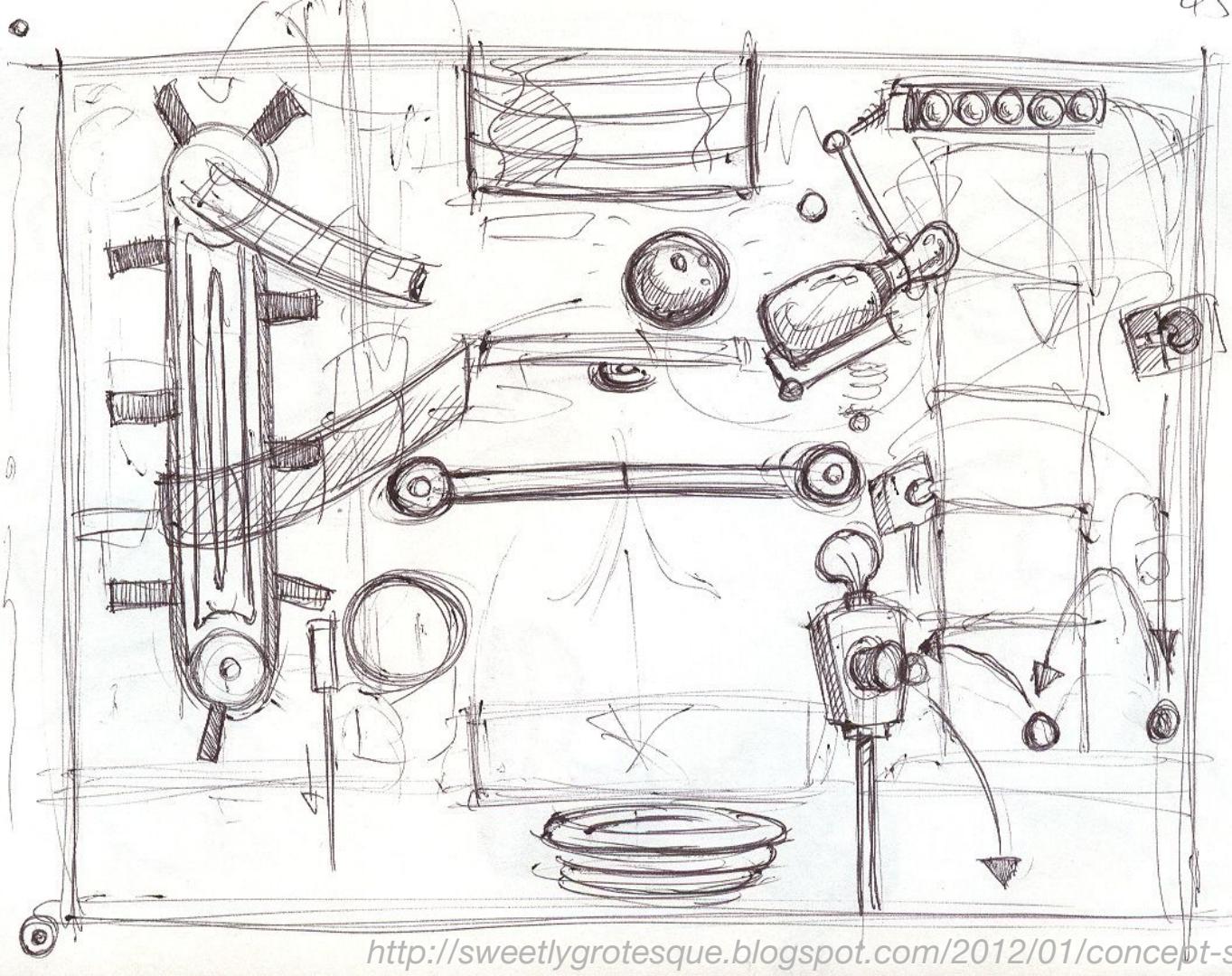
[Gpars: Concurrency in Java & Groovy by Ken Kousen](#)

['Concurrent Groovy with GPars' chapter from 'Groovy in Action' book](#)

[Sources](#)

[Samples](#)

[Slack channel](#)



Q/A

**THANK
YOU!**