Extreme Java - Concurrency Performance For Java 8 - V2.1

Dr Heinz M. Kabutz

Last updated 2016-10-20

© 2012-2016 Heinz Kabutz – All Rights Reserved

JavaSpecialists.EU

Copyright Notice

- © 2012-2016 Heinz Kabutz, All Rights Reserved
- No part of this course material may be reproduced without the express written permission of the author, including but not limited to: blogs, books, courses, public presentations.
- A license is hereby granted to use the ideas and source code in this course material for your personal and professional software development.
- Please contact heinz@javaspecialists.eu if you are in any way uncertain as to your rights and obligations.

Javaspecialists.eu

JavaSpecialists.EU

Day 1

1: Introduction

Short Introduction To Course Author

- **Dr Heinz Kabutz**
  - **Born in Cape Town, South Africa, now lives in Greece / Europe**
  - **Created The Java Specialists' Newsletter**
    - **http://www.javaspecialists.eu/archive/archive.html**
  - **One of the first Java Champions**
    - **http://javachampions.org**

Java™ Champions

Javaspecialists.eu

JavaSpecialists.EU

Introducing Your Instructor

- **This slide intentionally left blank**

Javaspecialists.eu

JavaSpecialists.EU

Comfort And Learning

- **We learn best when our basic needs are fulfilled**
- **Please let me know immediately if**
  - **The room is too cold or hot**
  - **You need oxygen**
  - **Your brain needs a break**
  - **Your nicotine or caffeine addiction needs attention**
  - **You don't understand something**
    - **Please interrupt me immediately!**

Javaspecialists.eu
JavaSpecialists.EU

Questions

- **Please please please please ask questions!**
- **Interrupt me at any time**
  - **Questions that are off-topic might be delayed until later**
  - **If so, please write down your question and we can look at it during exercise time**
- **There are some stupid questions**
  - **They are the ones you didn't ask**
  - **Once you've asked them, they are not stupid anymore**
- **The more you ask, the more we all learn**

Javaspecialists.eu

JavaSpecialists.EU

Exercises

- You don't learn cycling by just listening to lectures
- After each lecture, we do exercises
  - During exercises, real learning takes place
  - 30% of time spent applying our knowledge
- We give you hints and solutions
  - For example, for this introduction, you will find directories:
    - xj_conc.ch01_introduction.exercise1
    - xj_conc.ch01_introduction.solution1
  - Of course: do not look at the solutions until I tell you to

Javaspecialists.eu

JavaSpecialists.EU

Distractions

- **Cell Phones**
  - Please switch off unless wife is expecting child # 2+
  - If you do need to answer, please leave room
- **E-Mail & Internet**
  - Try to avoid reading and answering emails
- **Timekeeping**
  - Please try to come to the course on time in the mornings and after the breaks

Javaspecialists.eu

JavaSpecialists.EU

JavaSpecialists.Eu Certificate Of Training

- **Confirms *successful* completion of this course**
  - **e.g. http://www.javaspecialists.eu/courses/certificates/?uuid=eae4dfaa-c36b-4933-83c1-2f59b2822e3f**
- **To receive this you must:**
  - **Complete all the required exercises**
  - **Attend the entire course**
    - **You can miss half a day for medical reasons or family emergencies**
    - **Project deadlines and meetings are not accepted**

Javaspecialists.eu

JavaSpecialists.EU

Certificate of Training

This is to certify that

Sample Student

is hereby recognized for having successfully completed

The Java Concurrency in Practice Course

http://www.javaspecialists.eu/courses/concurrency.jsp

Instructor: Dr Heinz Kabutz
Date: 2011/07/04 to 2011/07/07
Location: Chania, Greece
URL: http://www.javaspecialists.eu/courses/certificates/?uuid=ea e4 dfaa -c36b-4933-83c1-2f59b2822e3f

Javaspecialists.eu
java training

Quick Introduction Of Everyone In Class

- **What is your name?**
  - **James Gosling**
- **What are you working on?**
  - **Saving planet earth with Java**
- **How long have you used Java?**
  - **I invented it**
- **Expectations from the course?**
  - **Figure out if IntelliJ is better than Emacs**
- **Something Personal?**
  - **Nerds wanted to date daughter just to meet me!**

Javaspecialists.eu

JavaSpecialists.EU

1.1: History Of Concurrency

Introduction

JavaSpecialists.EU

Let's Go Fast Fast Fast

- **In 2000, Intel predicted 10GHz chips on desktop by 2011**
  - **http://www.zdnet.com/news/taking-chips-to-10ghz-and-beyond/96055**
- **Core i7 990x hit the market early 2011**
  - **3.46GHz clock stretching up to 3.73 GHz in turbo mode**
  - **6 processing cores**
  - **Running in parallel, we get 22GHz of processing power!**

Javaspecialists.eu

JavaSpecialists.EU

Moore's Law

- Stated in 1965 that for the next 10 years, the number of transistors would double every two years
  - The prediction was only made for 10 years, but it is still true today
- Clock speed has levelled off
  - Heat buildup means we struggle to go beyond 4GHz
  - Moore's Law has often been misunderstood as clock speed doubling every 2 years
- The way to scale is to have lots of cores working together

Javaspecialists.eu

JavaSpecialists.EU

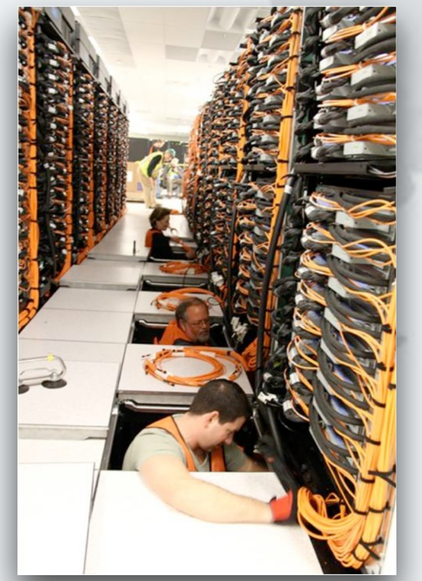
Japanese 'K' Computer

- **In June 2011, could calculate 8.2 petaFLOPS**
  - **8 200 000 000 000 000 floating point operations per second**
  - **Intel 8087 was 30 000 FLOPS, 273 billion times slower**
  - **548,352 cores from 68,544 2GHz 8-Core SPARC64 VIIIfx processors**
- **By November 2011, it had surpassed 10 petaFLOPS**

Javaspecialists.eu
JavaSpecialists.EU

"Sequoia" At Lawrence Livermore National Lab

- **Used by USA's National Nuclear Security Administration to simulate nuclear bombs**
- **June 2012: Delivers 16 petaflops**
  - **1.6 million cores, 1.6 GHz**
  - **1.6 petabytes of memory**

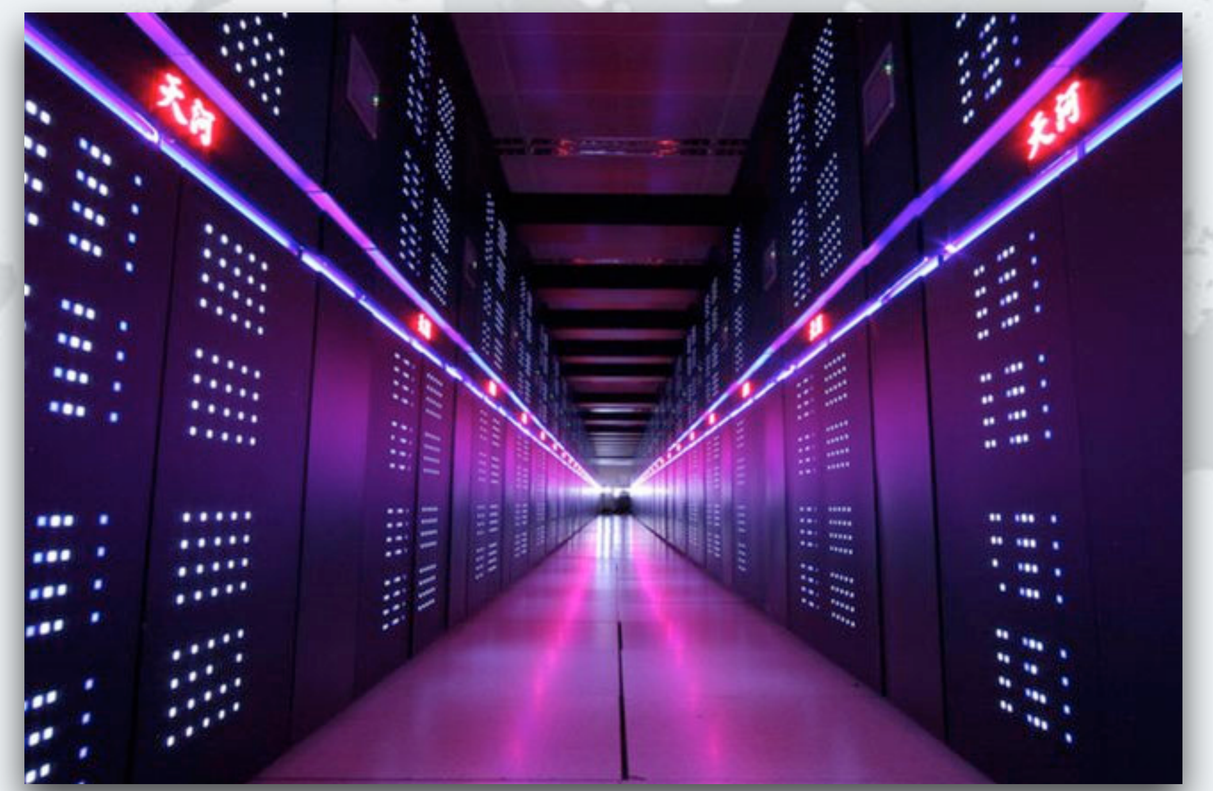


Javaspecialists.eu

JavaSpecialists.EU

"Tianhe-2" In Guangzhou, China

- **Used for simulation, analysis, and "government security"**
- **June 2014: Delivers 34 petaflops**
  - **3 million cores**
  - **1.34 petabytes of memory**
  - **1.8 GHz clock**



Javaspecialists.eu
JavaSpecialists.EU

1.2: Benefits Of Threads

Introduction

JavaSpecialists.EU

1.2: Benefits Of Threads

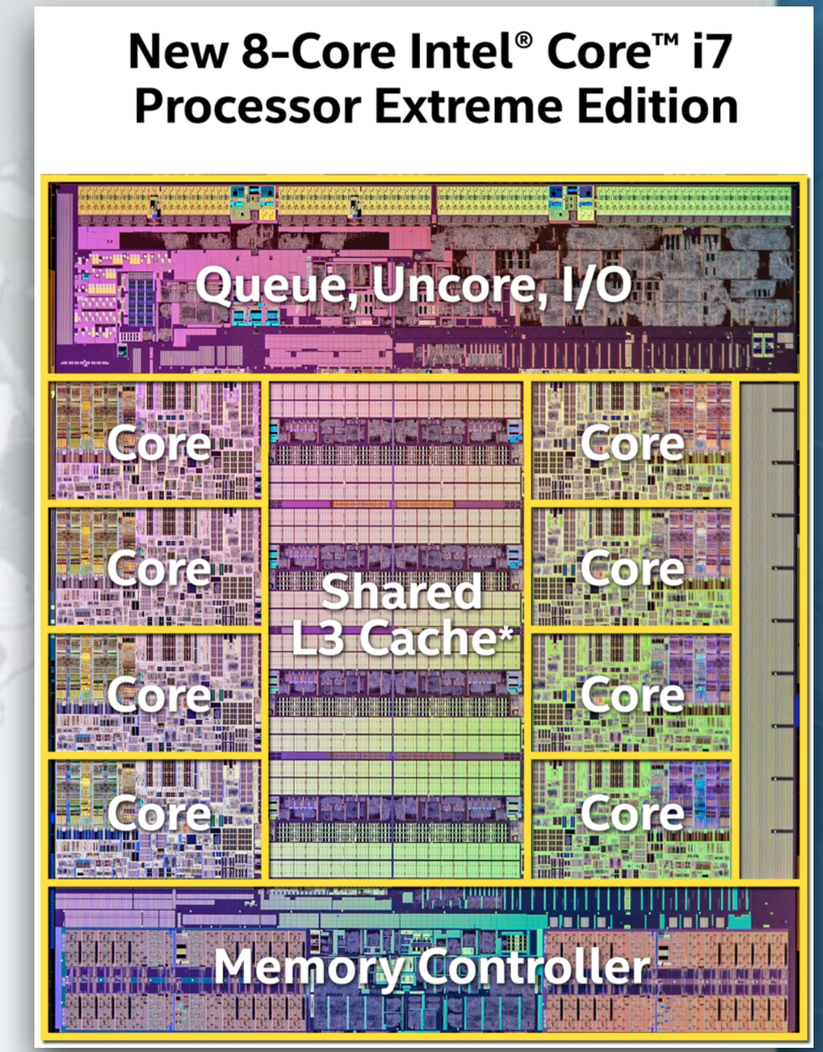
- **Threads make our lives as programmers easier:**
  - **Concurrent Programming**
    - **Independent sequential workflows are easier to write**
      - **Blocking vs non-blocking IO**
    - **During wait times our CPU can be kept busy**
    - **Easy to dispatch incoming requests to their own thread**
      - **Thread pools are a useful pattern here**
  - **Parallel Programming**
    - **Multiple cores can solve many difficult tasks faster**
      - **Support with Java 7 Fork/Join**
      - **Java 8 parallel streams utilize cores to solve tasks quicker**

Javaspecialists.eu

JavaSpecialists.EU

C.P.U. / Core / Hardware Thread

- **The Intel i7-5960X**
- **One CPU socket, 8 cores**
- **2.6 billion transistors**
- **Each core supports two hyperthreads**
  - **Each core can only execute a single instruction at a time, but the data is fetched in parallel**
- **Total of 16 threads**
- **Runtime.getRuntime().availableProcessors() = 16**



Javaspecialists.eu

JavaSpecialists.EU

Better Performance

- **Improved throughput**
  - A program with only one thread can run on only one processor.
    - Single threaded in Adobe CS4, but uses multiple threads in CS5
  - On a 2-processor system, a single-threaded program is giving up access to 1/2 the available CPU's
  - On a 100-processor system, it is giving up access to 99%!
- **Modern GPUs can have hundreds of cores**
  - NVIDIA Tesla C2075 has 448 CUDA cores
  - Can be used for basic number crunching
  - Access from Java with libraries such as jCUDA, Aparapi
    - Project Sumatra tried to offload JVM tasks to GPUs

Javaspecialists.eu

JavaSpecialists.EU

Simpler Modeling

- It is easier to write code that does only one thing at a time
  - Simpler to code, easier to test, less bugs
- We can split up our tasks into separate threads
- Leads to cleaner abstractions

Javaspecialists.eu

JavaSpecialists.EU

1.3: Risks Of Threads

Introduction

JavaSpecialists.EU

1.3: Risks Of Threads

- **Threads have become too easy to create and use**

```
public class MyThread extends Thread {
  public void run() {
    // your concurrent task here
  }
}
MyThread thread = new MyThread();
thread.start();
```

- **Fortunately, frameworks try to stop us from creating them**
  - **But every single piece of Java code still is executed by a thread**
  - **When threads are created by frameworks, we still do not know which thread our code will execute in!**
  - **We might see issues with contention, race conditions, deadlocks**

Javaspecialists.eu

JavaSpecialists.EU

Safety Vs Liveness

- **A program is thread safe if in a multi-threaded environment each thread behaves *consistently* with its behavior in a single-threaded environment (Naftalin)**
- **Safety**
  - **Nothing bad ever happens**
  - **For example, shared data should not get corrupted**
- **Liveness**
  - **Something good eventually happens**
  - **For example, we should not have deadlocks or livelocks**

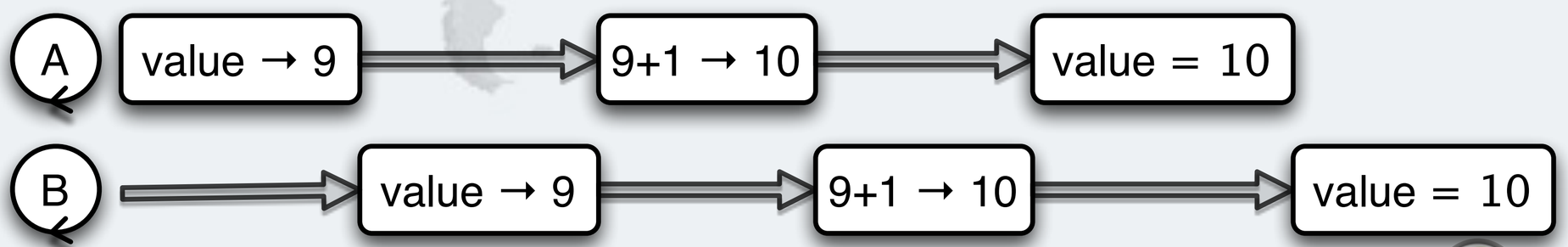
Javaspecialists.eu

JavaSpecialists.EU

Safety Hazards

- **Thread safety can be unexpectedly subtle.**
  - **Ordering of operations in threads can be unpredictable.**
- **This is supposed to generate a unique int**

```
@NotThreadSafe
public class UnsafeSequence {
  private int value;
  public int getNext() {
    return value++; // Should return unique int
  }
}
```

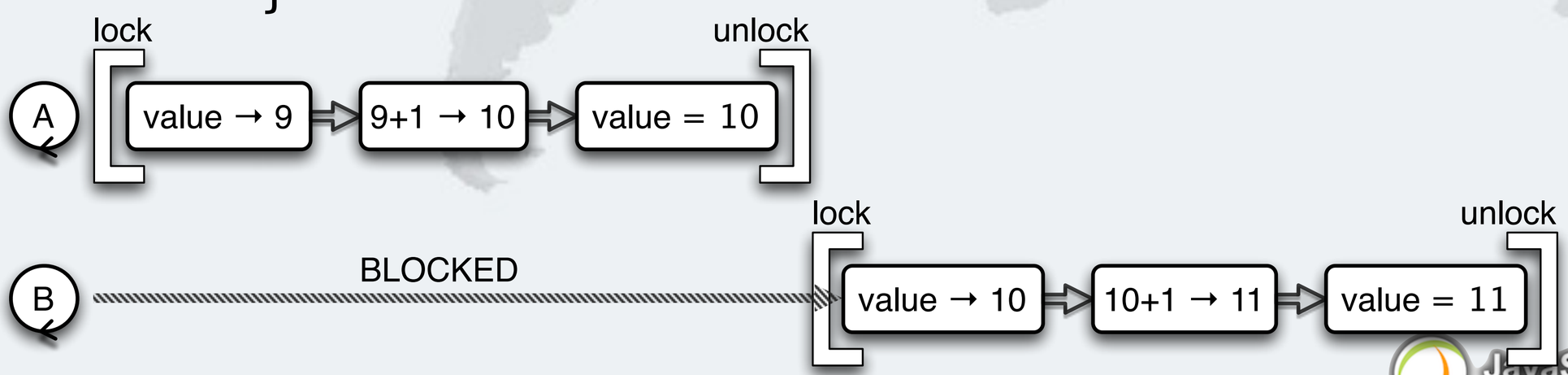


Javaspecialists.eu
JavaSpecialists.EU

Threadsafe Sequence

- **By making getNext() synchronized, we get rid of the race condition**

```
@ThreadSafe
public class Sequence {
  @GuardedBy("this")
  private int value;
  public synchronized int getNext() {
    return value++;
  }
}
```



Javaspecialists.eu
JavaSpecialists.EU

Thread Safety Annotations

- In JCiP, Brian Goetz defines useful class annotations
  - @Immutable - Object state cannot be changed once constructed
  - @ThreadSafe - Object state is protected by synchronization
    - @GuardedBy - denotes which lock guards the field
  - @NotThreadSafe - User has to ensure thread safety
- For example, if a class is annotated with @ThreadSafe
  - Users can use it with confidence
  - Maintainers are informed that thread safety must be preserved
  - Software analysis tools could identify possible coding errors

Javaspecialists.eu

JavaSpecialists.EU

1.4: Threads Are Everywhere

Introduction

JavaSpecialists.EU

1.4: Threads Are Everywhere

- **Even if you don't create a thread, the framework might!**
- **Code called from these threads must be thread safe.**
- **Java threads are created by:**
  - **The JVM**
    - **For housekeeping - garbage collection, finalization**
    - **And for running the main method**
  - **AWT & Swing**
    - **Event dispatch thread**
  - **Timer**
  - **Servlet Container, RMI, etc, etc, etc.**

Javaspecialists.eu

JavaSpecialists.EU

Concurrency Is Not An “Advanced” Feature!

- **Nearly all Java apps are multithreaded**
  - **Threading is not optional knowledge**
- **We need to all understand thread safety**

Thread safety is not about code, it is about shared, mutable, data!

Javaspecialists.eu

JavaSpecialists.EU

1.5: Short Java 7 & 8 Primer

1.5: Short Java 7 & 8 Primer

- **Java 7 has some nice syntactic sugar**
  - **Underscore separators to make numbers more readable**

```
// pre-Java 7
int upto = 1000 * 1000 * 1000;
long large_prime = 9223372036854775783L;
// Java 7
int upto = 1_000_000_000;
long large_prime = 9_223_372_036_854_775_783L;
```

  - **Type inference: Generics have become slightly more intelligent**
    - **But it cannot always deduce what the type parameter should be**

```
// pre-Java 7
Map<String, List<String>> data =
  new HashMap<String, List<String>>();
// Java 7
Map<String, List<String>> data = new HashMap<>();
```

Javaspecialists.eu

JavaSpecialists.EU

Java 8 Lambdas

- **Java 8 introduces lambdas**
  - **Functional interfaces have exactly one abstract method**

```
ExecutorService pool = Executors.newCachedThreadPool();
// Java 7 anonymous inner class for Runnable
pool.submit(new Runnable() {
  public void run() {
    System.out.println("I am a Runnable");
  }
});
// Java 8 lambda for Runnable
pool.submit(() -> System.out.println("I am a Runnable"));
```

Javaspecialists.eu

JavaSpecialists.EU

Passenger And Train Example

```
public class Passenger {
  public void onboard(Train train) {
    System.out.println("Onboard " + train);
  }
}

public class Train {
  public static Train create(Supplier<Train> supplier) {
    return supplier.get();
  }
  public static void paintBlue(Train train) {
    System.out.println("Painted blue " + train);
  }
  public void repair() {
    System.out.println("Repaired " + this);
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Using Lambdas with Trains

- **Lambda implementing Supplier<Train> constructs object**

```
Train train = Train.create(() -> new Train());
```

- **Converting Stream to List**

```
List<Train> trains = Stream.of(train)
                           .collect(Collectors.toList());
```

- **Static method with Train as parameter**

```
trains.forEach(t -> Train.paintBlue(t));
```

- **Instance method with no parameters**

```
trains.forEach(t -> t.repair());
```

- **Instance method in other class with Train as parameter**

```
trains.forEach(t -> p.onboard(t));
```

Javaspecialists.eu

JavaSpecialists.EU

Method References

- **When the parameter to method is the same as the lambda**

```
List<String> names = new ArrayList<>();
Collections.addAll(names, "Maxi", "Connie", "Bangie", "Efi");
names.forEach(s -> System.out.println( s ));
names.forEach(System.out::println); // method reference
```

- **When we are calling a method without parameters**

```
IntStream lengths = Stream.of("hello", "there", "world!")
    .mapToInt(s -> s.length());

IntStream lengths = Stream.of("hello", "there", "world!")
    .mapToInt(String::length);
```

Javaspecialists.eu

JavaSpecialists.EU

Method References for Train Example

- **Lambda implementing Supplier<Train> constructs object**

```
Train train = Train.create(Train::new);
```

- **Static method with Train as parameter**

```
trains.forEach(Train::paintBlue);
```

- **Instance method with no parameters**

```
trains.forEach(Train::repair);
```

- **Instance method in other class with Train as parameter**

```
trains.forEach(p::onboard);
```

- **Ambiguous method references won't compile**

Javaspecialists.eu

JavaSpecialists.EU

1.5: Short Java 7 & 8 Primer

- **Streams can filter, map and process objects**

```
Stream.of("Maxi", "Connie", "Bangie", "Efi")
    .filter(n -> n.matches(".*[eE].*")) // regular expression
    .sorted() // sort the entries
    .mapToInt(String::length) // return stream of lengths
    .forEach(System.out::println); // print them out
```

```
6
6
3
```

- **And we can even use primitives for streams**
  - **e.g. generate and print an endless stream of random values**

```
DoubleStream.generate(() -> ThreadLocalRandom.current().nextDouble())
   .forEach(System.out::println);
```

```
0.48762083504412546
0.8372583168483918
0.37309494547635647
0.2586716816782939
...
```

Javaspecialists.eu

JavaSpecialists.EU

1: Exercises

Annotations for Concurrency

JavaSpecialists.EU

Exercise 1.1: Class Annotations 1/3

- **Fire up your IDE and add class annotations**
  - **Run PersonTest to check that you've done it correctly**
  - **Annotations can be @Immutable, @ThreadSafe or @NotThreadSafe**

```
public class Person1 {
  private final String firstName;
  private final String surname;
  private final int age;

  public Person1(String firstName,
                 String surname, int age) {
    this.firstName = firstName;
    this.surname = surname;
    this.age = age;
  }
}
```

- **http://www.javaspecialists.eu/outgoing/xj/conc-j8.zip**

Javaspecialists.eu

JavaSpecialists.EU

Exercise 1.1: Class Annotations 2/3

```
public class Person2 {
  private final String firstName;
  private final String surname;
  private int age;

  public Person2(String firstName,
                 String surname, int age) {
    this.firstName = firstName;
    this.surname = surname;
    this.age = age;
  }

  public void birthday() {
    age = age + 1;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Exercise 1.1: Class Annotations 3/3

```
public class Person3 {
  private final String firstName;
  private final String surname;
  private int age;

  public Person3(String firstName,
                 String surname, int age) {
    this.firstName = firstName;
    this.surname = surname;
    this.age = age;
  }

  public synchronized void birthday() {
    age = age + 1;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

1: Introduction

Questions?

2: Thread Safety

Thread safety is all about shared data

JavaSpecialists.EU

2: Thread Safety

- **It is not safe to ignore the rules**
  - **Even if you get away with it for a long time**

Javaspecialists.eu

JavaSpecialists.EU

Shared Data In Multi-Threading

- **Multi-processing typically does not allow sharing of data**
  - **Inter-process communication is expensive**
  - **Limits the opportunities for parallelism**
- **Multi-threading allows sharing of fields or data**
  - **Communication between threads can be more light-weight**
  - **Can cause race conditions and data races**
  - **Thread safety is about managing access to shared, mutable state**
- **Each thread has its own stack memory**
  - **Contains call stack, local variables and parameters**
  - **We never have to worry about thread safety with these because they are not shared in Java**

Javaspecialists.eu
JavaSpecialists.EU

Synchronization

- **Shared *mutable* data needs correct *synchronization***
  - **This is not only done with** synchronized **keyword**
  - **Also includes volatile, atomics and explicit locks**
- **For the rest of the course, when we talk about *synchronization*, we mean the broader term**

Javaspecialists.eu

JavaSpecialists.EU

Your Program Probably Has Latent Defects

- **Your program *is* broken if several threads access shared mutable data without correct synchronization**
  - **Even if it appears to work, you might have just been lucky so far**
- **We can fix it in three ways:**
  - **Stop sharing state across threads**
  - **Make state immutable**
  - **Use synchronization whenever we access the shared mutable state**
    - **This might seem the easy choice, but often it is difficult**
- **Rather think of this up-front**
  - **Retrofitting can be hard work and error prone**

Javaspecialists.eu

JavaSpecialists.EU

2.1: Atomicity

Thread Safety

JavaSpecialists.EU

2.1: Atomicity

- **Non-atomic operations to fields may cause *lost updates***
- **Even something as simple as** count++ **is non-atomic**
  - **It consists of several JVM byte code instructions:**
    - **getfield // *read* current field value onto stack**
    - **lconst_1 // put constant 1 onto stack**
    - **ladd // *modify* value on stack**
    - **putfield // *write* value on stack back into field**
  - **We call this a *read-modify-write* operation**

Javaspecialists.eu

JavaSpecialists.EU

Broken Atomicity - Don't Do This!

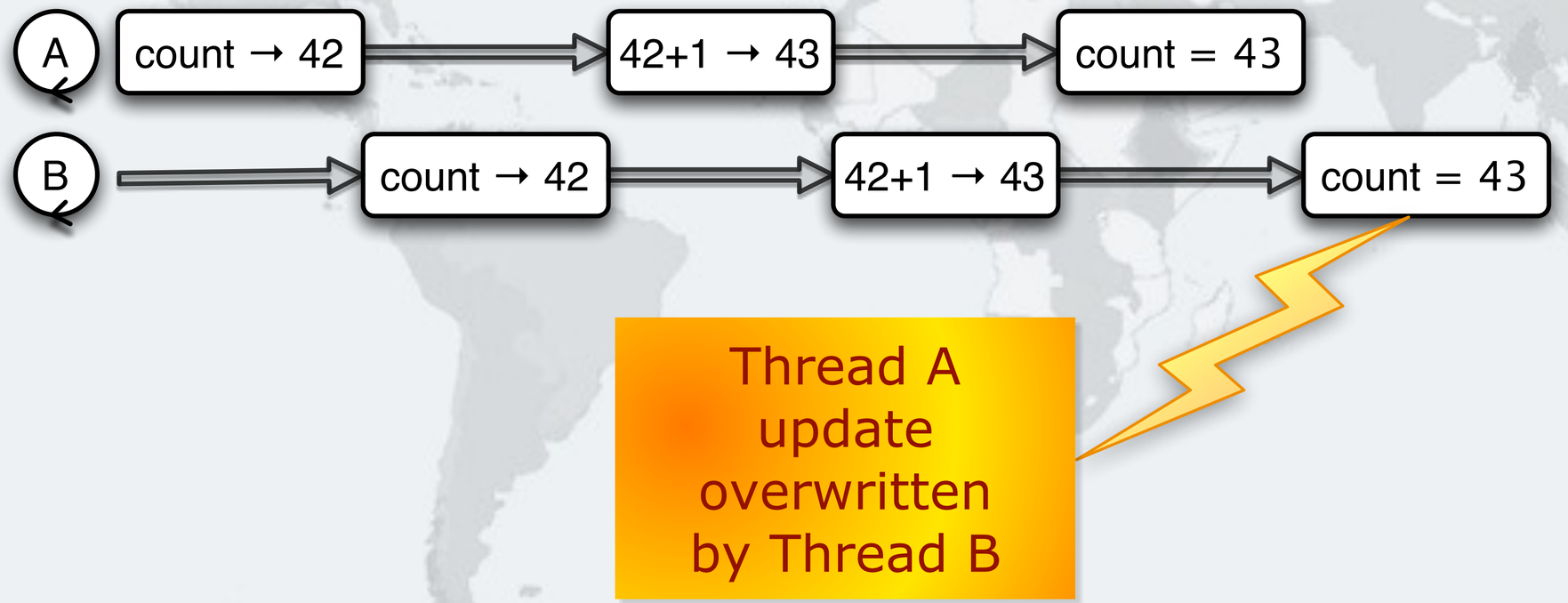
- **Servlet counts requests without correct synchronization**
  - **It can *lose updates***

```
@NotThreadSafe
public class UnsafeCountingFactorizer
    implements Servlet {
  private long count = 0;
  public long getCount() { return count; }
  public void service(ServletRequest req,
                      ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    ++count;
    encodeIntoResponse(resp, factors);
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

UnsafeCountingFactorizer Sequence Diagram



Javaspecialists.eu

JavaSpecialists.EU

Making Compound Actions Atomic

- **Holding a lock for the *entire duration* of the compound action will make it atomic**
  - **e.g. Compound actions like read-modify-write or check-then-act**
- **If we use locks, we need the *same* lock everywhere that field is accessed**
  - **Not just *writes* but also *reads!***

Every shared, mutable field should be guarded by exactly one lock. Make it clear to maintainers which lock that is.

Javaspecialists.eu

JavaSpecialists.EU

2.2: Visibility

Thread Safety

JavaSpecialists.EU

What Output Could We See?

```
public class WaitingForMeaningOfLife {
  private static boolean ready;
  private static int number;

  private static class ReaderThread extends Thread {
    public void run() {
      while (!ready) Thread.yield();
      System.out.println(number);
    }
  }

  public static void main(String... args) {
    new ReaderThread().start();
    number = 42;
    ready = true;
  }
}
```

42

no output

0

Javaspecialists.eu

JavaSpecialists.EU

Output for WaitingForMeaningOfLife

- **The number 42 is our expected output**
- **It might not show any output if ReaderThread caches field**
  - **Write to "ready" field is then not visible to ReaderThread**
- **We could also see the number 0**
  - **If the writes to fields ready and number are reordered**

Javaspecialists.eu

JavaSpecialists.EU

Bug In Java?

- **We would expect all threads to see the most up-to-date values in memory. Wrong!**
  - **Transferring data across processor cores is expensive**
  - **The operating system might move threads between different cores**
  - **Thus Java cannot guarantee that all threads see all changes**
- **To ensure that a memory write is visible across all threads, you must use synchronization!**
  - **Although it does not necessarily have to be *synchronized***

Javaspecialists.eu

JavaSpecialists.EU

2.3: Confinement

Thread Safety

JavaSpecialists.EU

2.3: Confinement

- **In this section we will look at three ways of introducing thread safety through confinement**
  - **Thread Confinement**
  - **Stack Confinement**
  - **Object Confinement**

Javaspecialists.eu

JavaSpecialists.EU

Thread Confinement

- Safe access of shared data needs synchronization
- To avoid synchronization, don't share!
  - A.K.A. "Thread confinement"
- No language level support to enforce confinement
  - Must be done via implementation
  - Be careful of accidentally leaking non-threadsafe instances

Javaspecialists.eu

JavaSpecialists.EU

ThreadLocal

- **ThreadLocal is a utility class that aids in enforcing thread confinement.**
- **ThreadLocal get and set methods associate values to one particular thread instance**
  - **e.g. DateFormat is not threadsafe, so we use thread confinement**

```
private static final ThreadLocal<DateFormat> df =
    ThreadLocal.withInitial(
        () -> new SimpleDateFormat("yyyy-MM-dd")
    );
public static String format(Date date) {
  return df.get().format(date);
}
```

Javaspecialists.eu

JavaSpecialists.EU

Stack Confinement

- **If the only reference to an object is local then there is no way to get that object to escape if not done in the method**
  - **e.g. we could also solve SimpleDateFormat like this:**

```
public static String format(Date date) {
  DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
  return df.format(date);
}
```

- **Since the object is only ever accessible from the stack and this only belongs to one thread, it does not need to be thread safe.**
- **Again, this is still somewhat fragile - what if future coders let the reference escape?**

Javaspecialists.eu

JavaSpecialists.EU

Object Confinement

- **We make the field holding our object private**
- **All methods accessing our object are synchronized**
  - **e.g. we could also solve SimpleDateFormat like this:**

```
@GuardedBy("this")
private final DateFormat df =
  new SimpleDateFormat("yyyy-MM-dd");
public synchronized String format(Date date) {
  return df.format(date);
}
```

- **Might introduce contention into our system if the method is called a lot**

Javaspecialists.eu

JavaSpecialists.EU

2.4: Immutability

Thread Safety

JavaSpecialists.EU

2.4: Immutability

- **Immutability is another way to ensure thread safety without synchronization**
- **Immutable object are *always* thread safe!**
- **Can only have one state**
- **Safe - alien code cannot modify state**

Javaspecialists.eu

JavaSpecialists.EU

Immutability

- An object is immutable if all of the following are true:
  - Its state cannot be modified after construction
  - All its fields are final
  - It is properly constructed
    - "this" reference does not escape during construction
- Immutable objects *can* modify state in constructor
  - May use mutable fields, but these may not be modified after construction

Javaspecialists.eu

JavaSpecialists.EU

Final Fields

- **Final fields cannot be reassigned**
  - **The objects they point to could still be mutable**
- **Java memory model guarantees initialization safety**
  - **"Happens-before" edge**
- **We always try to make classes as immutable as possible**
  - **Every little bit helps**

Javaspecialists.eu

JavaSpecialists.EU

2.5: Designing A Thread-Safe Class

Thread Safety

JavaSpecialists.EU

2.3: Designing A Thread-Safe Class

- **Encapsulation: [Booch - OOAD with Applications]**
  - **"the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation."**
- **Good encapsulation is one of our strongest weapons against concurrency bugs**
  - **Encapsulated fields and synchronization mechanisms**

Javaspecialists.eu

JavaSpecialists.EU

Steps For Designing A Thread-Safe Class

- **Designing a thread-safe class should include these steps**
  - **Identify the fields that form the object's state**
  - **Identify the invariants that constrain the state fields**
  - **Establish a policy for managing concurrent access to the state**
    - **This includes encapsulation and immutability**

Javaspecialists.eu

JavaSpecialists.EU

Simple Thread-Safe Counter With Invariant

- **We will design a thread-safe counter class**
  - **State**
    - **Primitive positive long, used as a counter**
  - **Synchronization policy**
    - **Acquire lock 'this' every time 'value' is touched**
    - **This is for both writes and reads of the fields**
  - **Enforced by**
    - **Encapsulation of all state**

Javaspecialists.eu

JavaSpecialists.EU

Simple Thread-Safe Counter With Invariant

```
@ThreadSafe
public final class Counter {
  // INVARIANT: value >= 0
  @GuardedBy("this")
  private long value = 0L;

  public synchronized long getValue() {
    return value;
  }

  public synchronized long increment() {
    if (value == Long.MAX_VALUE)
      throw new IllegalStateException("overflow");
    return ++value;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Post-Conditions

- **A post-condition must be true after every method has completed successfully**
  - **We can temporarily break these types of conditions, but this must be hidden and protected by the synchronization policy**
  - **e.g. Post-Condition for increment() method**

```
// POST-CONDITION: new.value == old.value + 1
public synchronized long increment() {
  if (value == Long.MAX_VALUE)
    throw new IllegalStateException("counter overflow");
  long old_value = value;
  long result = ++value;
  assert value == old_value + 1 : "Post-Condition fail";
  return result;
}
```

Javaspecialists.eu

JavaSpecialists.EU

Pre-Conditions Or State-Dependent Methods

- **A pre-condition must be true at start of method**
  - **For example, you cannot remove an element from an empty queue**
  - **Different policies might apply: Fail, Wait Until True, Timed wait, etc.**
- **Example of Counter.increment() with pre-condition**

```
// PRE-CONDITION: value < Long.MAX_VALUE
public synchronized long increment() {
  if (value == Long.MAX_VALUE)
    throw new IllegalStateException("counter overflow");
  // ... etc.
}
```

- **Chapter 12 is all about state-dependent methods**

Javaspecialists.eu

JavaSpecialists.EU

2: Exercises

Thread Safety

Exercise 2.1: Thread Confined Date Parsing

- **Make the DateFormat field thread confined**

```
@NotThreadSafe
public class DateFormatter {
    private final DateFormat df =
      new SimpleDateFormat("yyyy-MM-dd");

    public String format(Date date) {
        return df.format(date);
    }

    public Date parse(String date)
            throws ParseException {
        return df.parse(date);
    }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Exercise 2.2: Stack Confined Date Parsing

- **Make the DateFormat field stack confined**

```
@NotThreadSafe
public class DateFormatter {
    private final DateFormat df =
      new SimpleDateFormat("yyyy-MM-dd");

    public String format(Date date) {
        return df.format(date);
    }

    public Date parse(String date)
            throws ParseException {
        return df.parse(date);
    }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Exercise 2.3: Object Confined Date Parsing

- **Make the DateFormat field object confined**

```
@NotThreadSafe
public class DateFormatter {
    public DateFormat df =
      new SimpleDateFormat("yyyy-MM-dd");

    public String format(Date date) {
        return df.format(date);
    }

    public Date parse(String date)
            throws ParseException {
        return df.parse(date);
    }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Exercise 2.4: Immutable Date Parsing

- **Solve thread safety by using the immutable parser object DateTimeFormatter.ISO_LOCAL_DATE. Your methods would now use LocalDate.**

```
@NotThreadSafe
public class DateFormatter {
    private final DateFormat df =
      new SimpleDateFormat("yyyy-MM-dd");

    public String format(Date date) {
        return df.format(date);
    }

    public Date parse(String date)
            throws ParseException {
        return df.parse(date);
    }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Exercise 2.5: Bytegenerator (Optional)

- **This class misses values, figure out why and fix the code**

```
/** @author Neeme Praks, used with permission. */
public class ByteGenerator {
  private static final int INITIAL_VALUE = Byte.MIN_VALUE - 1;
  private final AtomicInteger counter = new AtomicInteger(INITIAL_VALUE);
  private final AtomicInteger resetCounter = new AtomicInteger(0);
  private final Object lock = new Object();
  public byte nextValue() {
    int next = counter.incrementAndGet();
    if (next > Byte.MAX_VALUE) {
      synchronized (lock) {
        int i = counter.get();
        // if value is still larger than max byte value, we reset it
        if (i > Byte.MAX_VALUE) {
          counter.set(INITIAL_VALUE);
          resetCounter.incrementAndGet();
        }
        next = counter.incrementAndGet();
      }
    }
    return (byte) next;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

2: Thread Safety

Questions?

3: Building Blocks

3: Building Blocks

- **In this section we will explore the rich set of tools provided in the standard Java Development Kit (JDK)**
  - **Synchronized Collections**
  - **Concurrent Collections**
  - **Blocking Queues**
  - **Synchronizers, such as CountDownLatch, Semaphore and Phaser**

Javaspecialists.eu

JavaSpecialists.EU

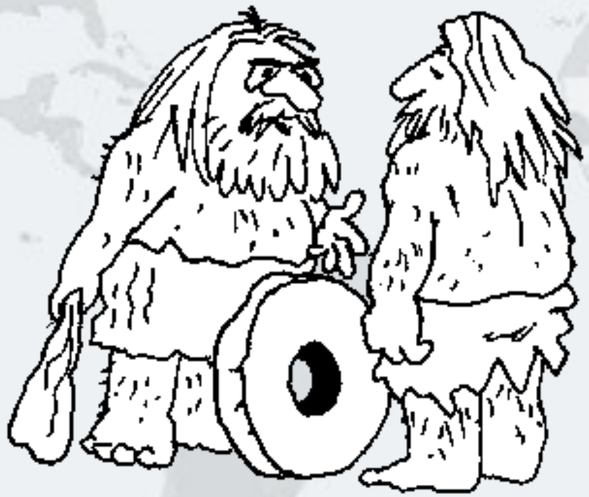
3.1: Synchronized Collections

Building Blocks

JavaSpecialists.EU

3.1: Synchronized Collections

- **Two popular choices for managing thread safety:**
  - **Write it yourself**



Do you really want to reinvent the wheel?

  - **Delegate to proven and tested thread-safe classes in the JDK**
    - **Probability that Vector and Hashtable have concurrency bugs?**
- **Typically we have to write some of it ourselves**
  - **Keeps us employed as programmers :-)**

Javaspecialists.eu

JavaSpecialists.EU

JDK Provided Building Blocks

- **JDK offers some basic concurrent building blocks**
  - **Thread-safe collections**
    - **The synchronized wrapper classes created by the Collections.synchronizedXxx factory methods**

```
List<String> safeList = Collections.synchronizedList(
    new ArrayList<>());
Map<Integer, String> safeMap = Collections.synchronizedMap(
    new HashMap<>());
```

    - **Also some older classes, such as Vector and Hashtable**

Javaspecialists.eu

JavaSpecialists.EU

Benefits Of Using Synchronized Collections

- **Collections are guaranteed to be thread-safe**
- **Easy to use**
  - **Reduces the burden on the client code to manage all the locking**
- **Well tested**
- **Allow client code to participate in synchronization policy**
  - **All the synchronized collections synchronize on "this"**

Javaspecialists.eu

JavaSpecialists.EU

Limitations With Synchronized Collections

- **The class is thread safe even without additional locking**
  - **But it doesn't necessarily behave as expected!**
- **Requires additional locking to guard compound actions**
  - **Iteration**
  - **Navigation**
  - **Conditional operations - put-if-absent**

Javaspecialists.eu

JavaSpecialists.EU

3.2: Concurrent Collections

Building Blocks

JavaSpecialists.EU

3.2: Concurrent Collections

- **When picking a collection, choose the easiest to use**
  - **If the collection is not shared, non-thread-safe collections are great!**
    - **ArrayList, LinkedList, HashMap, TreeMap**
  - **If the collection is shared but access is not too frequent, use the standard synchronized collections**
    - **Benefit: ease of use, low memory footprint**
  - **If the collection is shared and frequently accessed by multiple threads, use a concurrent collection**
    - **ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentLinkedQueue, etc.**
    - **Benefit: can be used by lots of threads without too much blocking**
    - **Disadvantage: might use a lot more memory**

Javaspecialists.eu

JavaSpecialists.EU

Synchronized vs Concurrent Collections

- **Synchronized collections**
  - All methods synchronize on the same lock
  - They hold the lock for the entire operation, even time consuming ones - e.g. List.contains or Map.get
- **Concurrent collections can offer dramatic scalability improvements with little risk!**
  - One thing to keep an eye on is memory consumption, especially with the ConcurrentHashMap

Javaspecialists.eu

JavaSpecialists.EU

Concurrent Collections

- Designed to be accessed by many threads at same time
- Use lock free mechanisms such as CompareAndSwap
- ConcurrentMap Interface
  - Adds compound "Check and swap methods" e.g. putIfAbsent
  - ConcurrentHashMap
- Other concurrent collections
  - ConcurrentSkipListMap, ConcurrentSkipListSet
  - ConcurrentLinkedQueue - lock free queue

Javaspecialists.eu

JavaSpecialists.EU

ConcurrentHashMap

- **ConcurrentHashMap uses "lock-striping" - dividing the table into different segments**
  - Higher throughput under concurrent access
  - Single-threaded access still fast
  - Uses more memory than plain HashMap
- **Iteration not fail-fast**
  - No ConcurrentModificationException but rather weakly-consistent
    - Will not return any entry more than once
    - Insertions may or may not be reflected
    - Updates to values that have not been reached by the iterator, that have been updated or removed, will definitely be reflected

Javaspecialists.eu

JavaSpecialists.EU

ConcurrentHashMap

- **isEmpty() and size() methods are only estimates**
  - **Can be out of date by the time result is seen**
- **Cannot lock map for exclusive access**
  - **"Dirty reads" might happen whilst iterating**
  - **But we typically have much better performance**

Javaspecialists.eu

JavaSpecialists.EU

Additional Atomic Map Operations

- **ConcurrentMap cannot be locked for exclusive access**
- **How do you write compound actions?**
  - **Synchronizing on the map won't help!**

```
if (!map.contains("123")) {
  map.put("123", newValue);
}
```

- **ConcurrentMap has methods for compound operations so you do not need to synchronize**

Javaspecialists.eu

JavaSpecialists.EU

ConcurrentMap Atomic Operations

```
public interface ConcurrentMap<K,V> extends Map<K,V> {
  V putIfAbsent(K key, V value);

  boolean remove(K key, V value);

  // Replace value if K is mapped to oldValue
  boolean replace(K key, V oldValue, V newValue);

  // Replace value if K is present in map
  V replace(K key, V newValue);

  V computeIfAbsent(K key, Function func);

  V computeIfPresent(K key, BiFunction func);

  V compute(K key, BiFunction func);

  V merge(K key, V value, BiFunction func);
}
```

Javaspecialists.eu

JavaSpecialists.EU

Fibonacci Using HashMap.put()

- **Can't use computeIfAbsent(), as function changes map**

```
public class Fibonacci implements Function<Integer, BigInteger> {
  public BigInteger apply(Integer n) {
    Map<Integer, BigInteger> cache = new HashMap<>();
    cache.put(0, BigInteger.ZERO);
    cache.put(1, BigInteger.ONE);
    return apply(n, cache);
  }

  public BigInteger apply(int n, Map<Integer, BigInteger> cache) {
    BigInteger result = cache.get(n);
    if (result == null) {
      result = apply(n - 1, cache).add(apply(n-2, cache));
      cache.put(n, result);
    }
    return result;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Java 8 ConcurrentHashMap

- **Different approach to the "lock striping"**
  - **Concurrency level is not used anymore**
  - **Table is arranged in a type of tree structure of nodes**
  - **Each node can either be an entry or a bin containing other nodes**
  - **It has better scalability than Java 5 ConcurrentHashMap**
    - **Plus uses less memory up front**
    - **But for large tables, they are similar in performance**
  - **The number of bins is related to the size of the map**

Javaspecialists.eu

JavaSpecialists.EU

Iteration – CopyOnWrite Collections

- **Allow fast iteration without contention**
- **Modifying these collections is very expensive**
- **Only use when collections are seldom modified**
  - **We need to understand how our collections will be used before making a choice**

Javaspecialists.eu

JavaSpecialists.EU

Eclipse Collections

- **Donated by Goldman Sachs to the Java world**
  - **https://www.eclipse.org/collections/**
- **Design**
  - **Minimize object creation**
  - **Work with large data sets**
  - **Support mutable and immutable collections**
- **Support Java 7+**
  - **One Caveat: Java 8 streams do not parallelize efficiently by default**
    - **Needs to be backwards compatible, so it uses default stream() method**
- **Coding katas show how to use collections**

Javaspecialists.eu

JavaSpecialists.EU

3.3: Blocking Queues & Producer-Consumers

Building Blocks

JavaSpecialists.EU

3.3: Blocking Queues & Producer-Consumers

- **We use blocking queues a lot with concurrency**
  - **The producer puts objects onto a blocking queue**
  - **The consumer takes them off the queue**
- **Block on take() from empty queue or put() to full queue**
- **All JDK queues support multi-producer-multi-consumer**
  - **For faster alternatives, see https://github.com/JCTools/JCTools**
    - **Supports various specialized queues, such as single-producer-single-consumer (SPSC)**

Javaspecialists.eu

JavaSpecialists.EU

BlockingQueue

- **Untimed blocking methods**
  - **void put(E e)**
    - **Puts the specified element on the queue, waiting indefinitely**
    - **Element may not be null**
  - **E take()**
    - **Returns the head of the queue, waiting indefinitely for data**
  - **Both methods might throw InterruptedException**

Javaspecialists.eu

JavaSpecialists.EU

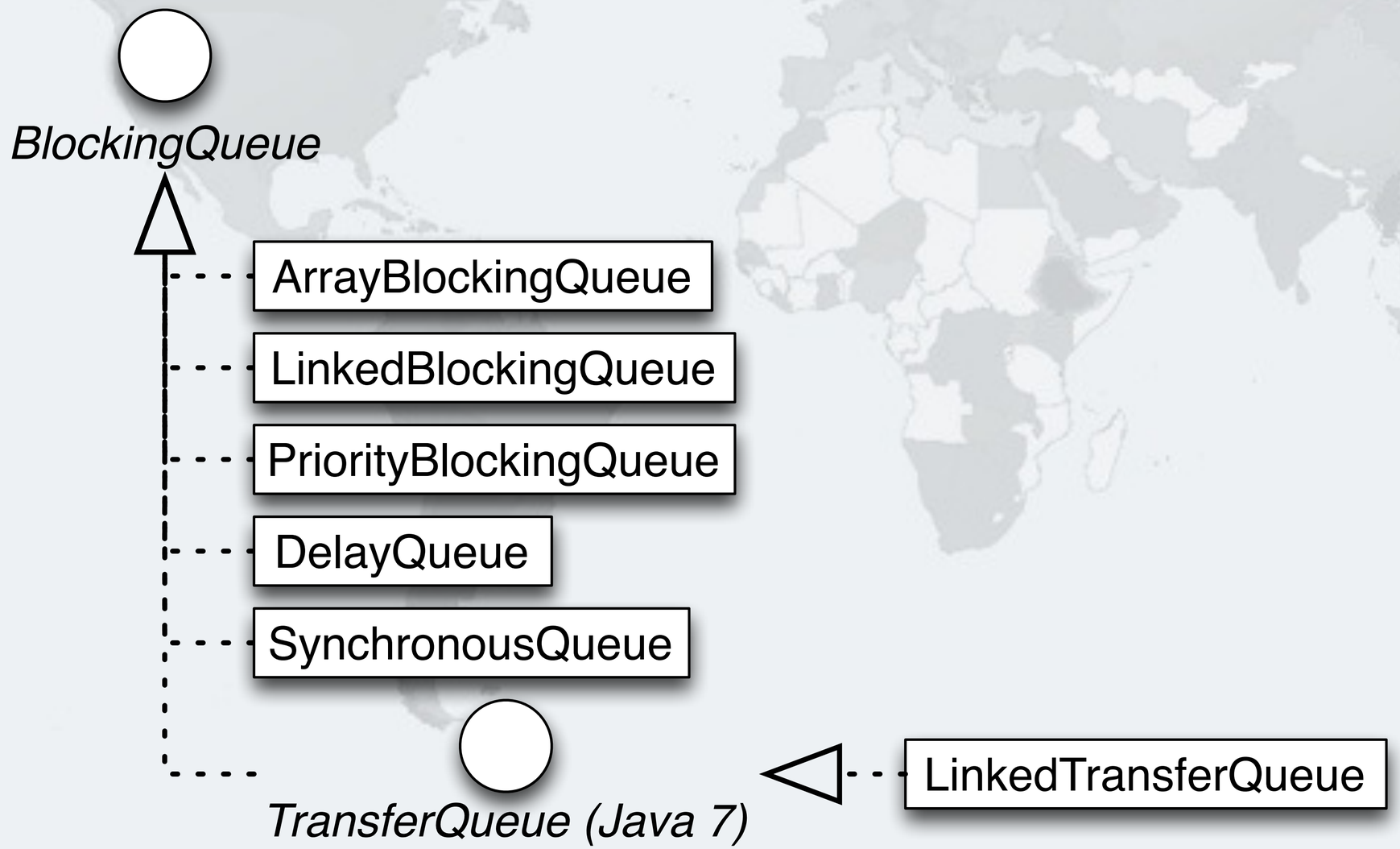
BlockingQueue

- **Timed blocking methods**
  - **boolean offer(E e, long timeout, TimeUnit unit)**
    - **Tries to put the element on the queue, waiting the specified timeout before giving up**
    - **Return value is a boolean success flag**
  - **E poll(long timeout, TimeUnit unit)**
    - **Returns the head of the queue, waiting the specified timeout for data if necessary**
    - **If no data after the timeout expires, returns null**
  - **Both methods might throw InterruptedException**

Javaspecialists.eu

JavaSpecialists.EU

Blocking Queues - Core Java Implementations



Javaspecialists.eu

JavaSpecialists.EU

TransferQueue (Java 7)

- **Single implementation LinkedTransferQueue**
- **A mixture between SynchronousQueue and LinkedBlockingQueue**
- **We can call put() and take() as with BlockingQueue**
- **transfer() waits for a consumer to take the element**
  - **If a consumer is already waiting, the element is transferred directly**

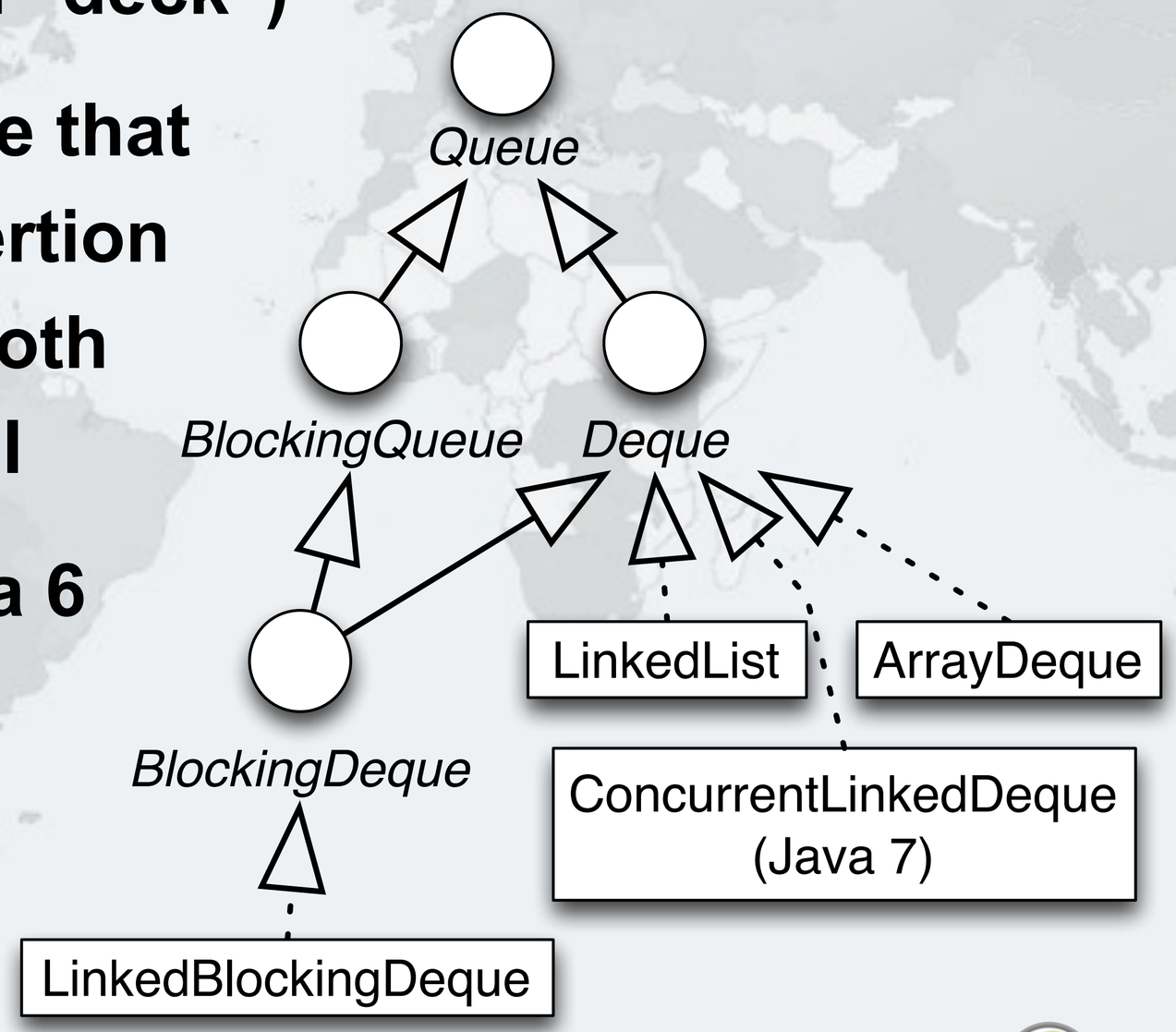
```
public interface TransferQueue<E> extends BlockingQueue<E> {
  boolean tryTransfer(E e);
  void transfer(E e) throws InterruptedException;
  boolean tryTransfer(E e, long timeout, TimeUnit unit)
    throws InterruptedException;
  boolean hasWaitingConsumer();
  int getWaitingConsumerCount();
}
```

Javaspecialists.eu

JavaSpecialists.EU

Deques

- Deque (pronounced "deck")
- Double ended queue that allows efficient insertion and removal from both the head and the tail
- Available since Java 6



Javaspecialists.eu
JavaSpecialists.EU

Deques

- **Double-ended queues, otherwise just like queues:**

```
void addFirst(E e) / void addLast(E e)
boolean offerFirst(E e) / boolean offerLast(E e)
E removeFirst() / E removeLast()
E pollFirst() / E pollLast()
E getFirst() / E getLast()
E peekFirst() / E peekLast()
boolean removeFirstOccurrence(Object o)
boolean removeLastOccurrence(Object o)
Iterator<E> descendingIterator()
```

- **In addition to the Queue and Collection methods**

Javaspecialists.eu

JavaSpecialists.EU

Good Non-Thread Safe Collections Defaults

- **List**
  - **ArrayList (low on memory, fast iteration)**
- **Map**
  - **HashMap (low on memory, fast lookup)**
- **Set**
  - **HashSet**
- **Queue**
  - **ArrayDeque**

Javaspecialists.eu

JavaSpecialists.EU

Good Defaults For Thread Safe Collections

- **List**
  - Collections.synchronizedList(ArrayList)
  - Lots of iteration - CopyOnWriteArrayList
- **Map**
  - ConcurrentHashMap (low contention, always available map)
  - If memory is bottleneck - Collections.synchronizedMap(HashMap)
  - If contention is bottleneck - Cliff Click's non-blocking wait-free table
- **Set**
  - Collections.newSetFromMap(new ConcurrentHashMap<>())
- **Queue / BlockingQueue**
  - LinkedBlockingQueue

Javaspecialists.eu

JavaSpecialists.EU

3.4: Synchronizers

Building Blocks

JavaSpecialists.EU

3.4: Synchronizers

- A synchronizer is any object that coordinates the control flow of threads based on its state
- For example
  - FutureTask, Exchanger, BlockingQueue, Semaphore, Phaser, CountDownLatch, CyclicBarrier, CompletableFuture
  - ReentrantLock, ReentrantReadWriteLock, StampedLock (ch 11)
- Sometimes easier to write your own (ch 12)

Javaspecialists.eu

JavaSpecialists.EU

Semaphores

- **Semaphores are used to synchronize resource access**
  - For example, limiting number of open database connections
- **Each semaphore contains a number of permits**
  - We acquire() a semaphore permit when we want to use a resource
    - If none is available, it will block
  - We release() the semaphore permit when we are done with it
- **Danger!**
  - Semaphore does not know which threads acquired them
  - Thus you can release() semaphores you never acquire()d
  - Or you could forget to release() due to an exception

Javaspecialists.eu

JavaSpecialists.EU

Semaphore

- **Concurrent Animation by Victor Grazi**
- **Note: the “of 4” text is misleading**

Javaspecialists.eu

JavaSpecialists.EU

CountDownLatch

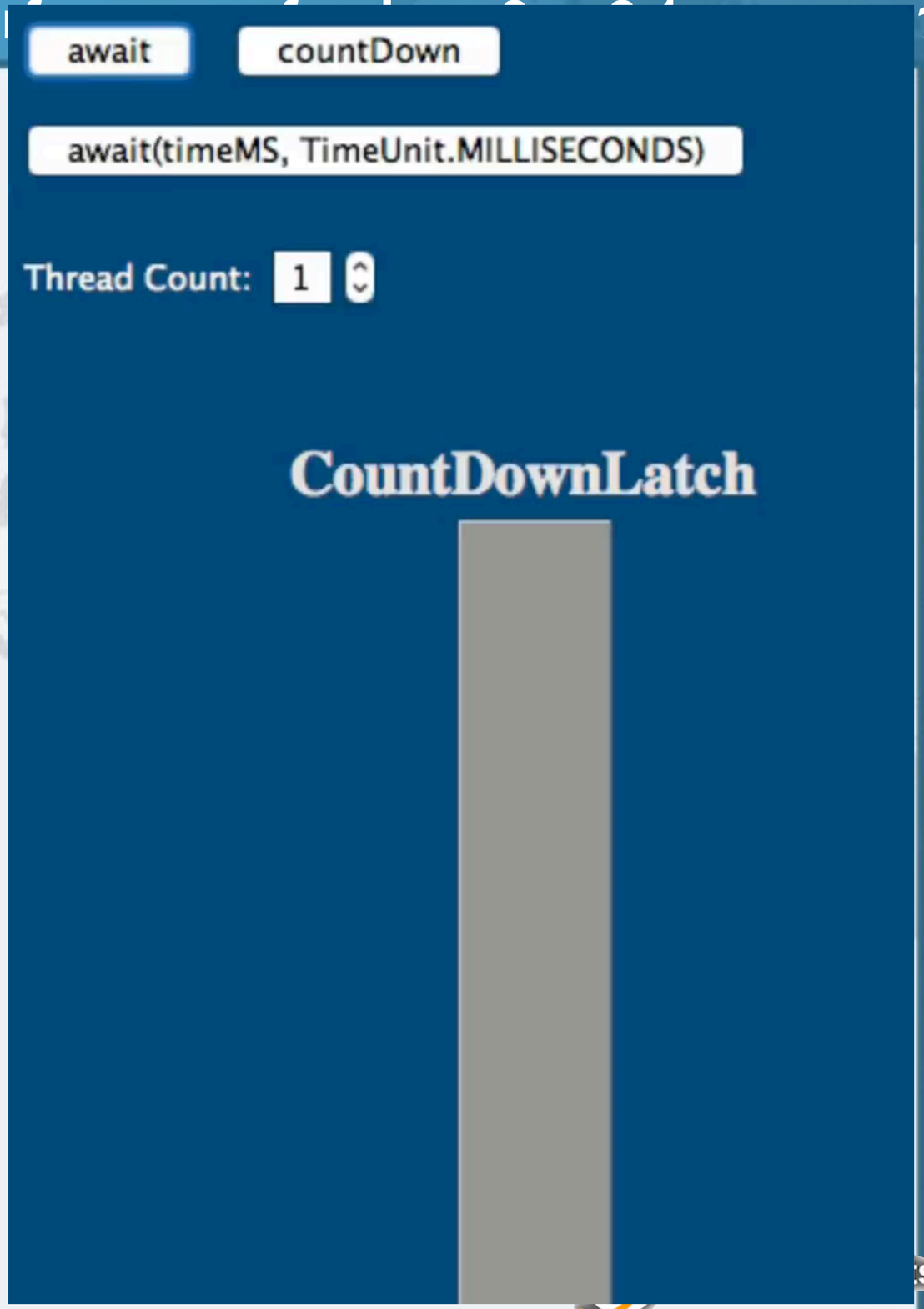
- **The CountDownLatch only allows threads to pass once countDown() has been called a number of times**
  - **e.g. Our server might need to start some thread pools, database pools, and other services before being generally available**
  - **If we know that there are 6 services to start, we can make a CountDownLatch of size 6**
  - **Any thread wanting to use our server would need to wait until the latch has been counted down to zero**

Javaspecialists.eu

JavaSpecialists.EU

CountDownLatch

- Concurrent Animation by Victor Grazi
- Threads are waiting until the count down latch is zero
  - Then they immediately continue running



Javaspecialists.eu

CountDownLatch: Waiting for Threads to Start

```
static void runTasks(List<Runnable> tasks)
    throws InterruptedException {
  int size = tasks.size() + 1;
  CountDownLatch latch = new CountDownLatch(size);
  for (Runnable task : tasks) {
    new Thread(() -> {
      try {
        latch.countDown();
        latch.await();
        System.out.println("Running " + task);
        task.run();
      } catch (InterruptedException e) { /* returning */ }
    }).start();
    Thread.sleep(1000);
  }
  latch.countDown();
}

// Note: Java 8 forEach() does not handle checked exceptions well
```

Javaspecialists.eu

JavaSpecialists.EU

CountDownLatch: Dealing with Interruptions

- **"Saving" interruptions until we can deal with them is a lot of work with CountDownLatch**

```
latch.countDown();
boolean wasInterrupted = Thread.interrupted();
while (true) {
  try {
    latch.await();
    break;
  } catch (InterruptedException e) {
    wasInterrupted = true;
  }
}
if (wasInterrupted) Thread.currentThread().interrupt();
System.out.println("Running: " + task);
task.run();
```

Javaspecialists.eu

JavaSpecialists.EU

Phasers

- **Allows threads to coordinate by phases**
  - **Similar to CountDownLatch, but more flexible**
- **Registration**
  - **Number of parties *registered* may vary over time**
    - **Same as *count* in CountDownLatch**
    - **A party can register/deregister itself at any time**
- **ManagedBlocker**
  - **Can be used in the ForkJoinPool**

Javaspecialists.eu

JavaSpecialists.EU

Phaser: Waiting For Threads To Start

- **The code for Phaser is simpler and more intuitive**

```
static void runTasks2(List<Runnable> tasks)
    throws InterruptedException {
  Phaser phaser = new Phaser(1); // we register ourselves
  for (Runnable task : tasks) {
    phaser.register(); // and we register all our new threads
    new Thread(() -> {
        phaser.arriveAndAwaitAdvance();
        System.out.println("Running: " + task);
        task.run();
    }).start();
    Thread.sleep(1000);
  }
  phaser.arriveAndDeregister(); // we let the main thread arrive
}
```

phaser.arrive() and phaser.arriveAndAwaitAdvance() also work here

Javaspecialists.eu

JavaSpecialists.EU

Synchronizers Summary

- **Semaphores**
  - Used to limit resources
- **CountDownLatch**
  - Makes threads wait until the latch has been counted down to zero
- **Phaser**
  - A flexible synchronizer in Java 7 to do latch and barrier semantics
    - With less code and better interrupt management
- **Others (not covered)**
  - FutureTask, Exchanger, CyclicBarrier

Javaspecialists.eu

JavaSpecialists.EU

3: Exercises

Building Blocks

JavaSpecialists.EU

Exercise 3.1: ConcurrentModificationException

- **We were sent the following code snippets**

```
public class AlertProvider {
  private final List<Alert> alerts =
    Collections.synchronizedList(new ArrayList<>(10));
  public Collection<Alert> getAlerts() {
    return Collections.unmodifiableCollection(alerts);
  }
  // ...
```

- **Causes a ConcurrentModificationException in Client**

```
public class Client {
  public void checkAlerts() {
    Collection<Alert> alerts = alertProvider.getAlerts(); // causes exception
    alerts.stream()
      .filter(alert -> alert.getLevel() != AlertLevel.GREEN) ...
```

- **Fix the code (either Client or AlertProvider) so that the exception does not happen anymore**

Javaspecialists.eu

JavaSpecialists.EU

Exercise 3.2: Prioritized Jobs In Thread Pool

- **Instead of using a LinkedBlockingQueue, we want to use a PriorityBlockingQueue inside our ExecutorService**
  - **Jobs need to be ordered by priority and insertion order**
  - **Most of the work is done already, you just need to finish the class PrioritizedFutureTask<T>**

Javaspecialists.eu

JavaSpecialists.EU

3: Building Blocks

Questions?

4: Task Execution

How to divide the work into units of execution

JavaSpecialists.EU

4: Task Execution

- **Units of execution in concurrent application are usually defined as discrete units of work, or *tasks***
  - This simplifies program organization
  - Makes error recovery easier
  - Provides a natural transaction boundary
  - Allows for parallelization of work
- **In this section we will look at**
  - Executing tasks in threads
  - The executor framework
  - Finding Exploitable Parallelism

Javaspecialists.eu

JavaSpecialists.EU

Executing Tasks In Threads

- **We need to identify what constitutes a task in our system**
  - **Ideally, tasks should be completely independent**
    - **Independence means the task does not need to share mutable state with other tasks or depend on the completion of other tasks**
    - **Independence allows us to farm out processing to lots of cores**
  - **Each task should represent a small part of the overall work**
- **We want to have good performance**
  - **Throughput: with more cores, we can serve more customers**
  - **Responsiveness: customers never have to wait long for a response**
  - **Graceful degradation: system should not crash under heavy load**
- **Good task boundaries can achieve this**

Javaspecialists.eu

JavaSpecialists.EU

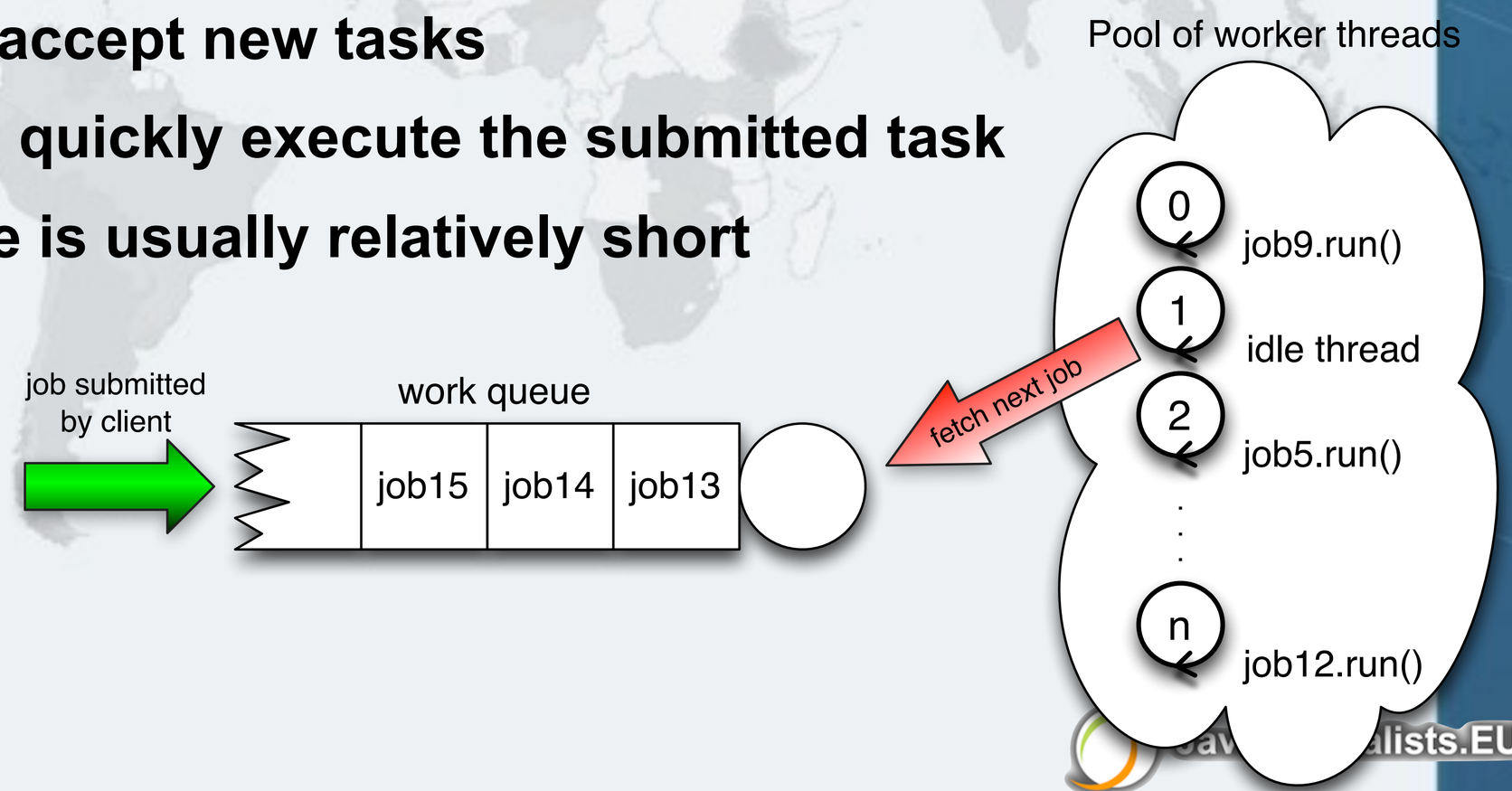
4.1: The Executor Framework

Task Execution

JavaSpecialists.EU

4.1: The Executor Framework

- **Starting a new thread can take micro or milliseconds**
  - **If a task is small, cost of thread creation swamps any benefits**
- **Thread Pools keep a *reasonable* number of threads warm**
  - **A queue is used to accept new tasks**
  - **Existing thread can quickly execute the submitted task**
  - **Hopefully the queue is usually relatively short**



Javaspecialists.eu

Types of ExecutorService

- **Create with java.util.concurrent.Executors Facade**
  - **Executors.newFixedThreadPool()**
    - **Keeps a permanent total number of threads**
    - **Threads are not started until we start submitting jobs - why?**
  - **Executors.newSingleThreadExecutor()**
    - **Like a FixedThreadPool, but with only one thread**
  - **Executors.newCachedThreadPool()**
    - **Creates new threads as needed and destroys old threads**
  - **Executors.newScheduledThreadPool()**
    - **Should be used instead of Timer**
  - **Executors.newSingleThreadScheduledExecutor()**
    - **Very similar to the java.util.Timer class, but more reliable**

Javaspecialists.eu

JavaSpecialists.EU

Executor Lifecycle

- **The Executor interface lacks shutdown methods**
  - **This could stop JVM from exiting if non-daemon threads are used**
- **The extended interface ExecutorService fixes that**

```
public interface ExecutorService extends Executor {
  void shutdown();
  List<Runnable> shutdownNow();
  boolean isShutdown();
  boolean isTerminated();
  boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException;
  <T> Future<T> submit(Callable<T> task);
  // ... additional convenience methods for task submission
}
```

Javaspecialists.eu

JavaSpecialists.EU

shutdown() vs shutdownNow()

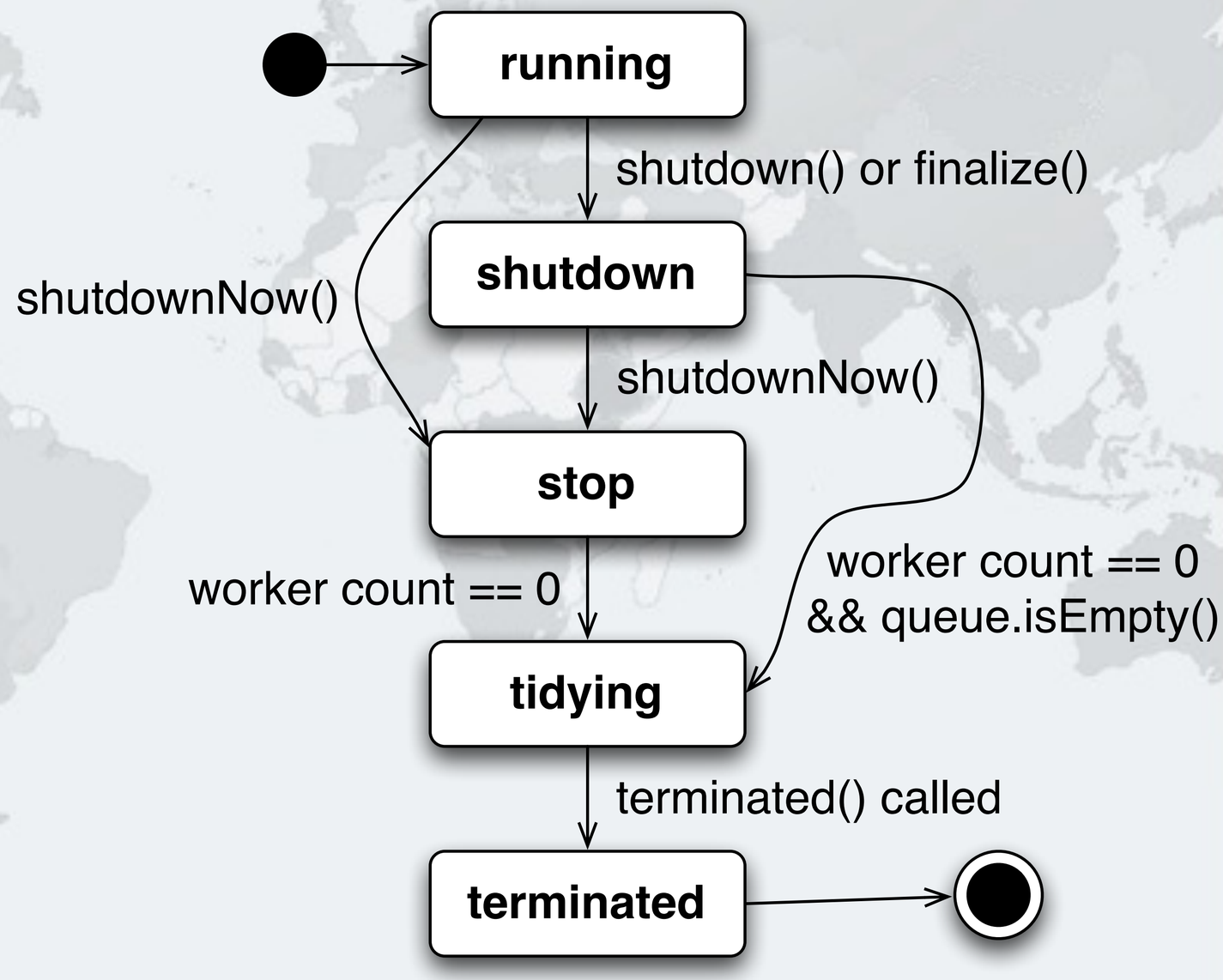
- **Threads don't stop immediately when you call shutdown()**
  - **You cannot submit new tasks**
  - **The existing queued tasks are all completed**
  - **awaitTermination() will wait until ExecutorService is terminated or return false if timeout expires - always check the return value!**
- **When you call shutdownNow(), threads are interrupted**
  - **You cannot submit new tasks**
  - **Queued tasks are returned from the method**
- **By default, ExecutorService will throw RejectedExecutionException for new tasks submitted**

Javaspecialists.eu

JavaSpecialists.EU

ThreadPoolExecutor State Machine

- **Note: This is the state machine for the thread pool, not the individual thread**



Javaspecialists.eu

JavaSpecialists.EU

Removing ThreadLocal after use

- **Memory: ThreadLocal values stay active between tasks**
  - **Executor threads typically stay alive and pick up the next task**
  - **Don't use ThreadLocal with threads managed by executors**
  - **Or if you have to, make sure you remove them at the end of the task:**

```
private static final ThreadLocal<String> userContext =
    new ThreadLocal<>();

public String call() {
  try {
    userContext.set(id);
    return processUserRequest();
  } finally {
    userContext.remove();
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

4.2: Finding Exploitable Parallelism

Task Execution

JavaSpecialists.EU

4.2: Finding Exploitable Parallelism

- **We need to find units of work that we can parallelize**
- **A single client request to a server is a natural unit of work**
  - **However, sometimes we can parallelize parts of the request**
  - **For example, some of the data could be retrieved in parallel**

Javaspecialists.eu

JavaSpecialists.EU

Page Renderer Example

- In our example we render a webpage in four distinct steps
  - We scan the source for any image info
  - We download the images
  - We render the text
  - We render the images
- Some steps might be reordered without affecting result

Javaspecialists.eu

JavaSpecialists.EU

Example: Sequential Page Renderer

- **Sequential processing is the simplest**
  - **Simply use the server thread to do all the work**

```
public class SingleThreadRenderer extends BasicRenderer {
  public void renderPage(CharSequence source) {
    List<ImageInfo> infos = scanForImageInfo(source); // step 1
    List<ImageData> images = infos.stream()
        .map(ImageInfo::downloadImage) // step 2
        .collect(Collectors.toList());
    renderText(source); // step 3
    images.forEach(this::renderImage); // step 4
  }
}
```

```
List<ImageData> images = new ArrayList<>();
for (ImageInfo info : infos) {
  images.add(info.downloadImage());
}
```

```
for (ImageData image : images) {
  this.renderImage(image);
}
```

Javaspecialists.eu

JavaSpecialists.EU

Callable And Future Structure

- **ExecutorService.submit(Callable<V>) returns Future<V>**

```
public interface Callable<V> {
  V call() throws Exception;
}
```

- **We can then retrieve result of call() or cancel task**

```
public interface Future<V> {
  boolean cancel(boolean mayInterruptIfRunning);
  boolean isCancelled();
  boolean isDone();
  V get() throws InterruptedException, ExecutionException,
    CancellationException;
  V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException,
           CancellationException, TimeoutException;
}
```

Javaspecialists.eu

JavaSpecialists.EU

Example: Page Renderer With Future

- **First step is to divide the page rendering into two tasks:**
  - **Rendering the text (CPU bound)**
  - **Downloading the images (I/O bound)**
- **In our FutureRenderer, we make a Callable to download all the images in the background**
  - **We might get better performance this way**
  - **If we are lucky, images are ready by the time we need them**
- **Future.get() may cause a bunch of exceptions**
  - **Due to the life cycle of the task**

Javaspecialists.eu

JavaSpecialists.EU

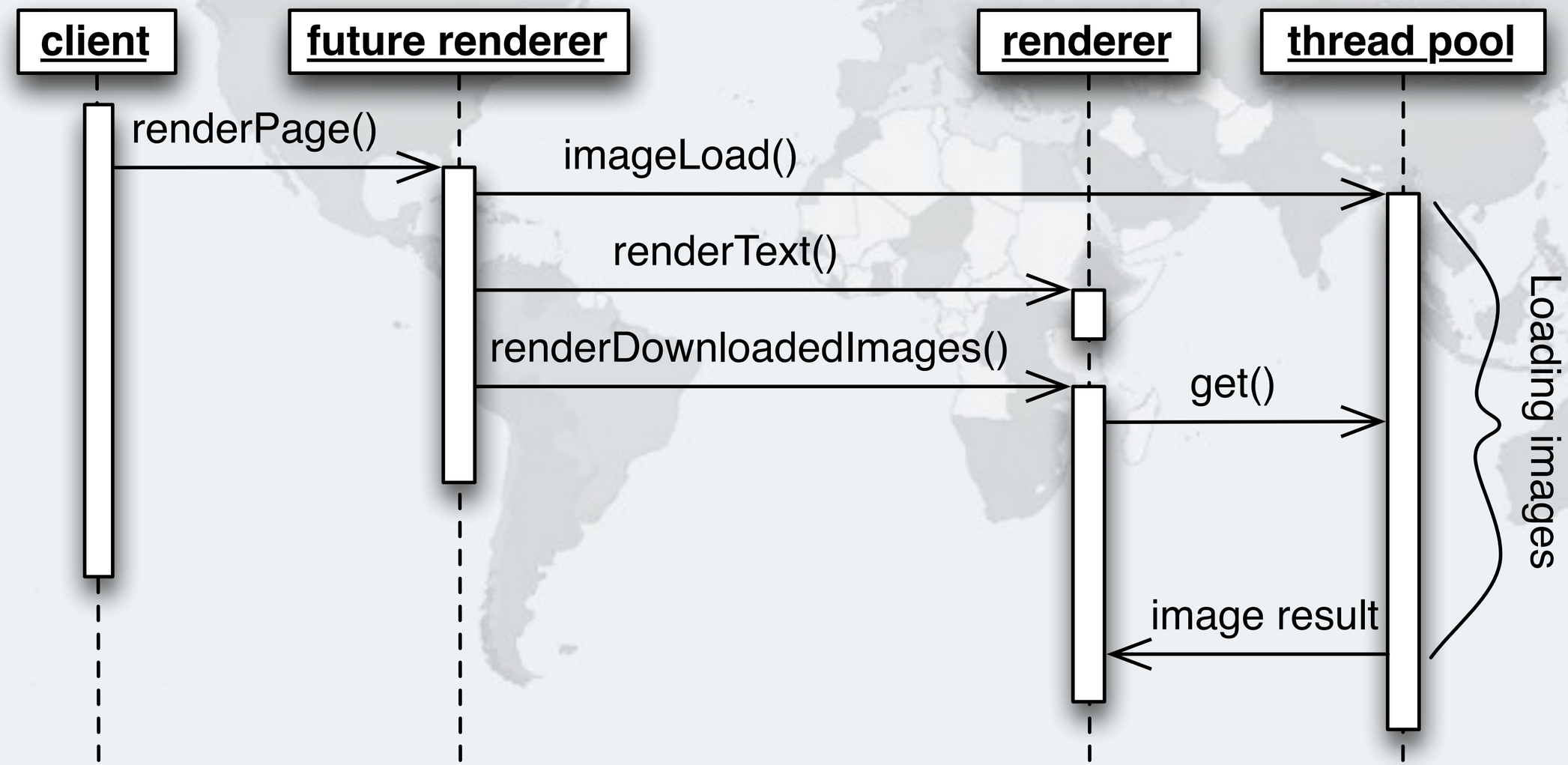
Download Images in Background Thread

```
public class FutureRenderer extends BasicRenderer {
  private final ExecutorService executor;

  public FutureRenderer(ExecutorService executor) {
    this.executor = executor;
  }

  public void renderPage(CharSequence source)
      throws ExecutionException, InterruptedException {
    List<ImageInfo> infos = scanForImageInfo(source); // step 1
    Callable<List<ImageData>> task = () -> {
      return infos.stream()
        .map(ImageInfo::downloadImage) // step 2
        .collect(Collectors.toList());
    };
    Future<List<ImageData>> images = executor.submit(task);
    renderText(source); // step 3
    images.get().forEach(this::renderImage); // step 4
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Sequence Diagram Of Future Renderer



Javaspecialists.eu

JavaSpecialists.EU

Parallelizing Heterogeneous Tasks

- **In factories, it is important to optimize machine usage**
  - **Since an extruder costs hundreds of thousands of dollars, it should never sit idle**
  - **For example, if you have two extruders each producing 1200 straws a minute and three corrugators producing between 500 and 800 straws a minute, how do you optimize your production?**
    - **It does not help to have two extruders running at full capacity**
      - **And you cannot run them too slow**
  - **Good capacity management can earn real $$$**
- **Similarly, we should optimize our available computer hardware**
  - **CPU, IO, Disk, etc.**

Javaspecialists.eu
cialists.EU

Assigning Heterogenous Tasks

- **Dividing work heterogeneously has its limitations**
  - **Our FutureRenderer would not be much more efficient than before**
  - **And code is more complicated**

Our system scales best when we can split the work into lots of equal chunks that can run in parallel.

Javaspecialists.eu

JavaSpecialists.EU

CompletionService: Executor & BlockingQueue

- **Sometimes we want to see results as they are complete**
- **Wrap an ExecutorService with a CompletionService**
- **CompletionService is parameterized with generics**
  - **Should only be used with one type of Callable**

Javaspecialists.eu

JavaSpecialists.EU

Example: Renderer with CompletionService

- **We now combine the FutureRenderer with the CompletionService**
  - **We can then render images as they are loaded**
- **Several CompletionServices can share one ExecutorService**
  - **However, a CompletionService should be for homogenous tasks**

Javaspecialists.eu

JavaSpecialists.EU

Renderer with CompletionService

```
public class Renderer extends BasicRenderer {
  private final ExecutorService executor;

  public Renderer(ExecutorService executor) {
    this.executor = executor;
  }

  public void renderPage(CharSequence source)
      throws ExecutionException, InterruptedException {
    List<ImageInfo> infos = scanForImageInfo(source); // step 1
    CompletionService<ImageData> cs =
        new ExecutorCompletionService<>(executor);
    infos.forEach(imageInfo -> {
          Callable<ImageData> task = imageInfo::downloadImage; // step 2
          cs.submit(task);
        }
    );
    renderText(source); // step 3
    for (int i = 0; i < infos.size(); i++) {
      renderImage(cs.take().get()); // step 4
    }
  }
}
```

Why can't we just use a parallelStream() ?

Javaspecialists.eu

JavaSpecialists.EU

CompletableFuture in Java 8

- **Flexible way to model complex concurrent workflows**
  - Unlike parallel streams, we can supply own Executor to async calls
- **CompletableFuture.supplyAsync(Supplier<U>)**
  - Similar to submit(Callable<U>) method in ExecutorService
- **CompletableFuture.runAsync(Runnable)**
  - Similar to submit(Runnable) method in ExecutorService
- **All or any**
  - CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)
  - CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)

Javaspecialists.eu

JavaSpecialists.EU

Chaining of CompletableFuture

- **We can build complicated asynchronous workflows**

```
CompletableFuture
  .supplyAsync(info::downloadImage, downloadPool)
  .thenAcceptAsync(this::renderImage, renderPool);
```

- **Methods in CompletionStage interface are**
  - **acceptEither, applyToEither, runAfterBoth, runAfterEither, thenAccept, thenAcceptBoth, thenApply, thenCombine, thenCompose, thenRun and whenComplete**
  - **Each method can be called asynchronously and with own Executor**
    - **Default is either common Fork/JoinPool or ThreadPerTaskExecutor**

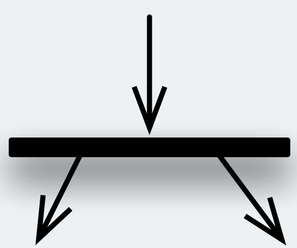
```
static final class ThreadPerTaskExecutor implements Executor {
  public void execute(Runnable r) { new Thread(r).start(); }
}
```

Javaspecialists.eu

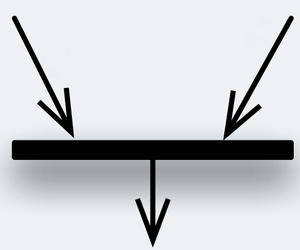
JavaSpecialists.EU

Modeling Control Flow

- **Our Renderer example could also be modelled by an UML Activity Diagram**
- **Fork**



- **Join**



start

CompletableFuture.allOf()

runAsync()

runAsync()

Render Text

Scan for Image Info

supplyAsync()

Download Image 1

Download Image 2

...

Download Image n

thenAcceptAsync(ImageData)

Render Image 1

Render Image 2

...

Render Image n

join()

join()

end

Javaspecialists.eu

JavaSpecialists.EU

Renderer With CompletableFuture

- **Independent tasks can now be executed concurrently**

```
private final ExecutorService renderPool;
private final ExecutorService downloadPool;

public void renderPage(CharSequence source) {
  CompletableFuture.allOf(
      CompletableFuture.runAsync(() ->
          scanForImageInfo(source) // step 1 - using renderPool
              .stream()
              .map(info ->
                  CompletableFuture
                                  // step 2 with downloadPool
                      .supplyAsync(info::downloadImage, downloadPool)
                      .thenAcceptAsync(this::renderImage,
                          renderPool) // step 4 - using renderPool again
              )
              .collect(Collectors.toList())
              .forEach(CompletableFuture::join), renderPool),
      CompletableFuture.runAsync(
          () -> renderText(source), renderPool) // step 3 - using renderPool
  ).join();
}
```

Javaspecialists.eu

JavaSpecialists.EU

CompletableFuture Tips

- **Can be complicated to write and debug**
  - **Always start by drawing an activity diagram**
  - **Try to keep code in sync with diagram**
- **On a single or dual core machine, it defaults to**

```
static final class ThreadPerTaskExecutor implements Executor {
  public void execute(Runnable r) { new Thread(r).start(); }
}
```

  - **Otherwise, it uses the common Fork/Join pool**
  - **Rather specify your own ExecutorService with async calls**
- **Make sure to eventually join() async tasks**
  - **Testing this can be tricky**

Javaspecialists.eu

JavaSpecialists.EU

4.3: Using Parallel Streams

Task Execution

JavaSpecialists.EU

4.3: Using Parallel Streams

- **In Java 8, collections can be transformed into streams**
  - **These can then easily be processed in parallel**
  - **But remember, only use parallel streams for CPU intensive tasks!!!**
    - **If you use them for IO intensive tasks (like downloading images), you will jam up the works for other parallel streams in the system**

Javaspecialists.eu

JavaSpecialists.EU

LongStream Producing Numbers

- **For example, to find all prime numbers within a range:**

```
public LongStream primeNumbers(long start, long end) {
  return LongStream.range(start, end)
      .filter(l -> Factorizer.factor(l).length == 1);
}
```

- **To solve this in parallel requires one small change:**

```
public LongStream primeNumbers(long start, long end) {
  return LongStream.range(start, end)
      .parallel()
      .filter(l -> Factorizer.factor(l).length == 1);
}
```

Javaspecialists.eu

JavaSpecialists.EU

Creating Streams From Collections

- **We can create streams from any collection:**

```
Collection<?> col = ...
col.stream()
   .filter(some predicate)
   .map(some conversion method)
   .forEach(System.out::println);
```

- **For example**

```
Collection<String> names = new ArrayList<>();
Collections.addAll(names,
    "Heinz", "Max", null, "Charalampos",
    "Vuyisile", null, "Kabutz");
names.stream()
    .filter(n -> n != null)
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

```
HEINZ
MAX
CHARALAMPOS
VUYISILE
KABUTZ
```

Javaspecialists.eu

JavaSpecialists.EU

Converting Streams To Parallel Streams

- **Streams can be run to a parallel stream with the parallel() call**
  - **Internally uses the Fork/Join framework (more tomorrow)**

```
IntSummaryStatistics stats =
  IntStream.range(0, 100_000_000)
    .parallel()
    .map(val -> ThreadLocalRandom.current().nextInt(1000))
    .summaryStatistics();
```

  - **On a 1-4-2 machine, parallel() completed in 160ms. Without parallel() it took 595ms**
  - **Alternative approach:**

```
IntSummaryStatistics stats =
    ThreadLocalRandom.current()
      .ints(100_000_000, 0, 1000)
      .parallel()
      .summaryStatistics();
```

Javaspecialists.eu

JavaSpecialists.EU

Common Fork/Join Pool

- **Parallel streams use the shared Fork/Join Pool**
  - Sized to # of hardware threads minus 1 (current thread also works)
  - We can get it with ForkJoinPool.commonPool()
- **Since the pool is used by all parallel streams in the JVM:**
  - Do not do any IO or synchronization (exception for Phaser)
  - Your JVM might not be the only application using cores
    - Change the size with

      -Djava.util.concurrent.ForkJoinPool.common.parallelism=<num>
  - http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html

Javaspecialists.eu

JavaSpecialists.EU

4: Exercises

Task Execution

JavaSpecialists.EU

Exercise 4.1: Parallel Factorizer

- **We have tried parallelizing this code for finding primes**

```
int NUMBERS_TO_CHECK = 20000;
long start = (1 << 19) - 1; // known Mersenne prime
int primes = 0;
for (int i = 0; i < NUMBERS_TO_CHECK; i++) {
  long[] factors = Factorizer.factor(start + i);
  if (factors.length == 1) {
    primes++;
  }
}
assertEquals(1514, primes);
```

- **However, our solution creates a new thread for each task**
  - **Change the ParallelFactorizer to use a fixed thread pool**
    - **Also consider a good size for each task**

Javaspecialists.eu

JavaSpecialists.EU

Exercise 4.2: Parallel Stream Factorizer

- **Instead of using ExecutorService, use a LongStream to filter the numbers**

```
int NUMBERS_TO_CHECK = 20000;
long start = (1 << 19) - 1; // known Mersenne prime
int primes = 0;
for (int i = 0; i < NUMBERS_TO_CHECK; i++) {
  long[] factors = Factorizer.factor(start + i);
  if (factors.length == 1) {
    primes++;
  }
}
assertEquals(1514, primes);
```

Javaspecialists.eu

JavaSpecialists.EU

4: Task Execution

Questions?

5: Cancellation

5. Cancellation

- **Starting a thread is very easy**

```
new Thread(job).start();
```

- **Stopping it again is not so easy**
  - **The only way a thread can stop is if it exits the run() method**
  - **Java 1.0, Sun gave us Thread.stop() and Thread.destroy() methods**
    - **destroy() was never implemented and stop() is broken and dangerous, don't ever use them**
  - **We need to use cooperative mechanisms to stop our threads**

Javaspecialists.eu

JavaSpecialists.EU

Reasons For Wanting To Cancel A Task

- **There can be many reasons why a task is canceled**
  - **User clicks “cancel” button**
  - **Time’s up for some calculation**
  - **One divide & conquer branch discovers a solution**
  - **Errors, such as disk full or out of memory**
  - **Shutdown**

Javaspecialists.eu

JavaSpecialists.EU

Cooperative Vs Preemptive Cancellation

- **Java does not have a practical mechanism for preemptively stopping a task**
  - **In Java, a stop() can easily break invariants, for example**

```
// INVARIANT: i == j
synchronized(lock) {
  i++;
  j++;
}
```

    - **If the thread is stopped after the call to i++, the invariant is broken**
    - **stop() can leave the data in an inconsistent state**
  - **Only the executing thread can decide on the best time to stop**

Javaspecialists.eu

JavaSpecialists.EU

"Cancellation Request" Flag

- **This cooperative mechanism uses a volatile boolean flag**
  - **Occasionally, this flag is checked to see if we should cancel**

```
@ThreadSafe
public class PrimeGenerator implements Runnable {
  private final List<BigInteger> primes =
    new CopyOnWriteArrayList<>();
  private volatile boolean cancelled = false;

  public void cancel() { cancelled = true; }

  public void run() {
    BigInteger p = BigInteger.ONE;
    while (!cancelled) {
      p = p.nextProbablePrime(); // this might take a while
      primes.add(p);
    }
  }

  public List<BigInteger> get() { return primes; }
}
```

Important!

Javaspecialists.eu

JavaSpecialists.EU

Printing A "Second" Worth Of Primes

- **We start the generator and cancel it after one second**

```
PrimeGenerator generator = new PrimeGenerator();
new Thread(generator).start();
try {
  TimeUnit.SECONDS.sleep(1);
} finally {
  generator.cancel();
}
System.out.println(generator.get());
```

Javaspecialists.eu

JavaSpecialists.EU

Cancellation Policy

- **Tasks that can be cancelled need a cancellation policy**
  - **How can other code request cancellation?**
    - **PrimeGenerator: client must call cancel()**
  - **When does the task check whether it is cancelled?**
    - **PrimeGenerator: checks the flag once per prime found**
  - **What actions does the task perform when it sees it is cancelled?**
    - **Does it throw an exception?**
    - **Does it return a partial result?**
    - **PrimeGenerator: stops looking for more primes and exits run()**

Javaspecialists.eu

JavaSpecialists.EU

Interruption

- **Using a volatile boolean cancellation field only works if the thread does not enter the WAITING state**
  - **e.g. calling put() or await() or take()**

```
public class BrokenPrimeProducer extends Thread {
  private final BlockingQueue<BigInteger> queue;
  private volatile boolean cancelled = false;
  public BrokenPrimeProducer(BlockingQueue<BigInteger> queue) {
    this.queue = queue;
  }

  public void run() {
    try {
      BigInteger p = BigInteger.ONE;
      while (!cancelled) queue.put(p = p.nextProbablePrime());
    } catch (InterruptedException consumed) { }
  }

  public void cancel() { cancelled = true; }
}
```

Javaspecialists.eu

JavaSpecialists.EU

How Does Interrupt Work?

- When we call Thread.interrupt(), a flag is set in the C layer
- If the thread is currently in WAITING state, it immediately returns with an InterruptedException
  - As a side effect, the interrupted status is set to false again
    - We need to do something similar if we throw the exception
- If the thread is not WAITING, then it continues running
  - Thread.currentThread().isInterrupted() will now return true
  - We can poll this like we did our volatile flag to cooperatively stop
- Calling interrupt() does not stop the target thread; it is simply a message that interruption was requested

Javaspecialists.eu

JavaSpecialists.EU

Policies In Dealing With InterruptedException

- **If an InterruptedException occurs, you should either**
  - **Pass it on to the caller, who might know better how to deal with it**
  - **Cancel immediately, abandoning the current work**
  - **Finish what you are doing currently and then cancel**
  - **Decide that you cannot cancel, in which case**
    - **Set the thread to not be interrupted, but remember that it was**
    - **Finish all the work**
    - **Set the thread to be interrupted again and return**
- **Simply ignoring the InterruptedException is not good**
  - **Methods up the call stack would not know about the interruption**
  - **Swallowing the exception is hiding that the interruption occurred**

Javaspecialists.eu
JavaSpecialists.EU

Static Thread.interrupted() Method

- **The Thread.interrupted() method returns current status of interrupted flag, but *also* sets the status false**
- **If you call it, you should typically either throw the InterruptedException or re-interrupt the thread later**
- **A best practice is to start every interruptible blocking method with the following**

```
if (Thread.interrupted())
  throw new InterruptedException();
```

  - **This allows the method to return immediately if the flag is set**
  - **It also maintains the rule that the flag is reset on exception**

Javaspecialists.eu

JavaSpecialists.EU

Fixing PrimeProducer

- **We can use the interrupted status to determine when to shut down the prime producer thread**

```
public class PrimeProducer extends Thread {
  private final BlockingQueue<BigInteger> queue;

  public PrimeProducer(BlockingQueue<BigInteger> queue) {
    this.queue = queue;
  }

  public void run() {
    try {
      BigInteger p = BigInteger.ONE;
      while (!Thread.currentThread().isInterrupted())
        queue.put(p = p.nextProbablePrime());
    } catch (InterruptedException consumed) {
      /* Allow thread to exit */
    }
  }

  public void cancel() { interrupt(); }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Cancellation Via Future

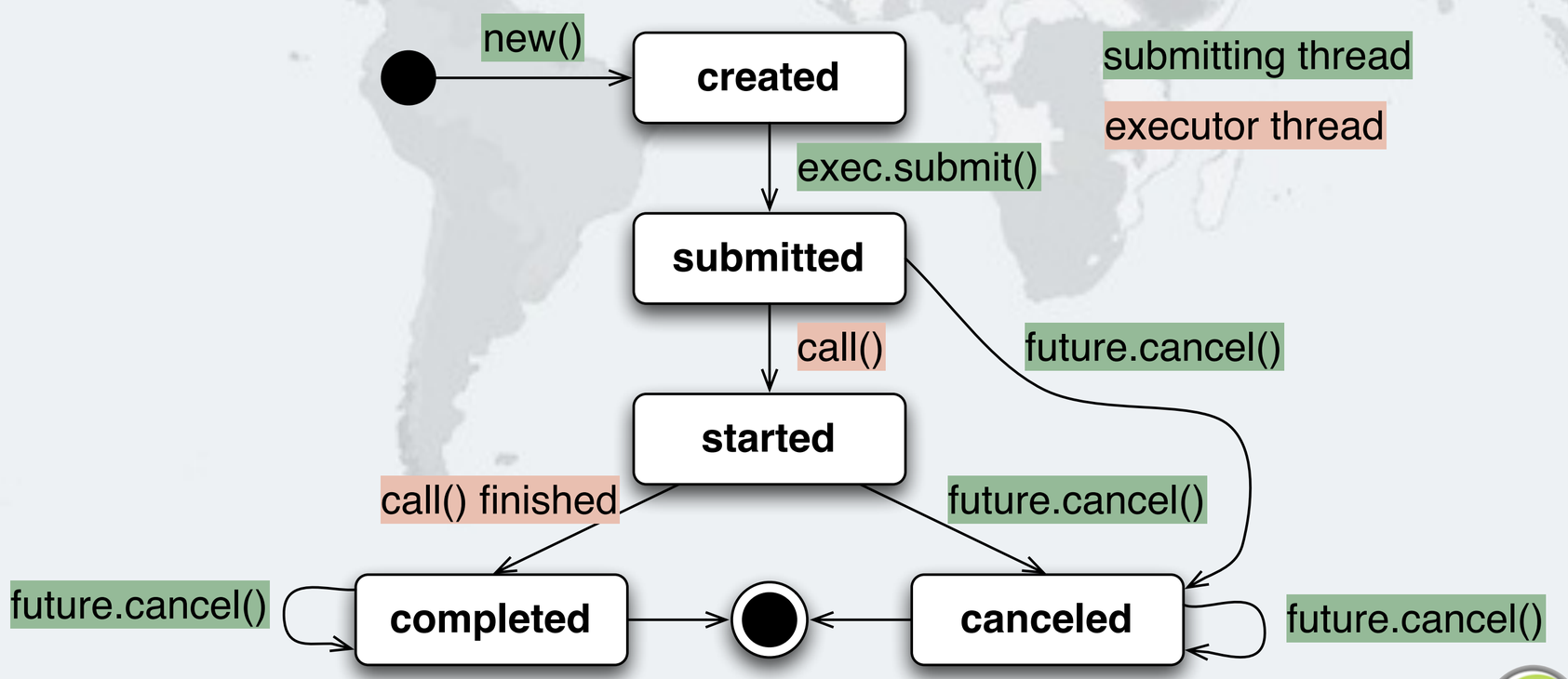
- Some problems can take forever to complete
- We might want to try for a while and then give up
- For example, a three dimensional box packing program
  - Well known NP complete problem, like the traveling salesman
  - We might find a solution quickly, or not at all
- The best way to cancel jobs is via the Future<E> interface
  - Much easier and safer than rolling your own

Javaspecialists.eu

JavaSpecialists.EU

Controlling Task Lifecycle

- **A callable task can be cancelled by Future.cancel()**
  - **If the task is submitted, it will go straight to the cancelled state**
  - **If the task is busy, it can be interrupted**
  - **If the task is already cancelled or complete, nothing is done**



Javaspecialists.eu

JavaSpecialists.EU

Helper Class For Canceling Streams

```
public class CancellationSupport {
  private static void cancelIfInterrupted() {
    if (Thread.currentThread().isInterrupted())
      throw new CancellationException("interrupted");
  }

  public static void cancelIfInterrupted(double d) {
    cancelIfInterrupted();
  }

  public static void cancelIfInterrupted(long l) {
    cancelIfInterrupted();
  }

  public static void cancelIfInterrupted(int i) {
    cancelIfInterrupted();
  }

  public static <T> void cancelIfInterrupted(T t) {
    cancelIfInterrupted();
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Add Cancelation to Stream With peek()

- **Stream peek() allows us to consume intermittent results**

```
public class CancellationSupportTest {
  public static void main(String... args) {
    Timer timer = new Timer(true);
    timer.schedule(new TimerTask() {
      private final Thread mainThread = Thread.currentThread();
      public void run() {
        System.out.println("Canceling");
        mainThread.interrupt();
      }
    }, 1000);
    DoubleStream.generate(
      () -> ThreadLocalRandom.current().nextDouble())
        .peek(CancellationSupport::cancelIfInterrupted)
        .forEach(System.out::println);
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Canceling Busy Jobs

- **The Future.cancel() method allows us to interrupt threads**

```
public static <E> Optional<E> timedRun(
    Callable<E> r, long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException {
  Future<E> task = taskExec.submit(r);
  try {
    E result = task.get(timeout, unit);
    return Optional.of(result); // does not support null!
  } catch (TimeoutException e) {
    task.cancel(true); return Optional.empty();
  } catch (InterruptedException e) {
    task.cancel(true); throw e;
  }
}
```

  - **We should throw the ExecutionException to the caller**
  - **The task should only be cancelled if we had a timeout**
  - **Future.cancel() takes a parameter *mayInterruptIfRunning***
    - **true means the thread is interrupted; false means it won't**

Javaspecialists.eu
JavaSpecialists.EU

Finding "Big" Prime Number Within 3 Seconds

- **We are looking for a prime that is more than 5 digits long**
  - **And if it does not come back within 3 seconds, we give up**

```
Optional<BigInteger> prime = timedRun(() -> {
  BigInteger p = BigInteger.ONE;
  while (!Thread.currentThread().isInterrupted()) {
    p = p.nextProbablePrime();
    if (p.toString().length() > 5) return p;
  }
  return null;
}, 3, TimeUnit.SECONDS);
System.out.println(
  prime.map(bi -> "Prime is " + bi)
    .orElse("Prime not found"));
```

  - **Avoid calling get() on Optional**
    - **http://www.javaspecialists.eu/archive/Issue238.html**

Javaspecialists.eu

JavaSpecialists.EU

Dealing With Non-Interruptible Blocking

- **Not all JDK library methods throw InterruptedException when we interrupt the thread whilst blocking**
  - **Synchronous socket I/O in java.io**
    - **Interrupting the thread has no effect, you need to close the socket**
  - **Synchronous I/O in java.nio**
    - **Interrupting the thread causes a ClosedByInterruptException**
    - **This only happens for InterruptibleChannel (most are)**
    - **Closing the channel causes an AynchronousCloseException**
  - **Asynchronous I/O with Selector.select()**
    - **Interrupting the thread causes a ClosedSelectorException**

Javaspecialists.eu

JavaSpecialists.EU

5: Exercises

Cancellation and Shutdown

JavaSpecialists.EU

Exercise 5.1: Make The Factorizer Cancelable

- **Change factor() to allow another thread to cancel it**

```
public class Factorizer {
  public long[] factor(long number)
      throws InterruptedException {
    Collection<Long> factors = new ArrayList<>();
    for (long factor = 2; factor <= number; factor++) {
      while (number % factor == 0) {
        factors.add(factor);
        number /= factor;
      }
    }
    long[] result = new long[factors.size()];
    int pos = 0;
    for (Long factor : factors) {
      result[pos++] = factor;
    }
    return result;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

5: Cancellation And Shutdown

Questions?

6: Applying Thread Pools

Using Java's built-in Executors to manage thread life cycles

JavaSpecialists.EU

6: Applying Thread Pools

- **Thread pools need to be sized correctly**
  - **too large will results in overloaded resources**
  - **too small will not utilize all the CPUs**

Javaspecialists.eu

JavaSpecialists.EU

Tasks And Execution Policies

- **Thread pools work best with *homogeneous*, *independent* and *thread-agnostic* tasks**
  - **Homogenous**
    - **Tasks take approximately the same amount of time**
  - **Independent**
    - **Tasks do not depend on results from other tasks**
  - **Thread-agnostic**
    - **ThreadLocal use, if any, is bound to the lifetime of the task**

Javaspecialists.eu

JavaSpecialists.EU

6.1: Sizing Thread Pools

Applying Thread Pools

JavaSpecialists.EU

6.1: Sizing Thread Pools

- **Thread pool size should not be hard coded**
  - How many cores is your server going to have in 5 years time?
    - Japanese "K" has over 500,000 cores
  - Either make it configurable or use Runtime.availableProcessors()
    - This usually gives number of hardware threads, not physical cores
    - Example: The Intel® Core™ i7-2820QM Processor (8M Cache, 2.30 GHz) has 4 cores and 8 threads using Hyper-Threading
      - Runtime.getRuntime().availableProcessors() will return 8

Javaspecialists.eu

JavaSpecialists.EU

Μέτρον Άριστον (Metron Ariston)

- **Everything in moderation**
- **Pool size too large**
  - **Threads compete for scarce CPU cycles**
  - **Out of memory errors can occur**
- **Pool size too small**
  - **Throughput suffers**
  - **Processors are not fully utilized**

Javaspecialists.eu

JavaSpecialists.EU

Sizing For Compute-Intensive Tasks

- **If the tasks are compute-intensive and independent**
  - **Threads for pool should be number of hardware threads + 1**
    - **But remember it might not be the *only* thread pool in the system!**
  - **There will always be times where a thread needs to wait, hence + 1**

Javaspecialists.eu

JavaSpecialists.EU

Sizing For Blocking Operations

- **We need the ratio of wait time over compute time (W/C)**
  - **can be estimated**
    - **W/C = %wait / (100 - %wait)**
- **With the following definitions:**
  - **N_{cpu} = number of CPUs (number of hardware threads)**
  - **U_{cpu} = target CPU utilization, between 0 and 1**
    - **Typically close to 1**
  - **W/C = ratio of wait over compute time**
- **Then the formula is**
  - **$N_{threads} = N_{cpu} \times U_{cpu} \times (1 + W/C)$**

Javaspecialists.eu

JavaSpecialists.EU

Example With Compute-Intensive Tasks

- **16 CPUs are busy trying to factor a large number**
  - $N_{cpu} = 16$
  - $U_{cpu} = 1$
  - W/C = 1 / 8
    - There is some dependency between results
- **Then the formula is**
  - $N_{threads} = N_{cpu} \times U_{cpu} \times (1 + W/C) = 16 \times 1 \times (1 + 1/8) = 18$
- **We should thus use 18 threads in this pool**
  - But 19 or 25 would also work, but 100 would be too many

Javaspecialists.eu

JavaSpecialists.EU

Example of Formula with IO-Intensive Tasks

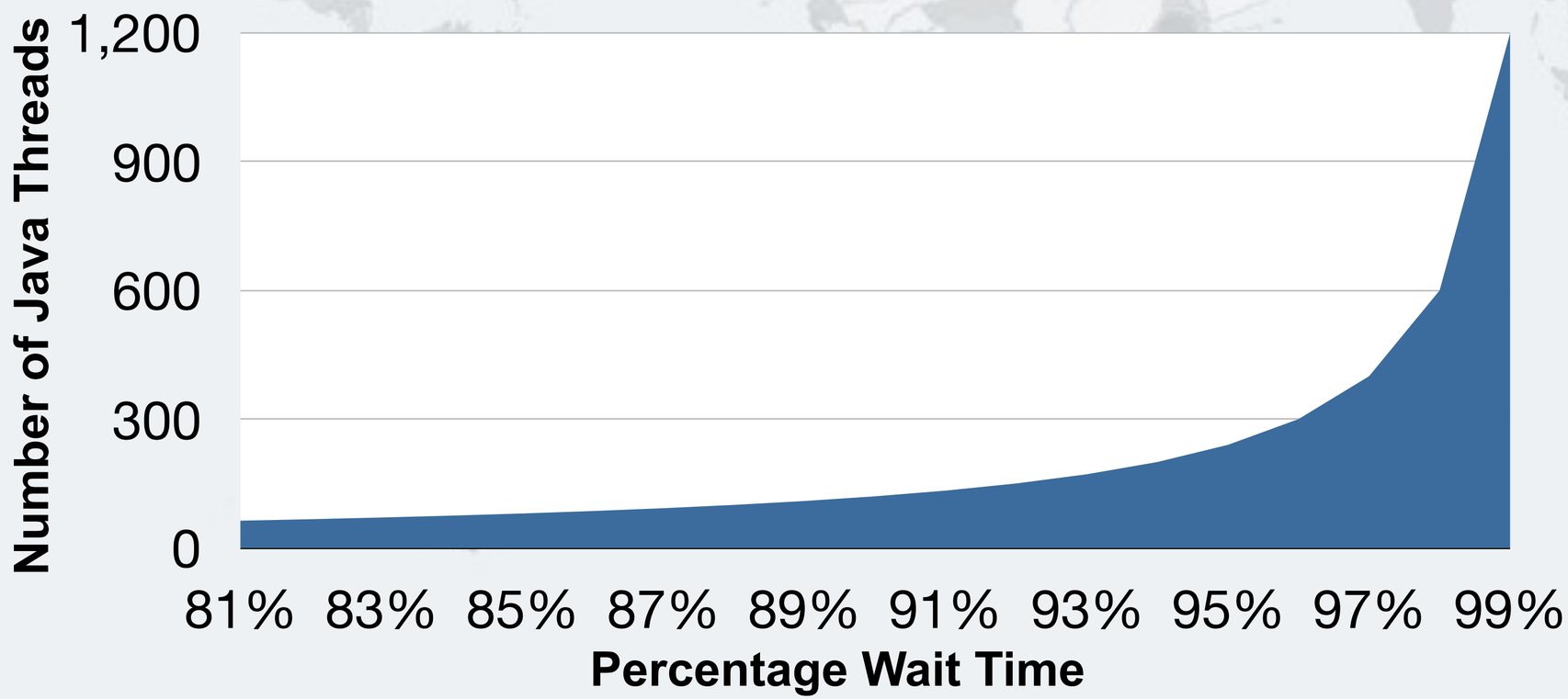
- **16 CPUs are serving web pages**
  - $N_{cpu} = 16$
  - $U_{cpu} = 0.75$
    - We want spare CPU cycles to be available in case of sudden load
  - W/C = 9
    - 90% of the time, we are waiting on IO
- **Then the formula is**
  - $N_{threads} = N_{cpu} \times U_{cpu} \times (1 + W/C) = 16 \times 0.75 \times (1 + 90/10) = 120$
- **We should thus configure our pool with 120 threads**

Javaspecialists.eu

JavaSpecialists.EU

Long Waits Need More Threads

- **As the percentage of wait time increases, we need more software threads to keep the hardware threads busy**
  - **16 CPUs, desired 75% CPU Utilization**



Javaspecialists.eu
JavaSpecialists.EU

Mixing CPU and IO Intensive Tasks

- **For different categories of tasks, it is better to have separate thread pools**
  - **CPU-intensive tasks go to a small thread pool**
  - **IO-intensive tasks go to a large or even unbounded pool**
    - **Some JVMs cannot cope with very many threads, be careful**
    - **Mac OS X Leopard limit set to 2560 threads, includes JVM threads**
      - **Unix-based JVMs: "sysctl -A | grep threads" for max number**
      - **Mac OS X Lion limits it to 2048**
    - **However, you could start hundreds of processes, each with thousands of threads if you really wanted to**

Javaspecialists.eu

JavaSpecialists.EU

6.2: Configuring ThreadPoolExecutor

Applying Thread Pools

JavaSpecialists.EU

6.2: Configuring ThreadPoolExecutor

- **Most of the creation methods in Executors.new* use the ThreadPoolExecutor with different configurations**
  - One specialization is the subclass ScheduledThreadPoolExecutor
- **Whenever possible, use default executors**
  - Executors.newFixedThreadPool
    - Fixed maximum number of threads, unbounded runnable queue
  - Executors.newCachedThreadPool
    - Unbounded number of threads, size zero runnable queue
    - Threads are stopped after being idle for one minute
  - Executors.newSingleThreadExecutor
    - Wraps a fixed thread pool of size 1 with an un-configurable pool

Javaspecialists.eu

JavaSpecialists.EU

Thread Creation And Teardown

- **The ThreadPoolExecutor has several tuning parameters:**
  - **Interactions between the parameters is not always obvious**
  - **corePoolSize - the pool will try to maintain this number of threads**
    - **Core threads are only started when tasks are submitted to the pool**
      - **Amortizes cost of constructing threads**
      - **Use prestartAllCoreThreads() to get them started up front**
      - **It won't create more threads than "core" until work queue is full**
  - **maximumPoolSize - upper bound on number of live threads in pool**
    - **This includes busy and idle threads**
  - **keepAliveTime - if a thread has been idle for longer than this, it can be terminated if the number of threads exceeds corePoolSize**

Javaspecialists.eu

JavaSpecialists.EU

ThreadPoolExecutor Constructor

- **The ThreadPoolExecutor constructor**
  - **threadFactory and handler parameters are optional**

```
public ThreadPoolExecutor(
    int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory,
    RejectedExecutionHandler handler) { ... }
```

Javaspecialists.eu

JavaSpecialists.EU

Example Configurations

- **Executors.newFixedThreadPool(nThreads)**
  - corePoolSize = maximumPoolSize = nThreads
  - keepAliveTime = 0L milliseconds
  - workQueue = unbounded LinkedBlockingQueue
- **Executors.newCachedThreadPool()**
  - corePoolSize = 0
  - maximumPoolSize = Integer.MAX_VALUE
  - keepAliveTime = 60L seconds
  - workQueue = SynchronousQueue

Javaspecialists.eu

JavaSpecialists.EU

Which One Should You Choose?

- **The cached thread pool if you know task insertion rate**
  - **It has better queueing performance due to a hand-over locking mechanism in the synchronous queue**
    - **The non-fair mechanism uses a stack, which returns the values in the reverse order of how they were submitted**
  - **However, there is no limit on the number of threads**
- **The fixed sized thread pool if you want to limit the number of concurrent tasks**
  - **Good choice for network services**
  - **Perhaps use a bounded blocking queue if tasks are independent**

Javaspecialists.eu

JavaSpecialists.EU

Saturation Policies

- **The default policy for dealing with a bounded queue is to throw an unchecked exception**
  - **This is called the abort policy.**
- **Thread pool executor provides 4 different such policies:**
  - **abort policy**
  - **caller runs policy**
  - **discard policy**
  - **discard oldest policy**
- **Saturation policy is applied when thread pool has been shut down and when it is temporarily overloaded**

Javaspecialists.eu

JavaSpecialists.EU

AbortPolicy

- **This policy simply throws an exception when queue is full**

```
public static class AbortPolicy
    implements RejectedExecutionHandler {
  public void rejectedExecution(
      Runnable r, ThreadPoolExecutor e) {
    throw new RejectedExecutionException();
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

CallerRunsPolicy

- **This policy uses the submitting thread to run the task**
  - **This effectively throttles the caller from submitting more tasks**
  - **However, this task will now be executed out-of-order**

```
public static class CallerRunsPolicy
    implements RejectedExecutionHandler {
  public void rejectedExecution(
      Runnable r, ThreadPoolExecutor e) {
    if (!e.isShutdown()) {
      r.run();
    }
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Discardpolicy

- **This policy silently discards the job**
  - **This is probably the worst policy to use**

```
public static class DiscardPolicy
    implements RejectedExecutionHandler {
  public void rejectedExecution(
      Runnable r, ThreadPoolExecutor e) {
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

DiscardOldestPolicy

- **A useful policy when the oldest task is no longer relevant**
  - **When a system is overloaded, it is often a good idea to discard work that is no longer necessary**

```
public static class DiscardOldestPolicy
    implements RejectedExecutionHandler {
  public void rejectedExecution(
      Runnable r, ThreadPoolExecutor e) {
    if (!e.isShutdown()) {
      e.getQueue().poll();
      e.execute(r);
    }
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Thread Factories

- **Threads in the pools are created using a thread factory**

```
public interface ThreadFactory {
  Thread newThread(Runnable r);
}
```

- **Executors.defaultThreadFactory() used by default**
  - **This makes the threads non-daemons and sets priority to normal**
    - **Usually, threads inherit their creator's daemon status and priority**
    - **Otherwise threads could abruptly terminate or have wrong priority**
  - **The default factory also gives a unique name to each worker thread**

Javaspecialists.eu

JavaSpecialists.EU

6: Exercises

Applying Thread Pools

JavaSpecialists.EU

Exercise 6.1: ThreadPoolSupplier

- **Return a new ThreadPoolExecutor with characteristics**
  1. **Core pool size of 10, maximum pool size of 100, timeout 5 seconds**
  2. **Task queue should be bounded queue of 1000.**
  3. **Threads have priority of Thread.MAX_PRIORITY-2, be non-daemon and be named by pattern such as pool-1-thread-1#description**
  4. **If we exceed capacity, let the caller run the job himself.**
  5. **If we shut down, reject new jobs with RejectedExecutionException**

```
public class ThreadPoolSupplier implements
    Function<String, ThreadPoolExecutor> {
  public ThreadPoolExecutor apply(String description) {
    return (ThreadPoolExecutor) Executors.newFixedThreadPool(10);
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Exercise 6.2: Thread Pool MBean (Opt)

- **Complete the code for the MBean that can control and manage ThreadPoolExecutors**
  - **We must be able to unregister bean when the thread pool is done**
  - **We should be able to shut down the pool and prestart core threads**
  - **Should see the core pool size, keep alive time, pool size, active count, largest pool size, task count and completed task count**
- **Run ThreadPoolStatsTest, and connect with JConsole**

```
-Dcom.sun.management.jmxremote.port=21000
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
```

- **Try what happens when you shut down individual pools or change the core pool size**

Javaspecialists.eu

JavaSpecialists.EU

Exercise 6.3: Sizing ThreadPoolExecutor (Opt)

- **Implement the ThreadPoolFactory to construct a fixed thread pool with the number of threads according to:**
  - **CPUs: This is the number of hardware threads that you want to use**
  - **Utilization: How busy you want to be, a number between 0.0 and 1.0**
  - **Wait time: Amount of time that each thread would need to wait**
  - **Maximum threads: Limit of how many threads to create**

Javaspecialists.eu

JavaSpecialists.EU

6: Applying Thread Pools

Questions?

Day 2

7: Fork/Join

Support for parallel decomposition

JavaSpecialists.EU

7: Fork/Join

- Some problems can be solved faster in parallel
- A large task can often be split up into smaller tasks
  - These in turn can be broken up into even smaller tasks
  - Until we reach a task that is so small that it is faster to do it sequentially than to break it up further
    - We call this *recursive decomposition* or *divide and conquer*
- A single task queue, such as with ThreadPool, would introduce contention
  - It would be better to have many work queues, to reduce contention

Javaspecialists.eu

JavaSpecialists.EU

ForkJoinPool and ForkJoinTask

- **The most important classes to know are**
  - **ForkJoinPool is a new ExecutorService written specifically for managing fork/join tasks**
    - **We call invoke() with our ForkJoinTask and wait for the result**
    - **Note: the threads are always daemon threads**
  - **ForkJoinTask is the task that is submitted to the ForkJoinPool**
    - **If the task to be done is relatively small, do it sequentially**
    - **Otherwise, we'd fork off a second task that could then be done in parallel by another thread from the ForkJoinPool**

Javaspecialists.eu

JavaSpecialists.EU

Work-Stealing in ForkJoinPool

- **One of the benefits we have in the ForkJoinPool over traditional ExecutorServices is work-stealing**
  - **If a thread finishes all its tasks, it "steals" jobs from other queues**
    - **Much like in a supermarket**
  - **If we allow too much stealing, then the management overhead will be greater than the benefit**
  - **If we allow too little, some threads will be idle**

Javaspecialists.eu

JavaSpecialists.EU

ForkJoinTask Methods

```
public abstract class ForkJoinTask<V> implements Future<V> {
  V invoke();
  ForkJoinTask<V> fork();
  V join();
  Throwable getException()
  static void invokeAll(ForkJoinTask<?>... tasks)

  // lifecycle methods
  boolean cancel(boolean mayInterruptIfRunning)
  boolean isDone()
  boolean isCancelled()
  boolean isCompletedAbnormally()
  boolean isCompletedNormally()

  // adapter methods (also for Callable)
  static ForkJoinTask<?> adapt(Runnable runnable)
}
```

Javaspecialists.eu

JavaSpecialists.EU

RecursiveTask vs RecursiveAction

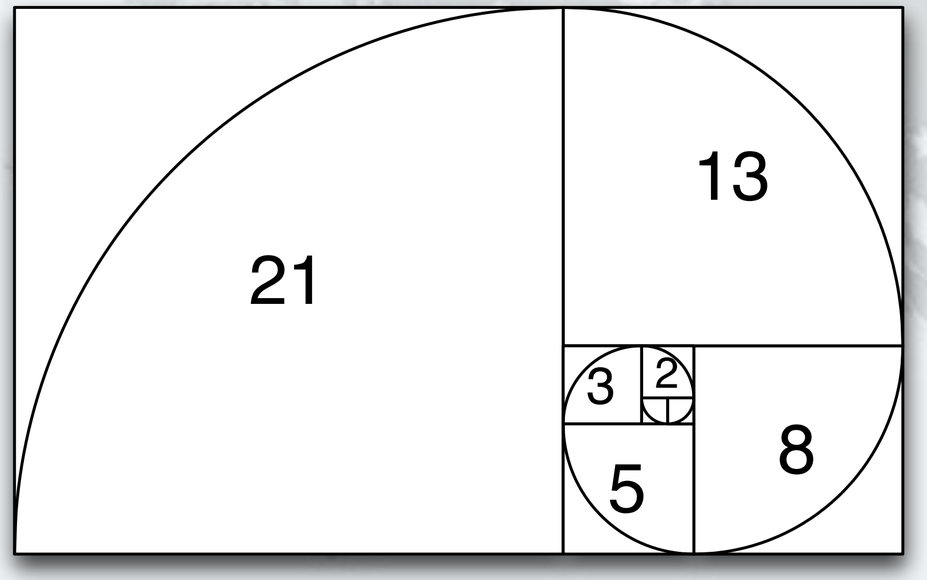
- **ForkJoinTask has two subclasses**
  - **RecursiveTask<V> returns a value V from compute()**
    - **Typically we would use this, avoids shared state**
  - **RecursiveAction**
    - **This does not return a value, so we need to have shared mutable state**

Javaspecialists.eu

JavaSpecialists.EU

Introducing Fibonacci

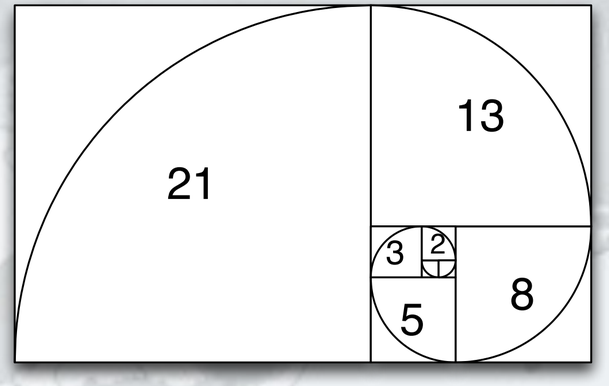
- **Number sequence named after Leonardo of Pisa**
  - **F0 = 0**
  - **F1 = 1**
  - **Fn = Fn-1 + Fn-2**
- **Thus the next number is equal to the sum of the two previous numbers**
  - **e.g. 0, 1, 1, 2, 3, 5, 8, 13, 21, …**
- **The numbers get large very quickly**



Javaspecialists.eu
JavaSpecialists.EU

First Attempt At Writing A Fibonacci Method

- **Taking our recursive definition**
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$



- **Our first attempt writes a basic recursive function**

```
public long f(int n) {
  if (n <= 1) return n;
  return f(n-1) + f(n-2);
}
```

- **But this has exponential time complexity**
  - f(n+10) is 1000 slower than f(n)

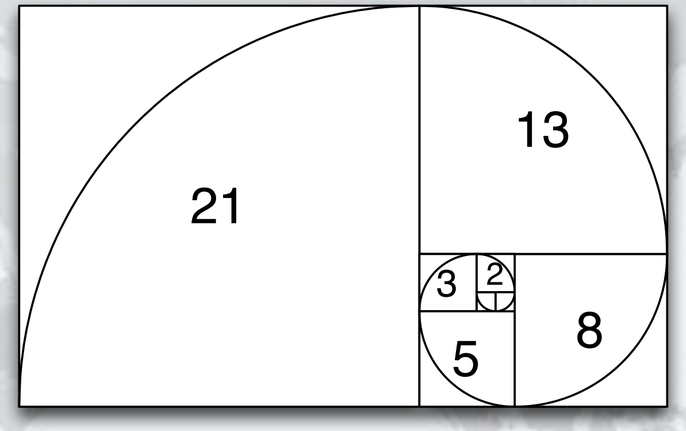
Javaspecialists.eu

JavaSpecialists.EU

Second Attempt At Writing A Fibonacci Method

- **Instead of a recursive method, we use iteration:**

```
public static long f(int n) {
  long n0 = 0, n1 = 1;
  for (int i = 0; i < n; i++) {
    long temp = n1;
    n1 = n1 + n0;
    n0 = temp;
  }
  return n0;
}
```

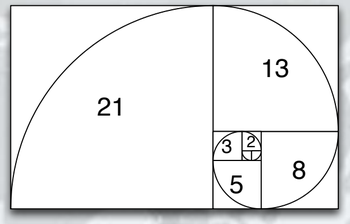


- **This algorithm has linear time complexity**
  - **Solved f(1_000_000_000) in 1.7 seconds**
    - **However, the numbers overflow so the result is incorrect**
    - **We can use BigInteger, but its add() is also linear, so time is quadratic**
    - **We need a better algorithm**

Javaspecialists.eu
JavaSpecialists.EU

Third Attempt Using Dijkstra's Sum Of Squares

- **Dijkstra noted the following formula for Fibonacci**
  - $F_{2n-1} = F_{n-1}^2 + F_n^2$
  - $F_{2n} = (2 \times F_{n-1} + F_n) \times F_n$

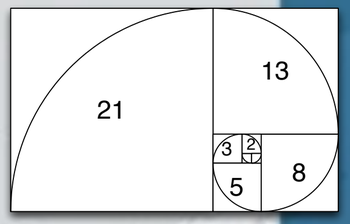


- **Logarithmic time complexity and can be parallelized**
  - Java 8 uses better BigInteger multiply() algorithms
    - Karatsuba complexity is $O(n^{1.585})$
    - 3-way Toom Cook complexity is $O(n^{1.465})$
    - Previous versions of Java had complexity $O(n^2)$
    - Unfortunately multiply() in BigInteger is only available single-threaded

Javaspecialists.eu

JavaSpecialists.EU

Djikstra's Sum of Squares with BigInteger




```
public BigInteger f(int n) {
  if (n == 0) return BigInteger.ZERO;
  if (n == 1) return BigInteger.ONE;
  if (n % 2 == 1) { // f(2n-1) = f(n-1)² + f(n)²
    int half_n = (n + 1) / 2;
    BigInteger fn_1 = f(half_n - 1);
    BigInteger fn = f(half_n);
    return fn_1.multiply(fn_1).add(fn.multiply(fn));
  } else { // F(2n) = ( 2 F(n-1) + F(n) ) F(n)
    int half_n = n / 2;
    BigInteger fn_1 = f(half_n - 1);
    BigInteger fn = f(half_n);
    return fn_1.shiftLeft(1).add(fn).multiply(fn);
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

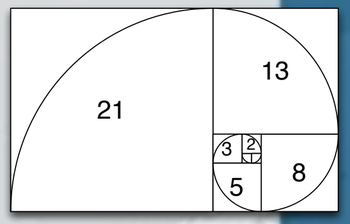
Fork/Join Vs Compute

- **If we are in a ForkJoinPool thread**
  - **If we want the task to be run in a separate thread, we call fork() to start it off and later join() to retrieve the result**
    - **Remember to always join() all the tasks that you fork**
  - **If we want the task to run in our current thread, we call invoke()**
- **Otherwise defaults to common Fork/Join worker pool**

Javaspecialists.eu

JavaSpecialists.EU

Creating RecursiveTask for Fibonacci

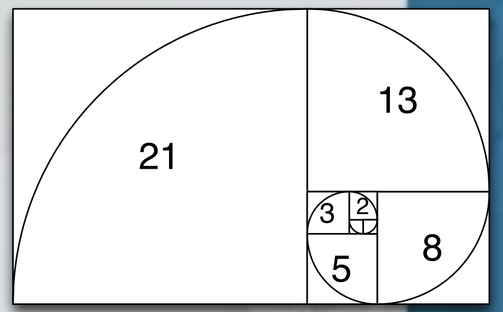



```
public class Fibonacci {
  private final class FibonacciTask
      extends RecursiveTask<BigInteger> {
    private final int n;

    private FibonacciTask(int n) {
      this.n = n;
    }

    protected BigInteger compute() {
      return f(n);
    }
  }
```

Javaspecialists.eu

JavaSpecialists.EU

We Now Patch FibonacciTask Into Old Code

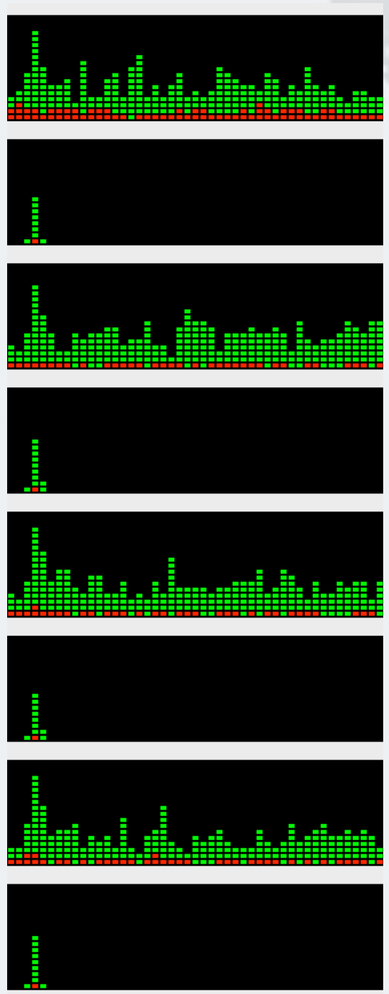



```
public BigInteger f(int n) {
  if (n == 0) return BigInteger.ZERO;
  if (n == 1) return BigInteger.ONE;
  if (n % 2 == 1) { // f(2n-1) = f(n-1)^2 + f(n)^2
    int half_n = (n + 1) / 2;
    FibonacciTask fn_1Task = new FibonacciTask(half_n - 1);
    fn_1Task.fork();
    BigInteger fn = f(half_n);
    BigInteger fn_1 = fn_1Task.join();
    return fn_1.multiply(fn_1).add(fn.multiply(fn));
  } else { // F(2n) = ( 2 F(n-1) + F(n) ) F(n)
    int half_n = n / 2;
    FibonacciTask fn_1Task = new FibonacciTask(half_n - 1);
    fn_1Task.fork();
    BigInteger fn = f(half_n);
    BigInteger fn_1 = fn_1Task.join();
    return fn_1.shiftLeft(1).add(fn).multiply(fn);
  }
}
}
```

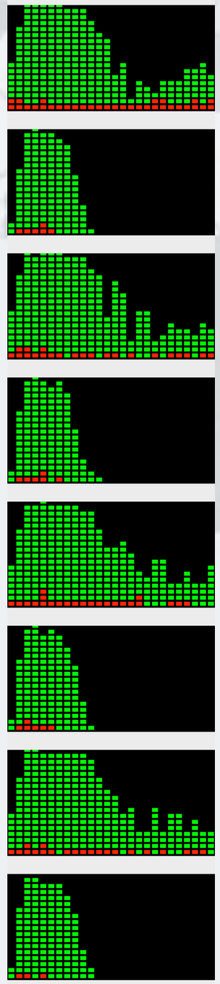
Javaspecialists.eu
JavaSpecialists.EU

CPU Utilization Fibonacci(100_000_000)

- **Single-threaded took 45 secs**



- **ForkJoin took 26 secs - why not 8x faster?**



Javaspecialists.eu

JavaSpecialists.EU

Using Our Own ForkJoinPools

- **If a ForkJoinTask is forked by a non-FJPool thread, it uses ForkJoinPool.commonPool()**
- **We can use our own pool by launching the code inside it**

```
ForkJoinPool pool = new ForkJoinPool(32);
BigInteger val = pool.submit(() -> fib.f(10_000_000)).get();
```

  - **This trick also works for parallel streams**

Javaspecialists.eu

JavaSpecialists.EU

ManagedBlocker

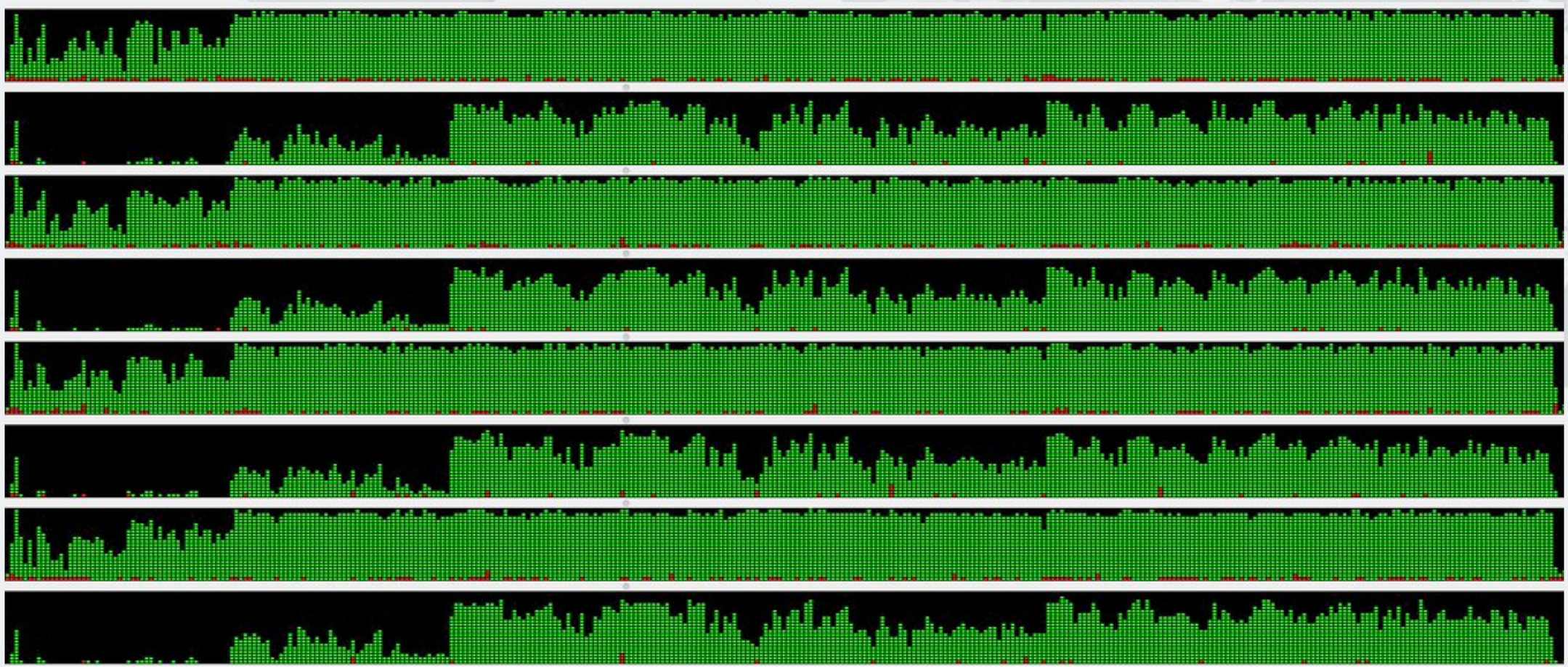
- **ForkJoinPool is configured with *desired parallelism***
  - **Number of active threads**
  - **ForkJoinPool mostly used with CPU intensive tasks**
- **If one of the FJ Threads has to block, a new thread can be started to take its place**
  - **This is done with the ManagedBlocker**
  - **See http://www.javaspecialists.eu/archive/Issue223.html**

Javaspecialists.eu

JavaSpecialists.EU

Fibonacci(1_000_000_000)

- **Parallel BigInteger.multiply(), reserved caching scheme**
  - **Notice low utilization at beginning of run**

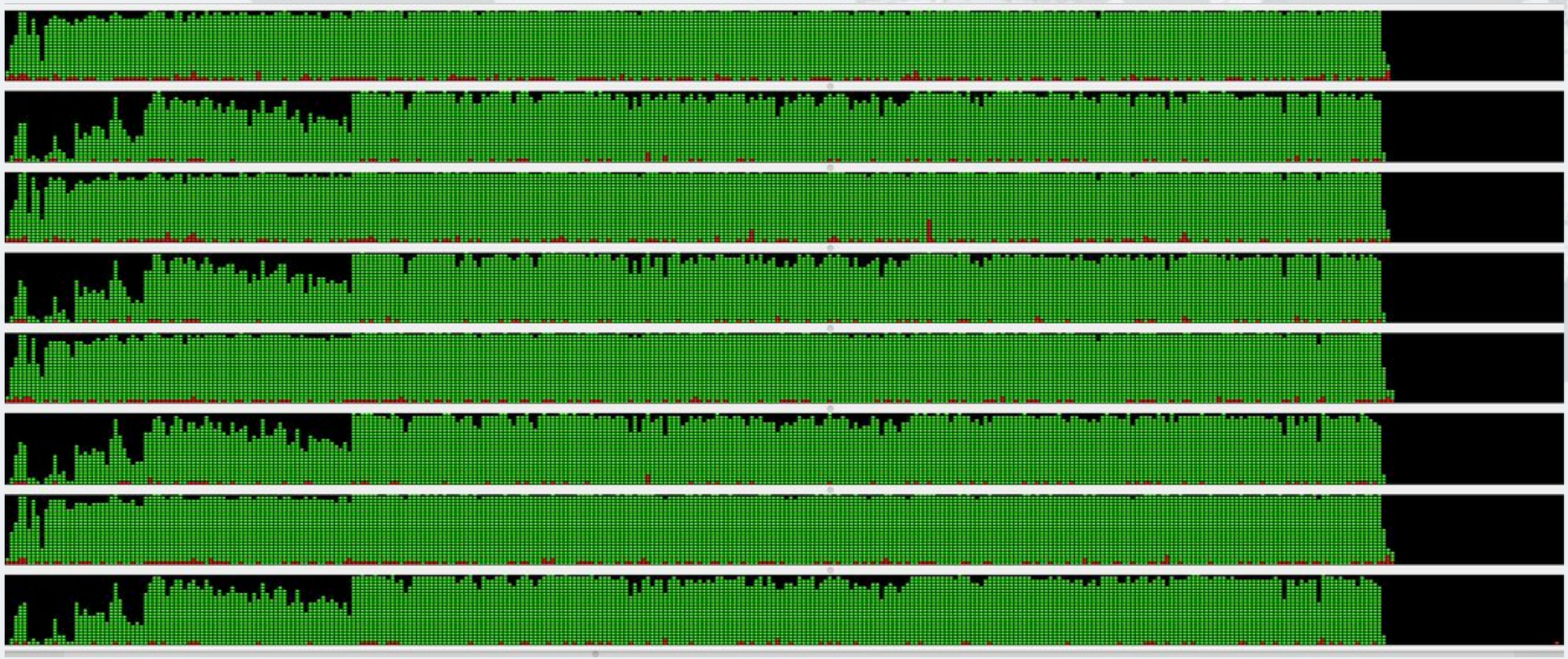


Javaspecialists.eu

JavaSpecialists.EU

With ManagedBlocker

- **Utilization of cores is higher at beginning of run**
  - **Completes 8% faster**



Javaspecialists.eu

JavaSpecialists.EU

Canceling A Task

- Fork/Join uses futures to do their work
- When we want to give up on a task, we call cancel()
  - Interrupting the thread won't work, as we do not know which thread will be executing our task
- Inside the compute() method
  - Must regularly call isCancelled() in long-running tasks
  - Would then typically throw a CancellationException
- Note: all the threads in the ForkJoinPool are daemons
  - Thus they might mask bugs in your code
  - Your threads might stay around forever

Javaspecialists.eu

JavaSpecialists.EU

Use Cases Of Fork/Join

- **Supports parallel streams in Java 8**
- **Arrays has a parallel methods using Fork/Join**
  - **Arrays.parallelPrefix()**
    - **Cumulates, in parallel, each element of the given array in place, using the supplied function**
  - **Arrays.parallelSetAll()**
    - **Set all elements of the specified array, in parallel, using the provided generator function to compute each element.**
  - **Arrays.parallelSort()**
    - **Uses a parallel merge sort**
    - **Only works with *more* than 2 cores, otherwise sequential sort**

Javaspecialists.eu

JavaSpecialists.EU

7: Exercises

Fork/Join

JavaSpecialists.EU

Exercise 7.1: Puzzle Solver With Fork/Join

- **Your job is to use fork/join to solve this puzzle**
  - **Instruments must match**
  - **We already have a plain recursive solution that works**
  - **You can build on that or write your own**

Javaspecialists.eu

JavaSpecialists.EU

Exercise 7.2: Streams Instead Of Fork/Join

- **The NumberSearcher is looking for longs that cannot be represented without loss of precision in a double. It creates a bunch of random numbers and then uses Fork/Join to count the values.**
- **Instead of using Fork/Join, use parallel streams**
  - **Besides making the code easier to understand and shorter, your performance should also improve a bit**

Javaspecialists.eu

JavaSpecialists.EU

7: Fork/Join

Questions?

8: Avoiding Liveness Hazards

Safety first!

JavaSpecialists.EU

8: Avoiding Liveness Hazards

- **Fixing safety problems can cause liveness problems**
  - **Don't indiscriminately sprinkle "synchronized" into your code**
- **Liveness hazards can happen through**
  - **Lock-ordering deadlocks**
    - **Typically when you lock two locks in different orders**
    - **Requires global analysis to make sure your order is consistent**
      - **Lesson: only ever hold a single lock per thread!**
  - **Resource deadlocks**
    - **e.g. due to bounded queue blocking threads**
- **Thread deadlocked in BLOCKED state can never recover**

Javaspecialists.eu

JavaSpecialists.EU

8.1: Deadlock

Avoiding Liveness Hazards

JavaSpecialists.EU

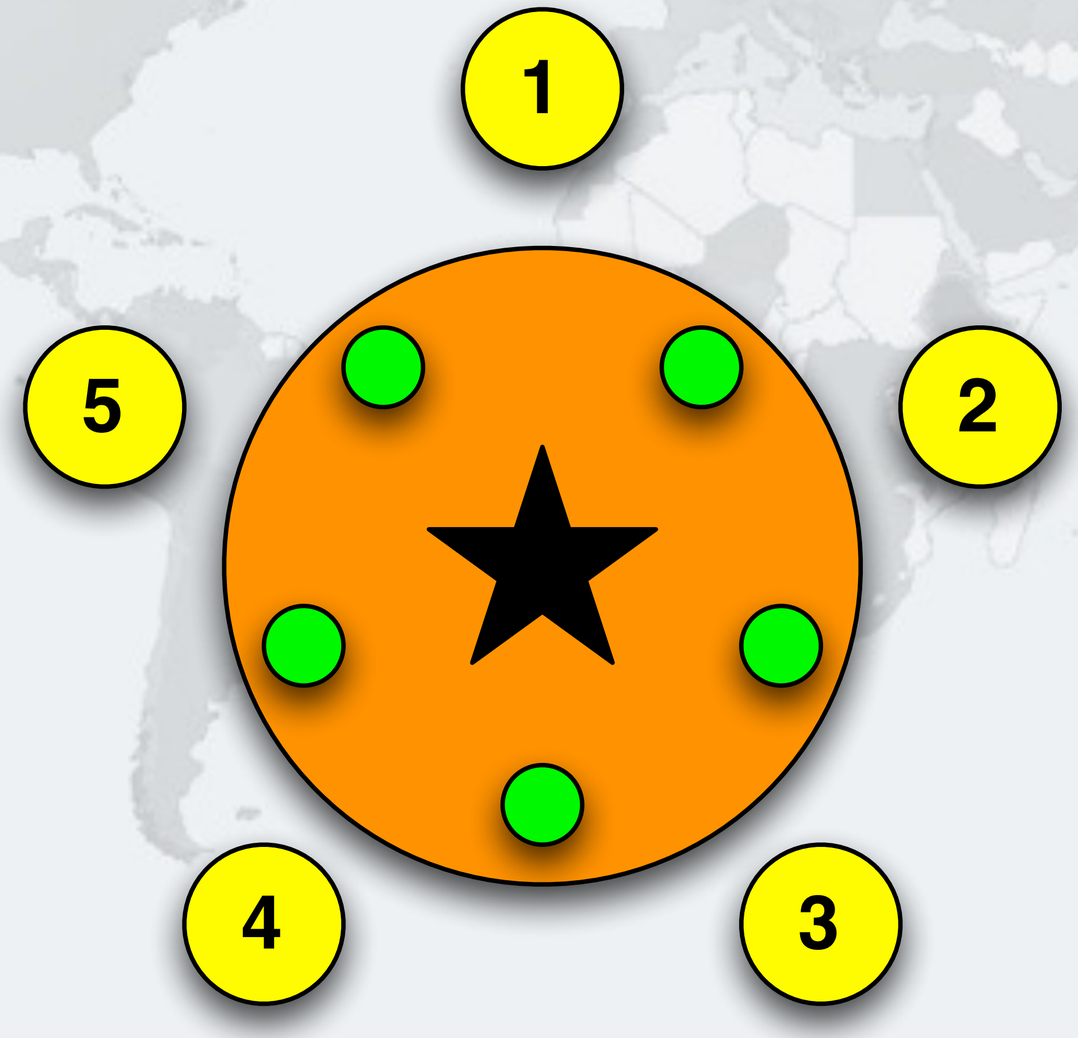
8.1: Deadlock

- **Classic problem is that of the "dining philosophers"**
  - **We changed that to the "drinking philosophers"**
    - **That is where the word "symposium" comes from**
      - **sym - together, such as "symphony"**
      - **poto - drink**
    - **Ancient Greek philosophers used to get together to drink & think**
- **In our example, a philosopher needs two glasses to drink**
  - **First he takes the right one, then the left one**
  - **When he finishes drinking, he returns them and carries on thinking**

Javaspecialists.eu

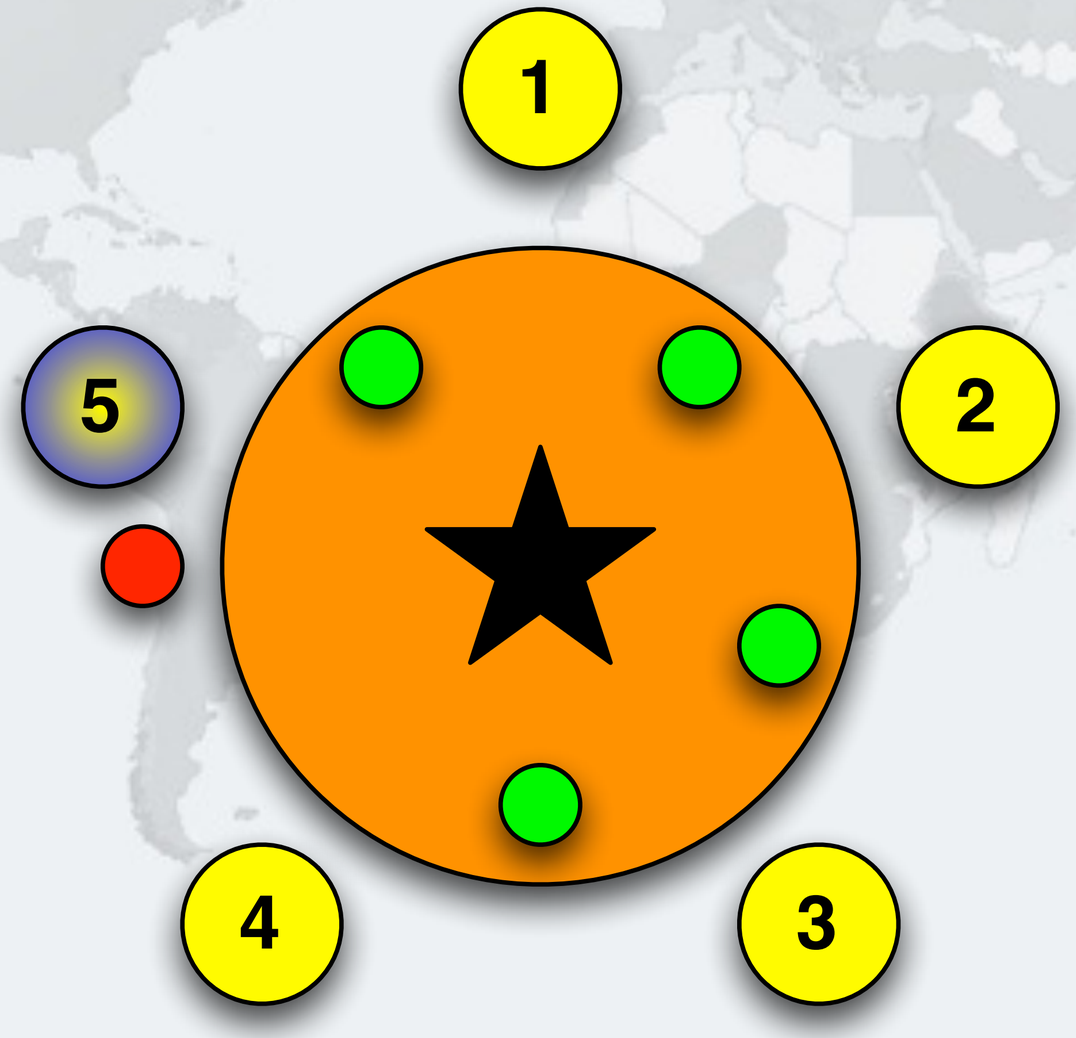
JavaSpecialists.EU

Table Is Ready, All Philosophers Are Thinking



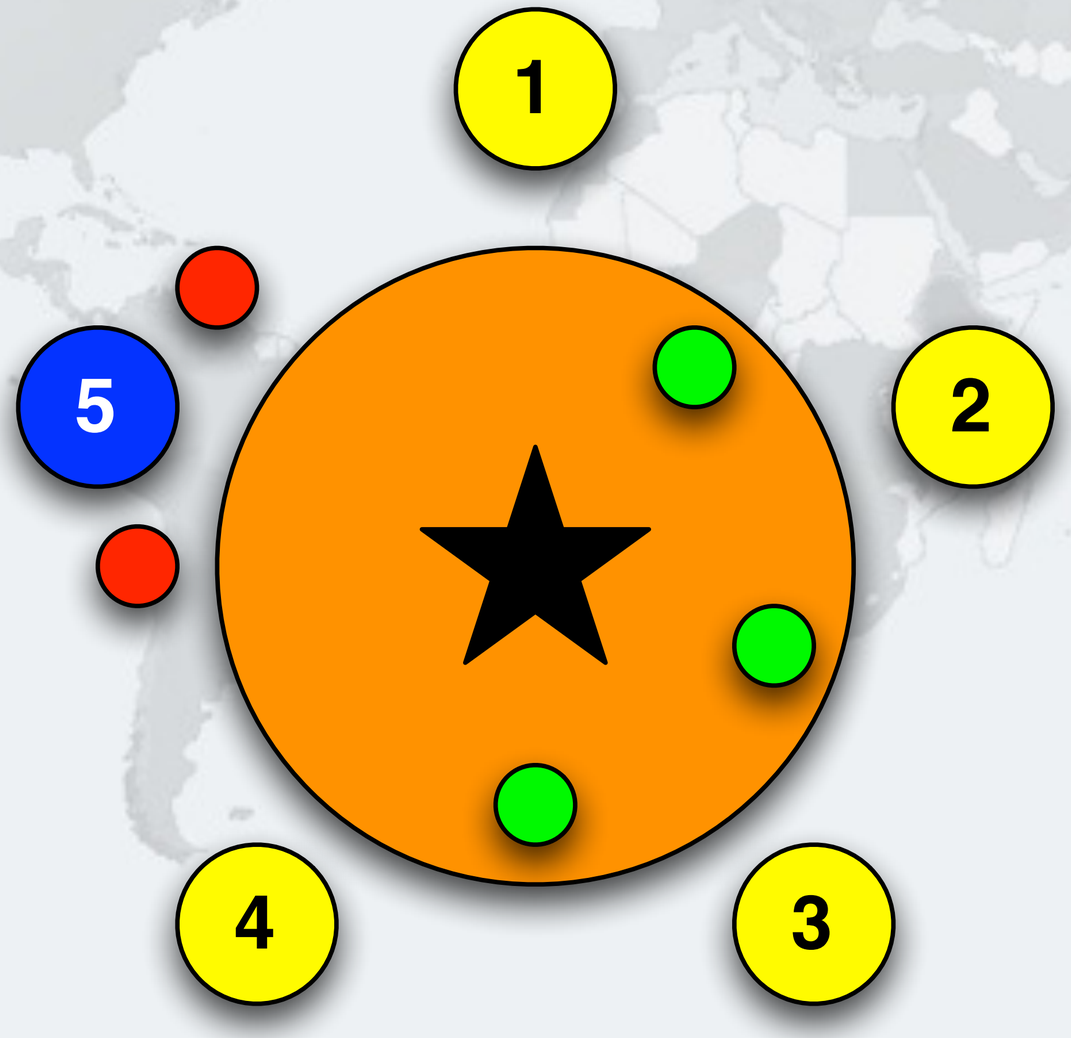
Javaspecialists.eu
JavaSpecialists.EU

Philosopher 5 Wants To Drink, Takes Right Cup



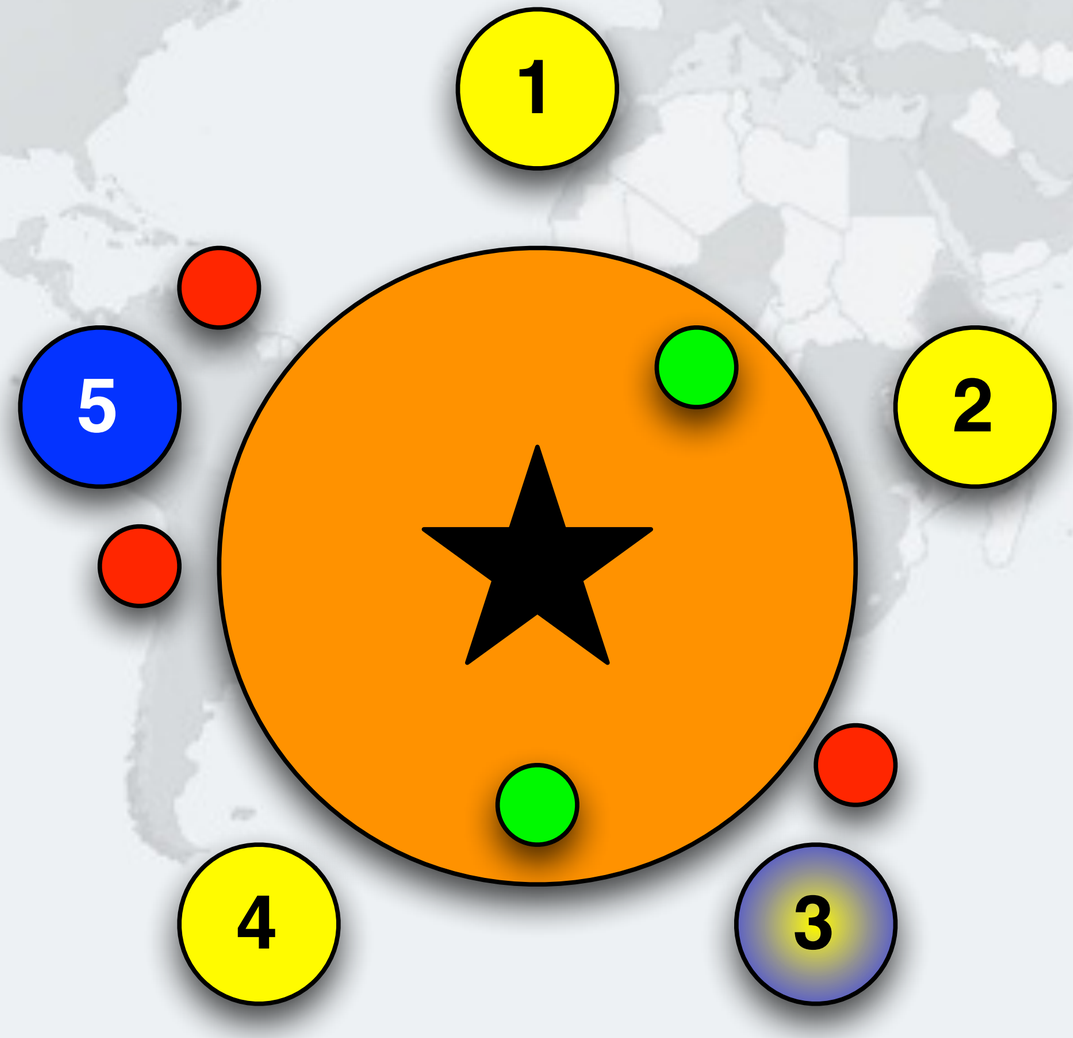
Javaspecialists.eu
JavaSpecialists.EU

Philosopher 5 Is Now Drinking With Both Cups



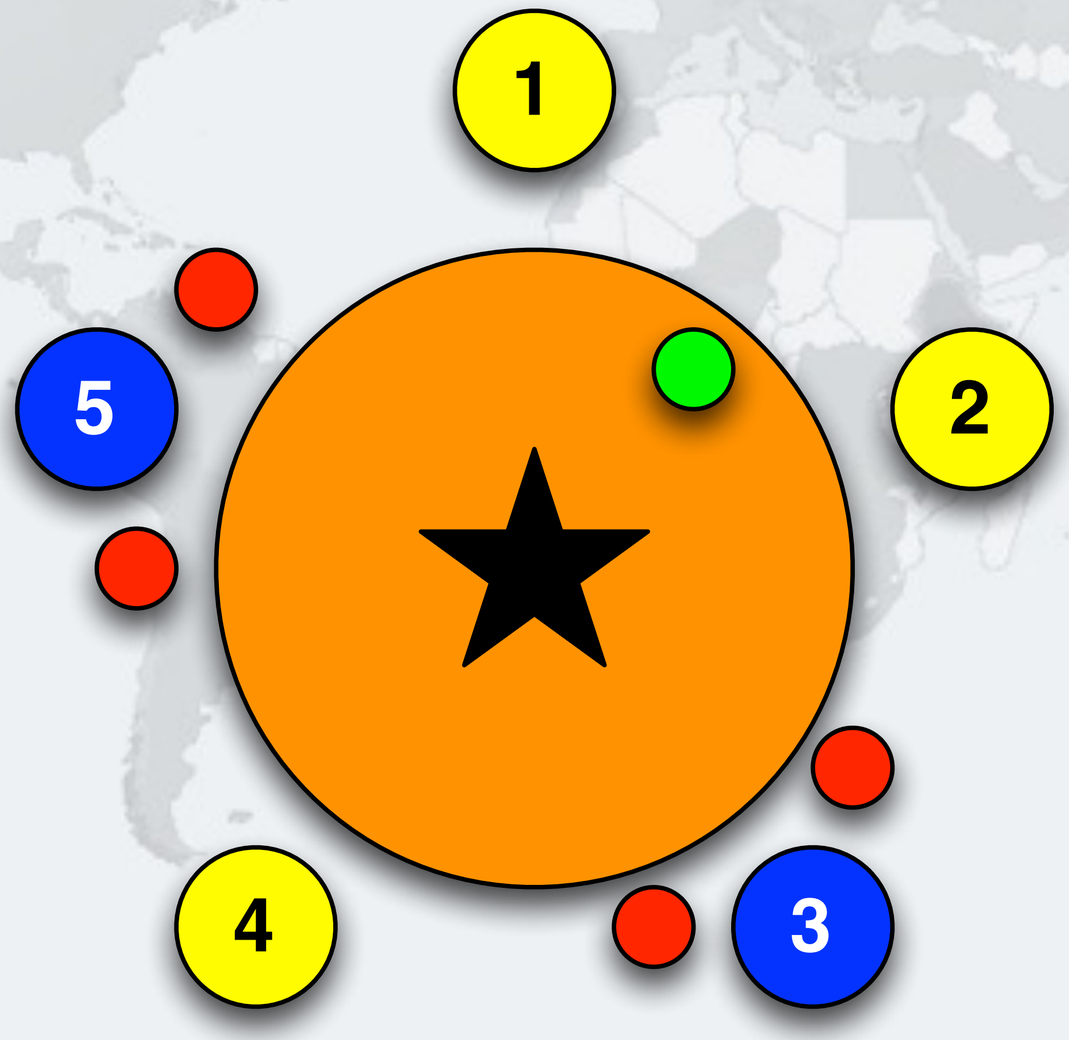
Javaspecialists.eu
JavaSpecialists.EU

Philosopher 3 Wants To Drink, Takes Right Cup



Javaspecialists.eu
JavaSpecialists.EU

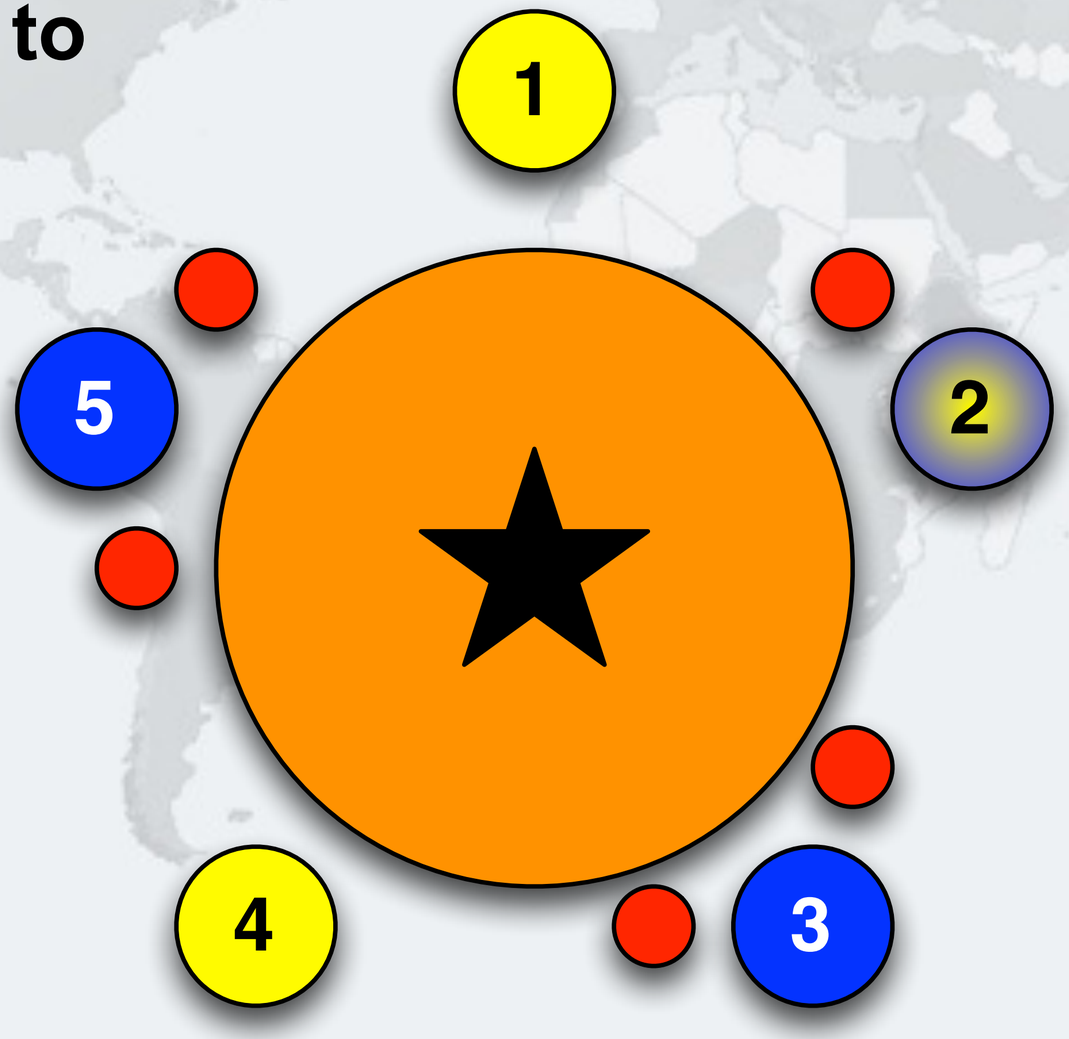
Philosopher 3 Is Now Drinking With Both Cups



Javaspecialists.eu
JavaSpecialists.EU

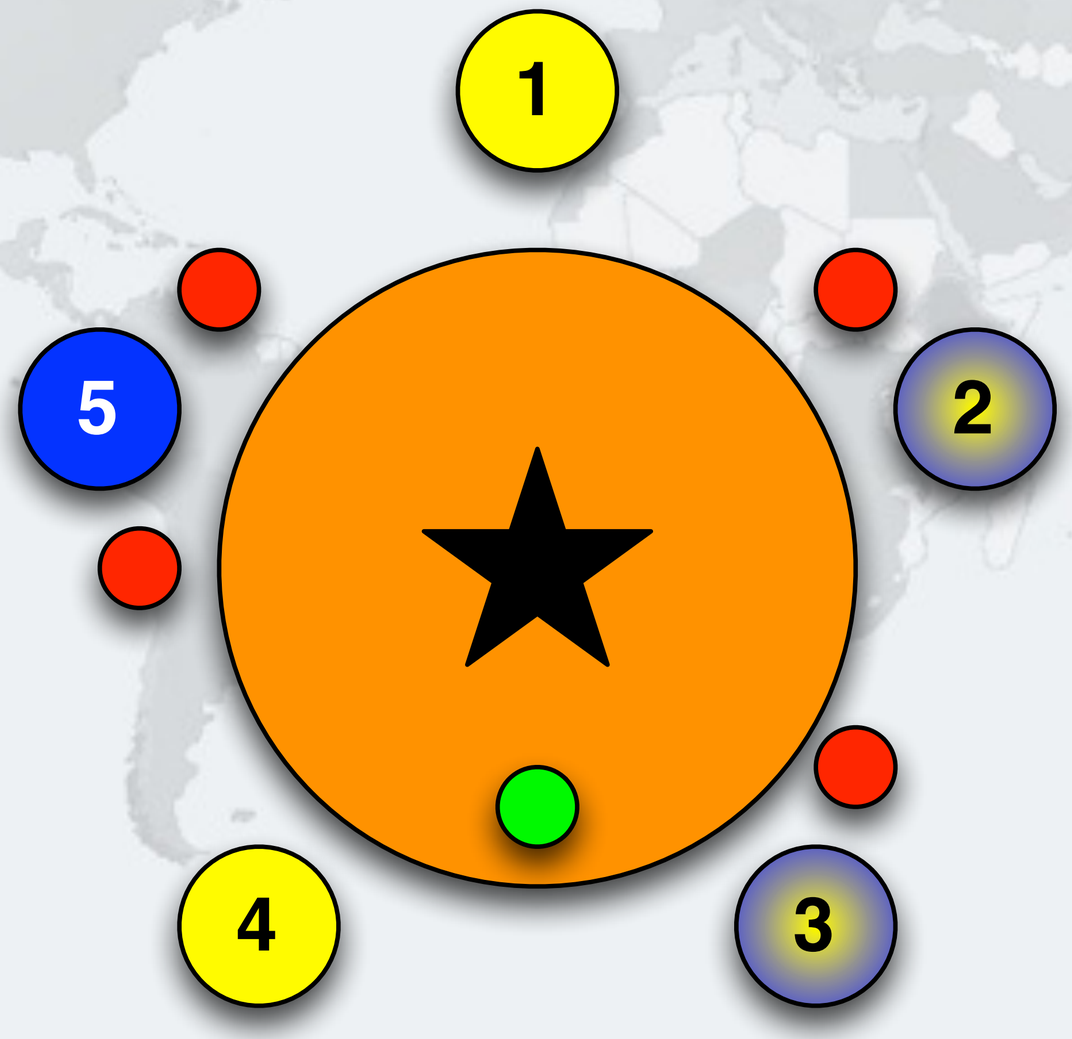
Philosopher 2 Wants To Drink, Takes Right Cup

- **But he has to wait for Philosopher 3 to finish his drinking session**



Javaspecialists.eu
JavaSpecialists.EU

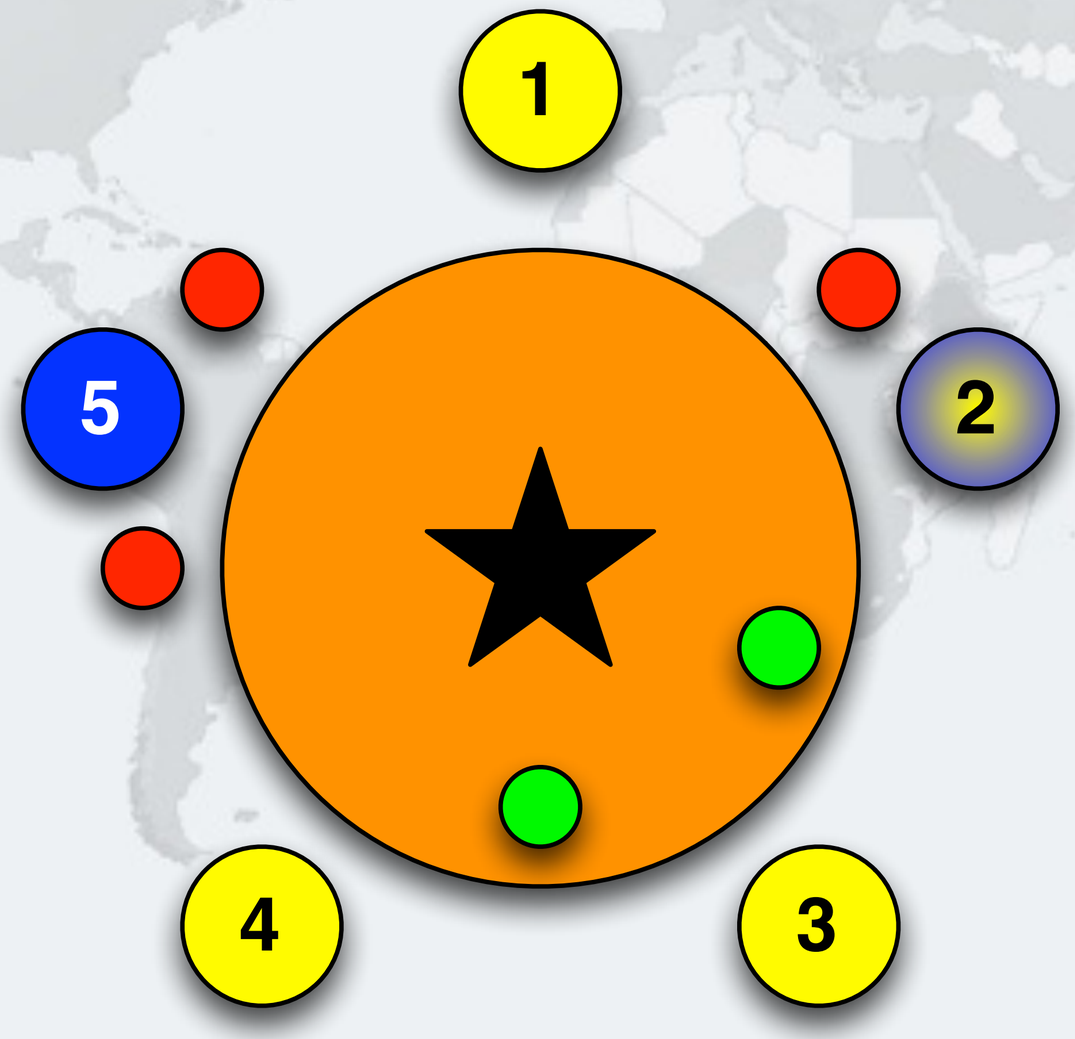
Philosopher 3 Finished Drink, Returns Left Cup



Javaspecialists.eu

JavaSpecialists.EU

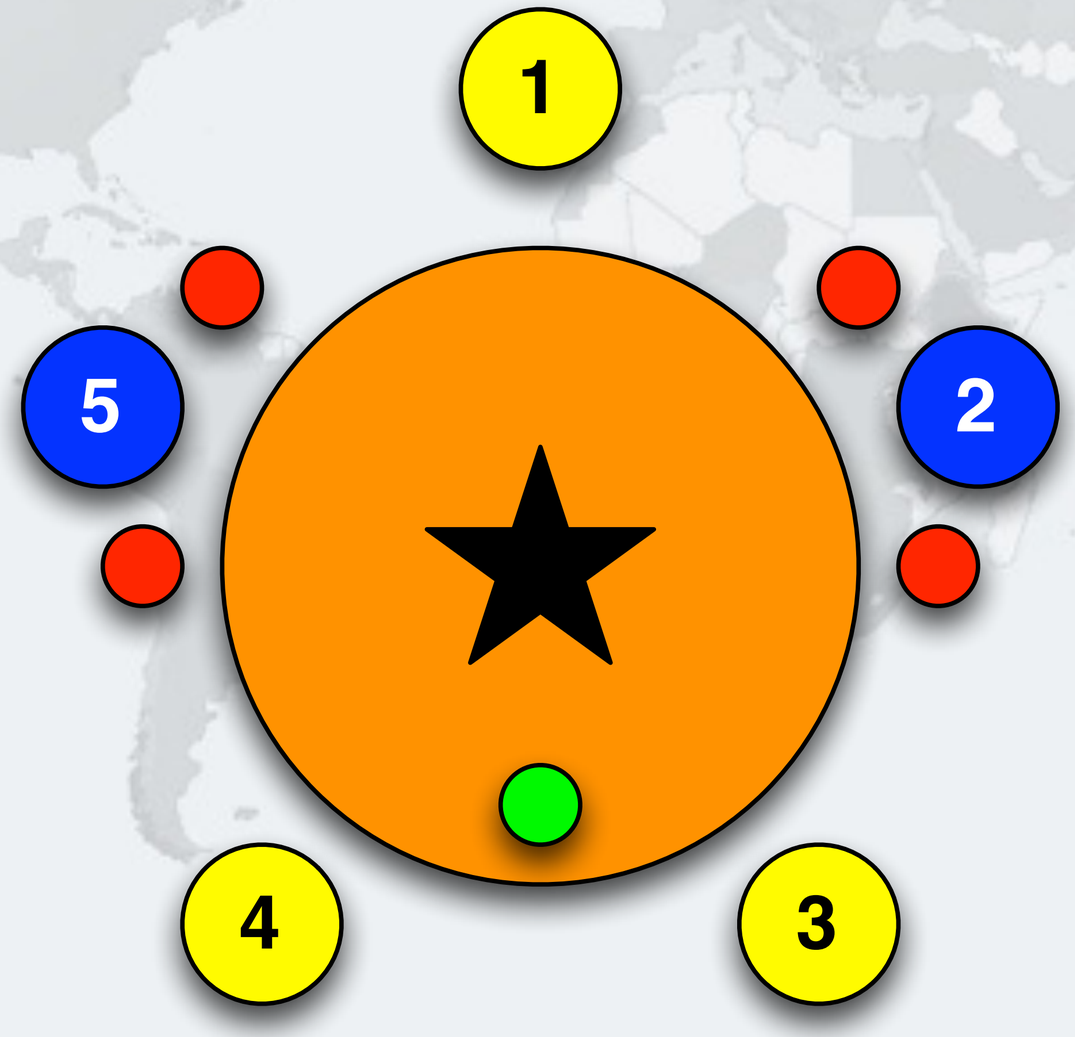
Philosopher 3 Returns Right Cup



Javaspecialists.eu

JavaSpecialists.EU

Philosopher 2 Is Now Drinking With Both Cups



Javaspecialists.eu

JavaSpecialists.EU

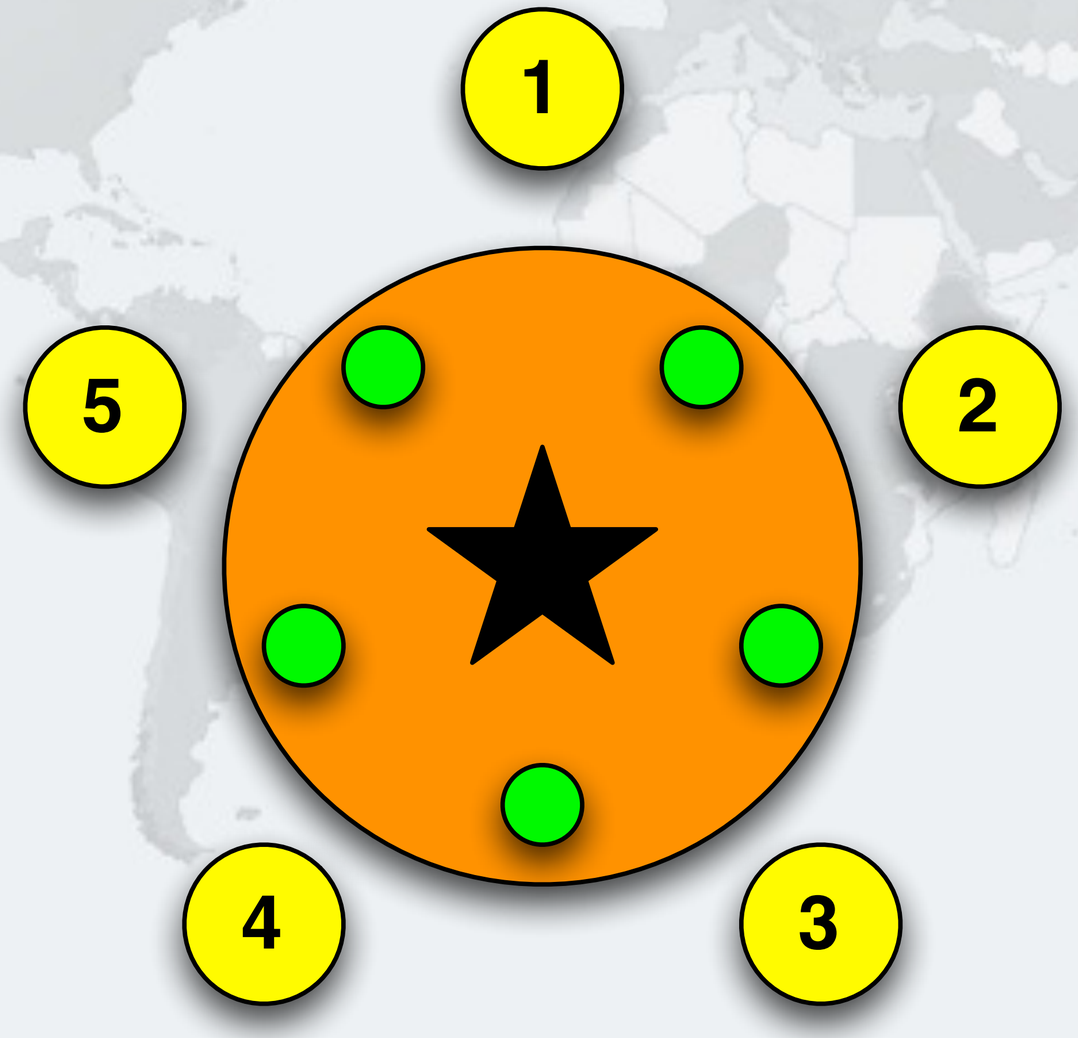
Drinking Philosophers In Limbo

- **The standard rule is that every philosopher first picks up the right cup, then the left**
  - **If all of the philosophers want to drink and they all pick up the right cup, then they all are holding one cup but cannot get the left cup**

Javaspecialists.eu

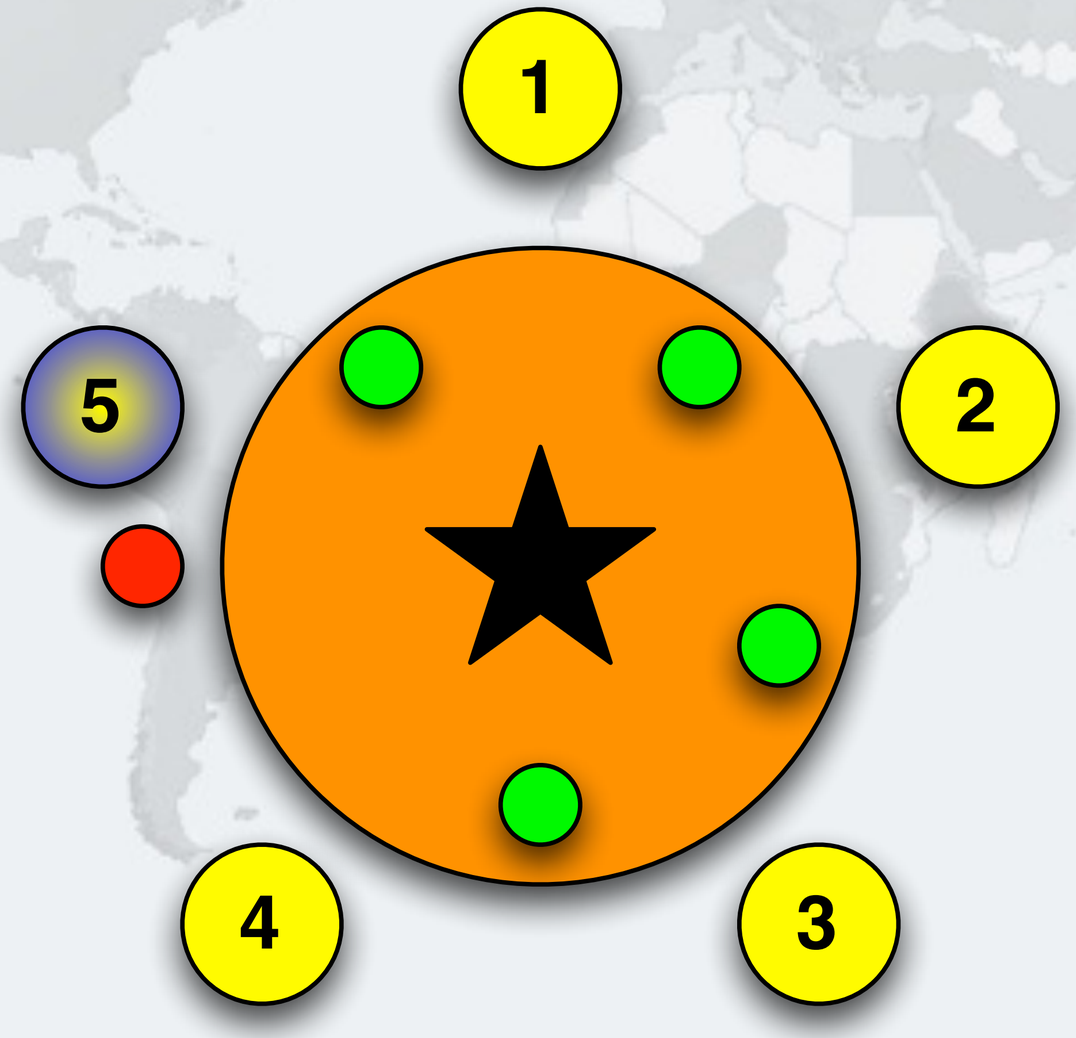
JavaSpecialists.EU

A Deadlock Can Easily Happen



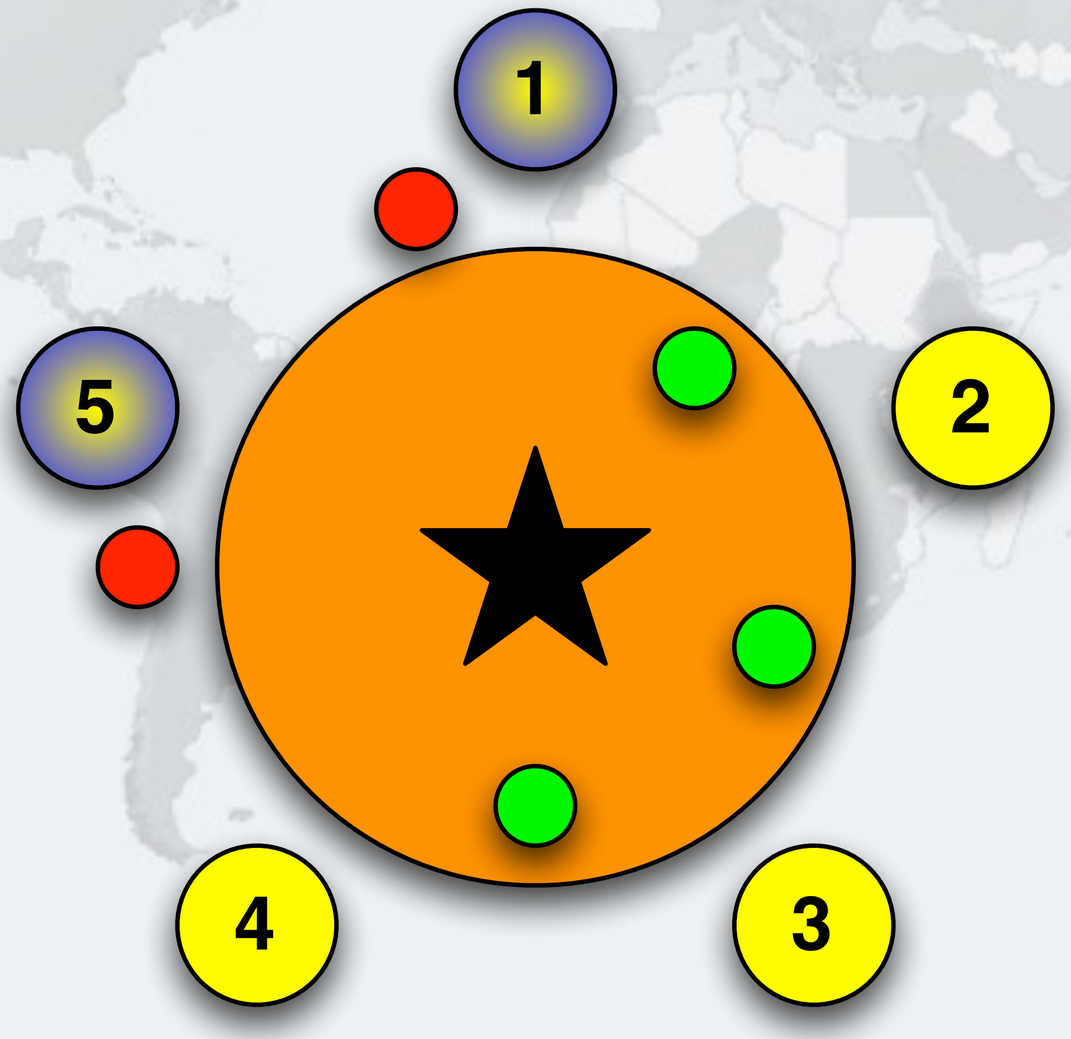
Javaspecialists.eu
JavaSpecialists.EU

Philosopher 5 Wants To Drink, Takes Right Cup



Javaspecialists.eu
JavaSpecialists.EU

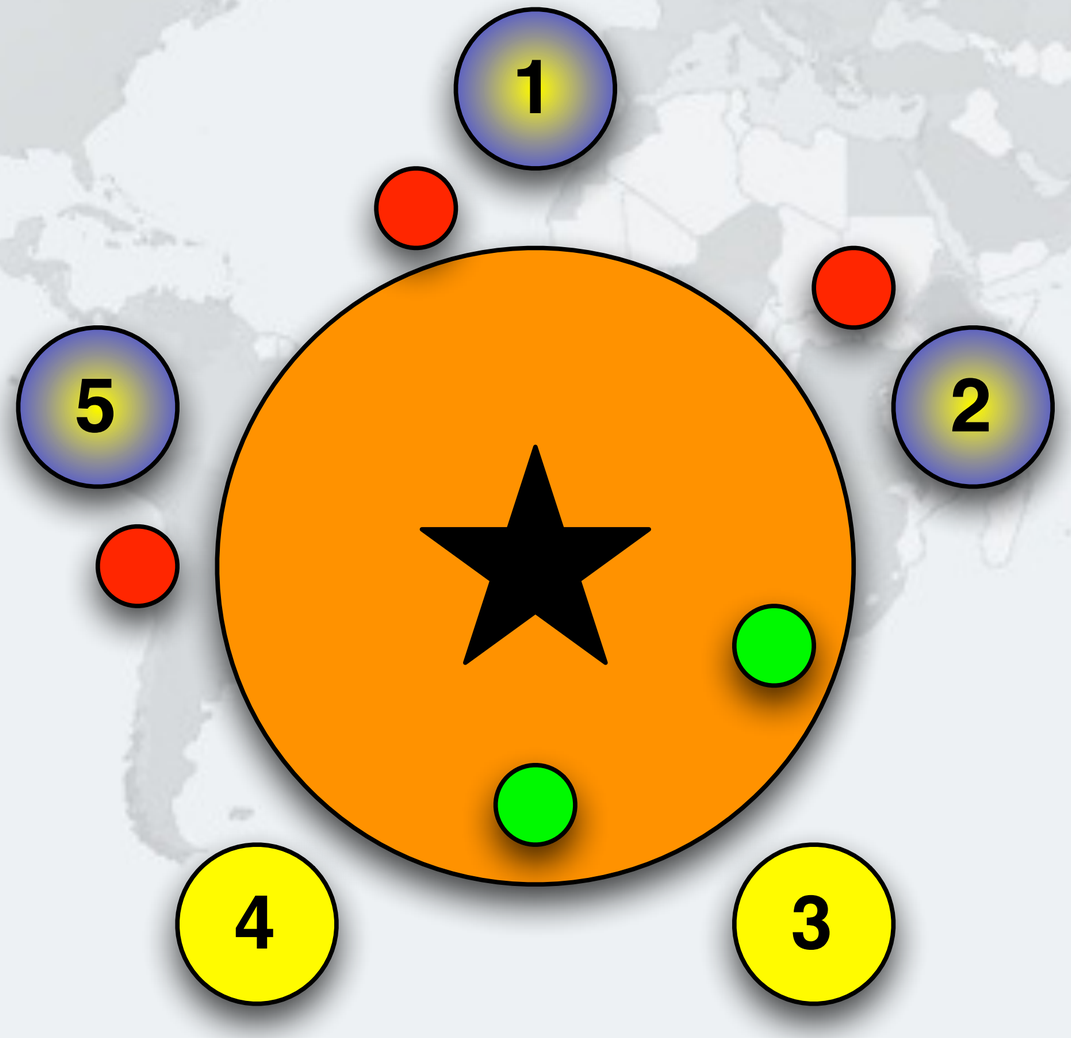
Philosopher 1 Wants To Drink, Takes Right Cup



Javaspecialists.eu

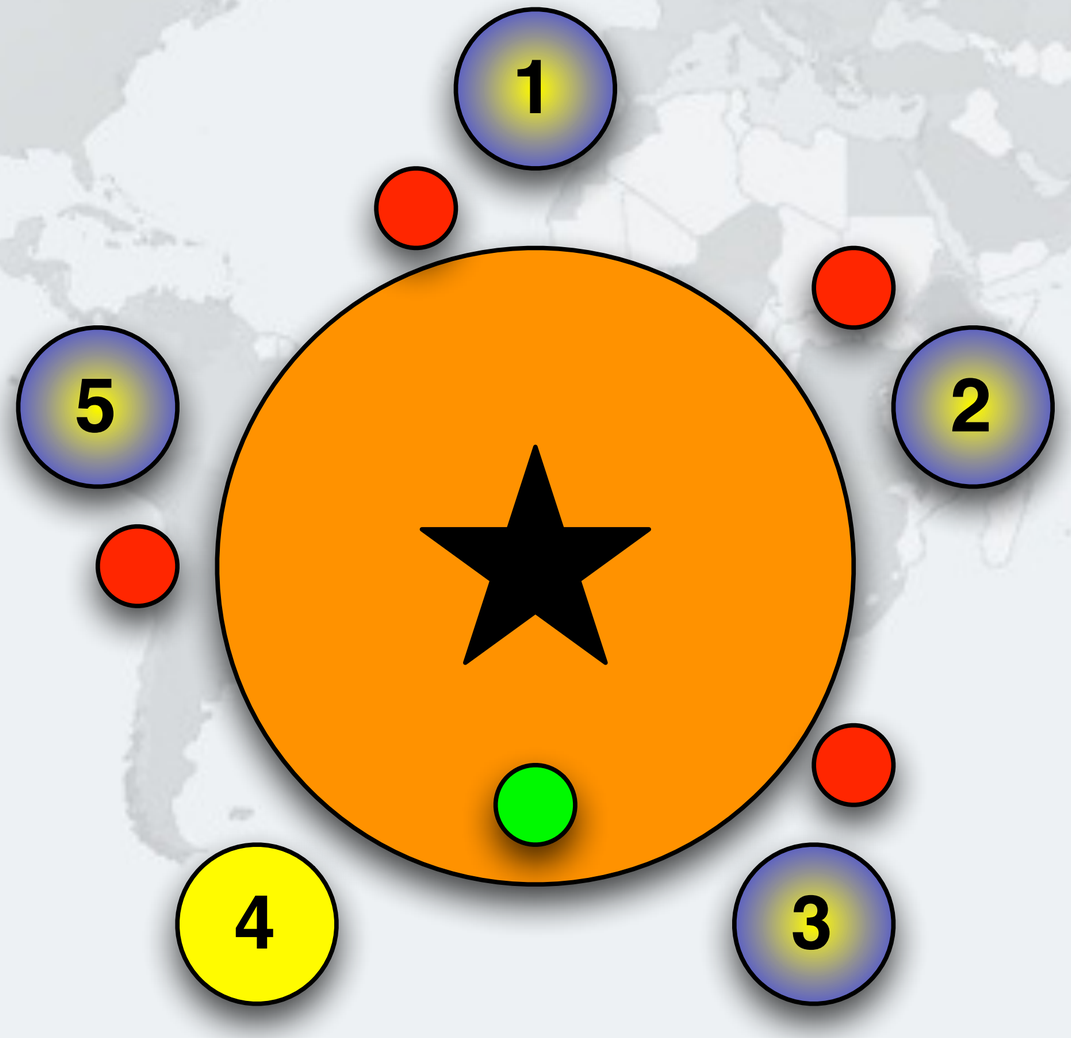
JavaSpecialists.EU

Philosopher 2 Wants To Drink, Takes Right Cup



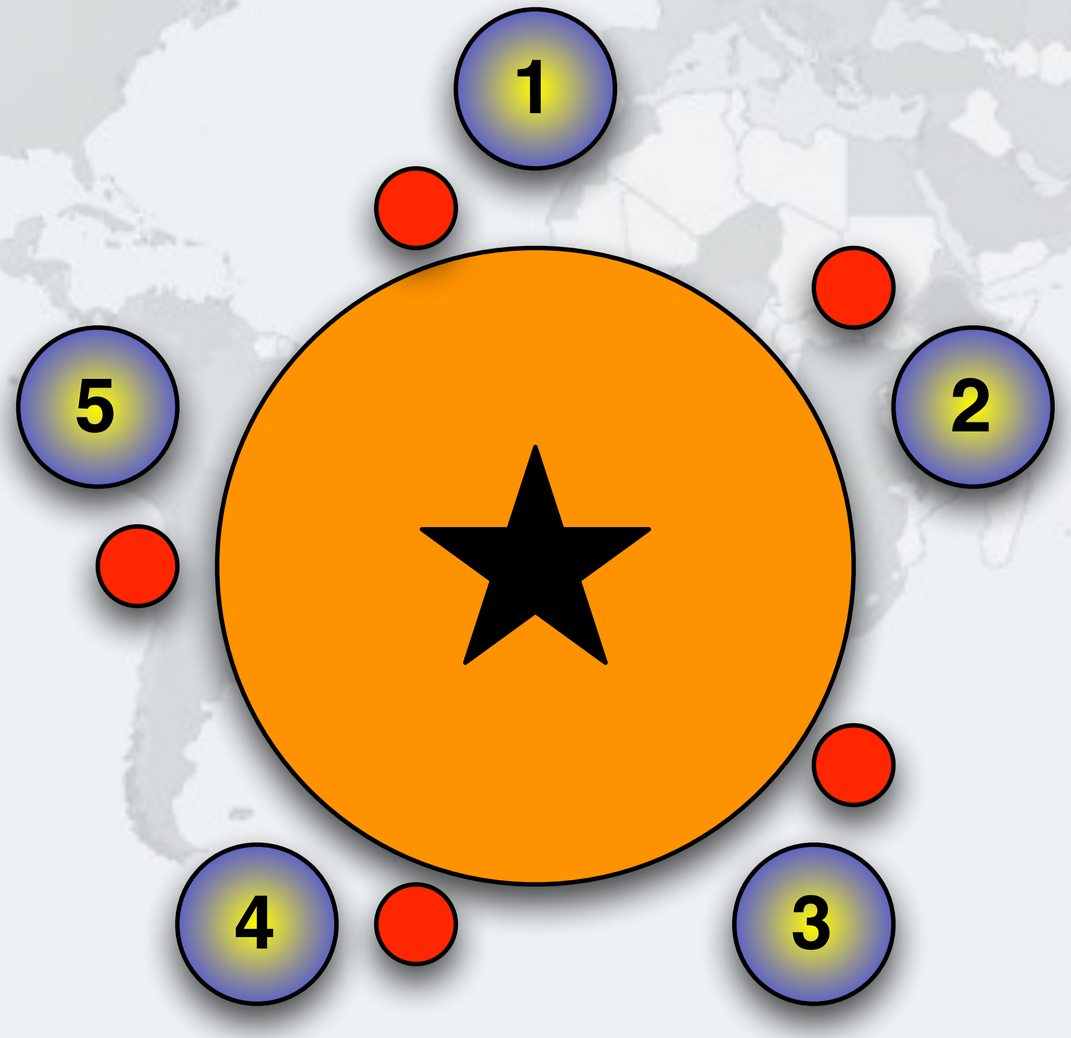
Javaspecialists.eu
JavaSpecialists.EU

Philosopher 3 Wants To Drink, Takes Right Cup



Javaspecialists.eu
JavaSpecialists.EU

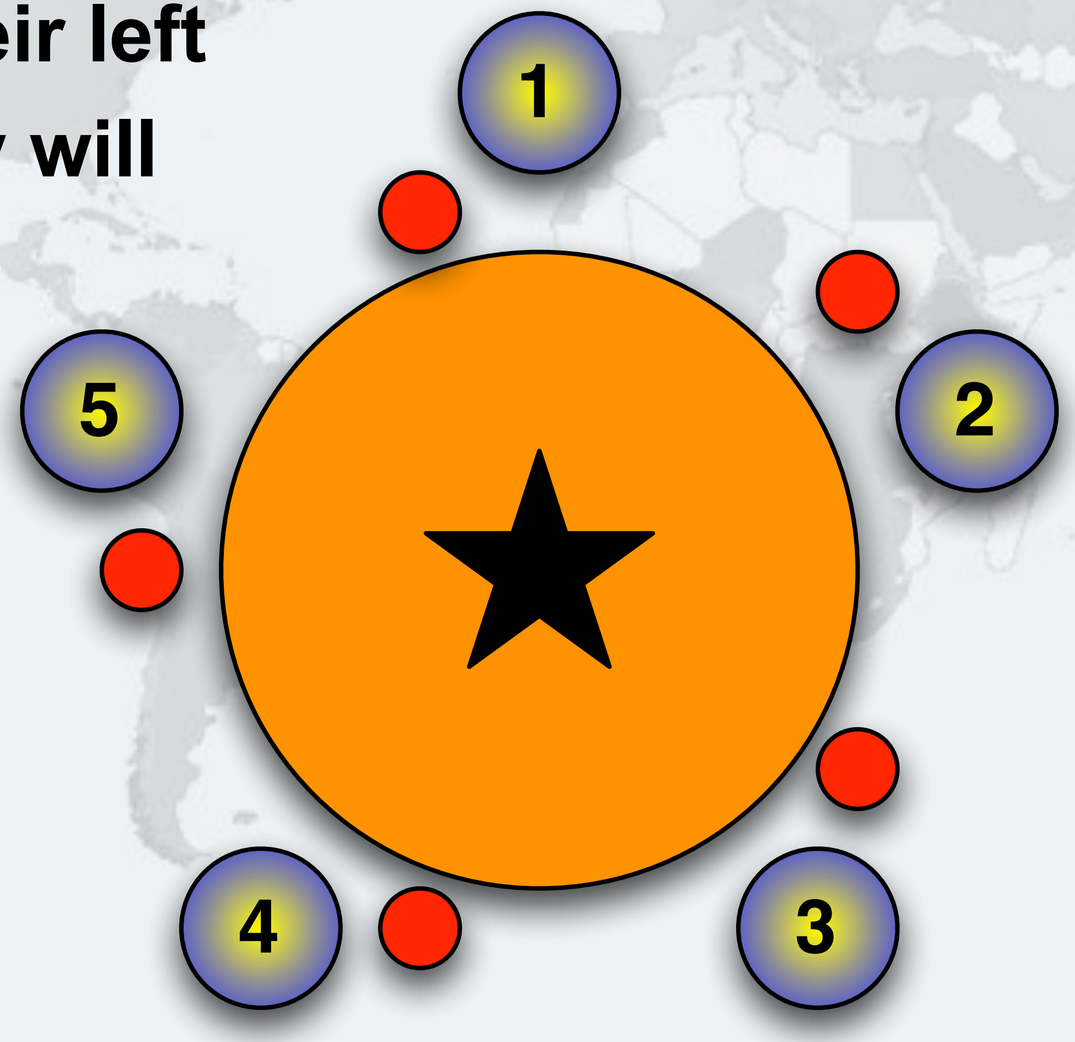
Philosopher 4 Wants To Drink, Takes Right Cup



Javaspecialists.eu
JavaSpecialists.EU

Deadlock!

- All the philosophers are waiting for their left cups, but they will never become available



Javaspecialists.eu

JavaSpecialists.EU

Resolving Deadlocks

- Deadlocks can be discovered automatically by searching the graph of call stacks, looking for circular dependencies
  - ThreadMXBean can find deadlocks for us, but cannot fix them
- In databases, the deadlock is resolved by one of the queries being aborted with an exception
  - The query could then be retried
- Java does not have this functionality
  - When we get a deadlock, there is no clean way to recover from it
  - Prevention is better than the cure

Javaspecialists.eu

JavaSpecialists.EU

How Do We Discover Deadlocks?

- **A lot of Java code contains subtle locking bugs**
  - **Calling methods in different orders could cause a *deadly embrace***
  - **Calling alien methods could cause a call-back**
  - **Limiting resources can cause deadlocks with dependent actions**
- **Most of the time, deadlocks do not manifest themselves**
  - **Usually never during testing**
  - **Seldom during production, only if the system is really busy**
    - **Often you will need to run the application for 5 days before it happens, usually on a Friday afternoon to ruin your weekend**

Javaspecialists.eu

JavaSpecialists.EU

Global Order With Boozing Philosophers

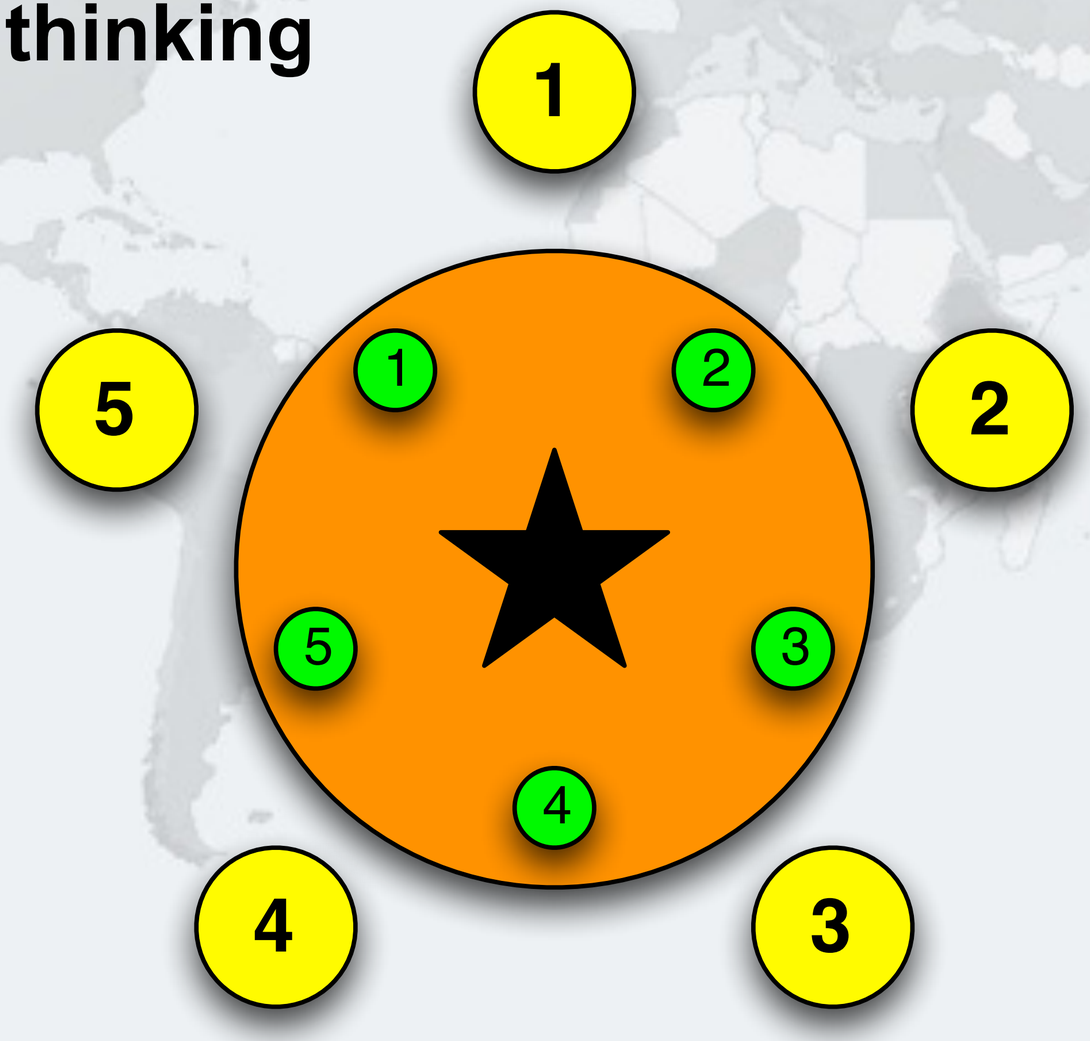
- **We can solve the deadlock with the "dining philosophers" by requiring that locks are always acquired in a set order**
  - **For example, we can make a rule that philosophers always first take the cup with the largest number**
  - **And return the cup with the lowest number first**

Javaspecialists.eu

JavaSpecialists.EU

Global Lock Ordering

- **We start with all the philosophers thinking**

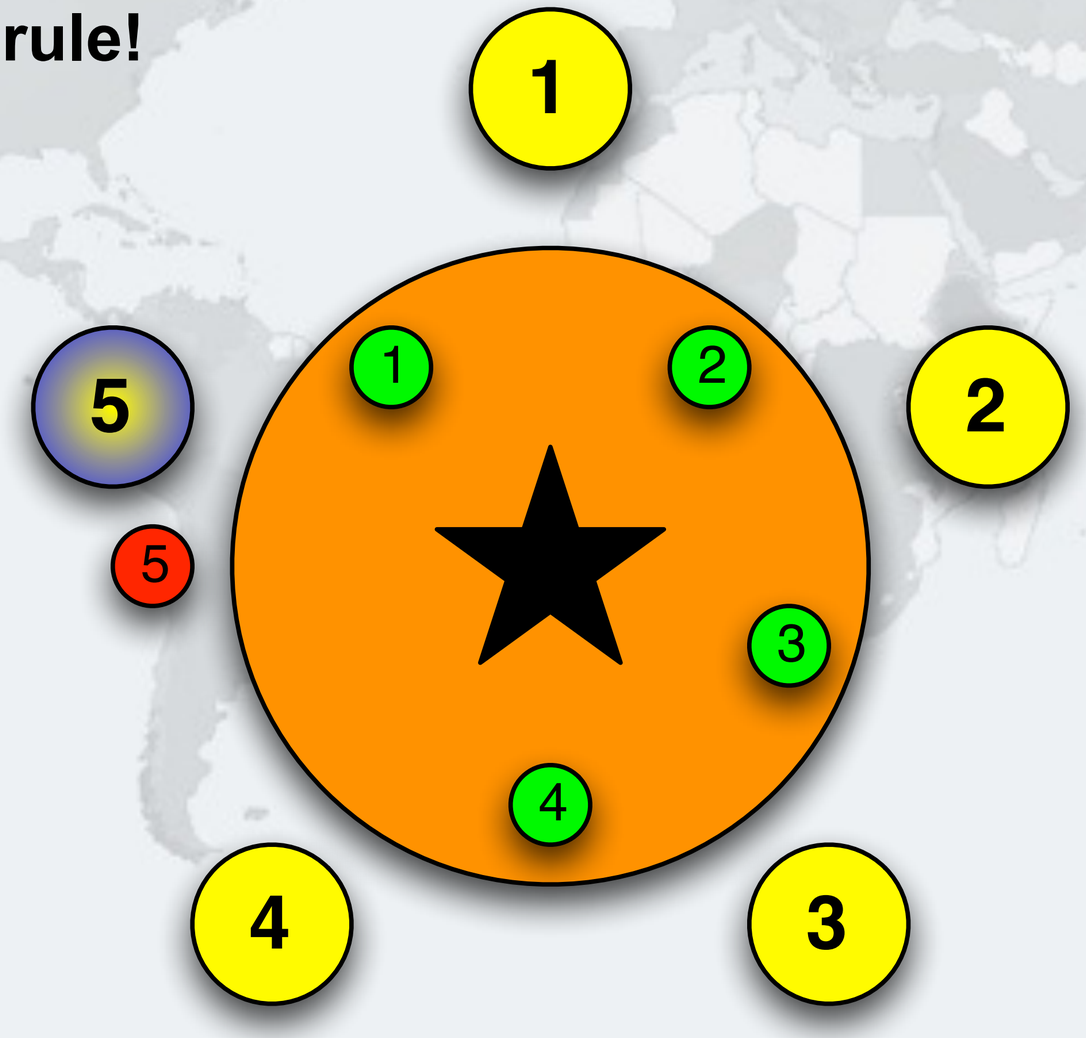


Javaspecialists.eu

JavaSpecialists.EU

Philosopher 5 Takes Cup 5

- **Cup 5 has higher number**
  - **Remember our rule!**

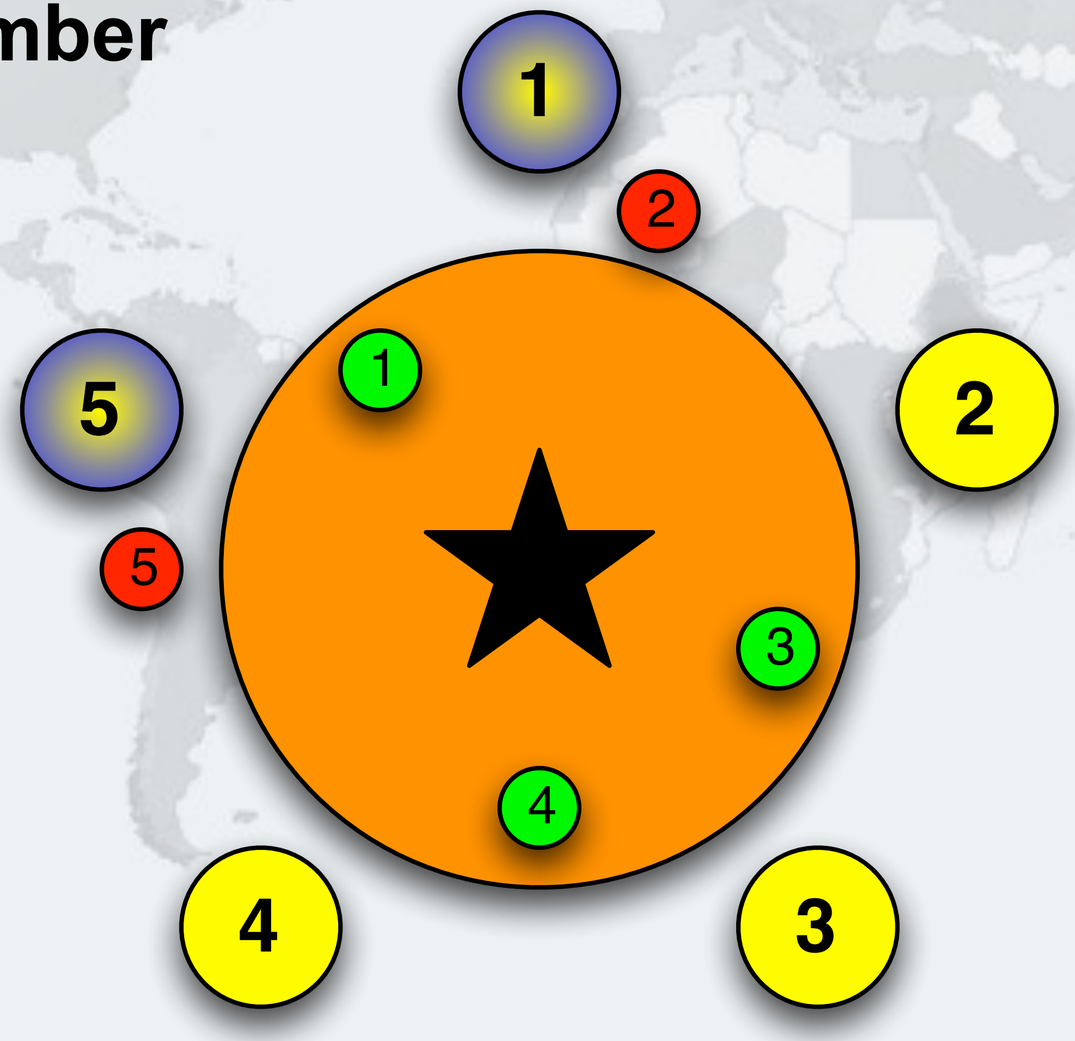


Javaspecialists.eu

JavaSpecialists.EU

Philosopher 1 Takes Cup 2

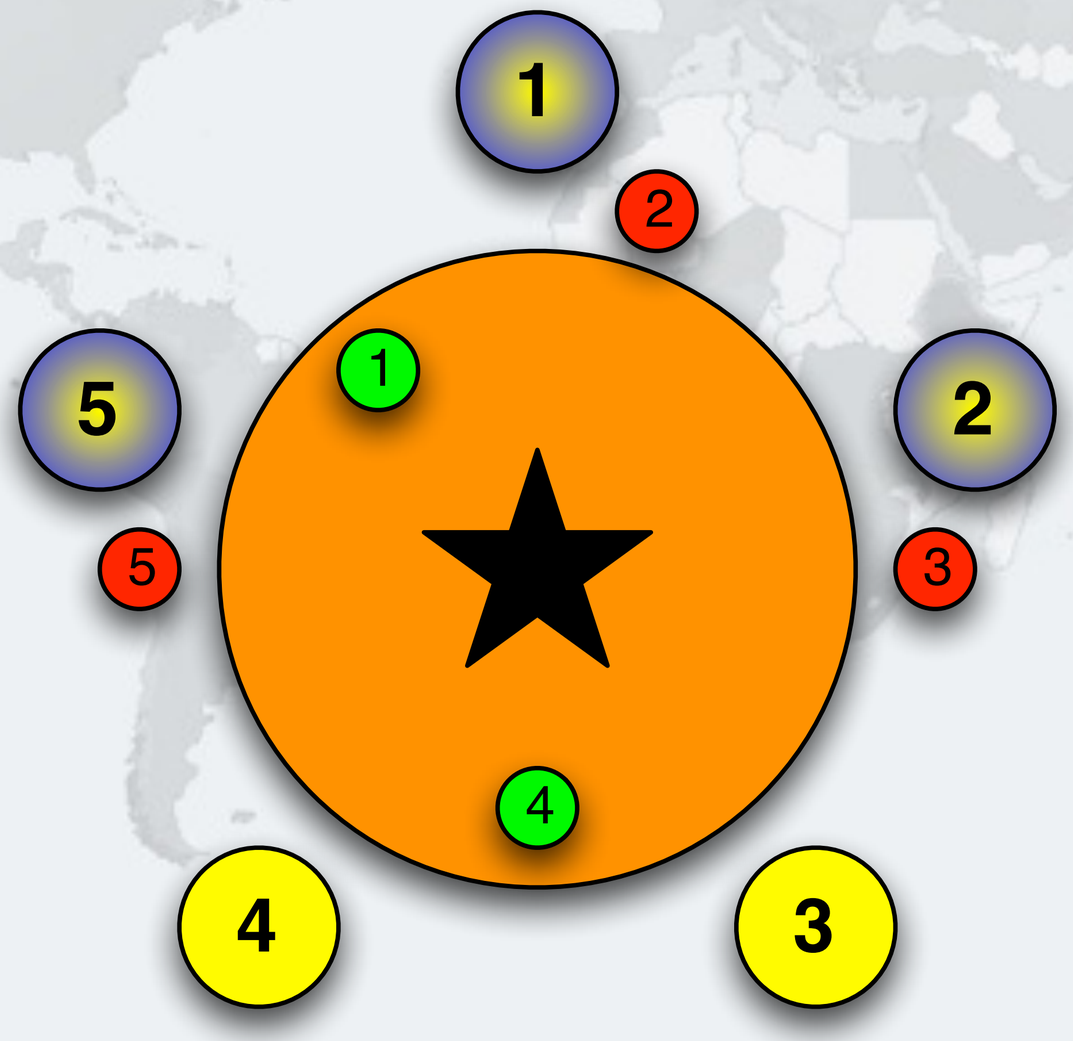
- **Must take the cup with the higher number first**
  - **In this case cup 2**



Javaspecialists.eu

JavaSpecialists.EU

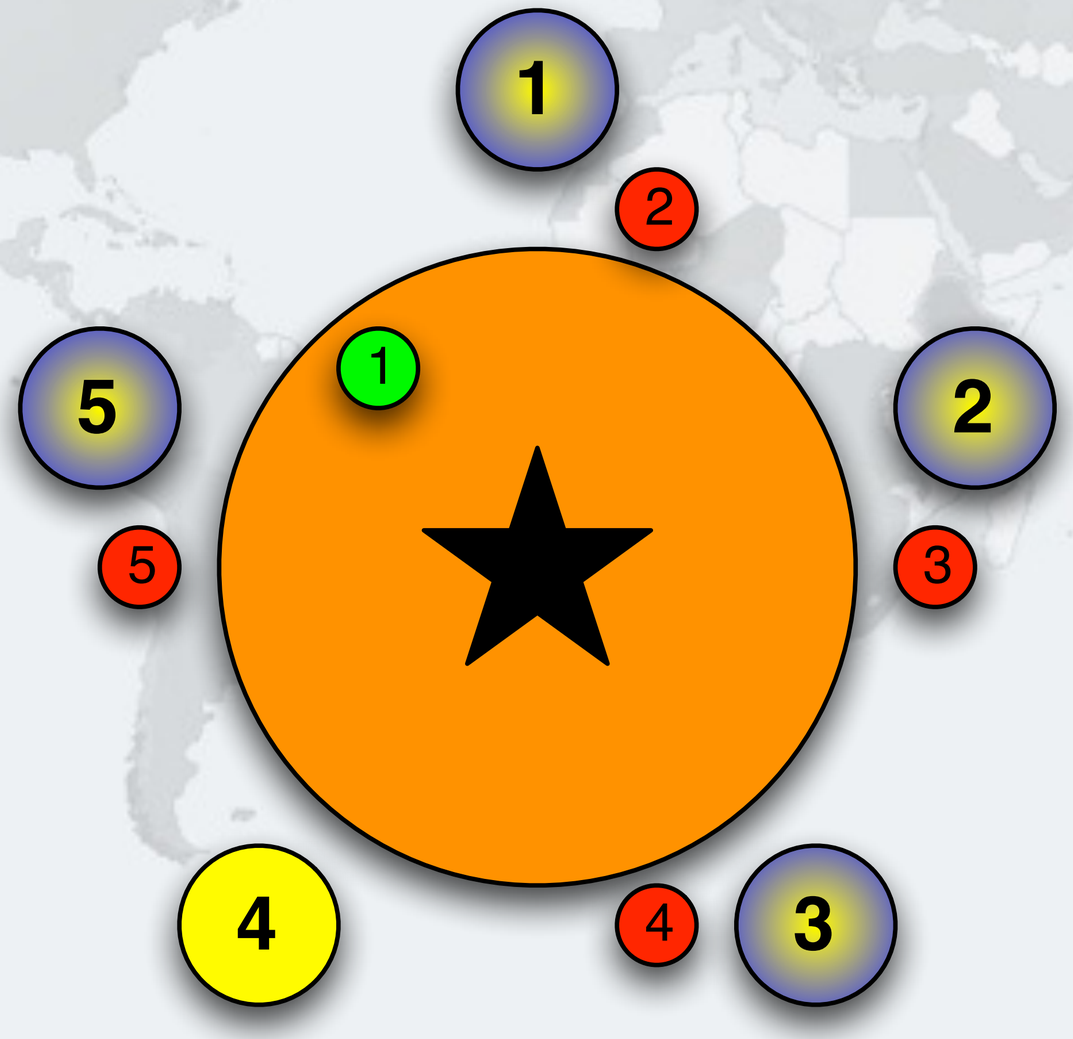
Philosopher 2 Takes Cup 3



Javaspecialists.eu

JavaSpecialists.EU

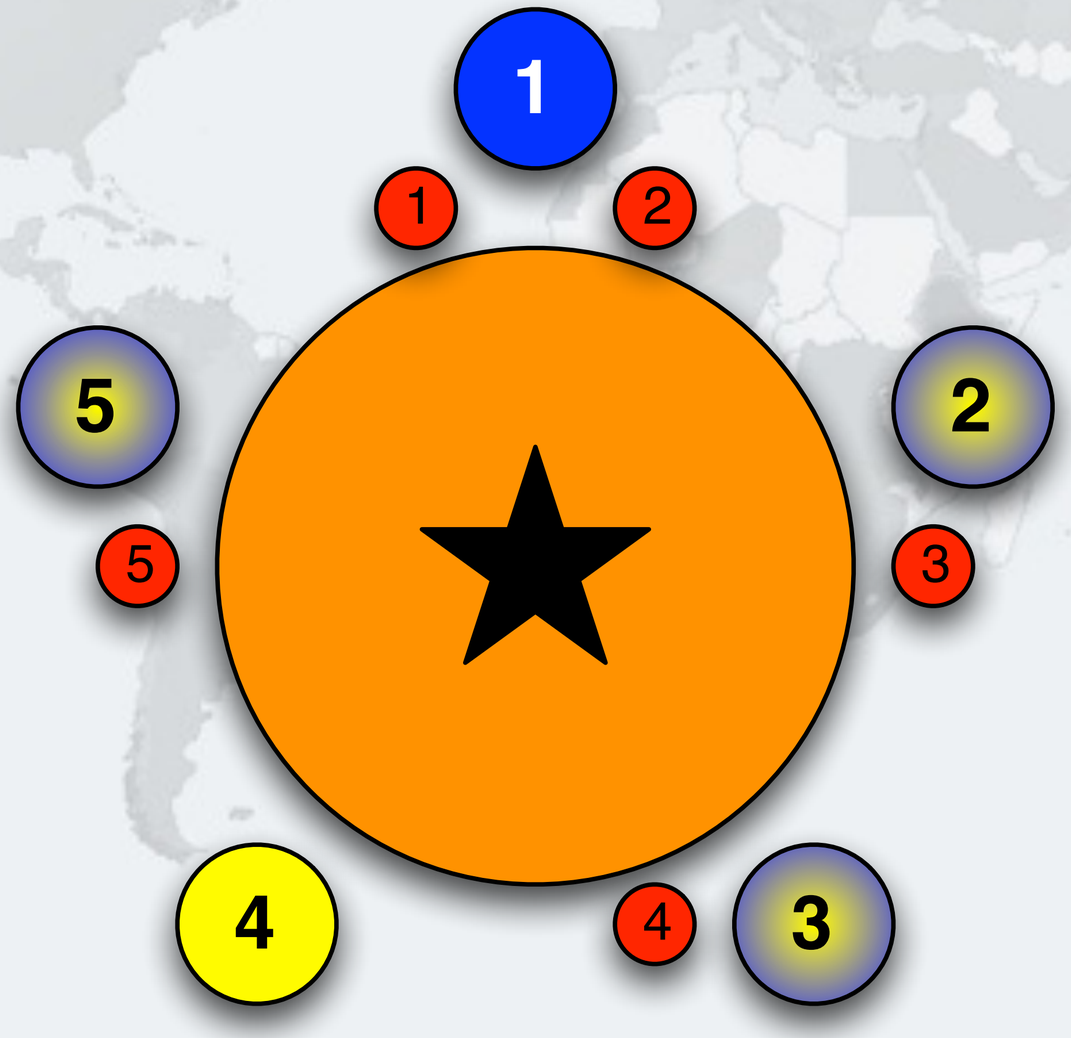
Philosopher 3 Takes Cup 4



Javaspecialists.eu

JavaSpecialists.EU

Philosopher 1 Takes Cup 1 - Drinking

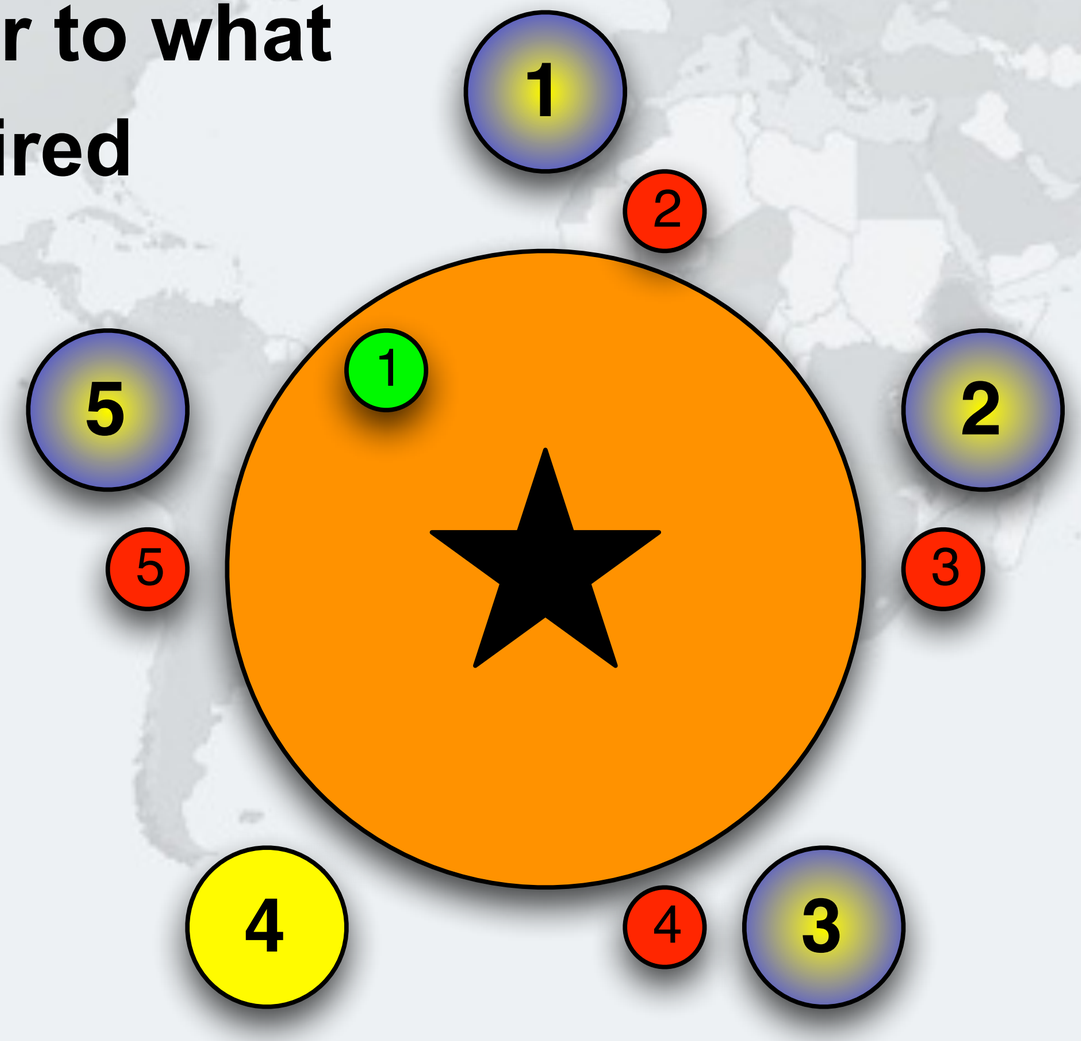


Javaspecialists.eu

JavaSpecialists.EU

Philosopher 1 Returns Cup 1

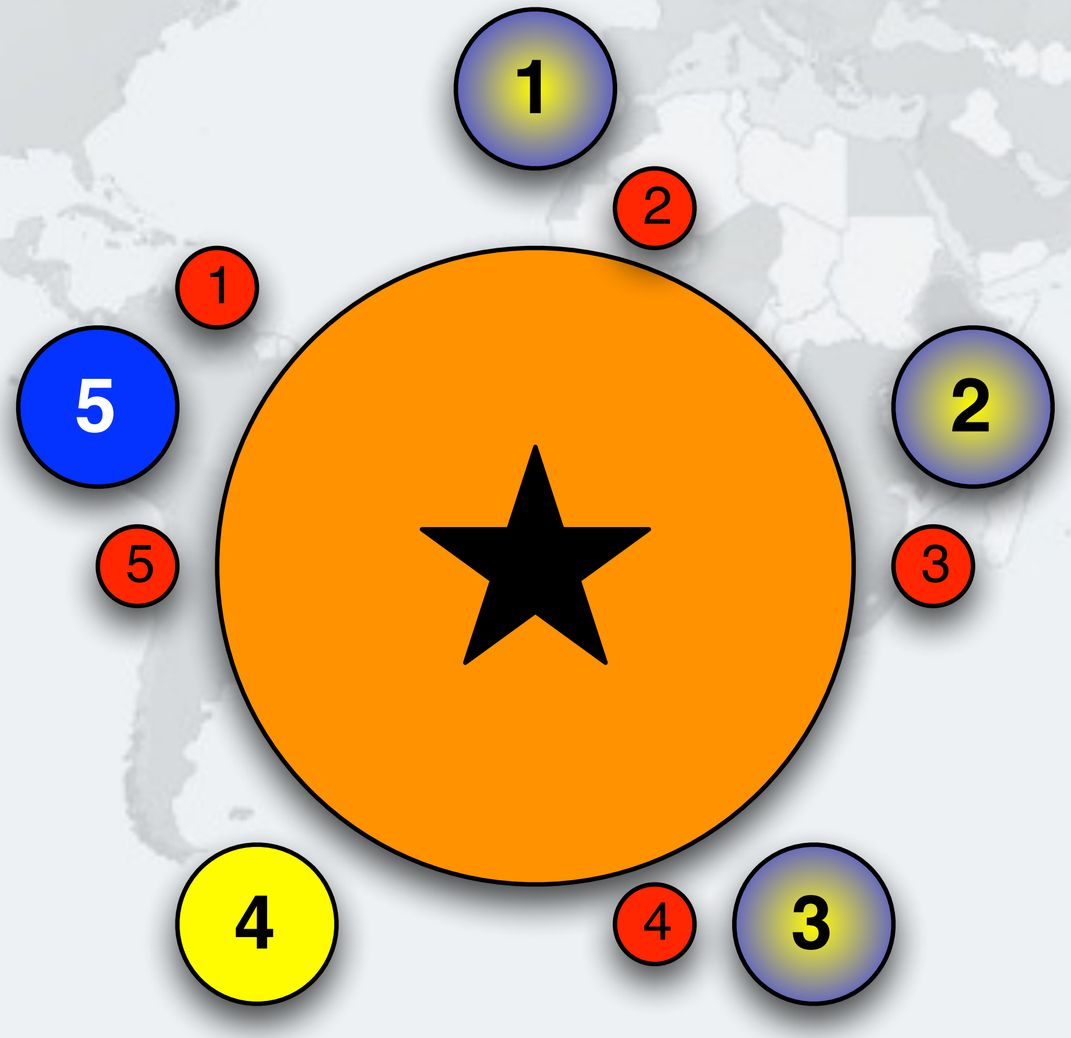
- **Cups are returned in the opposite order to what they are acquired**



Javaspecialists.eu

JavaSpecialists.EU

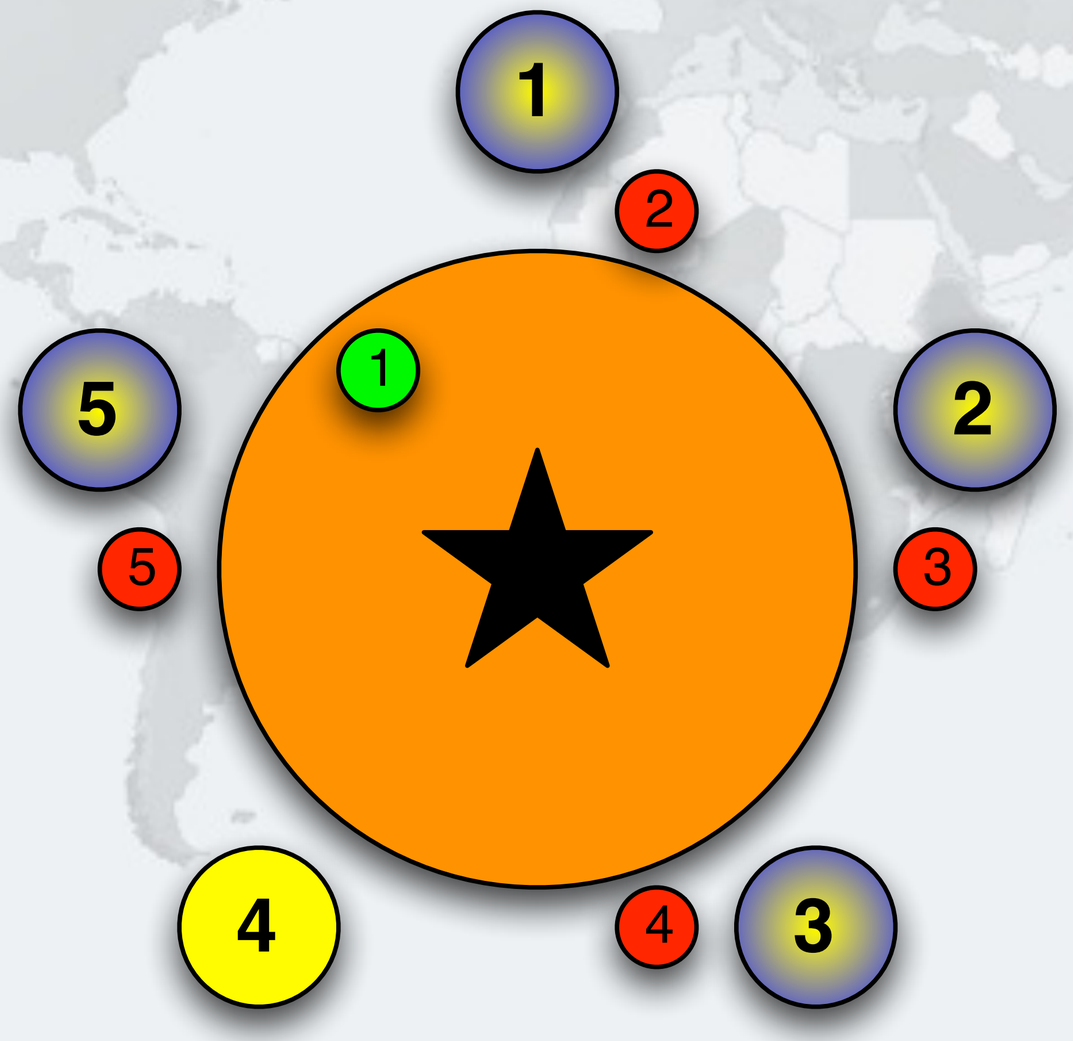
Philosopher 5 Takes Cup 1 - Drinking



Javaspecialists.eu

JavaSpecialists.EU

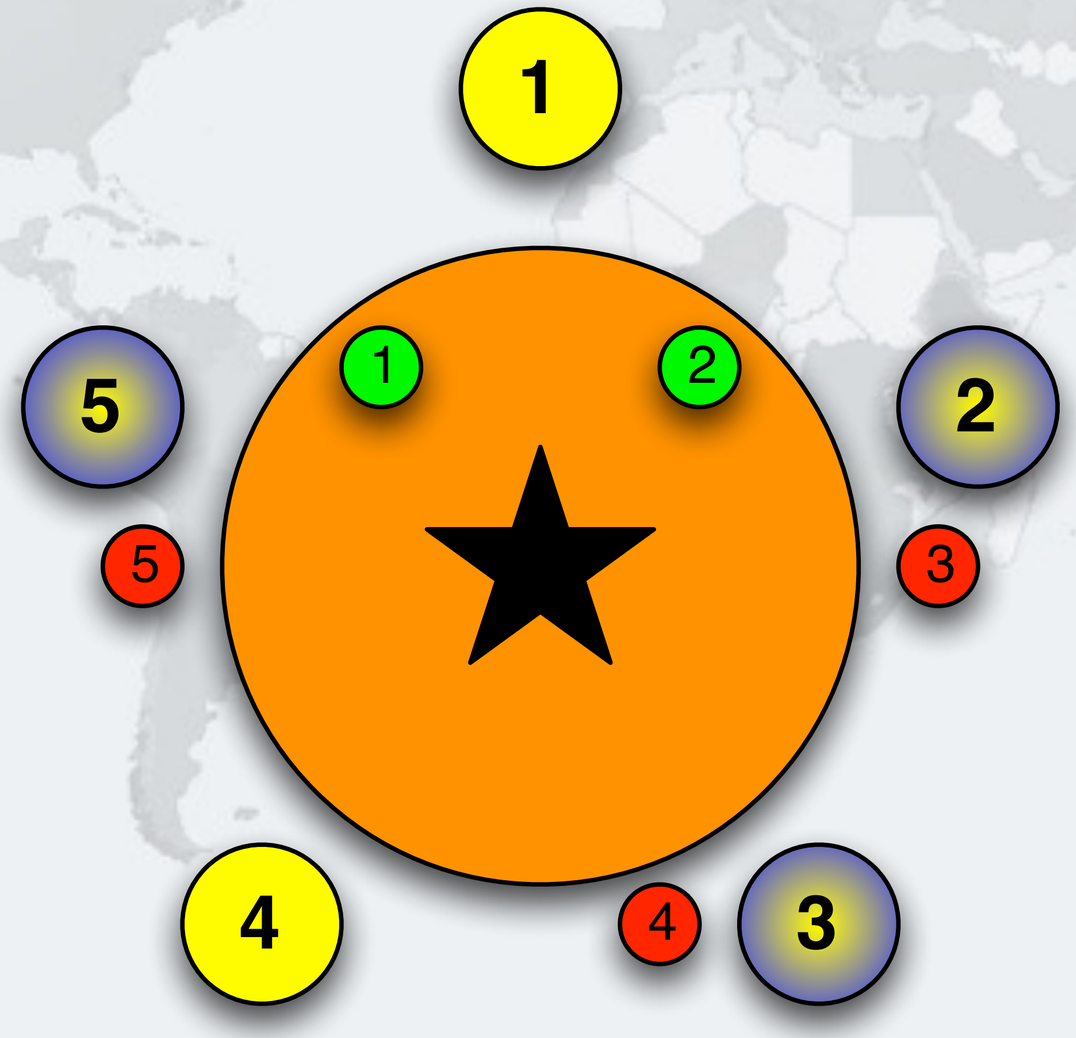
Philosopher 5 Returns Cup 1



Javaspecialists.eu

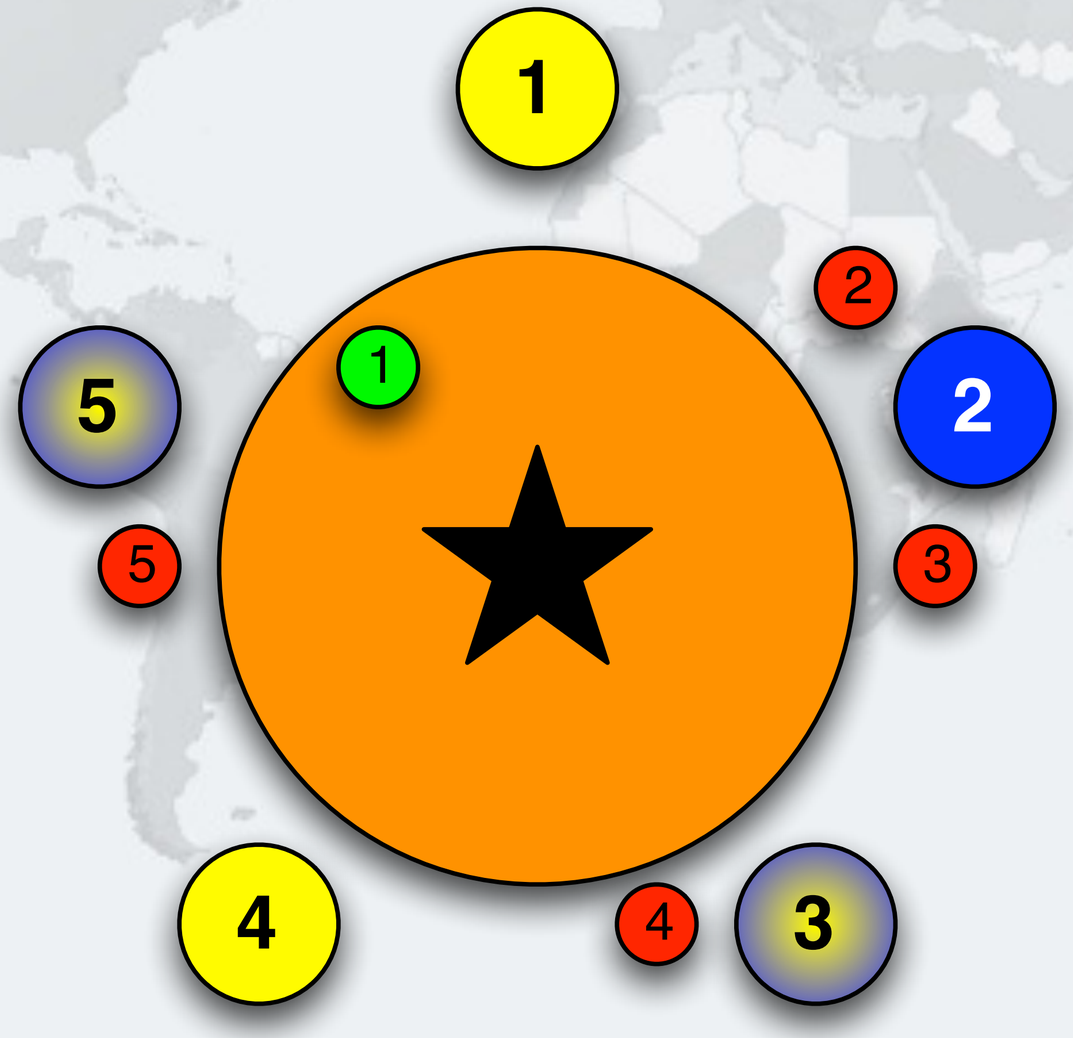
JavaSpecialists.EU

Philosopher 1 Returns Cup 2



Javaspecialists.eu
JavaSpecialists.EU

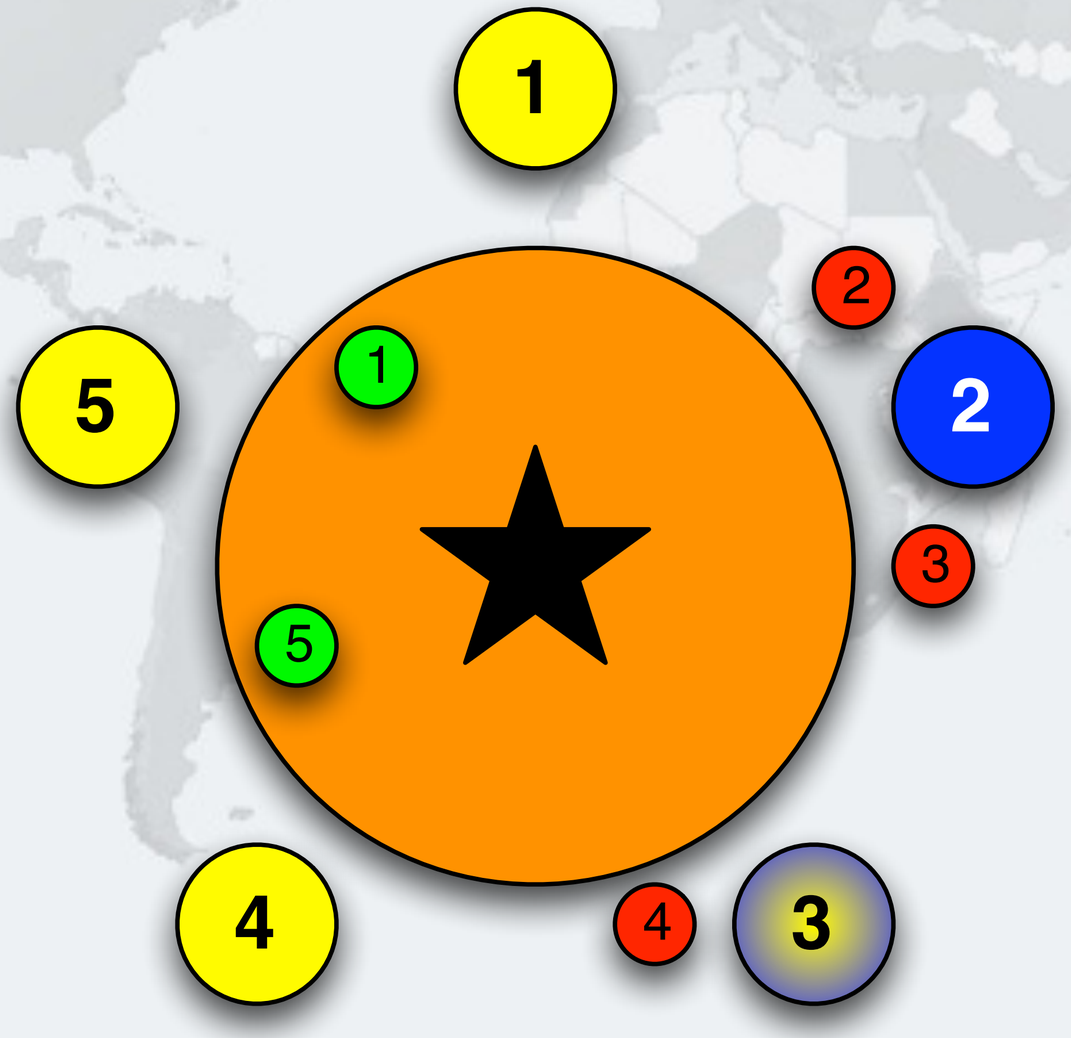
Philosopher 2 Takes Cup 2 - Drinking



Javaspecialists.eu

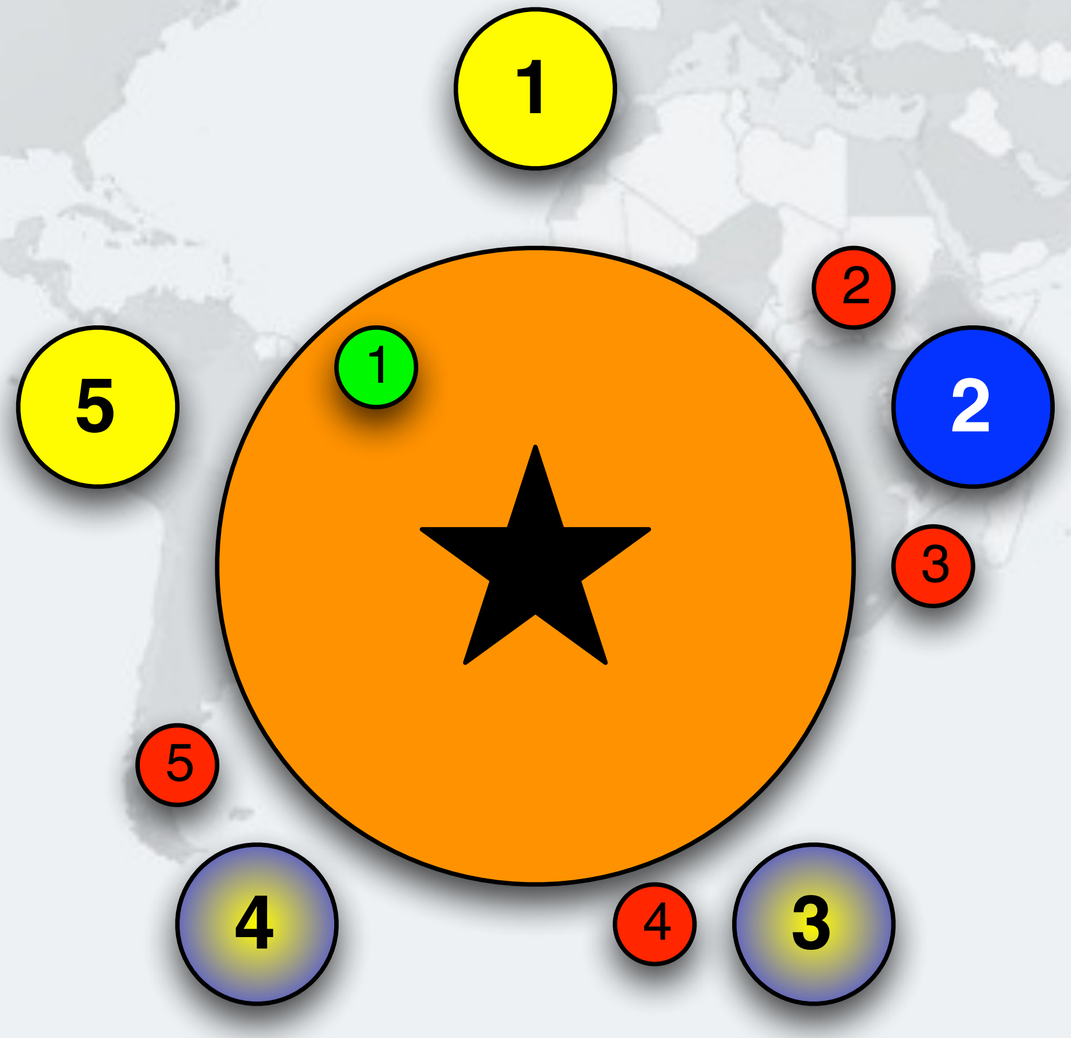
JavaSpecialists.EU

Philosopher 5 Returns Cup 5



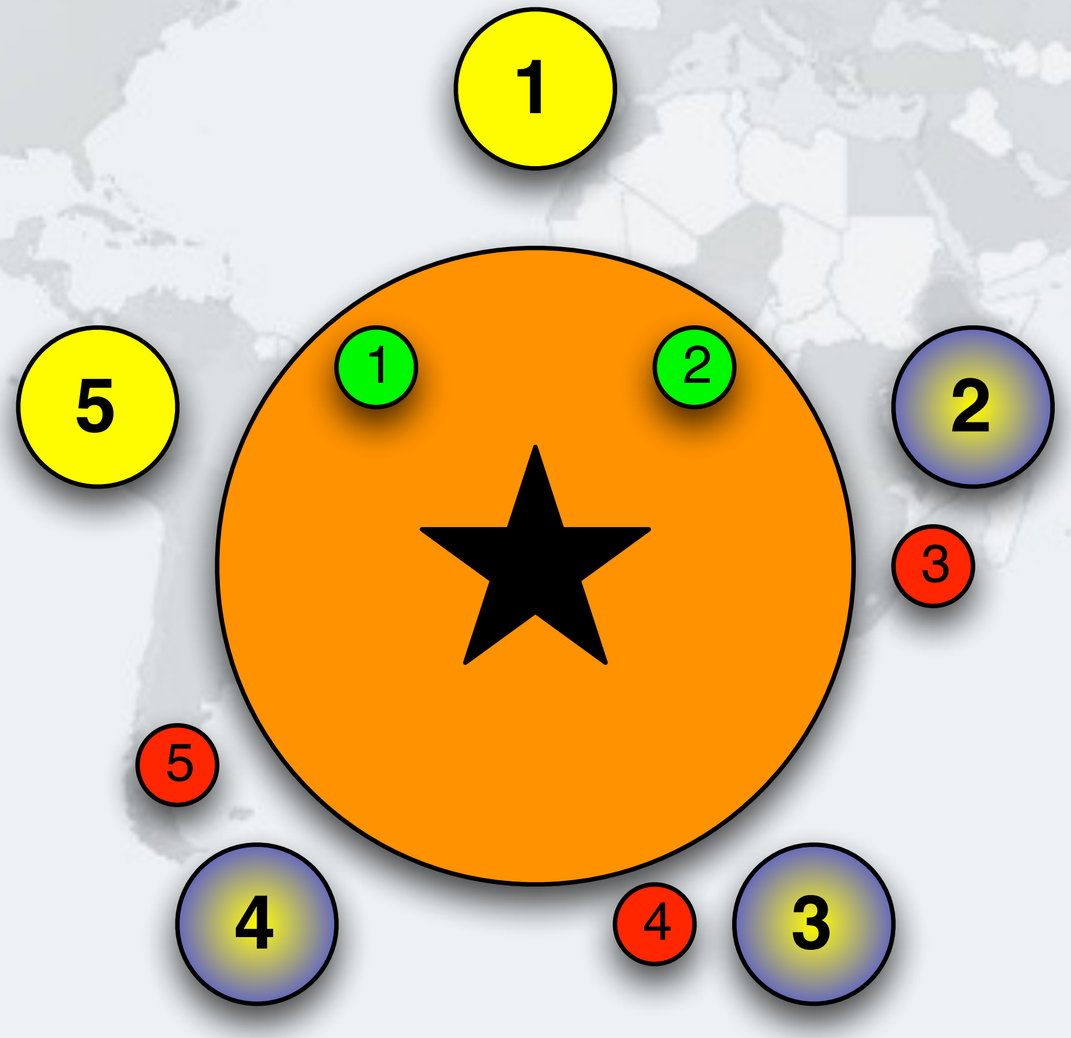
Javaspecialists.eu
JavaSpecialists.EU

Philosopher 4 Takes Cup 5



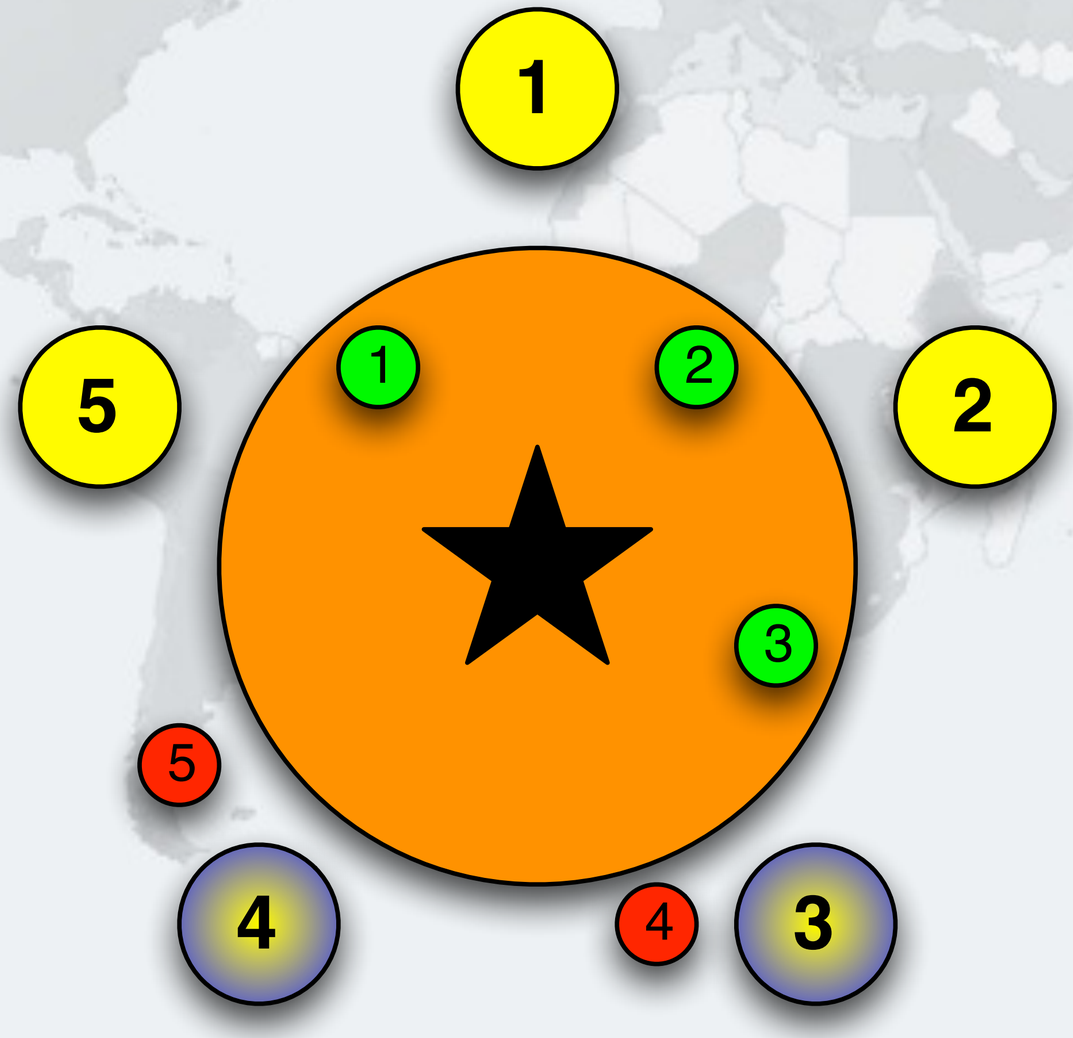
Javaspecialists.eu
JavaSpecialists.EU

Philosopher 2 Returns Cup 2



Javaspecialists.eu
JavaSpecialists.EU

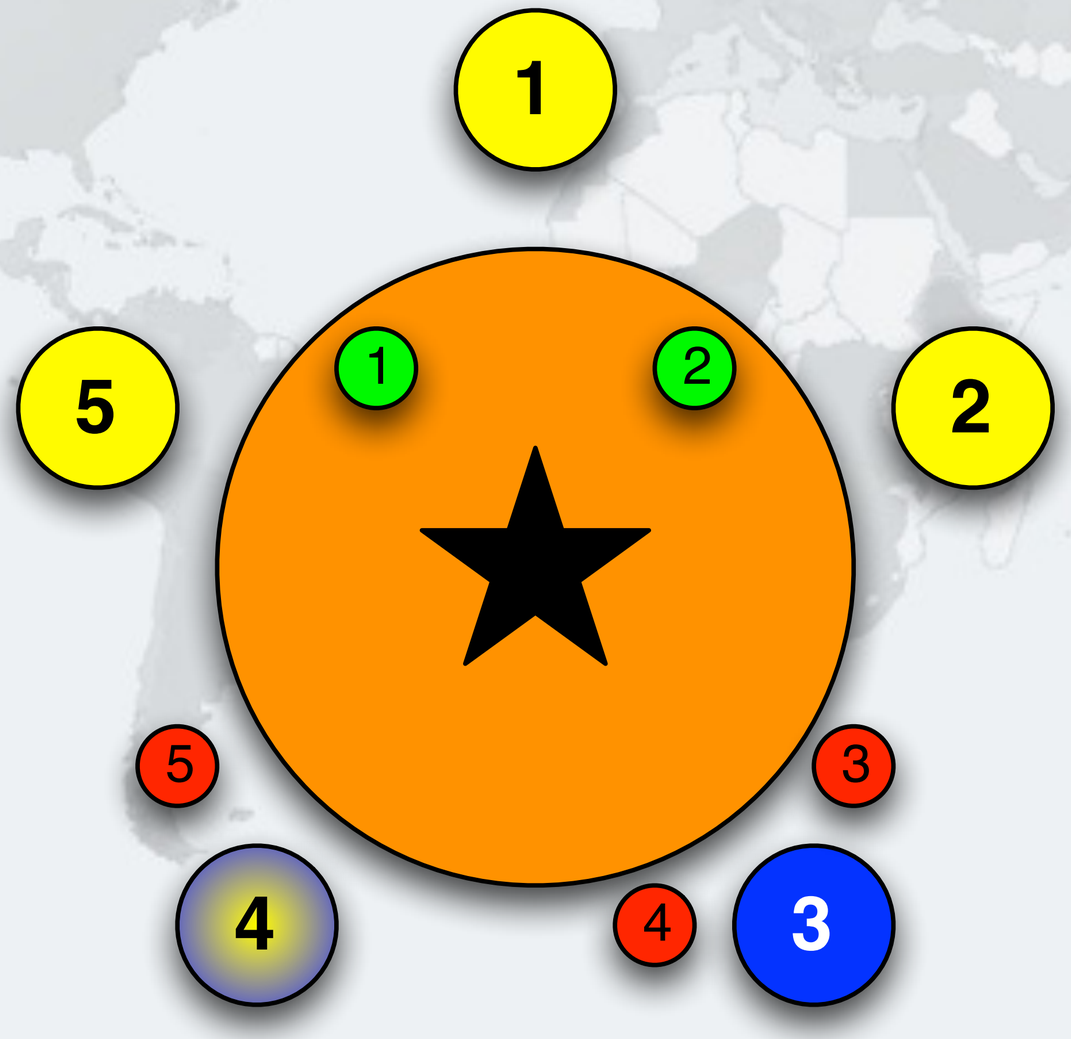
Philosopher 2 Returns Cup 3



Javaspecialists.eu

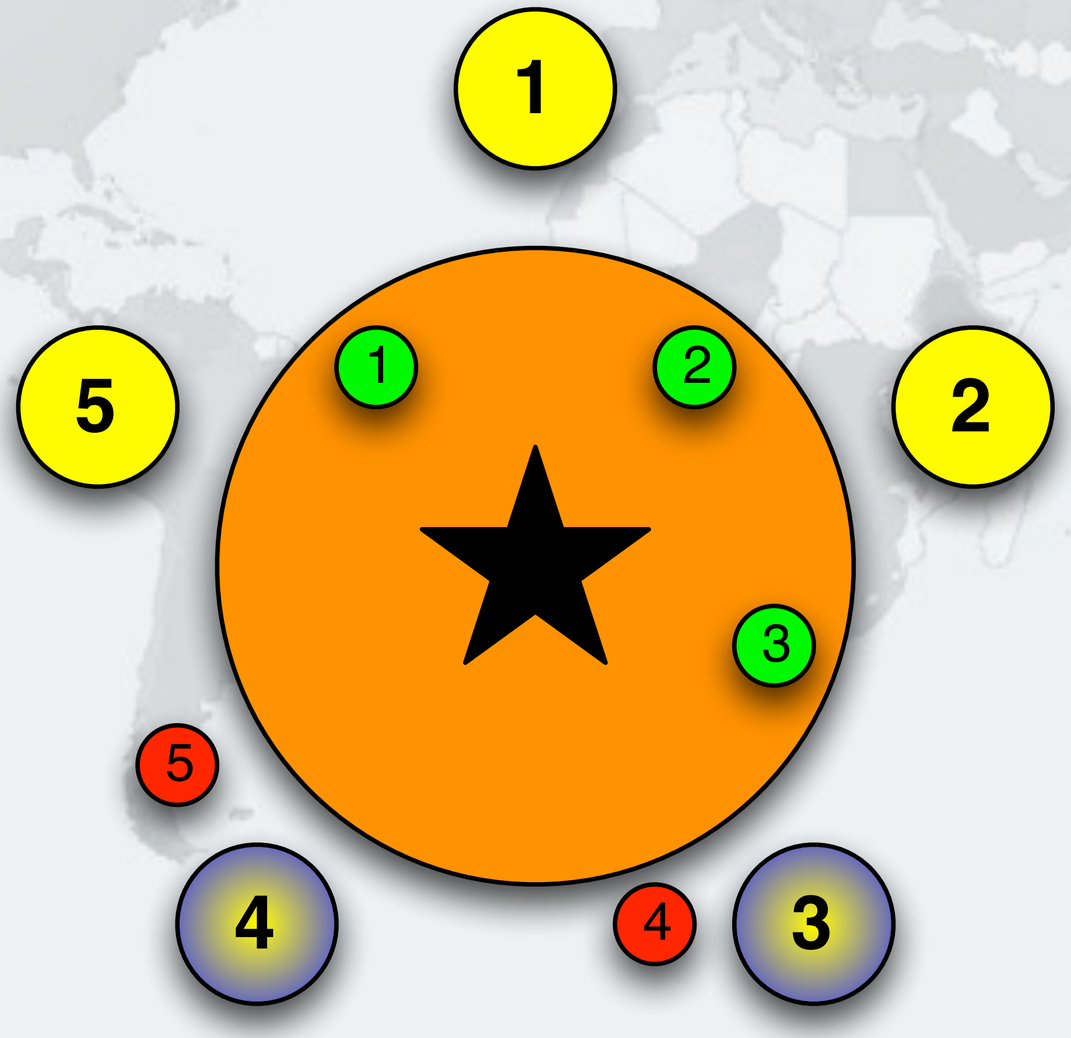
JavaSpecialists.EU

Philosopher 3 Takes Cup 3 - Drinking



Javaspecialists.eu
JavaSpecialists.EU

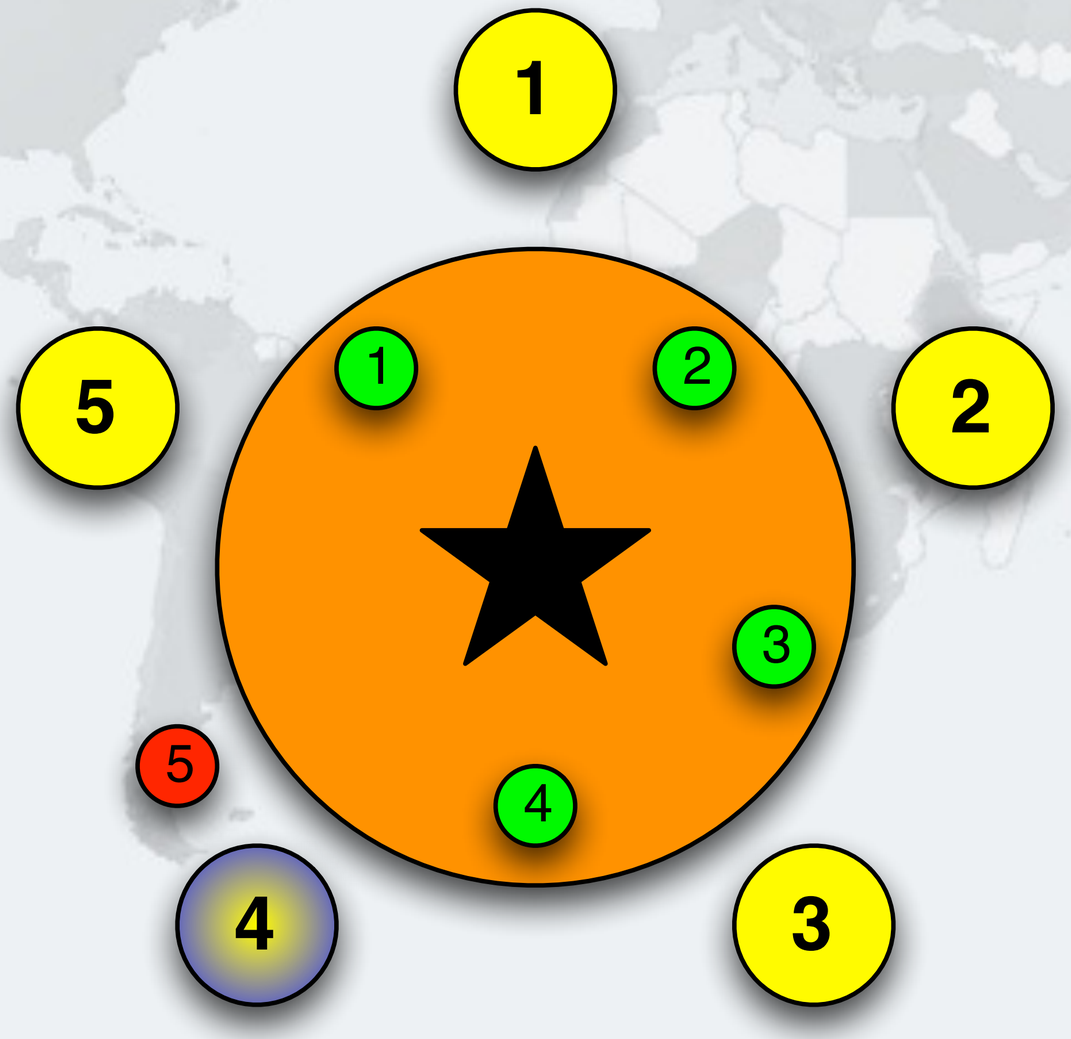
Philosopher 3 Returns Cup 3



Javaspecialists.eu

JavaSpecialists.EU

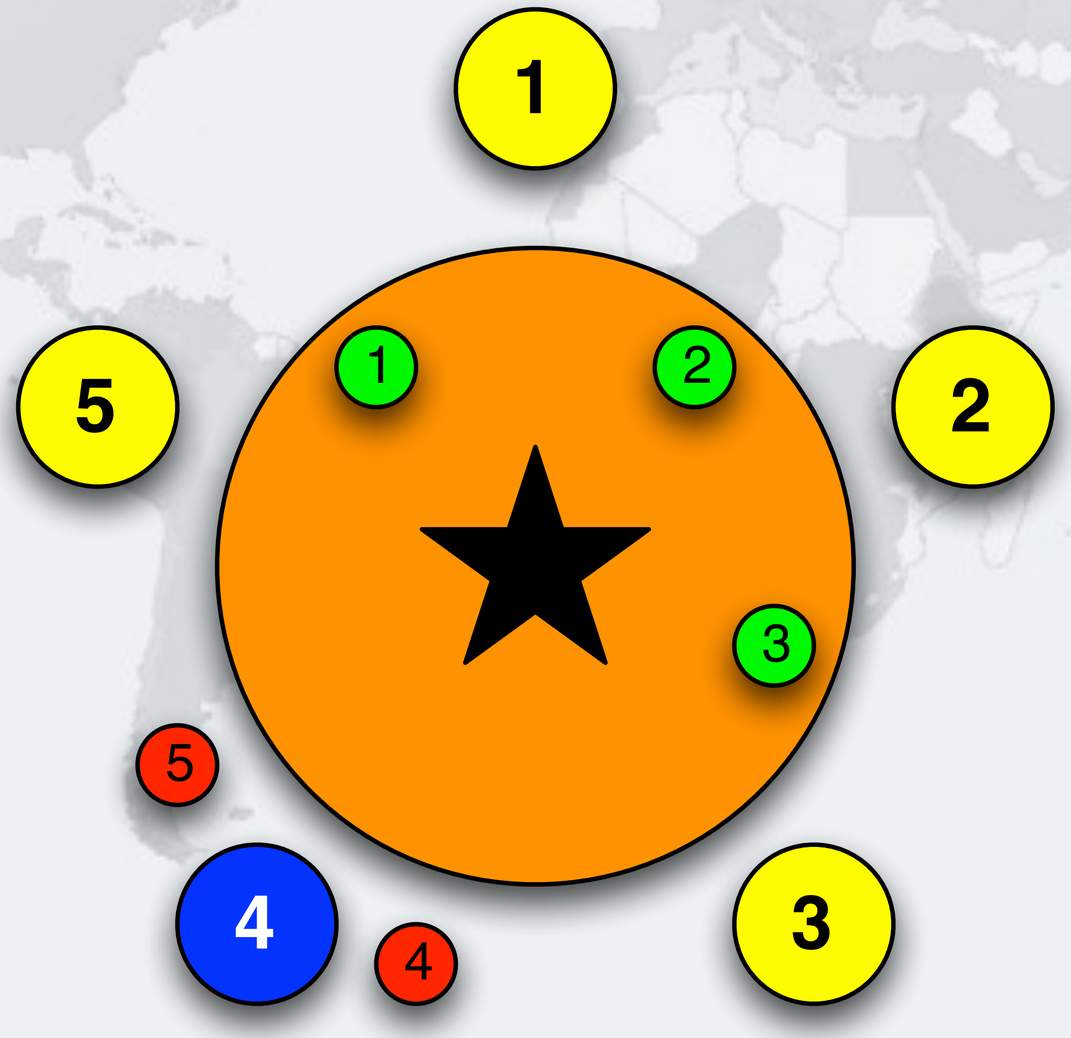
Philosopher 3 Returns Cup 4



Javaspecialists.eu

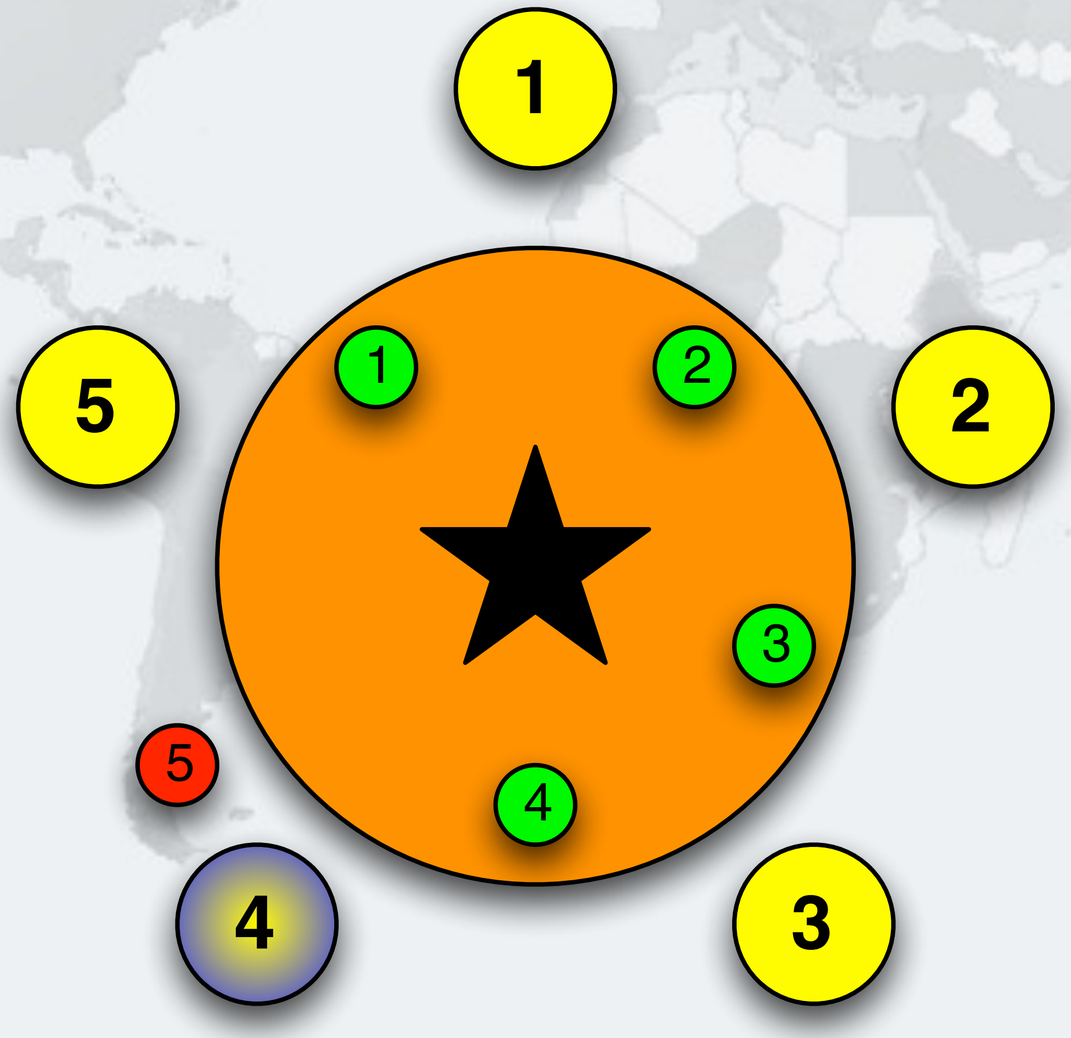
JavaSpecialists.EU

Philosopher 4 Takes Cup 4 - Drinking



Javaspecialists.eu
JavaSpecialists.EU

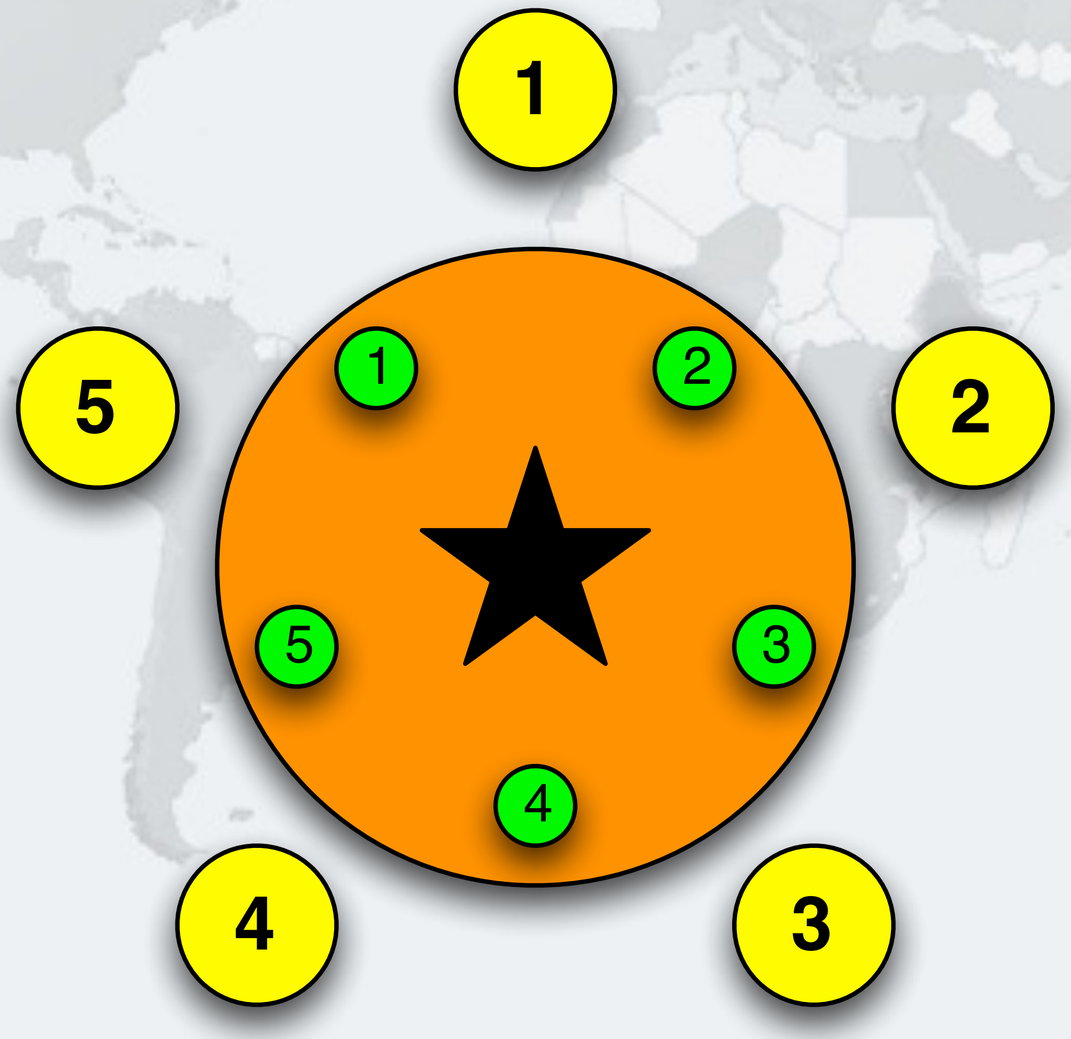
Philosopher 4 Returns Cup 4



Javaspecialists.eu

JavaSpecialists.EU

Philosopher 4 Returns Cup 5



Javaspecialists.eu
JavaSpecialists.EU

Dynamic Lock Order Deadlocks

- **Often, it is not obvious what the lock instances are, e.g.**

```
public boolean transferMoney(
    Account from, Account to,
    DollarAmount amount) {
  synchronized (from) {
    synchronized (to) {
      return doActualTransfer(from, to, amount);
    }
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Causing The Deadlock Transferring Money

- **Giorgos has accounts in Switzerland and in Greece**
  - **He keeps on transferring money between them**
    - **Whenever new taxes are announced, he brings money into Greece**
    - **Whenever he gets any money paid, he transfers it to Switzerland**
    - **Sometimes these transfers can coincide**
- **Thread 1 is moving money from UBS to Alpha Bank**

```
transferMoney(ubs, alpha, new DollarAmount(1000));
```

- **Thread 2 is moving money from Alpha Bank to UBS**

```
transferMoney(alpha, ubs, new DollarAmount(2000));
```

- **If this happens at the same time, it can deadlock**

Javaspecialists.eu

JavaSpecialists.EU

Fixing Dynamic Lock-Ordering Deadlocks

- **The locks for transferMoney() are outside our control**
  - **They could be sent to us in any order**
- **We can *induce* an ordering on the locks**
  - **For example, we can use System.identityHashCode() to get a number representing this object**
    - **Since this is a 32-bit int, it is possible that two different objects have exactly the same identity hash code**
    - **In that case, we have a static lock to avoid a deadlock**
    - **We can test this with -XX:hashCode=2, which always returns the same identity hash code of 1**

Javaspecialists.eu

JavaSpecialists.EU

```
public boolean transferMoney(Account from, Account to,
                             DollarAmount amount) {
  int fromHash = System.identityHashCode(from);
  int toHash = System.identityHashCode(to);
  if (fromHash < toHash) {
    synchronized (from) {
      synchronized (to) {
        return doActualTransfer(from, to, amount);
      }
    }
  } else if (fromHash > toHash) {
    synchronized (to) {
      synchronized (from) {
        return doActualTransfer(from, to, amount);
      }
    }
  } else {
    synchronized (tieLock) {
      synchronized (from) {
        synchronized (to) {
          return doActualTransfer(from, to, amount);
        }
      }
    }
  }
}
```

```
private static final Object tieLock =
  new Object();
```

Javaspecialists.eu

JavaSpecialists.EU

Deadlocks Between Cooperating Objects

- **In this example, the deadlock is more subtle**
  - **Taxi is an individual taxi with a location and a destination**
  - **Dispatcher represents a fleet of taxis**
- **Spot the deadlock**

Javaspecialists.eu

JavaSpecialists.EU

Taxi, Representing An Individual Vehicle

```
public class Taxi {
  @GuardedBy("this")
  private Point location, destination;
  private final Dispatcher dispatcher;

  public Taxi(Dispatcher dispatcher) {
    this.dispatcher = dispatcher;
  }

  public synchronized Point getLocation() {
    return location;
  }

  public synchronized void setLocation(
      Point location) {
    this.location = location;
    if (location.equals(destination))
      dispatcher.notifyAvailable(this);
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

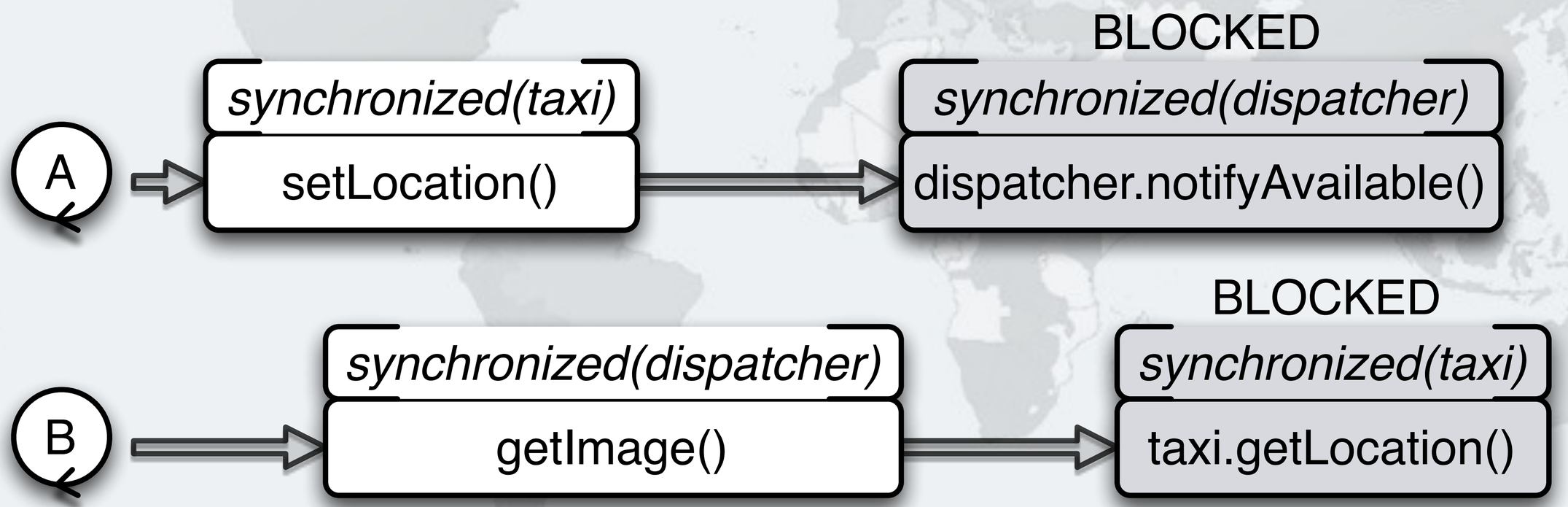
Dispatcher: Managing A Fleet Of Taxis

```
public class Dispatcher {
  @GuardedBy("this")
  private final Set<Taxi> taxis = new HashSet<>();
  @GuardedBy("this")
  private final Set<Taxi> availableTaxis = new HashSet<>();

  public synchronized void notifyAvailable(Taxi taxi) {
    availableTaxis.add(taxi);
  }

  public synchronized Image getImage() {
    Image image = new Image();
    taxis.forEach(
        taxi -> image.drawMarker(taxi.getLocation()));
   return image;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

How To Deadlock The Taxi Industry



Javaspecialists.eu

JavaSpecialists.EU

Open Calls

- Calling an *alien method* with a lock held is difficult to analyze and therefore risky
- Both Taxi and Dispatcher break this rule
- Calling a method with no locks held is called an *open call*
  - Makes it much easier to reason about liveness

Javaspecialists.eu

JavaSpecialists.EU

Refactored Taxi.setLocation()

- **We should not call *alien methods* whilst holding locks**
- **In our first solution, we split the method up into parts that need the lock and those that call alien methods**

```
public void setLocation(Point location) {
  boolean reachedDestination;
  synchronized (this) {
    this.location = location;
    reachedDestination = location.equals(destination);
  }
  if (reachedDestination) {
    dispatcher.notifyAvailable(this);
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Refactored Dispatcher.getImage()

- **In our second solution, we make a copy of the set to prevent race conditions**
  - **Alternatively we could use a concurrent hash set**

```
public Image getImage() {
  Set<Taxi> copy;
  synchronized (this) {
    copy = new HashSet<>(taxis);
  }
  Image image = new Image();
    copy.forEach(
        taxi -> image.drawMarker(taxi.getLocation()));
  return image;
}
```

Javaspecialists.eu

JavaSpecialists.EU

Refactored Taxi.getLocation()

- **A 3rd solution is to make location volatile**
  - **Now we do not need getLocation() to be synchronized anymore**

```
private volatile Point location;

public Point getLocation() {
  return location;
}
```

- **Any one of these three solutions would remove deadlock**

Javaspecialists.eu

JavaSpecialists.EU

Benefit Of Open Calls

- **Strive to use open calls throughout your program**
- **Programs that rely on open calls are far easier to analyze for deadlock-freedom than those that allow calls to alien methods with locks held**
- **Alien method calls with lock held are probably the biggest cause of deadlocks "in the field"**

Javaspecialists.eu

JavaSpecialists.EU

Open Call In Vector

- **In Sun Java 6 Vector.writeObject() is synchronized**
  - **This is to provide thread safety during writing**

```
private synchronized void writeObject(ObjectOutputStream s)
    throws IOException {
  s.defaultWriteObject();
}
```

  - **However, since it calls alien "defaultWriteObject()" it can deadlock**
    - **http://www.javaspecialists.eu/archive/Issue184.html**

Javaspecialists.eu

JavaSpecialists.EU

IBM Avoids This Problem With An Open Call

```
private void writeObject(ObjectOutputStream stream)
    throws IOException {
  Vector<E> cloned = null;
  // this specially fix is for a special dead-lock in customer
  // program: two vectors refer each other may meet dead-lock in
  // synchronized serialization. Refer CMVC-103316.1
  synchronized (this) {
    try {
      cloned = (Vector<E>) super.clone();
      cloned.elementData = elementData.clone();
    } catch (CloneNotSupportedException e) {
      // no deep clone, ignore the exception
    }
  }
  cloned.writeObjectImpl(stream);
}

private void writeObjectImpl(ObjectOutputStream stream)
   throws IOException {
 stream.defaultWriteObject();
}
```

Javaspecialists.eu

JavaSpecialists.EU

OpenJDK 7 Also Uses An Open Call

```
private void writeObject(ObjectOutputStream s)
    throws IOException {
  final ObjectOutputStream.PutField fields = s.putFields();
  final Object[] data;
  synchronized (this) {
    fields.put("capacityIncrement", capacityIncrement);
    fields.put("elementCount", elementCount);
    data = elementData.clone();
  }
  fields.put("elementData", data);
  s.writeFields();
}
```

Javaspecialists.eu

JavaSpecialists.EU

8.2: Avoiding And Diagnosing Deadlocks

Avoiding Liveness Hazards

JavaSpecialists.EU

8.2: Avoiding And Diagnosing Deadlocks

- **If you only ever acquire one lock, you cannot get a lock-ordering deadlock**
  - This is the easiest way to avoid deadlocks, but not always practical
- **If you need to acquire multiple locks, include lock ordering in your design**
  - Important to specify and document possible lock sequences
  - Identify where multiple locks could be acquired
  - Do a global analysis to ensure that lock ordering is consistent
    - This can be extremely difficult in large programs
- **Use open calls whenever possible**
  - Do not call alien methods whilst holding a lock

Javaspecialists.eu

JavaSpecialists.EU

Unit Testing For Lock Ordering Deadlocks

- **Code typically has to be called many times before a deadlock happens**
- **How many times do you need to call it to prove that there is no deadlock?**
  - **Nondeterministic unit tests are bad - they should either always pass or always fail**

Javaspecialists.eu

JavaSpecialists.EU

Adding A Sleep To Cause Deadlocks

- **In the transferMoney() method, a deadlock occurs if after the first lock is granted, the first thread is swapped out and another thread requests the second lock**
- **We can force this to happen by sleeping a short while after requesting the first lock**

```
public class Bank {
  public boolean transferMoney(Account from, Account to,
                               DollarAmount amount) {
    synchronized (from) {
      sleepAWhileForTesting();
      synchronized (to) {
        return doActualTransfer(from, to, amount);
      }
    }
  }
  protected void sleepAWhileForTesting() {}
}
```

Javaspecialists.eu

JavaSpecialists.EU

In Our Unit Test We Override The Class

- **We make the sleepAWhileForTesting() method sleep**
  - **In production, when we use only the normal Bank, the empty method will be optimized away by the HotSpot compiler**

```
public class SlowBank extends Bank {
  private final long timeout;
  private final TimeUnit unit;
  public SlowBank(long timeout, TimeUnit unit) {
    this.timeout = timeout;
    this.unit = unit;
  }
  protected void sleepAWhileForTesting() {
    try {
      unit.sleep(timeout);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Verifying Thread Deadlocks

- **ThreadMXBean has two methods for finding deadlocks**
  - **findMonitorDeadlockedThreads()**
    - **Includes only "monitor" locks, i.e. synchronized**
    - **Only way to find deadlocks in Java 5**
  - **findDeadlockedThreads()**
    - **Includes "monitor" and "owned" (Java 5) locks**
    - **Preferred method to test for deadlocks**
    - **But, does *not* find deadlocks between semaphores**
  - **See http://www.javaspecialists.eu/archive/Issue130.html**

Javaspecialists.eu

JavaSpecialists.EU

```
public class BankDeadlockTest {
  private final static ThreadMXBean tmb =
      ManagementFactory.getThreadMXBean();

  private void checkThatThreadTerminates(Thread thread)
      throws InterruptedException {
    for (int i = 0; i < 2000; i++) {
      thread.join(50);
      if (!thread.isAlive()) return;
      if (isThreadDeadlocked(thread.getId())) {
        fail("Deadlock detected!");
      }
    }
    fail(thread + " did not terminate in time");
  }

  private boolean isThreadDeadlocked(long tid) {
    long[] ids = tmb.findDeadlockedThreads();
    if (ids == null) return false;
    for (long id : ids) {
      if (id == tid) return true;
    }
    return false;
  }
```

Javaspecialists.eu

JavaSpecialists.EU

```
@Test
public void testForTransferDeadlock()
    throws InterruptedException {
  Account alpha = new Account(new DollarAmount(1000));
  Account ubs = new Account(new DollarAmount(1000000));
  Bank bank = new SlowBank(100, TimeUnit.MILLISECONDS);

  Thread alphaToUbs = new Thread(() ->
      bank.transferMoney(alpha, ubs, new DollarAmount(100))
      , "alphaToUbs");
  Thread ubsToAlpha = new Thread(() ->
      bank.transferMoney(ubs, alpha, new DollarAmount(100))
      , "ubsToAlpha");

  alphaToUbs.start();
  ubsToAlpha.start();

  checkThatThreadTerminates(alphaToUbs);
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Output With Broken transferMoney() Method

- **We see the deadlock within about 100 milliseconds**

```
junit.framework.AssertionFailedError: Deadlock detected!
  at BankDeadlockTest.checkThatThreadTerminates(BankDeadlockTest.java:20)
  at BankDeadlockTest.testForTransferDeadlock(BankDeadlockTest.java:52)
```

- **If we fix the transferMoney() method, it also completes within about 100 milliseconds**
  - **This is the time that we are sleeping for testing purposes**
- **Remember that the empty sleepAWhileForTesting() method will be optimized away by HotSpot**

Javaspecialists.eu

JavaSpecialists.EU

Deadlock Analysis With Thread Dumps

- The ThreadMXBean can be invoked directly to find deadlocks between monitors or Java 5 locks
- However, we can also cause a thread dump in many ways:
  - Ctrl+Break on Windows or Ctrl-\ on Unix
  - Invoking "kill -3" on the process id
  - Calling jstack on the process id
    - Only shows deadlocks since Java 6
- Intrinsic locks typically show more information of where they were acquired than the explicit Java 5 locks

Javaspecialists.eu

JavaSpecialists.EU

Deadlock Analysis With Thread Dumps

- **Thread dump from a real system (names changed)**
- **It is useful to have unique threads names**
- **The stack trace confirms the deadlock**

```
Found one Java-level deadlock:
=============================
"ApplicationServerThread-0":
  waiting to lock monitor 0x080f0cdc
    (object 0x650f7f30, a MumbleDBConnection),
  which is held by "ApplicationServerThread-1"

"ApplicationServerThread-1":
  waiting to lock monitor 0x080f0ed4
    (object 0x6024ffb0, a MumbleDBCallableStatement),
  which is held by "ApplicationServerThread-0"
```

Javaspecialists.eu

JavaSpecialists.EU

Stack Information Shows Where It Comes From

```
Java stack information for the threads listed above:
===================================================
"ApplicationServerThread-0":
    at MumbleDBConnection.remove_statement
    - waiting to lock <0x650f7f30> (a MumbleDBConnection)
    at MumbleDBStatement.close
    - locked <0x6024ffb0> (a MumbleDBCallableStatement)
    ...

"ApplicationServerThread-1":
    at MumbleDBCallableStatement.sendBatch
    - waiting to lock <0x6024ffb0>
                                (a MumbleDBCallableStatement)
    at MumbleDBConnection.commit
    - locked <0x650f7f30> (a MumbleDBConnection)
    ...

Found 1 deadlock.
```

Javaspecialists.eu

JavaSpecialists.EU

What Caused The Deadlock?

- **Inside the JDBC driver, different calls acquired locks in different orders**
  - **JDBC vendor was trying to build a thread-safe driver**
    - **But then ended up writing a potential deadlock**
  - **This could be fixed in the JDBC driver by imposing a global order**
- **However, in the system the JDBC connection was shared by multiple threads**
  - **This caused the bug to appear**
- **Solution: single threaded access to each individual connection**

Javaspecialists.eu

JavaSpecialists.EU

Thread Dump Caveats

- **Thread dump are not always accurate**
  - **It might appear that two threads are in the same synchronized block**
    - **Or that all threads are blocked and no one can enter**
  - **We can have differences in thread states between Java and native**

```
"pool-1-thread-1" #11 runnable [0x00007000016d8000]
   java.lang.Thread.State: WAITING (on object monitor)
```

- **Not all deadlocks automatically shown in thread dump**
  - **Resource deadlock, e.g. thread blocked forever on a Semaphore**
  - **Class loading deadlock, e.g. thread accessing static method or field in static initializer**
  - **ReentrantLock where thread has exited without unlocking**
  - **ReentrantReadWriteLock upgrading of lock**

Javaspecialists.eu

JavaSpecialists.EU

8.3: Livelock

Avoiding Liveness Hazards

JavaSpecialists.EU

8.3: Livelock

- Thread is running, but still not making progress
- Typically forever retrying a failed operation
  - Eventually you need to give up
- Often occurs in transactional messaging applications, where the messaging infrastructure rolls back a transaction if a message cannot be processed successfully, and puts it back at the head of the queue.
  - This form of livelock often comes from overeager error-recovery code that mistakenly treats an unrecoverable error as a recoverable one.

Javaspecialists.eu

JavaSpecialists.EU

Real-World Scenario

- Two polite people meet in a narrow corridor. Each steps to the side to make room for the other. They keep on doing this at the same time, never getting past each other.
  - Fortunately people are not that stupid
    - But computers are!
- Can happen especially in code that tries to recover from a deadlock situation
  - Only possible with Java 5 locks, in a controlled fashion

Javaspecialists.eu

JavaSpecialists.EU

Livelock In Intellij Idea

WildFactorizer.java – [Concurrency] – [~/svn/courses/concurrency/samples/Concurrency] – IntelliJ IDEA (Nika) IU-111.255

Concurrency > src > concurrency > ch06_task_execution > solution_6_1 > WildFactorizer

Project concurrency.ch06_task_execution.solution_6_1

Project

- Concurrency (~/svn/courses/concurrency/samples/Concurrency)
  - lib
  - src
    - concurrency
      - ch01_introduction
      - ch02_thread_safety
        - exercise_2_1
        - solution_2_1
      - ch03_sharing_objects
      - ch04_composing_objects
        - exercise_4_1
        - exercise_4_2
        - solution_4_1
      - ch05_building_blocks
        - exercise_5_1
        - exercise_5_2
        - solution_5_1
        - solution_5_2
      - ch06_task_execution
        - exercise_6_1
        - solution_6_1
          - WildFactorizer
      - ch07_cancellation_and_shutdown

Favorites

- Concurrency

CLIThread | exercise_7_2/CLITest | exercise_4_1/ByteGenerator | exercise_4_2/BankAccount | exercise_4_1/ByteGeneratorTest | InsufficientFundsException | solution_4_1/ByteGenerator | WildFactorizer | solution_4_1/ByteGeneratorTest | ConsoleInputReadTask

```
1  package concurrency.ch06_task_execution.solution_6_1;
2
3  import concurrency.math.*;
4
5  import java.util.concurrent.atomic.*;
6
7  public class WildFactorizer {
8    public static void main(String[] args) {
9      long start = (1 << 19) - 1; // known Mersenne prime
10     AtomicLong next = new AtomicLong(start);
11     for (int i = 0; i < 10000; i++) {
12       long number = next.getAndIncrement();
13       long[] factors = Factorizer.factor(number);
14       if (factors.length == 1) {
15         System.out.println(factors[0]);
16       }
17     }
18   }
19 }
```

Compilation aborted (moments ago) | 7:20 | UTF-8 | 171M of 2043M

Javaspecialists.eu
Javaspecialists.EU

Livelock Due To Wait/Notify

```
public class WaitNotifyLivelock {
  private boolean state = false;

  public synchronized void waitFor() {
    while (!state) {
      try {
        wait(); // only lets go of lock if it is not interrupted
      } catch (InterruptedException e) {
        // re-interrupting without exiting is a mistake
        Thread.currentThread().interrupt();
      }
    }
  }

  public synchronized void notifyIt() {
    state = true;
    notifyAll();
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

8: Exercises

Avoiding Liveness Hazards

JavaSpecialists.EU

Exercise 8.1: Solve Deadlock via Lock Ordering

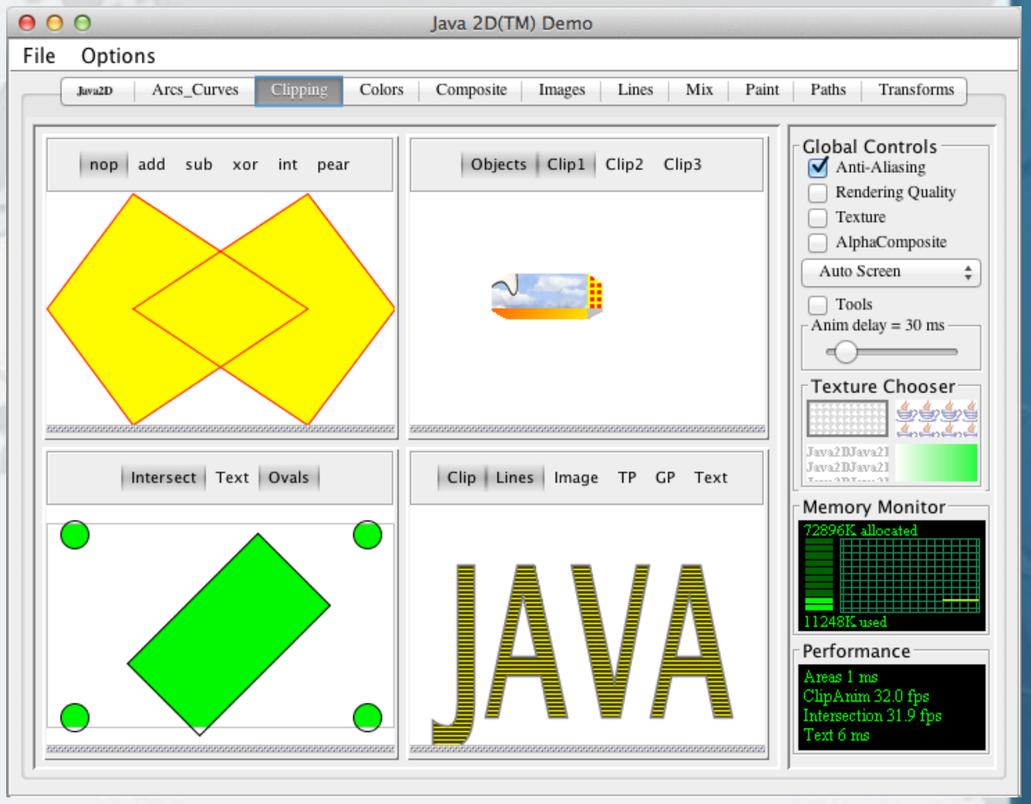
- **We want to solve the drinking philosophers**
  - **Capture a stack trace with `jps` and `jstack pid > stack1.txt`**
  - **Verify the deadlock involves the left and right locks**
  - **Define a global ordering for the locks that would prevent deadlock**
    - **We are synchronizing on the Krasi objects**
    - **Define a global ordering for Krasi objects by implementing Comparable and providing a unique number to sort on (Krasi.java)**
    - **Change the code to use the global ordering (Thinker.java)**
  - **Verify that the deadlock has now disappeared**

Javaspecialists.eu

JavaSpecialists.EU

Exercise 8.2: Find And Eliminate Deadlock

- **Start our modified Java2Demo**
  - **Capture thread dump with `jstack`**
  - **Use Java2Demo for a while until it deadlocks**
  - **Capture another thread dump and compare to the first one with diff**
    - **This should show you where the problem is inside your code**
  - **Fix the problem and verify that it has been solved**



Javaspecialists.eu

JavaSpecialists.EU

8: Avoiding Liveness Hazards

Questions?

9: Testing Concurrent Programs

It is impossible to exhaustively test concurrent programs

JavaSpecialists.EU

9: Testing Concurrent Programs

- **Concurrency bugs manifest relatively rarely**
  - **Usually happen in production at the worst possible time (24th Dec)**
    - **Production usually has richer data and faster hardware**
- **In practice, we *cannot* prove that our Java program is free from race conditions and deadlocks**
- **If Java was less powerful, such as a Petri Net, we could prove some characteristics**
  - **But then we could not use it for *real* software development**

Javaspecialists.eu

JavaSpecialists.EU

How To Test Concurrent Code?

- **We just said that we cannot prove our code is bug free**
  - **Testing only proves the *presence* of bugs, not their *absence***
- **We can test our code like this:**
  - **Automatic tooling**
    - **We use programs that try to discover code paths that result in race conditions or deadlocks, such as JChord**
  - **Bulk updates**
    - **We call the methods of a class many times, hoping to end in some expected state**
  - **Single updates, with time delays**
    - **We call the methods only once, but insert delays that guarantee a race condition or deadlock will happen**

Javaspecialists.eu

JavaSpecialists.EU

Sample Code: BankAccount

- **In this chapter, we will test the following class**

```
public class BankAccount {
  private int balance;
  public BankAccount(int balance) {
    this.balance = balance;
  }
  public void deposit(int amount) {
    balance += amount;
  }
  public void withdraw(int amount) {
    deposit(-amount);
  }
  public int getBalance() { return balance; }
  public boolean transferTo(BankAccount other, int amount) {
    if (amount > balance) return false;
    withdraw(amount);
    other.deposit(amount);
    return true;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

A Simple Race Condition: Bank Account

- **We call deposit(100) and withdraw(100) a million times from 16 threads**

Javaspecialists.eu

JavaSpecialists.EU

A Simple Race Condition: Bank Account

- **Number of race conditions detected**
  - **Single-core machine: 3 (0.000009375%)**
  - **Dual-core Macbook Pro: 48361 (0.15%)**
  - **8-core Server: 60425 (0.18%)**
- **The chance of detecting a race condition in a normal system is very small**
  - **Faster systems with more cores exhibit the bugs more frequently**
  - **But the same field needs to be accessed by lots of threads**
- **Context switching happens at a safepoint**
  - **On a single-core machine, it is heavily biased to switch at a method boundary**

Javaspecialists.eu

JavaSpecialists.EU

Types of Concurrent Tests

- **Correctness**
  - **Race Conditions**
    - **Class' behavior conforms to its specification**
    - **Check that invariants are not broken**
  - **Liveness**
    - **Neither deadlocks nor livelocks are possible**
- **Performance**
  - **We test performance meets requirements**
    - ***Throughput*: Rate at which some concurrent tasks is completed**
    - ***Latency*: Delay between a request for and completion of action**
    - ***Scalability*: How adding more CPUs increases throughput**

Javaspecialists.eu

JavaSpecialists.EU

9.1: Testing For Correctness

Testing Concurrent Programs

JavaSpecialists.EU

9.1: Testing For Correctness

- **We want to test that our class is free of data races, visibility issues, deadlocks and livelocks**
- **This is in addition to standard unit tests that check invariants and pre-/post-conditions**

Javaspecialists.eu

JavaSpecialists.EU

Automatic Tooling

- We use tooling to find code paths leading to data races
- Google's JChord
  - https://bitbucket.org/pag-lab/jchord
- ThreadSafe by Contemplate
  - http://www.contemplateltd.com/threadsafe

Javaspecialists.eu

JavaSpecialists.EU

Tools That Detect Common Bugs

- **IntelliJ IDEA allows us to analyze our codebase and find common programming errors**
  - **It even allows us to correct a lot of mistakes automatically**
  - **http://www.jetbrains.com/idea/**
- **FindBugs is an excellent tool for finding a lot of mistakes**
  - **http://findbugs.sourceforge.net/**

Javaspecialists.eu

JavaSpecialists.EU

JChord

- **JChord needs a class with a main(String...) method that calls the methods from multiple threads, for example:**

```
public class BankAccountAutomaticTest {
  public static void main(String... args) {
    BankAccount account = new BankAccount(1000);
    Runnable depositWithdraw = () -> {
      account.deposit(100);
      account.withdraw(100);
    };
    new Thread(depositWithdraw).start();
    new Thread(depositWithdraw).start();
    System.out.println(account.getBalance());
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Running JChord

- **To run JChord, download the source release**
  - **Our work directory needs to contain a chord.properties file**

```
chord.main.class=BankAccountAutomaticTest
chord.class.path=classes
chord.src.path=src
```

  - **We then inspect our code with the JChord ANT file**

```
ant -Dchord.work.dir=banking -Dchord.run.analyses=datarace-java run
```

  - **Generates a file dataraces_by_fld.html in chord_output directory**
    - **In our case, it shows that there is a race condition on balance field**

Datarace Reports (Grouped By Field)		
Details	Trace 1	
	Thread	Memory Access
1. Dataraces on eu.javaspecialists.course.concurrency.ch12_testing_concurrent_programs.BankAccount.balance		

Javaspecialists.eu

JavaSpecialists.EU

JChord Quirks

- **We might see *a lot of* false positives or *benign* races**
  - **In the Java2D Demo, only 14,530 lines, but found 131,184 data races!**
- **Needs a lot of memory, might need 64-bit machine**
  - **Java2D demo needed at least 4GB to complete**
- **It might take a long time without any meaningful results**
  - **BankAccountAutomaticTest took 43s to complete**
  - **Java2D Demo used 7m 23s**
- **JChord project seems to have stalled**

Javaspecialists.eu

JavaSpecialists.EU

Bulk Updates

- **After calling all methods many times, we check end state**

```
public class BankAccountBasicBulkUpdateTest {
  @Test
  public void testBulkUpdates() throws InterruptedException {
    ExecutorService pool = Executors.newFixedThreadPool(2);
    BankAccount account = new BankAccount(1000);
    Runnable depositWithdraw = () -> {
     for (int i = 0; i < 100_000_000; i++) {
        account.deposit(100);
        account.withdraw(100);
      }
    };
    pool.submit(depositWithdraw);
    pool.submit(depositWithdraw);
    pool.shutdown();
    assertTrue(pool.awaitTermination(1, TimeUnit.MINUTES));
    assertEquals(1000, account.getBalance());
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Server Hotspot Interference With Test

- **We expected the bulk update test to always fail**
- **However, the Server HotSpot Compiler is clever**
  - **It inlines the calls to deposit(100) and withdraw(100)**

```
account.balance += 100;
account.balance += -100;
```

  - **This is then further optimized to**

```
account.balance += 0;
```

  - **And then eliminated completely as dead code**

Test	Time elapsed	Usage Delta	Usage Before	Usage After	Results
testBulkUpdates	0.013 s	880 Kb	4,869 Kb	5,749 Kb	Passed

Tests Passed: 1 passed

Total time: 0.013 s

Javaspecialists.eu

JavaSpecialists.EU

Controlling Hotspot And Just-In-Time Compiling

- **Several switches affect HotSpot**
  - **-server is default on 64-bit or unix based JVMs**
  - **-client default on 32-bit non-unix JVMs, e.g. Windows**
    - **Need to sometimes use -d32 to enable 32-bit JVM**
- **We can also control how the JIT should work**
  - **-Xmixed (default)**
    - **The code is optimized and compiled according to profiling data**
  - **-Xint**
    - **The code is always run in *interpreted mode***
  - **-Xcomp**
    - **The code is always compiled, but not optimized by profiling data**

Javaspecialists.eu

JavaSpecialists.EU

Turning Off Optimizations

- **We can run our test with the -client and -d32 flags**
  - **This reduces the risk of our test being eliminated as dead code**

Test	Time elapsed	Usage Delta	Usage Before	Usage After	Results
testBulkUpdates	0.404 s	-3,042 Kb	4,141 Kb	1,099 Kb	Assertion

Tests Failed: 0 passed, 1 failed
Total time: 0.404 s

- **Or use -Xcomp flag to compile all the classes up front**
  - **It does not use the profiling information gathered by HotSpot**

Test	Time elapsed	Usage Delta	Usage Before	Usage After	Results
testBulkUpdates	2.424 s	1,325 Kb	5,760 Kb	7,085 Kb	Assertion

Tests Failed: 0 passed, 1 failed
Total time: 2.424 s

Javaspecialists.eu

JavaSpecialists.EU

Randomizing Bulk Operations

- **We can deposit and withdraw random amounts**
  - **As long as we have enough random amounts, the HotSpot compiler is not clever enough to eliminate our code**
- **Step 1: Generate random ints**

```
private int[] generateRandomInts() {
  return ThreadLocalRandom.current().ints(500_000).toArray();
}
```

- **Step 2: Call deposit and withdraw with random values, making sure you end up at the same original balance**

Javaspecialists.eu

JavaSpecialists.EU

```
@Test
public void testRandomUpdates() throws InterruptedException {
  ExecutorService pool = Executors.newFixedThreadPool(2);
  CountDownLatch latch = new CountDownLatch(2);
  BankAccount account = new BankAccount(1000);
  Runnable depositWithdraw = () -> {
    int[] amounts = generateRandomInts();
    latch.countDown();
    try { latch.await(); }
    catch (InterruptedException e) { return; }
    for (int i = 0; i < 10; i++) {
      for (int j = 0; j < amounts.length; j+=2) {
        account.deposit(amounts[j]);
        account.withdraw(amounts[j+1]);
      }
      for (int j = 0; j < amounts.length; j+=2) {
        account.withdraw(amounts[j]);
        account.deposit(amounts[j+1]);
      }
    }
  };
  pool.submit(depositWithdraw);
  pool.submit(depositWithdraw);
  pool.shutdown();
  assertTrue(pool.awaitTermination(1, TimeUnit.MINUTES));
  assertEquals(1000, account.getBalance());
}
```

Javaspecialists.eu

JavaSpecialists.EU

Results Of Our Randomization

- **With our random amounts deposited and withdrawn, the code is not recognized as having no effect**
  - **Server HotSpot runs the code and we see the race condition**

Test	Time elapsed	Usage Delta	Usage Before	Usage After	Results
testRandomUpdates	0.106 s	5,231 Kb	4,867 Kb	10,098 Kb	Assertion

Tests Failed: 0 passed, 1 failed

Total time: 0.106 s

Javaspecialists.eu

JavaSpecialists.EU

Testing Visibility Of Field

- **In our bank account, we can also test getBalance()**

```
public class BankAccountVisibilityTest {
  @Test
  public void testGetBalance() throws InterruptedException {
    BankAccount account = new BankAccount(1000);
    ExecutorService pool = Executors.newCachedThreadPool();
    pool.submit(() -> {
      while (account.getBalance() == 1000) ;
    });
    Thread.sleep(100);
    account.deposit(100);
    pool.shutdown();
    assertTrue("updates not visible",
        pool.awaitTermination(1, TimeUnit.SECONDS));
    assertEquals(1100, account.getBalance());
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Visibility Testing And Hotspot Optimizations

- **The Server HotSpot compiler will typically inline the getBalance() method call and cache the value locally**

Test	Time elapsed	Usage Delta	Usage Before	Usage After	Results
testGetBalance	1.135 s	440 Kb	4,420 Kb	4,861 Kb	Assertion

Tests Failed: 0 passed, 1 failed
Total time: 1.135 s

- **Client HotSpot or running with -Xcomp will not fail**

Test	Time elapsed	Usage Delta	Usage Before	Usage After	Results
testGetBalance	0.217 s	881 Kb	5,318 Kb	6,199 Kb	Passed

Tests Passed: 1 passed
Total time: 0.217 s

- **We need to test with *both* Client *and* Server HotSpot**
  - **Maybe start several Jenkins instances with different configurations**

Javaspecialists.eu

JavaSpecialists.EU

Single Update, With Time Delays

- **Instead of repeatedly calling methods, hoping to discover a race condition, we can also force the error to happen**
  - **Similar to the way we tested for deadlock**
- **We rewrite our deposit method to pause at the exact time needed to cause a race condition**

```
public void deposit(int amount) {
  int localBalance = balance;
  localBalance += amount;
  timeDelayForTesting();
  balance = localBalance;
}

protected void timeDelayForTesting() {
  // default is to do nothing
}
```

Javaspecialists.eu

JavaSpecialists.EU

Pros And Cons Of Various Approaches

- None of the approaches can give us complete confidence
  - A tool like JChord might warn us of errors that can never happen in production
    - How can we fix 131184 race conditions in 14530 lines of code?
      - The code does run, although there might be bugs
  - Bulk updates can be optimized away by the Server HotSpot
    - How do we know that we have called the methods often enough?
  - A single update with a delay timer means we have to know that we have a race condition in the first place
    - This is really only useful for writing regression tests
    - Also to *prove* that a race condition exists (in someone else's code)
      - It can help to give confidence that the race condition is repaired

Javaspecialists.eu

JavaSpecialists.EU

Repairing the BankAccount

- **Our first attempt to fix the problem: add synchronized**

```
public class BankAccount {
  private int balance;
  public BankAccount(int balance) {
    this.balance = balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
  public void withdraw(int amount) {
    deposit(-amount);
  }
  public int getBalance() { return balance; }
  public boolean transferTo(BankAccount other, int amount) {
    if (amount > balance) return false;
    withdraw(amount);
    other.deposit(amount);
    return true;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Tests Pass / Fail

- **JChord fails**
  - **It recognizes that getBalance() is read from a different thread to the one calling deposit()**
- **IntelliJ IDEA fails**
  - **Name: private field int balance**
  - **Problem synopsis: Field balance is accessed in both synchronized and unsynchronized contexts at line 6**
- **Bulk update tests pass**
- **Single update test passes**
- **BankAccountVisibilityTest fails**

Javaspecialists.eu

JavaSpecialists.EU

Let's Synchronize *All* The Methods

```
public class BankAccount {
  private int balance;
  public BankAccount(int balance) {
    this.balance = balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
  public synchronized void withdraw(int amount) {
    deposit(-amount);
  }
  public synchronized int getBalance() { return balance; }
  public synchronized boolean transferTo(
      BankAccount other, int amount) {
    if (amount > balance) return false;
    withdraw(amount);
    other.deposit(amount);
    return true;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Tests Pass / Fail

- JChord passes
- IntelliJ IDEA passes
- Bulk update tests pass
- Single update test passes
- Visibility test passes

Javaspecialists.eu

JavaSpecialists.EU

Have Another Look at BankAccount

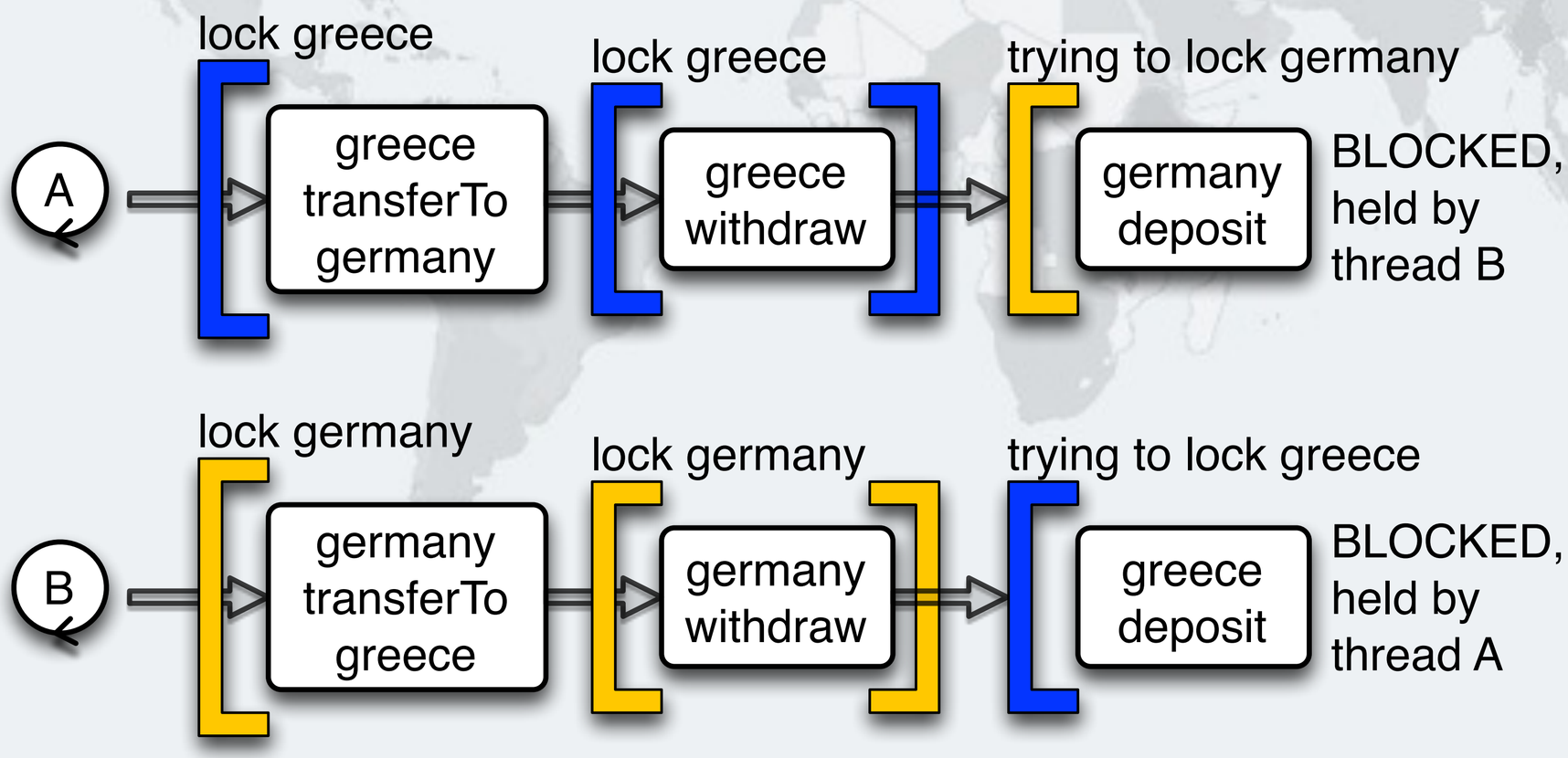
```
public class BankAccount {
  private int balance;
  public BankAccount(int balance) {
    this.balance = balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
  public synchronized void withdraw(int amount) {
    deposit(-amount);
  }
  public synchronized int getBalance() { return balance; }
  public synchronized boolean transferTo(
      BankAccount other, int amount) {
    if (amount > balance) return false;
    withdraw(amount);
    other.deposit(amount);
    return true;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Deadlock Is A Liveness Problem

- **A program that causes a deadlock is not correct**
  - **Performance is about timing information, not liveness**



Javaspecialists.eu

JavaSpecialists.EU

Testing For Deadlocks With Bulk Updates

```
public class BankAccountBasicBulkTransferTest {
  @Test
  public void testBulkTransfers() throws InterruptedException {
    ExecutorService pool = Executors.newCachedThreadPool();
    BankAccount greece = new BankAccount(1_000_000);
    BankAccount germany = new BankAccount(1_000_000);
    pool.submit(() -> {
     for (int i = 0; i < 1_000_000; i++)
        greece.transferTo(germany, 1);
    });
    pool.submit(() -> {
     for (int i = 0; i < 1_000_000; i++)
        germany.transferTo(greece, 1);
    });
    pool.shutdown();
    assertTrue("possible deadlock",
      pool.awaitTermination(10, TimeUnit.SECONDS));
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Testing For Deadlocks

- We might find deadlocks with tools like JChord
  - Or in production
  - Or during normal unit testing
- Deadlocks can be solved by changing our system to be single-threaded
  - This is how Swing avoids deadlocks
- Run an automatic deadlock detector on critical code
  - http://www.javaspecialists.eu/archive/Issue160.html
  - Warning: not all deadlocks can be found automatically

Javaspecialists.eu

JavaSpecialists.EU

9.2: Testing For Performance

Testing Concurrent Programs

JavaSpecialists.EU

9.2: Testing For Performance

- We want to test that our code completes within the expected amount of time
  - Microbenchmarking, more about this in chapter 15
- For concurrency, we also want to test throughput and utilization of our cores
- In Java the garbage collector runs interleaved with code
  - Sometimes hard to tell what our cores are doing
  - More about this in chapter 14

Javaspecialists.eu

JavaSpecialists.EU

```
public class MathRandomTest {
  private final static int THREADS = 8;

  private static double someRandomValue() {
    return Math.random();
  }

  private static void callRandomLots() {
    double total = 0;
    for (int i = 0; i < 10_000_000; i++) {
      total += someRandomValue();
    }
    System.out.println(total); // ensures total can't be eliminated
  }

  public static void main(String... args) {
    ExecutorService pool = Executors.newCachedThreadPool();
    for (int i = 0; i < THREADS; i++) {
      pool.submit(MathRandomTest::callRandomLots);
    }
    pool.shutdown();
  }
}
```

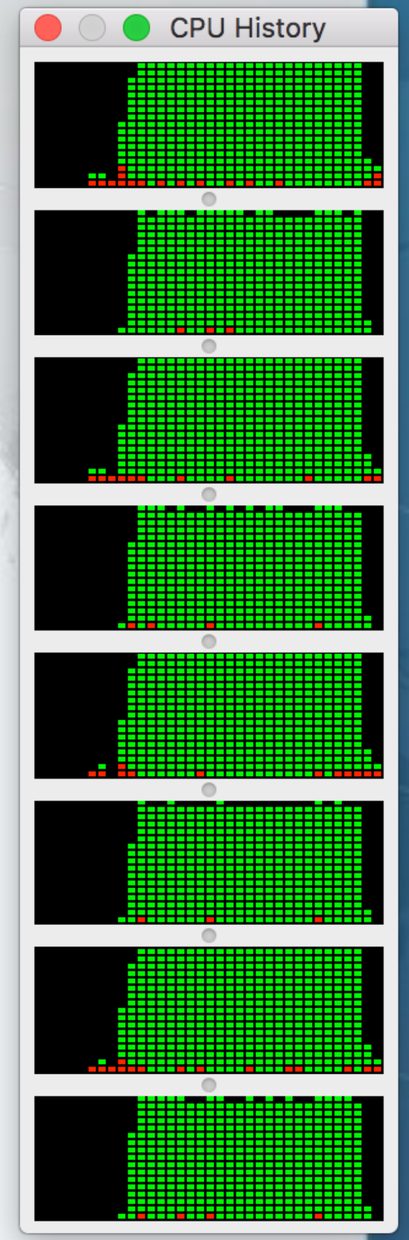
Javaspecialists.eu

JavaSpecialists.EU

Run Results with 1-4-2 MacBookPro

- **With 4 cores and 8 hardware threads, it finished in 22 seconds**
- **How long do you think it will take if we make someRandomValue() synchronized?**

```
private synchronized static double someRandomValue() {
  return Math.random();
}
```

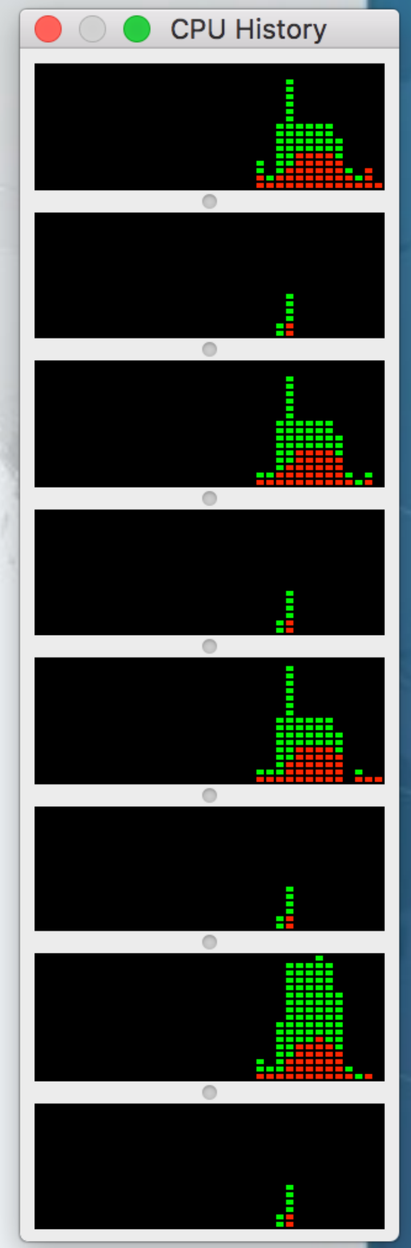


Javaspecialists.eu

JavaSpecialists.EU

With someRandomValue() synchronized

- **Making the method synchronized made it faster!**
  - **Now completes in 6 seconds**



Javaspecialists.eu

JavaSpecialists.EU

Even Better - Thread Confined

- **ThreadLocalRandom - thread confined random numbers**

```
private static double someRandomValue() {
  return ThreadLocalRandom.current().nextDouble();
}
```

- **Code completed in 39 milliseconds**
- **We will talk more about these topics in the next chapter**

Javaspecialists.eu

JavaSpecialists.EU

9: Exercises

Testing Concurrent Programs

JavaSpecialists.EU

Exercise 9.1: HandoverQueue test

- **Devise test for our half-baked queue**
  - **First some single-threaded correctness tests**
    - **Add 10 elements with offer() and confirm with poll() that they are returned in the same order**
  - **Then some multi-threaded tests, using runConcurrently()**
    - **We should run the code in several threads, 1_000_000 calling**
      - **q.offer(somevalue); // check the element was added**
      - **q.size(); // check the result is within range**
      - **q.peek(); // check the value is what you would expect**
      - **q.poll(); // check the value is what you would expect**

Javaspecialists.eu

JavaSpecialists.EU

Exercise 9.2: LinkedTransferQueue Test

- **We discovered a bug in the LinkedTransferQueue**
  - **Your job is to reproduce the test**
  - **We expect our test to fail for LinkedList, but pass for ConcurrentLinkedQueue and LinkedBlockingQueue**
    - **LinkedTransferQueue should also work, but it's broken**
- **The test will do the following:**
  - **One thread will repeatedly convert the collection to a stream and print out each element**
    - **Note: you need to use the stream() method and *then* forEach()**
  - **Another thread will repeatedly add and remove alerts**

Javaspecialists.eu

JavaSpecialists.EU

9: Testing Concurrent Programs

Questions?

10: Performance And Scalability

Making your concurrent code really fast!

JavaSpecialists.EU

10: Performance And Scalability

- **Remember that Intel predicted 10GHz chips by 2011**
  - **http://www.zdnet.com/news/taking-chips-to-10ghz-and-beyond/96055**
- **Intel Core i7 990x hit the market early 2011**
  - **3.46GHz clock stretching up to 3.73 GHz in turbo mode**
  - **6 processing cores**
  - **Running in parallel, we get 22GHz of processing power!**
- **Japanese 'K' Computer June 2011**
  - **548,352 cores from 68,544 2GHz 8-Core SPARC64 VIIIfx processors**
  - **8.2 petaFLOPS**
    - **8 200 000 000 000 000 floating point operations per second**
    - **Intel 8087 was 30 000 FLOPS, 273 billion times slower**

Javaspecialists.eu

JavaSpecialists.EU

Which Is Faster, "K" Or One Simple Intel "I7"

- **Intel i7 has a total speed of 3.73GHz x 6 = 22 GHz**
- **K has a total speed of 2GHz x 548352 = 1096704 GHz**
- **Which is faster?**
  - **If we can parallelize our algorithm, then K is 50,000 times faster**
  - **But if it has to run in serial, then one Intel i7 is almost twice as fast**

Javaspecialists.eu

JavaSpecialists.EU

10.1: Thinking About Performance

Performance and Scalability

JavaSpecialists.EU

10.1: Thinking About Performance

- **We want to utilize all our CPUs with application code**
  - **Having too many serial sections means not all CPUs are utilized**
    - **Took 51 seconds to complete**

  - **Too much locking means we are busy with system code (red)**
    - **Took 745 seconds to complete**

Javaspecialists.eu

JavaSpecialists.EU

Performance Vs Scalability

- **We can measure application performance in many ways**
  - **Latency: How fast *one* unit of work runs**
  - **Throughput: How many units of work can be done per unit of time**
- **A system is *scalable* when the throughput increases with more computing resources such as CPUs or faster IO**

Javaspecialists.eu

JavaSpecialists.EU

How Fast Vs How Much

- **In traditional performance optimizations, we try to make our code run *faster***
  - e.g. change $O(n^2)$ algorithm to $O(n \log n)$, or caching old results
- **When we tune for scalability, we want to parallelize work**
  - Thus by adding more CPUs, we can complete more work
- **Many code tricks that make code run *faster* also make it less scalable**
  - For example, maintaining previous results adds synchronization

Javaspecialists.eu

JavaSpecialists.EU

Evaluating Performance Tradeoffs

- **Before starting to tune any system, find out what the performance requirements are**
  - **Are the requirements low latency? Or high throughput?**
  - **How many clients do you want to support at the same time?**
  - **What is the maximum wait time for your clients?**
- **Do not extrapolate**
  - **Complexity is seldom linear, especially when we include unexpected side effects!**

Javaspecialists.eu

JavaSpecialists.EU

10.2: Amdahl's And Little's Laws

Performance and Scalability

JavaSpecialists.EU

10.2: Amdahl's Law

- **Some problems can be solved faster by parallelizing portions of it**
  - **If N is the number of processors and F the portion of code that has to be run in serial, then the speedup is**

$$\text{Speedup} \leq \frac{1}{F + \frac{(1 - F)}{N}}$$

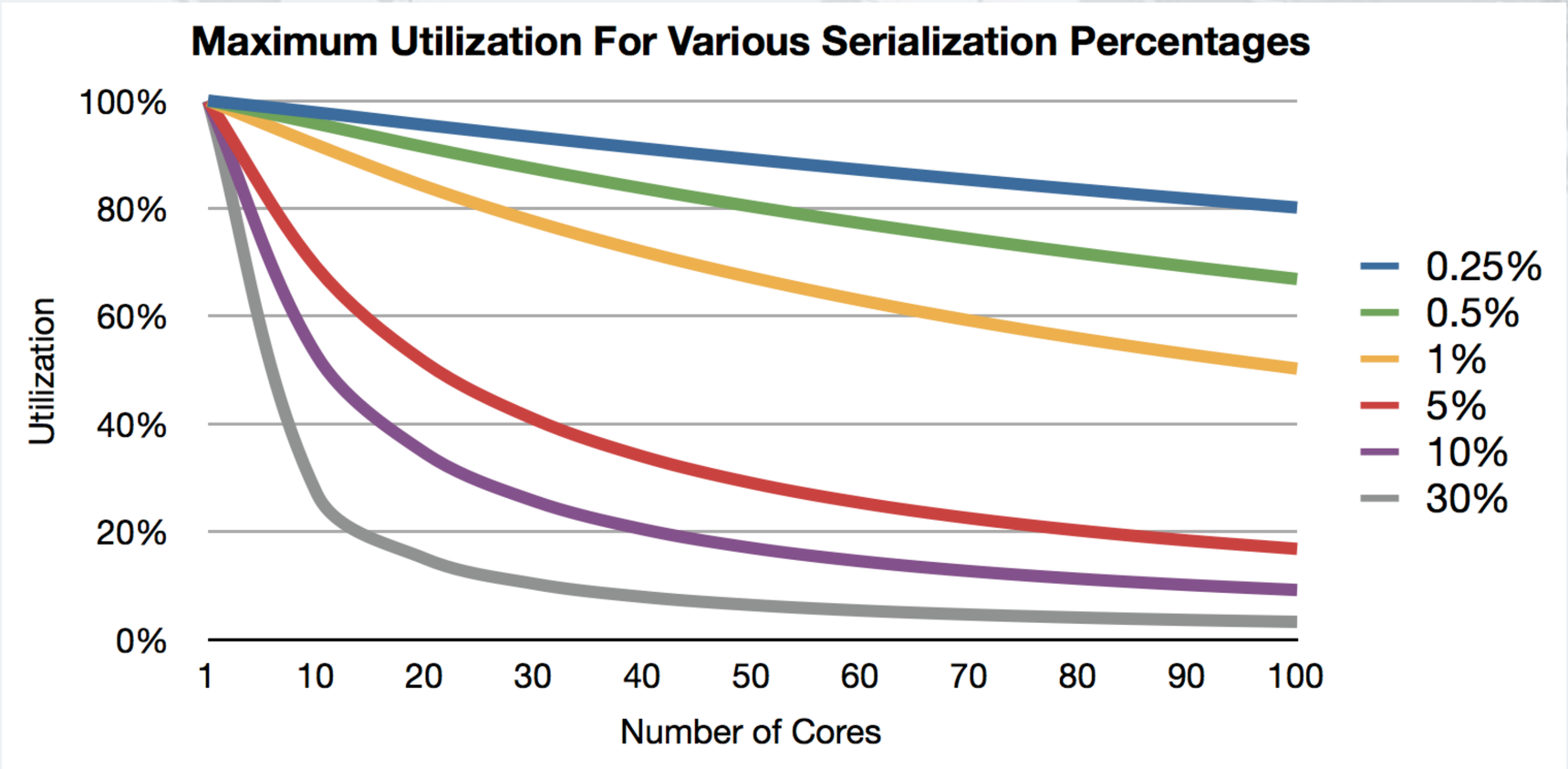
  - **More workers can *plant* watermelons faster and *harvest* them faster**
  - **But no amount of additional workers can make them *grow* any faster**
  - **The *growing* is our serial section**

Javaspecialists.eu

JavaSpecialists.EU

Utilization According To Amdahl

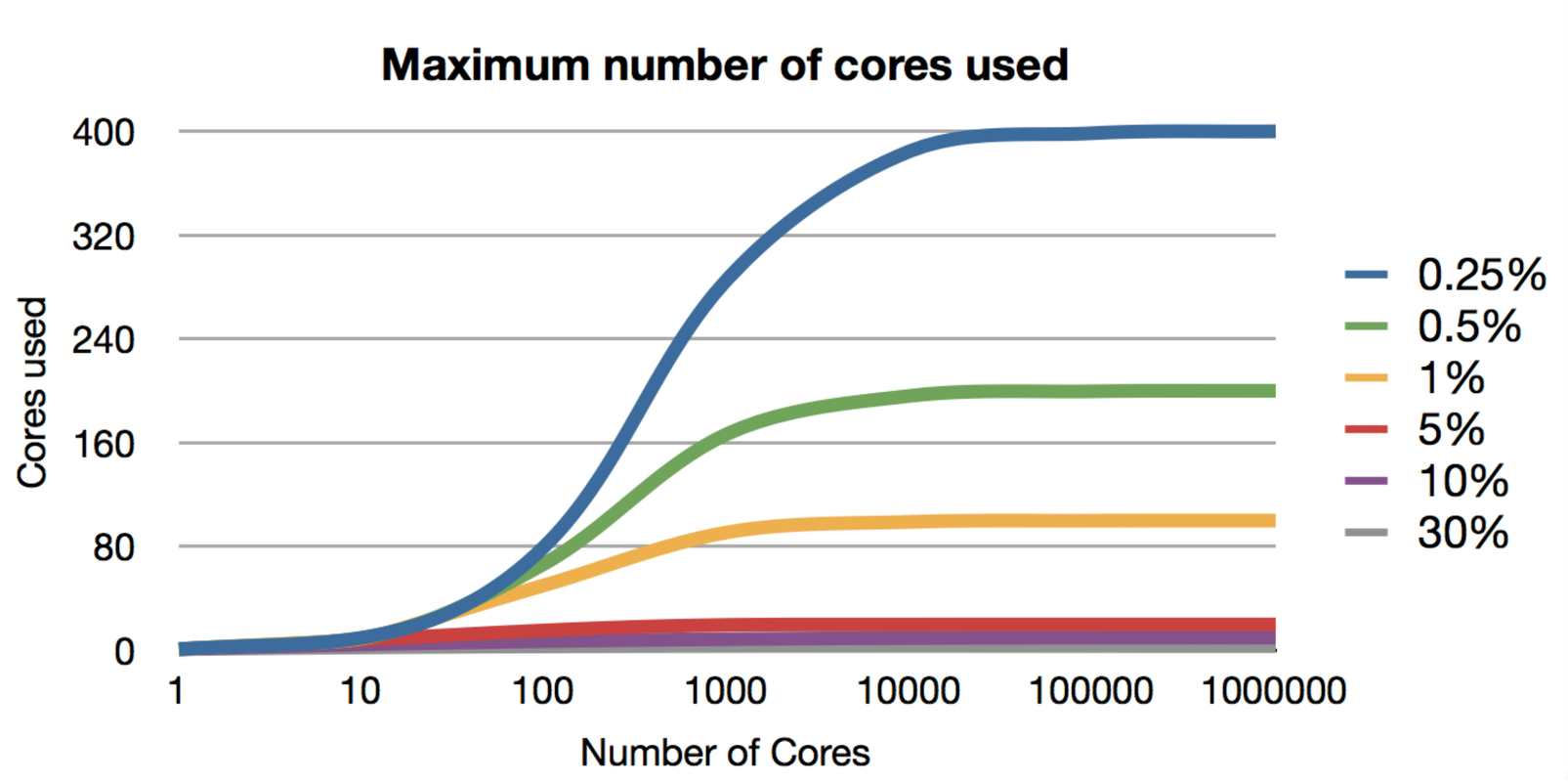
- **Even with a small section needing to run in serial, we are limited to how we can speed up our program**



Javaspecialists.eu
JavaSpecialists.EU

Maximum Number Of Cores That Can Be Used

- **The number of cores utilized is calculated as 1/F as N tends towards infinity**
  - **For a serial portion 0.25%, we can never use more than 400 cores**



Javaspecialists.eu
JavaSpecialists.EU

Problems With Amdahl's Law In Practice

- We cannot predict what portion of code is running serial
  - We can explain why our system does not scale, but not make accurate predictions
- Amdahl's law shows what happens with infinite cores
  - Not useful in practice
- It assumes the amount of data remains the same
  - Usually with more utilization, the data also is more

Javaspecialists.eu

JavaSpecialists.EU

Little's Law

- A better law for modeling real systems is Little's Law
  - The long-term average number of customers in a *stable system* L is equal to the long-term average effective arrival rate, λ, multiplied by the average time a customer spends in the system, W
    - $L = \lambda W$
  - Throughput is the inverse of service time
- If your service time is 1ms and you have one server, then the maximum throughput is 1000 transactions per second
  - To increase throughput, add more servers or decrease service time
- Good paper showing how the law can be used in practice
  - http://ie.technion.ac.il/serveng/Lectures/Little.pdf

Javaspecialists.eu
JavaSpecialists.EU

Little's Law In Business

- **In USA, restaurants try to limit the time spent by patrons**
  - **As you walk in, a greeter shows you to the table - load balancer**
  - **Next someone comes with some water**
  - **Another person comes to take your order, and yet another brings it**
  - **Everyone is optimized to bring your order as quickly as possible**
    - **And get you the cheque even faster**
  - **In Crete, you can sit in a restaurant all day drinking one coffee**
- **Smaller W ⇒ higher throughput of customers ⇒ more $$$**

Javaspecialists.eu

JavaSpecialists.EU

Threading And Little's Law

- **We can even apply Little's Law to a section of our code**
  - **A synchronized section only lets one thread in at a time**
    - **$\lambda = L/W$**
    - **L is 1, since the code is synchronized**
    - **W is how long we are holding the lock for**
    - **If W takes 20ms, maximum throughput is 1/0.02 = 50 calls per second**
  - **No matter how many CPUs are in system, we are restricted by Little!**

Javaspecialists.eu

JavaSpecialists.EU

10.3: Costs Introduced By Threads

Performance and Scalability

JavaSpecialists.EU

10.3: Costs Introduced By Threads

- **Single-threaded programs do not have to synchronize, context switch or use locks to protect data**
- **Threads can give faster performance, but there is a cost**

Javaspecialists.eu

JavaSpecialists.EU

Context Switching Good And Bad

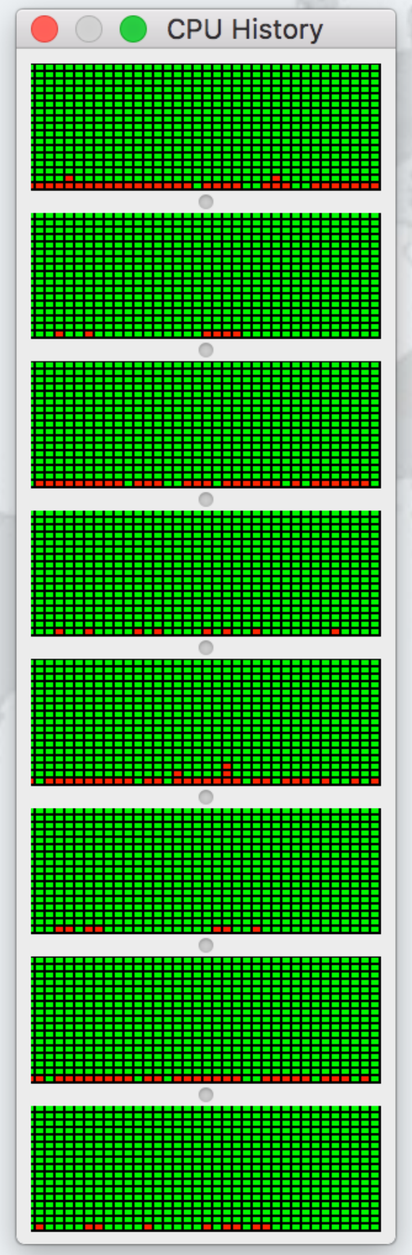
- In an *involuntary context switch* the time quantum is used up and the operating system swaps it out (good)
  - Can happen in a single clock cycle with hyperthreading
  - These types of context switches usually don't cause a lot of issues
  - Time quantum is typically about 100 ms
- In a *voluntary context switch* thread needs to block and gives up control (bad)
  - e.g. encounters a lock that is held by another thread or waits for IO
  - Can take thousands or tens of thousands of clock cycles
  - Typically seen by a lot of system time in CPU graph

Javaspecialists.eu

JavaSpecialists.EU

What Is Happening Here?

- **#TH 53/16**

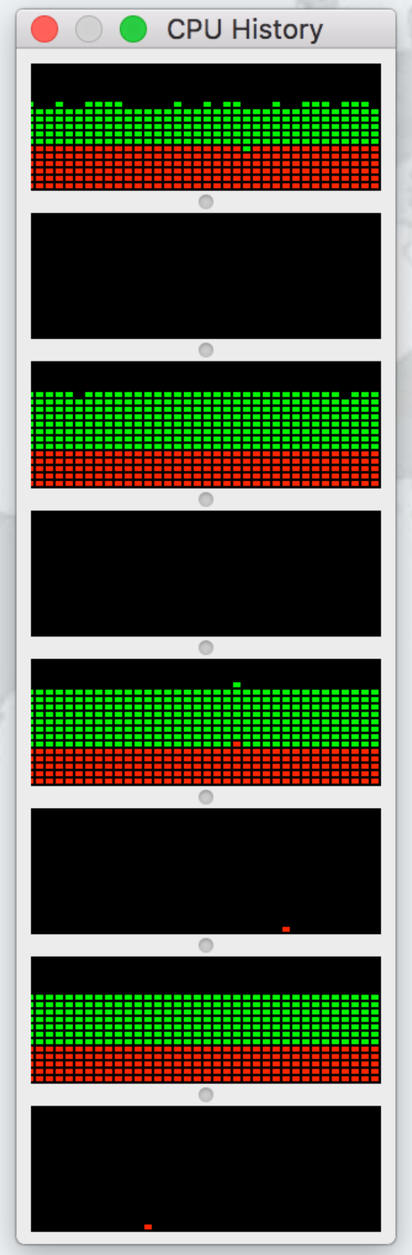


Javaspecialists.eu

JavaSpecialists.EU

What Is Happening Here?

- **#TH 55/2**

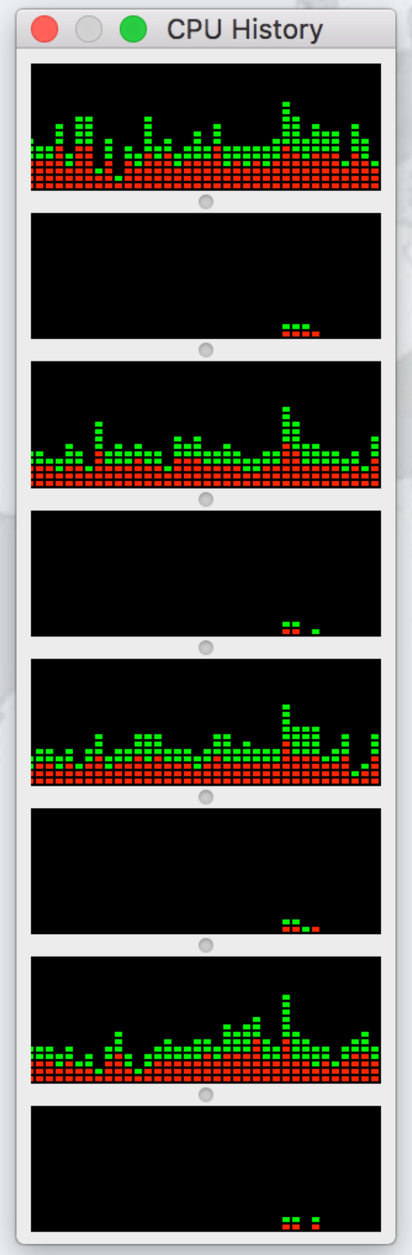


Javaspecialists.eu

JavaSpecialists.EU

What Is Happening Here?

- **#TH 37/1**



Javaspecialists.eu

JavaSpecialists.EU

What Is Happening Here?

- **#TH 53/16**

Javaspecialists.eu

JavaSpecialists.EU

Memory Synchronization

- **Java uses *memory barriers* to ensure that fields are flushed and caches invalidated**
  - **We use *volatile* and *synchronized* to place memory barriers**
  - **Memory barriers slow us down**
  - **They also limit how our code can be optimized**
- **Uncontended locks can be optimized away by HotSpot**
  - **Escape Analysis can determine that the object is never used outside of the method**
    - **The object can then be constructed on the stack or in the registers**
    - **Locking can be removed**
  - **Don't worry about uncontended locks**

Javaspecialists.eu

JavaSpecialists.EU

10.4: Reducing Lock Contention

Performance and Scalability

JavaSpecialists.EU

10.4: Reducing Lock Contention

- **The biggest threat to scalability is the exclusive lock**
  - **Amdahl's Law shows that even a small section of serial code will limit the amount of speedup we can achieve**
  - **And with Little's Law L=λW, the serial section always has L=1**
    - **Thus λ=1/W**
- **Our aim would need to be to reduce contended locks**
  - **But of course ensuring that the code is still *safe***

Safety First

Safety First

Safety First

Javaspecialists.eu

JavaSpecialists.EU

How To Reduce Lock Contention

- **We have three main ways to reduce contention**
  - **Reduce the duration that locks are held**
  - **Reduce frequency with which locks are requested**
  - **Replace exclusive locks with non-locking thread-safe mechanisms**
- **Of course, we can also avoid synchronization by making our data thread confined or stack confined**

Javaspecialists.eu

JavaSpecialists.EU

How To Find Lock Contention

- **The JVM has hooks built in for finding contention on synchronized locks**
  - **A number of tools such as YourKit use this to find contention**
- **Finding contention on Java 5 ReentrantLock needs work**
  - **Take a number of thread dumps with `jstack -l`**
  - **Look for threads that are WAITING on the same lock**
- **Also look for threads waiting on semaphores**
  - **Have the permits perhaps been lost?**

Javaspecialists.eu

JavaSpecialists.EU

Narrowing Lock Scope ("Get In, Get Out")

- **We should always hold locks for as short as possible**
  - **Our performance is limited by how long we hold the locks**
    - **If the lock is held for 2 ms, throughput is maximum of 500 tx/s**
    - **If it is held for only 1ms, throughput can increase to 1000 tx/s**

Javaspecialists.eu

JavaSpecialists.EU

AttributeStore With a Long Critical Section

- **We are locking the entire matching method, even the regular expression pattern matching**

```
@ThreadSafe
public class AttributeStore {
  @GuardedBy("this")
  private final Map<String, String> attributes =
    new HashMap<>();
  public synchronized boolean userLocationMatches(
      String name, String regexp) {
    String key = "users." + name + ".location";
    String location = attributes.get(key);
    if (location == null)
      return false;
    else
      return Pattern.matches(regexp, location);
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

A Better Way to Write "userLocationMatches"

- **Faster is to lock only the portion that is necessary**
  - **In addition, we are encapsulating the lock by using a private field**

```
public boolean userLocationMatches(
    String name, String regexp) {
  String key = "users." + name + ".location";
  String location;
  synchronized (attributes) {
    location = attributes.get(key);
  }
  if (location == null)
    return false;
  else
    return Pattern.matches(regexp, location);
}
```

Javaspecialists.eu

JavaSpecialists.EU

Or Use a ConcurrentHashMap

- **The ConcurrentHashMap is non-blocking on reads**
  - **The serial section is reduced to just a memory barrier via volatile**

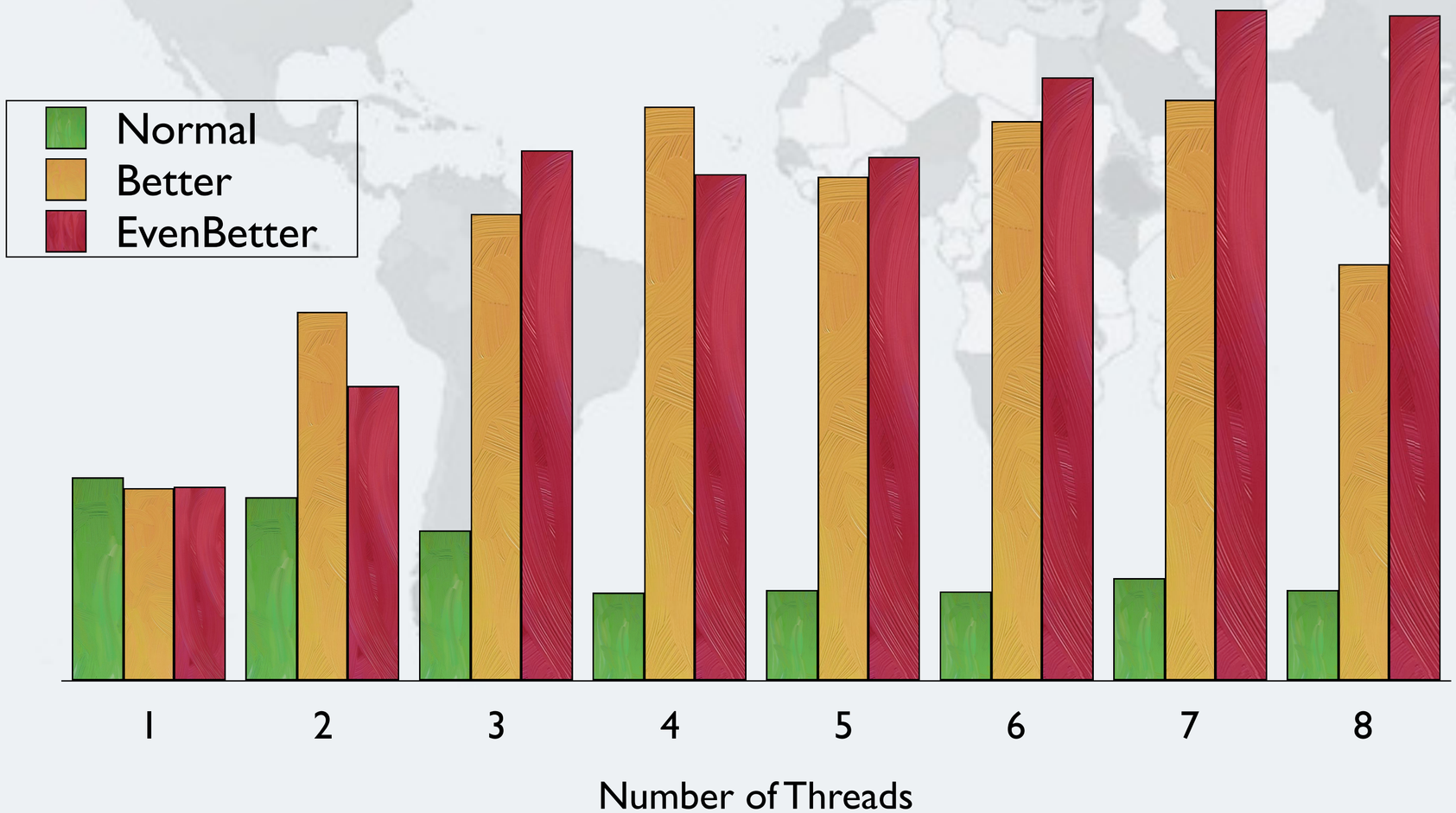
```
@ThreadSafe
public class EvenBetterAttributeStore {
  private final Map<String, String> attributes =
      new ConcurrentHashMap<>();

  public boolean userLocationMatches(
      String name, String regexp) {
    String key = "users." + name + ".location";
    String location = attributes.get(key);
    if (location == null)
      return false;
    else
      return Pattern.matches(regexp, location);
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

AttributeStore Performance Comparisons

- **Throughput for a million lookups on an 8-core machine**



Javaspecialists.eu

JavaSpecialists.EU

Reducing Lock Granularity

- **We can use *lock splitting* or *lock striping* to reduce contention**
- **Lock splitting**
  - **locking unrelated parts of the class with different locks**
  - **e.g. LinkedBlockingQueue, ConcurrentLinkedQueue**
- **Lock striping**
  - **Breaking up one piece of data into many pieces to reduce probability that two threads need to write to the same piece**
  - **e.g. LongAdder, ConcurrentHashMap**

Javaspecialists.eu

JavaSpecialists.EU

LinkedBlockingQueue in JDK Lock Splitting

```
public class LinkedBlockingQueue<E> extends AbstractQueue<E>
    implements BlockingQueue<E>, java.io.Serializable {
  private final AtomicInteger count = new AtomicInteger();
  transient Node<E> head;
  /** Tail of linked list. Invariant: last.next == null */
  private transient Node<E> last;

  /** Lock held by take, poll, etc */
  private final ReentrantLock takeLock = new ReentrantLock();

  /** Wait queue for waiting takes */
  private final Condition notEmpty = takeLock.newCondition();

  /** Lock held by put, offer, etc */
  private final ReentrantLock putLock = new ReentrantLock();

  /** Wait queue for waiting puts */
  private final Condition notFull = putLock.newCondition();

// ...
```

Javaspecialists.eu

JavaSpecialists.EU

ConcurrentLinkedQueue in JDK Lock Splitting

- **General purpose unbounded MPMC queue**
- **Separate pointers for head and tail**
- **offer() and poll() update independent fields using CompareAndSet (CAS)**

```
public class ConcurrentLinkedQueue<E>
    extends AbstractQueue<E>
    implements Queue<E>, java.io.Serializable {
  transient volatile Node<E> head;
  private transient volatile Node<E> tail;
```

- **Note: ConcurrentLinkedQueue does not maintain a count**
  - **size() method is O(n) - also not always correct**

Javaspecialists.eu

JavaSpecialists.EU

ConcurrentHashMap Lock Striping

- **Lots of segments instead of one large array**
  - **Pre Java 8 we had a fixed number of segments**
    - **Specified by the concurrencyLevel constructor parameter**
  - **Since Java 8, we have a tree structure of segments**
    - **Dynamically can change the number of segments as per need**
    - **Uses less memory for small tables**

Javaspecialists.eu

JavaSpecialists.EU

LongAdder Lock Striping

- **Stores values inside an array of cells**
  - **Threads can update *their* cell without causing contention**
  - **Each cell is "padded" to make sure it lives in its own cache line**
- **Results are *merged* when we call longValue()**

Javaspecialists.eu

JavaSpecialists.EU

Alternatives To Exclusive Locks

- We can use more scalable alternatives to exclusive locks
  - ReadWriteLock allows several threads to read at the same time but only for one to write
  - Some of the concurrent collections allow better scalability
    - They typically use a combination of volatile and compare-and-set
  - Immutable objects reduce the need for locking
  - Atomic fields provide volatile access and compare-and-set
- Contended fields based on compare-and-set can have worse performance due to too many retries

Javaspecialists.eu

JavaSpecialists.EU

Monitoring CPU Utilization

- We can use several tools for monitoring our system
  - Unix
    - vmstat for the overall system performance
    - mpstat to see what each processor is busy with
    - pidstat gives you information about each individual thread
  - Windows
    - taskmgr for summary view of CPU, memory, disk, network
      - Make sure to enable correct columns in task view
    - perfmon for detailed views of performance
  - Mac OS X
    - iStat for summary of performance
    - Activity monitor for detailed view

Javaspecialists.eu

JavaSpecialists.EU

Unix "mpstat"

- **We are running a test with 32 threads on 8-core system**
- **What is mpstat telling us?**

```
Linux 2.6.32-37-server _x86_64_ (8 CPU)

12:32:43 PM  CPU   %usr  %sys  %idle
12:32:44 PM  all  28.04  2.45  69.51
12:32:45 PM  all  26.61  2.46  70.93
12:32:46 PM  all  28.27  1.18  70.44
12:32:47 PM  all  25.80  3.20  71.01
12:32:48 PM  all  27.63  2.20  70.17
12:32:49 PM  all  26.33  1.77  71.90
12:32:50 PM  all  26.98  2.76  70.26
```

Javaspecialists.eu
JavaSpecialists.EU

Processors Are Idle Waiting For Work

- **We can see most of the cores are idle. Only 27% of the CPUs are busy, even though we have 32 threads on 8 cores.**

```
Linux 2.6.32-37-server _x86_64_ (8 CPU)

12:32:43 PM  CPU   %usr  %sys  %idle
12:32:44 PM  all  28.04  2.45  69.51
12:32:45 PM  all  26.61  2.46  70.93
12:32:46 PM  all  28.27  1.18  70.44
12:32:47 PM  all  25.80  3.20  71.01
12:32:48 PM  all  27.63  2.20  70.17
12:32:49 PM  all  26.33  1.77  71.90
12:32:50 PM  all  26.98  2.76  70.26
```

Javaspecialists.eu

JavaSpecialists.EU

Unix "vmstat"

- **Same test as before, do any values stand out?**

```
procs  -----------memory---------- -swap- --io-- --system-- ----cpu----
 r  b  swpd    free   buff   cache  si so bi  bo   in    cs us sy id wa
 3  0     0 2666092 223300 4388744   0  0  0  68 1506 56459 24  2 74  0
 3  0     0 2644168 223300 4388744   0  0  0   8 1298 61687 31  2 68  0
 3  0     0 2643668 223300 4388744   0  0  0   0 1296 60977 25  1 73  0
 4  0     0 2644064 223300 4388744   0  0  0  12 1311 59997 27  2 71  0
 2  0     0 2643660 223300 4388748   0  0  0   8 1423 68424 25  2 73  0
```

- **To make it easier, here is a quiet system**

```
procs -----------memory---------- -swap- --io-- --system-- ----cpu-----
 r  b swpd    free   buff  cache   si so bi  bo   in    cs us sy  id wa
 0  0    0 2661188 223524 4388964   0  0  0   0  228   212  0  0 100  0
 0  0    0 2660800 223524 4388968   0  0  0   0  135   141  0  0 100  0
 0  0    0 2660676 223524 4388968   0  0  0   0   83    83  0  0 100  0
 0  0    0 2660676 223524 4388968   0  0  0   0  103    91  0  0 100  0
 0  0    0 2660676 223524 4388968   0  0  0   0  170   157  0  0 100  0
```

Javaspecialists.eu

JavaSpecialists.EU

Unix "vmstat"

- **The runnable queue (procs/r) is non-zero, meaning we have too many threads for the number of CPUs**
  - **After all, we are trying to run 32 threads on 8 cores**
- **The context switching (system/cs) is large, telling us that threads are not using their time quantum**

procs		----------memory----------				-swap-		--io--		--system--		----cpu----			
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
3	0	0	2666092	223300	4388744	0	0	0	68	1506	56459	24	2	74	0
3	0	0	2644168	223300	4388744	0	0	0	8	1298	61687	31	2	68	0
3	0	0	2643668	223300	4388744	0	0	0	0	1296	60977	25	1	73	0
4	0	0	2644064	223300	4388744	0	0	0	12	1311	59997	27	2	71	0
2	0	0	2643660	223300	4388748	0	0	0	8	1423	68424	25	2	73	0

Javaspecialists.eu

JavaSpecialists.EU

Why Might the CPUs Not Be Fully Loaded?

- **There are several reasons the CPUs might not get hot**
  - **Insufficient load**
    - **The test data set might be too small**
    - **Our test script might not be adequately loading the system**
    - **Our test environment might not be powerful enough**
  - **I/O bound**
    - **If the application is disk-bound you will see a lot of disk io**
    - **Windows: perfmon or taskmgr, Unix: iostat or vmstat**
  - **Externally bound**
    - **We might be waiting for the database or a web service**
    - **Use a sampling profiler to see what our threads are waiting for**
  - **Lock contention - more next slide**

Javaspecialists.eu

JavaSpecialists.EU

How To Find "Hot Locks"

- **Profiling**
  - **A profiling tool like YourKit shows the most contended locks**
- **Thread dumps**
  - **A cheap way of finding "hot locks" is to take several thread dumps**
  - **A heavily contended lock will usually show up several times**

```
"pool-9-thread-2" prio=10 runnable
   java.lang.Thread.State: RUNNABLE
        at SynchronizedOuter.someMethod
        - locked <0x00000007555c3de8>
        at SynchronizedInnerOuterTest$2.callMethod
        at SynchronizedInnerOuterTest$2.run

"pool-9-thread-1" prio=10 waiting for monitor entry
   java.lang.Thread.State: BLOCKED
        at SynchronizedOuter.someMethod
        - locked <0x00000007555c3de8>
        at SynchronizedInnerOuterTest$2.callMethod
        at SynchronizedInnerOuterTest$2.run
```

Javaspecialists.eu

JavaSpecialists.EU

Hotspot Options For Lock Performance

- **We can control how HotSpot does locking**
  - **-XX:+DoEscapeAnalysis**
    - **Elides locks on objects that cannot escape**
  - **-XX:+EliminateLocks**
    - **Does lock coarsening using roach motel semantics**
  - **-XX:+UseBiasedLocking**
    - **Locks are assumed to be given to a single thread**
      - **This might have to be undone if another thread needs the lock**
      - **Can cause bad latency due to safe pointing**

Javaspecialists.eu

JavaSpecialists.EU

10: Performance And Scalability

Questions?

11: Explicit Locks

Java 5 locks for controlling synchronization

JavaSpecialists.EU

11: Explicit Locks

- **Java 5 added the Lock, based on Doug Lea's Synch**
  - **Implementation is the ReentrantLock**
- **This is *not* a replacement for intrinsic locks, but offers additional features**

Javaspecialists.eu

JavaSpecialists.EU

11.1: Lock And Reentrantlock

Explicit Locks

JavaSpecialists.EU

11.1: Lock And Reentrantlock

- **The Lock interface offers different ways of locking:**
  - **Unconditional, polled, timed and interruptible**

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit unit)
    throws InterruptedException;
  void unlock();
  Condition newCondition();
}
```

- **Lock implementations must have same memory-visibility semantics as intrinsic locks (synchronized)**

Javaspecialists.eu

JavaSpecialists.EU

Using The Explicit Lock

- **We have to call unlock() in a finally block**
  - **Every time, without exception**
    - **There are FindBugs detectors that will look for forgotten "unlocks"**

```
private final Lock lock = new ReentrantLock();

public void update() {
  lock.lock(); // this should be before try
  try {
    // update object state
    // catch exceptions and restore
    // invariants if necessary
  } finally {
    lock.unlock();
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Polled And Timed Lock Acquisition

- **Instead of unconditional lock, we can tryLock()**

```
if (lock.tryLock()) {
  try {
    balance = balance + amount;
  } finally {
    lock.unlock();
  }
} else {
  // alternative path
}
```

- **We can also try to lock for a specified period**

```
lock.tryLock(long, TimeUnit)
```

Javaspecialists.eu

JavaSpecialists.EU

Using Try-Lock To Avoid Deadlocks

- **Deadlocks happen when we lock multiple locks in different orders**
- **We can avoid this by using tryLock()**
  - **If we do not immediately get the lock, we sleep for a random time and then try again**
  - **It is important to let go of any locks we might hold, otherwise we can get a livelock instead of a deadlock**
- **Unsafe in OpenJDK had a tryMonitorEnter() method**
  - **Gave us the tryLock functionality with implicit locks**
  - **Deprecated in Java 8, removed in Java 9**
  - **http://www.javaspecialists.eu/archive/Issue194.html**

Javaspecialists.eu

JavaSpecialists.EU

Using tryLock() to Avoid Deadlocks

```
while (true) {
  fromAcct.lock.lock();
  try {
    if (toAcct.lock.tryLock()) {
      try {
        if (fromAcct.getBalance().compareTo(amount) < 0) {
          return false;
        } else {
          fromAcct.debit(amount);
          toAcct.credit(amount);
          return true;
        }
      } finally {
        toAcct.lock.unlock();
      }
    }
  } finally {
    fromAcct.lock.unlock();
  }
  LockSupport.parkNanos(System.nanoTime() & 0xffff);
}
```

Javaspecialists.eu

JavaSpecialists.EU

11.2: Performance Considerations

Explicit Locks

JavaSpecialists.EU

11.2: Performance Considerations

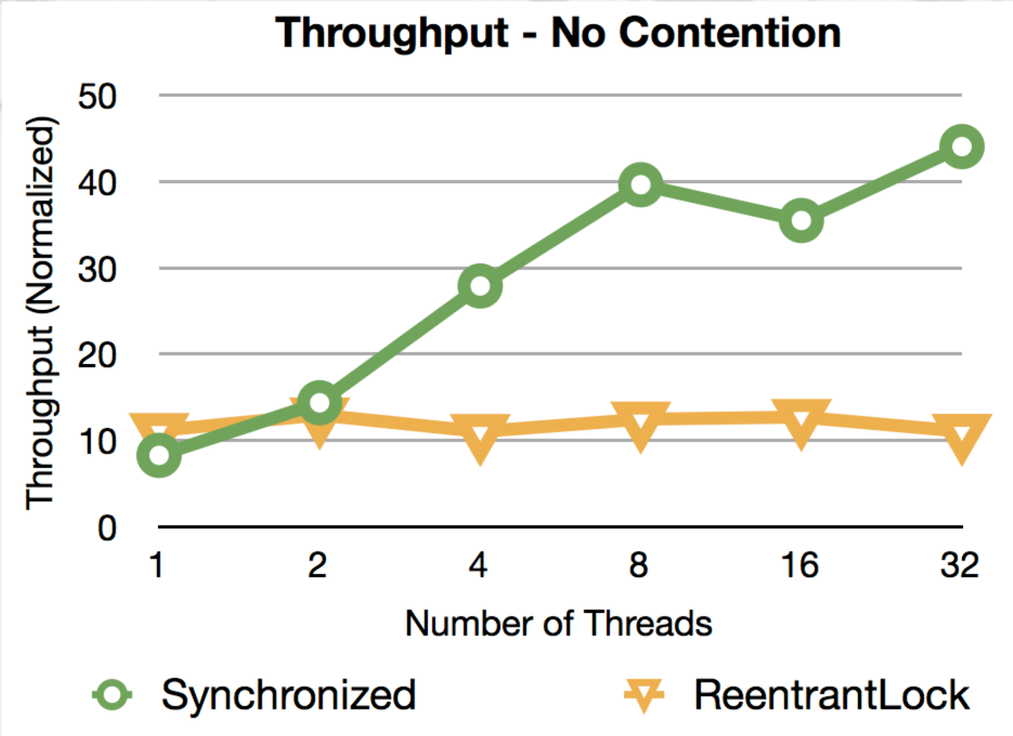
- In Java 5, ReentrantLock was faster than intrinsic locks (synchronized) for contended locks
  - "Contended" means several threads are trying to acquire same lock
- In Java 6, intrinsic locks were improved
  - Now they are about the same speed
- Performance is a moving target

Javaspecialists.eu

JavaSpecialists.EU

Uncontended Performance

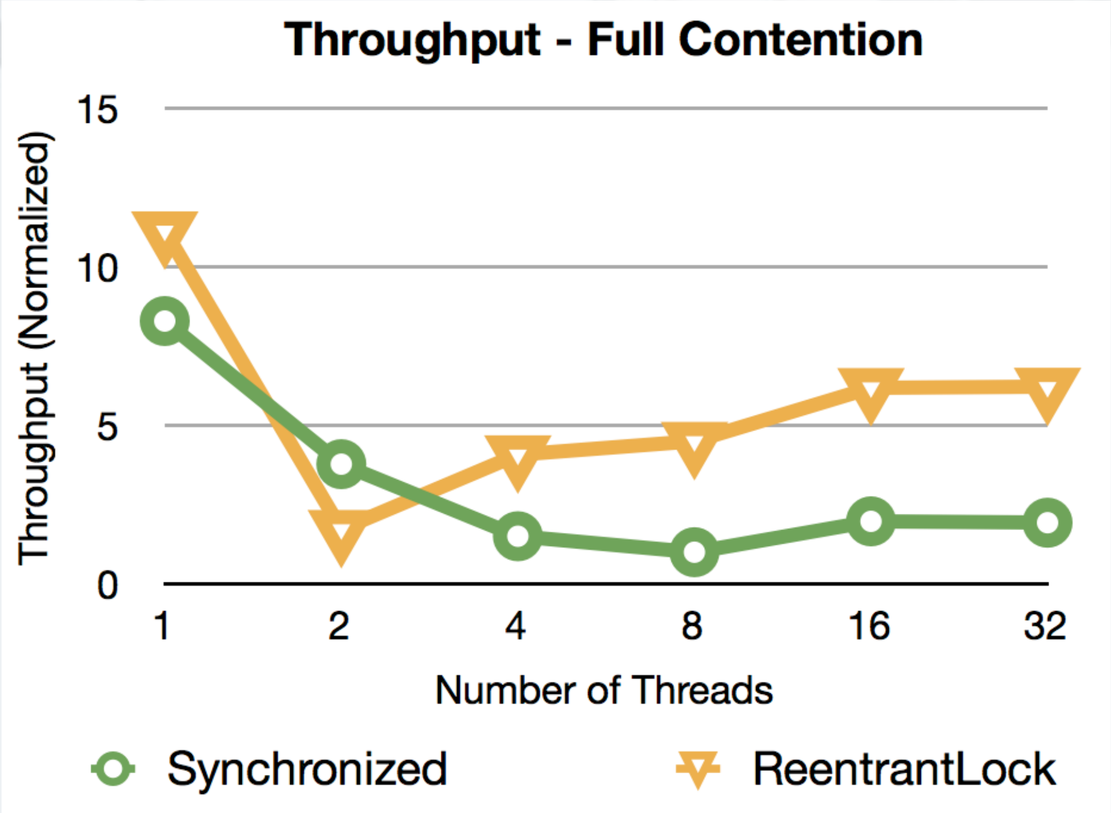
- **When there is no contention at all, synchronized can be optimized away through biased locking**
  - **Thus it beats explicit locks**



Javaspecialists.eu
JavaSpecialists.EU

Contended Performance

- **When there is a lot of contention, ReentrantLock can still beat synchronized, even in Java 6**



Javaspecialists.eu

JavaSpecialists.EU

11.3: Synchronized Vs Reentrantlock

Explicit Locks

JavaSpecialists.EU

11.3: Synchronized Vs Reentrantlock

- **ReentrantLock and intrinsic locks have the same memory semantics**
- **Reentrant locks has more features**
  - **Polled locks, timed locks, interruptible locks and "fairness"**
- **However, synchronized has significant advantages**
  - **Few programmers structure the try-finally block correctly:**

```
lock.lock();
try {
  // do operation
} finally {
  lock.unlock();
}
```

```
synchronized(monitor) {
 // do operation
}
```

Javaspecialists.eu

JavaSpecialists.EU

Bad Try-Finally Blocks

- **Either no try-finally at all**

```
lock.lock();
// operation
lock.unlock();
```

- **Or the lock is locked inside the try block**

```
try {
  lock.lock();
  // operation
} finally {
  lock.unlock();
}
```

```
try {
  lock.lockInterruptibly();
  // operation
} finally {
  lock.unlock();
}
```

- **Or unlock() call is forgotten in some places altogether!**

```
lock.lock();
// operation
```

Javaspecialists.eu

JavaSpecialists.EU

When To Use Reentrantlock

- **Use it when you need**
  - **lock.tryLock()**
  - **lock.tryLock(timeout)**
  - **lock.lockInterruptibly()**
  - **Multiple condition variables for one lock**
- **Otherwise, prefer synchronized**
  - **From Java 9 CopyOnWriteArrayList, now using synchronized**

```
/**
 * The lock protecting all mutators.  (We have a mild preference
 * for builtin monitors over ReentrantLock when either will do.)
 */
final transient Object lock = new Object();
```

Javaspecialists.eu

JavaSpecialists.EU

11.4: Read-Write Locks

Explicit Locks

JavaSpecialists.EU

11.4: Read-Write Locks

- **ReadWriteLock interface**
  - The writeLock() is only given to one thread at a time
  - The readLock() is given to lots of threads at the same time
    - Much better when mostly reads are happening
  - Implemented in ReentrantReadWriteLock
- **Need a good understanding of system to decide between Lock and ReadWriteLock**
  - Should only use when read block is at least 2000 instructions
  - ReadWriteLock can cause starvation
    - http://www.javaspecialists.eu/archive/Issue165.html

Javaspecialists.eu

JavaSpecialists.EU

Bank Account With Readwritelock

```
import java.util.concurrent.locks.*;

public class BankAccountWithReadWriteLock {
  private final ReadWriteLock lock =
      new ReentrantReadWriteLock();
  private double balance;
  public void deposit(double amount) {
    lock.writeLock().lock();
    try     { balance = balance + amount; }
    finally { lock.writeLock().unlock(); }
  }
  public double getBalance() {
    lock.readLock().lock();
    try     { return balance; }
    finally { lock.readLock().unlock(); }
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Bank Account With Volatile And Synchronized

```
public class BankAccountWithVolatileRead {
  private volatile double balance;
  public synchronized void deposit(double amount) {
    balance = balance + amount;
  }
  public double getBalance() {
    return balance;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

ReentrantReadWriteLock Up/Downgrading

- **We can *downgrade* a write lock to a read lock**

```
09:      rwlock.writeLock().lock();
10:      rwlock.readLock().lock();
11:      rwlock.writeLock().unlock();
12:      rwlock.readLock().unlock();
```

- **We cannot *upgrade* a read lock to a write lock**
  - **Imagine if two threads both hold the read lock and try do this**

```
14:      rwlock.readLock().lock();
15:      rwlock.writeLock().lock(); // deadlock here!!!
16:      rwlock.readLock().unlock();
17:      rwlock.writeLock().unlock();
```

```
"main" #1 waiting on condition
  java.lang.Thread.State: WAITING (parking)
    at jdk.internal.misc.Unsafe.park(Native Method)
    ...
    at UpDowngrading.main(UpDowngrading.java:15)
```

Javaspecialists.eu

JavaSpecialists.EU

11.5: StampedLock

Explicit Locks

JavaSpecialists.EU

What is StampedLock?

- **Java 8 synchronizer**
- **Allows optimistic reads**
  - **ReentrantReadWriteLock only has pessimistic reads**
- **Not reentrant**
  - **This is *not* a feature**
- **Use to enforce invariants across multiple fields**
  - **For simple classes, synchronized/volatile is easier and faster**

Javaspecialists.eu

JavaSpecialists.EU

Pessimistic Exclusive Lock (Write)

```
public class StampedLock {
  long writeLock() // never returns 0, might block

  // returns new write stamp if successful; otherwise 0
  long tryConvertToWriteLock(long stamp)

  void unlockWrite(long stamp) // needs write stamp

// and a bunch of other methods left out for brevity
```

Javaspecialists.eu

JavaSpecialists.EU

Pessimistic Non-Exclusive Lock (Read)

```
public class StampedLock { // continued ...
  long readLock() // never returns 0, might block

  // returns new read stamp if successful; otherwise 0
  long tryConvertToReadLock(long stamp)

  void unlockRead(long stamp) // needs read stamp

  void unlock(long stamp) // unlocks read or write
```

Javaspecialists.eu

JavaSpecialists.EU

Optimistic Non-Exclusive Read (No Lock)

```
public class StampedLock { // continued ...
  // could return 0 if a write stamp has been issued
  long tryOptimisticRead()

  // return true if stamp was non-zero and no write
  // lock has been requested by another thread since
  // the call to tryOptimisticRead()
  boolean validate(long stamp)
```

Javaspecialists.eu

JavaSpecialists.EU

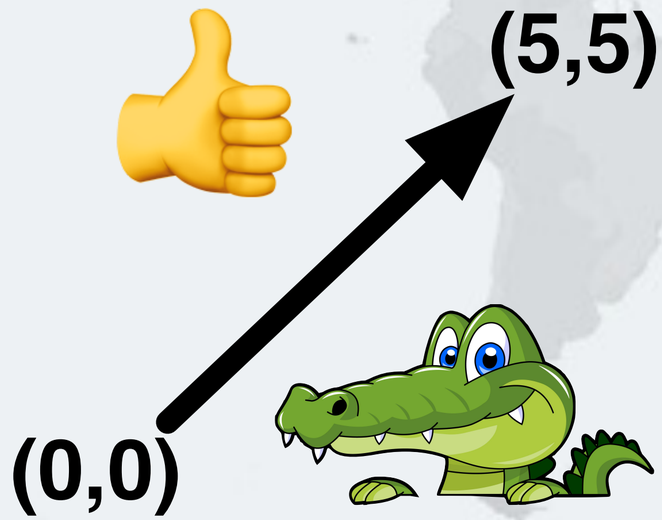
Sifis the Cretan Crocodile (RIP)

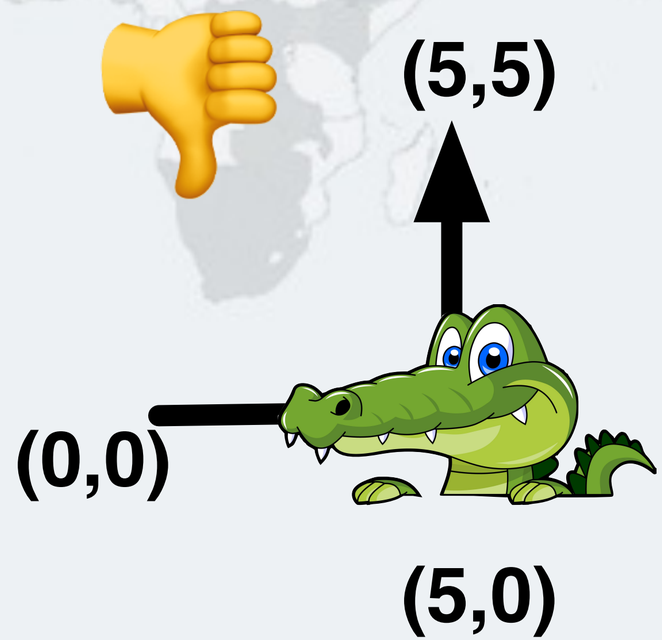
- **Poor critter was roaming around Crete**
  - **The pet became too big**
  - **Or hungry**
- **Eventually died in our cold winter months**

Javaspecialists.eu

Introducing The Position Class

- **When moving from (0,0) to (5,5), we want to travel in a diagonal line**
  - **We don't want to ever see our position at (0,5) or especially (5,0)**





Javaspecialists.eu

JavaSpecialists.EU

Moving Our Position

- **Similar to the way ReentrantLock would be written**

```
public class Position {
  private double x, y;
  private final StampedLock sl = new StampedLock();

  // method is modifying x and y, needs exclusive lock
  public void move(double deltaX, double deltaY) {
    long stamp = sl.writeLock();
    try {
      x += deltaX;
      y += deltaY;
    } finally {
      sl.unlockWrite(stamp);
    }
  }
```

Javaspecialists.eu

JavaSpecialists.EU

Optimistic Read

- **Avoids pessimistic read locking**
- **Better throughput than ReadWriteLock**

Javaspecialists.eu

JavaSpecialists.EU

Code Idiom For Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(currentState1, currentState2);
}
```

Javaspecialists.eu

JavaSpecialists.EU

Code Idiom For Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(currentState1, currentState2);
}
```

We get a stamp to use for the optimistic read

Javaspecialists.eu

JavaSpecialists.EU

Code Idiom For Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(currentState1, currentState2);
}
```

We read field values into local fields

Javaspecialists.eu

JavaSpecialists.EU

Code Idiom For Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(currentState1, currentState2);
}
```

Next we validate that no write locks have been issued in the meanwhile

Javaspecialists.eu

JavaSpecialists.EU

Code Idiom For Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(currentState1
}
```

If they have, then we don't know if our state is clean

Thus we acquire a pessimistic read lock and read the state into local fields

Javaspecialists.eu

JavaSpecialists.EU

Code Idiom For Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(currentState1, currentState2);
}
```

Javaspecialists.eu

JavaSpecialists.EU

Optimistic Read In Our Position Class

```
public double distanceFromOrigin() {
  long stamp = sl.tryOptimisticRead();
  double currentX = x, currentY = y;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentX = x;
      currentY = y;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return Math.hypot(currentX, currentY);
}
```

The shorter the code path from tryOptimisticRead() to validate(), the better the chances of success

Javaspecialists.eu

JavaSpecialists.EU

StampedLock Idioms are Difficult to Master

- **Instead, we can define static helper methods**
  - **Gang-of-Four Facade Pattern**
- **Lambdas make helper methods pluggable**

Javaspecialists.eu

JavaSpecialists.EU

Moving With StampedLockIdioms

- **The old move() method**

```
public void move(double deltaX, double deltaY) {
  long stamp = sl.writeLock();
  try {
    x += deltaX;
    y += deltaY;
  } finally {
    sl.unlockWrite(stamp);
  }
}
```

- **Now looks like this**

```
public void move(double deltaX, double deltaY) {
  StampedLockIdioms.writeLock(sl, () -> {
    x += deltaX;
    y += deltaY;
  });
}
```

Javaspecialists.eu

JavaSpecialists.EU

Our StampedLockIdioms

- **We simply call writeJob.run() inside the locked section**

```
public class StampedLockIdioms {
  public static void writeLock(StampedLock sl,
                               Runnable writeJob) {
    long stamp = sl.writeLock();
    try {
      writeJob.run();
    } finally {
      sl.unlockWrite(stamp);
    }
  }
  // ...
```

- **Checked exceptions would be an issue though**

Javaspecialists.eu

JavaSpecialists.EU

Optimistic Read Using StampedLockIdioms

- **Our old distanceFromOrigin**

```
public double distanceFromOrigin() {
  long stamp = sl.tryOptimisticRead();
  double currentX = x, currentY = y;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentX = x;
      currentY = y;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return Math.hypot(currentX, currentY);
}
```

Javaspecialists.eu

JavaSpecialists.EU

Optimistic Read Using StampedLockIdioms

- **Becomes this new mechanism**

```
public double distanceFromOrigin() {
  double[] current = new double[2];
  return StampedLockIdioms.optimisticRead(sl,
      () -> {
        current[0] = x;
        current[1] = y;
      },
      () -> Math.hypot(current[0], current[1]));
}
```

Javaspecialists.eu

JavaSpecialists.EU

Our StampedLockIdioms.optimisticRead()

- **The reading.run() call would probably be inlined**

```
public static <T> T optimisticRead(
        StampedLock sl,
        Runnable reading,
        Supplier<T> computation) {
  long stamp = sl.tryOptimisticRead();
  reading.run();
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      reading.run();
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return computation.get();
}
```

Javaspecialists.eu

JavaSpecialists.EU

11: Exercises

Explicit Locks

JavaSpecialists.EU

Exercise 11.1: Better WalkingCollection

- **Fix the WalkingCollection to allow multiple threads to call the iterate() method concurrently and to still be threadsafe**
- **Watch out for liveness issues**

Javaspecialists.eu

JavaSpecialists.EU

Exercise 11.2: tryLock() for Solving Deadlocks

- **We will now try (!) to avoid a deadlock in the Thinker**
  - **Use Lock.tryLock() to avoid blocking on the inner lock**
  - **Verify that the deadlock has now disappeared**

Javaspecialists.eu

JavaSpecialists.EU

Exercise 11.3: StampedLock with IntList (Opt)

- **IntList uses synchronized to protect its state**
- **Use StampedLock optimistic non-exclusive read with**
  - **size()**
  - **get()**
  - **You might need to get a pessimistic non-exclusive lock**
- **Use StampedLock pessimistic exclusive write with**
  - **trimToSize()**
  - **add()**
  - **remove()**
- **Might also be faster (hardware dependent)**

Javaspecialists.eu

JavaSpecialists.EU

11: Explicit Locks

Questions?

Day 3

12: Building Custom Synchronizers

Advanced topic on how to build your own locks and things

JavaSpecialists.EU

12: Building Custom Synchronizers

- **A state-dependent class has some precondition that must be true before a method can be called**
  - **It could block until the precondition has been fulfilled**
  - **Or give up after a specified timeout**
  - **Or give up immediately**

Javaspecialists.eu

JavaSpecialists.EU

Using State-Dependent Library Classes

- **The easiest way to construct a state-dependent class is to build it with existing state-dependent *library* classes**
  - **For example with**
    - **FutureTask**
    - **CountdownLatch**
    - **Semaphore**
    - **BlockingQueue**
    - **Exchanger**
    - **etc.**
  - **These library classes do not always do exactly what we need**
    - **We need to thus also understand the low-level building blocks**

Javaspecialists.eu

JavaSpecialists.EU

12.1: Managing State Dependence

Building Custom Synchronizers

JavaSpecialists.EU

12.1: Managing State Dependence

- **In single-threaded programs**
  - **A false pre-condition based on the object's state will remain false**
    - **The only sensible thing to do is fail immediately**
- **In multi-threaded programs**
  - **A false pre-condition based on state might become true later**
    - **We could thus either fail immediately or block for a while to see if the state changes**
    - **e.g. an empty queue might receive elements by a different thread**

Javaspecialists.eu

JavaSpecialists.EU

Condition Queues With Intrinsic Locks

- **The wait/notify methods are related to synchronized**
  - **We need to hold the intrinsic lock if which we want to call wait(), notify() or notifyAll()**
    - **wait() - release the lock, block until *something happens* or timeout**
    - **notify() - *something happened***
- **Tips on wait/notify**
  - **Favour notifyAll() over notify()**
    - **"Thundering Herd Problem" is not usually an issue in real world**
    - **Less risk of one thread missing a notify**
  - **Always write the wait() code as if you were using notifyAll()**

Javaspecialists.eu

JavaSpecialists.EU

Structure Of Blocking State-Dependent Actions

- **Blocking state-dependent methods typically work like this**
  - **It is obviously very important that we release lock whilst waiting**

```
void blockingAction() throws InterruptedException {
  acquire lock on object state
  while (precondition does not hold) {
    release lock
    wait until precondition might hold
    optionally fail if interrupted or timeout expires
    reacquire lock
  }
  perform action and signal others that might wait
  release lock
}
```

Javaspecialists.eu

JavaSpecialists.EU

Example of an ArrayBoundedQueue

```
@ThreadSafe
public class ArrayBoundedQueue<V> {
  @GuardedBy("this")
  private final V[] buf;
  @GuardedBy("this")
  private int tail, head, count;

  public ArrayBoundedQueue(int capacity) {
    this.buf = (V[]) new Object[capacity];
  }

  public synchronized boolean isFull() {
    return count == buf.length;
  }

  public synchronized boolean isEmpty() {
    return count == 0;
  }
```

Javaspecialists.eu

JavaSpecialists.EU

take() Waits Until The Queue is *Not* Empty

```
// BLOCKS-UNTIL: !isEmpty()
public synchronized V take()
    throws InterruptedException {
  while (isEmpty()) wait();
  notifyAll();
  return doTake();
}

private V doTake() {
  assert !isEmpty() : "Queue empty";
  assert Thread.holdsLock(this);
  V v = buf[head];
  buf[head] = null;
  if (++head == buf.length) head = 0;
  --count;
  return v;
}
```

Javaspecialists.eu

JavaSpecialists.EU

put() Waits Until The Queue is *Not* Full

```
// BLOCKS-UNTIL: !isFull()
public synchronized void put(V v)
    throws InterruptedException {
  while (isFull()) wait();
  doPut(v);
  notifyAll();
}

private void doPut(V v) {
  assert !isFull() : "Queue full";
  assert Thread.holdsLock(this);
  buf[tail] = v;
  if (++tail == buf.length) tail = 0;
  ++count;
}
}
```

Javaspecialists.eu

JavaSpecialists.EU

12.2: Using Condition Queues

Building Custom Synchronizers

JavaSpecialists.EU

The Condition Predicate

- **The first step is to identify and document what the *condition predicate* is**
  - **This is the pre-condition that makes the method state-dependent**
- **ArrayBoundedQueue has these condition predicates**
  - **take() : queue is not empty**
    - **e.g. you can only take an element from a queue if it is not empty; otherwise you must wait until the condition is true**
  - **put() : queue is not full**
- **Condition predicates should always be documented**
  - **There is no language support for condition predicates**

Javaspecialists.eu

JavaSpecialists.EU

Three Way Relationship

- **There is a relationship between the lock, the condition predicate and the condition queue**
  - **In ArrayBoundedQueue, the state is guarded by "this" and the condition queue also uses "this"**
  - **The take method synchronizes on "this" and then tests queue size**
    - **If non-empty, it removes the first element**
      - **At this point it still holds the intrinsic lock**
    - **If empty, it calls wait() on the intrinsic lock**
      - **This suspends the thread and releases the lock**
      - **Once wait returns, it reacquires lock and tests condition predicate *again***
  - ***Note: a thread waking from wait() has to contend for the lock with all other threads - it does not get priority on the lock***

Javaspecialists.eu

JavaSpecialists.EU

Waking Up Too Soon

- **We can wake up from wait() when**
  - **another thread sends us a notify() or notifyAll()**
  - **the timeout expires**
  - **another thread calls interrupt() on our thread**
  - **very rarely, the OS or the JVM might send a *spurious wakeup***
- **Our wait() code should always work even with notifyAll()**
  - **That way you cover spurious wakeups and lost races for the lock**

Javaspecialists.eu

JavaSpecialists.EU

```
@ThreadSafe
public class UnboundedQueue<V> {
  private final Object lock = new Object();
  @GuardedBy("lock")
  private final List<V> elements = new LinkedList<V>();
  public V take() throws InterruptedException {
    synchronized(lock) {
      while (elements.isEmpty()) {  // BLOCKS-UNTIL: not-empty
        try {
          lock.wait();
        } catch (InterruptedException e) {
          lock.notify(); // propagate to non-interrupted thread
          throw e;
        }
      }
      return elements.remove(0);
    }
  }
  public void put(V v) {
    synchronized(lock) {
      elements.add(v);
      lock.notify(); // wake up a single thread
    }
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

When to Use Single notify() vs notifyAll()

- **We can only use notify() instead of notifyAll() when**
  - **Uniform waiters: Only one condition predicate is associated with the condition queue**
    - **UnboundedQueue only has one condition - queue !empty**
      - **We can use notify()**
    - **ArrayBoundedQueue uses two conditions - queue !empty & !full**
      - **We *must* use notifyAll()**
  - **One-in, one-out: A notification means only one thread can proceed**
    - **If the UnboundedQueue is empty and we add one element, only one thread can proceed to take it out, we can use notify()**
    - **In a bank account, if withdraw blocks if** amount > balance**, then with a large deposit, lots of threads could proceed, must use notifyAll()**

Javaspecialists.eu

JavaSpecialists.EU

12.3: Explicit Condition Objects

Building Custom Synchronizers

JavaSpecialists.EU

12.3: Explicit Condition Objects

- **Condition is a higher-level construct for doing condition queues with explicit locks**

```
public interface Condition {
  void await() throws InterruptedException;
  void awaitUninterruptibly();
  long awaitNanos(long nanosTimeout) throws InterruptedException;
  boolean await(long time, TimeUnit unit)
      throws InterruptedException;
  boolean awaitUntil(Date deadline) throws InterruptedException;
  void signal();
  void signalAll();
}
```

Javaspecialists.eu

JavaSpecialists.EU

Intrinsic Condition Queue Vs Java 5 Condition

- **Intrinsic condition queues have disadvantages**
  - Each lock can only have a single condition queue
    - ArrayBoundedQueue needs to use notifyAll() to wake both wait()s
  - When the timed wait() returns, we do not know whether it timed out or whether we received a signal
- **Explicit condition queue associated with an explicit lock**
  - But each Lock can have lots of Conditions
    - Such as "queue full" and "queue empty"
  - In addition, we can find out how long the timed wait took to return
  - Method names were changed, wait/notify/notifyAll are final in Object
  - We can also wait uninterruptibly

Javaspecialists.eu

JavaSpecialists.EU

ConditionBoundedQueue

- **Our next queue uses two Condition objects to signal the notFull and notEmpty conditions**

```
@ThreadSafe
public class ConditionBoundedQueue<V> {
  private final Lock lock = new ReentrantLock();
  private final Condition notFull = lock.newCondition();
  private final Condition notEmpty = lock.newCondition();
  @GuardedBy("lock")
  private final V[] items;
  @GuardedBy("lock")
  private int tail, head, count;

  public ConditionBoundedQueue(int size) {
    items = (V[]) new Object[size];
  }
```

Javaspecialists.eu

JavaSpecialists.EU

take() Waits Until The Queue is Not Empty

```
// BLOCKS-UNTIL: (count != 0)
public V take() throws InterruptedException {
  lock.lock();
  try {
    while (count == 0) notEmpty.await();
    notFull.signal();
    return doTake();
  } finally {
    lock.unlock();
  }
}
private V doTake() {
  V v = items[head];
  items[head] = null;
  if (++head == items.length) head = 0;
  --count;
  return v;
}
```

Javaspecialists.eu

JavaSpecialists.EU

put() Waits Until The Queue is Not Full

```
// BLOCKS-UNTIL: (count != items.length)
public void put(V v) throws InterruptedException {
  lock.lock();
  try {
    while (count == items.length) notFull.await();
    doPut(v);
    notEmpty.signal();
  } finally {
    lock.unlock();
  }
}
private void doPut(V v) {
  items[tail] = v;
  if (++tail == items.length)
    tail = 0;
  ++count;
}
}
```

Javaspecialists.eu

JavaSpecialists.EU

Timed Conditions

- **The await() method allows us to specify a timeout**
  - **long awaitNanos(long nanosTimeout)** ***[useful]***
    - **returns the difference between nanosTimeout and actual wait time**
      - **This might be negative if it waited too long**
      - **Or positive if it waited too short**
        - » **Maybe received a signal and then lost the race for the lock?**
  - **boolean await(long time, TimeUnit unit)** ***[not useful]***
    - **returns true if it received a signal, false if it timed out and InterruptedException if it was interrupted**
  - **boolean awaitUntil(Date deadline)** ***[not useful]***
    - **returns false if the deadline has elapsed; true if it received a signal**

Javaspecialists.eu

JavaSpecialists.EU

Calling condition.awaitNanos()

- **If we want to ensure that a method waits for a specific time, we need to re-wait if there is still time left**
  - **The awaitNanos() method might return early**

```
public boolean aMethod(long timeout, TimeUnit unit)
      throws InterruptedException {
  long nanos = unit.toNanos(timeout);
  lock.lock();
  try {
    while (!conditionBeingWaitedFor()) {
      if (nanos <= 0L) return false;
      nanos = theCondition.awaitNanos(nanos);
    }
    doSomething();
    return true;
  } finally {
    lock.unlock();
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

12: Exercises

Building Custom Synchronizers

JavaSpecialists.EU

Exercise 12.1: FutureResultIterable

- **We want to write an Iterable that allows us to iterate over completed jobs and block on uncompleted jobs**
  - **Thanks to Rabea Gransberger for the idea**
- **You can use any synchronization mechanism**
  - **Hint: It is possible to solve with simple wait/notify constructs**

Javaspecialists.eu

JavaSpecialists.EU

12: Building Custom Synchronizers

Questions?

13: Atomic Variables And Nonblocking Synchronization

Making your application run even faster!

JavaSpecialists.EU

13: Atomic Variables And Nonblocking Synchronization

- **Utility classes in java.util.concurrent such as ConcurrentHashMap offer great performance**
  - A lot of the gains are from nonblocking and atomic synchronization
- **Nonblocking algorithms use Compare-And-Swap (CAS) to ensure data integrity**
  - These are more complicated than lock-based
  - But can offer better performance
- **Deadlock usually cannot happen, but livelock can in rare situations**

Javaspecialists.eu

JavaSpecialists.EU

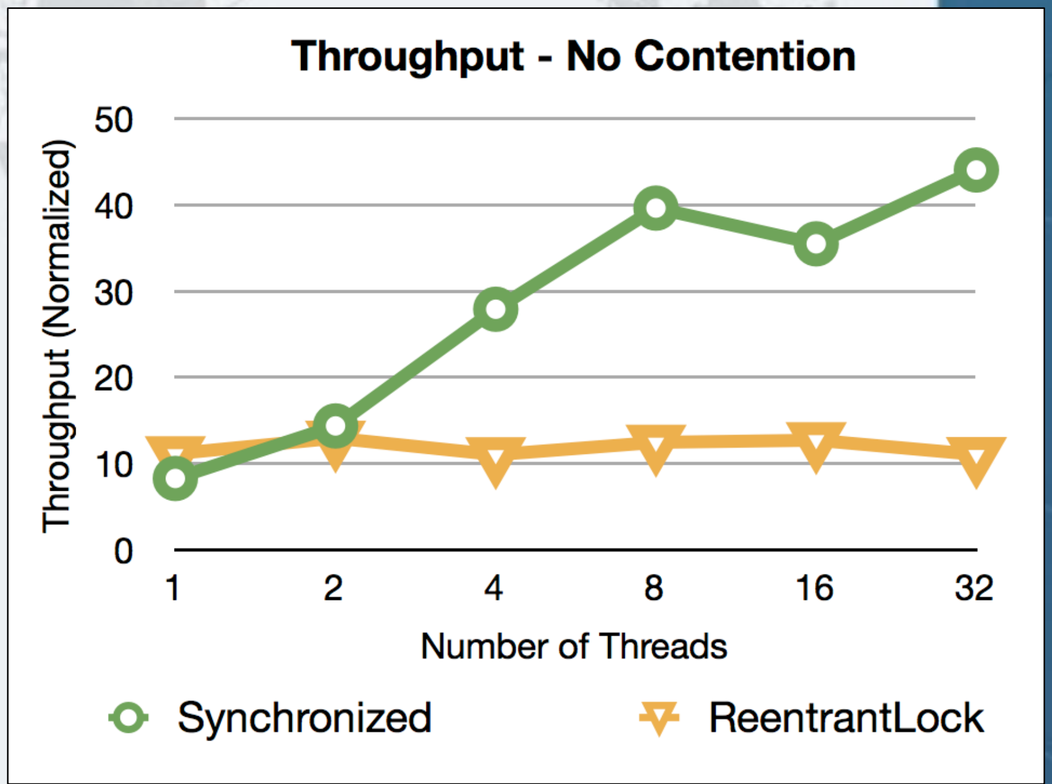
13.1: Disadvantages Of Locking

Atomic Variables and Nonblocking Synchronization

JavaSpecialists.EU

13.1: Disadvantages Of Locking

- **Locking ensures the thread has exclusive access to a memory location**
  - **Our work cannot be overwritten by another rogue thread**
- **If the lock is uncontended, the JVM can often eliminate it**
  - **This is done with biased locking or even lock eliding**
  - **Sometimes the JVM will let the thread spin for a while before locking**
  - **We can see that with no contention, throughput on *synchronized* is very good**



Javaspecialists.eu

Volatile Vs Locking

- Volatile is a light-weight synchronization mechanism
  - No context switching occurs
- Volatile ensures that changes to memory are visible
- Operations like += on volatile fields are also not atomic
  - Usually, volatile fields corrupt even quicker than normal fields
  - If we have a check-then-act or a read-modify-write operation, we need atomic classes and CAS to ensure changes are not overwritten
- Locking can result in deadlocks and priority inversion
  - Priority inversion: a low-priority thread is holding a lock, thus preventing a high-priority thread from entering a critical section

Javaspecialists.eu

JavaSpecialists.EU

Volatile Vs Synchronized Visibility

- **From a memory visibility perspective**
  - **Writing a volatile field is like exiting a** synchronized
  - **Reading a volatile field is like entering a** synchronized
- **It is possible to write clever code using this trick**
  - **e.g. ConcurrentHashMap**
  - **However, not advisable for most programmers**
  - **Volatile does not solve race conditions - it makes them worse!**

Javaspecialists.eu

JavaSpecialists.EU

13.2: Hardware Support For Concurrency

Atomic Variables and Nonblocking Synchronization

JavaSpecialists.EU

13.2: Hardware Support For Concurrency

- **Exclusive locking is a *pessimistic* technique**
- **For fine-grained work, *optimistic mechanisms* are better**
  - **"Everything is going to be alright"**
  - **We try an operation and if it does not succeed, we try again**
  - **This depends on *collision detection* instead of explicit locking**
- **Modern processors have multi-processing instructions**
  - **Compare And Swap (CAS), Load Linked / Store Conditional**

Javaspecialists.eu

JavaSpecialists.EU

Compare-and-Swap (CAS)

- **Most processor architectures use Compare-And-Swap**
  - **For example, Intel, AMD and Sparc**
  - **Others such as Alpha AXP, IBM PowerPC, MIPS and ARM, implement it with a pair of instructions**
    - **load-linked and store-conditional**
- **Compare-and-swap does the following atomically**
  - **We pass in memory location, expected value and new value**
  - **CAS updates memory to new value only if it contains expected value**
  - **The old value is returned always**
- **Compare-and-set is similar, but returns true or false**
  - **CAS in Java is usually compare-and-set**

Javaspecialists.eu

JavaSpecialists.EU

CompareAndSet (CAS) Support in the JVM

- **Available via atomic classes**
  - **Internally uses sun.misc.Unsafe**
- **The CAS methods are compiled as efficiently as possible**
  - **If the hardware supports it, it becomes a single CPU instruction**
    - **Even though it is declared as a native method call, we do not have the call cost overhead of JNI**
  - **If it does not support it, it is typically implemented as a spin lock**
- **The JDK uses this in a number of classes in the java.util.concurrent package**

Javaspecialists.eu

JavaSpecialists.EU

Java 9 VarHandles Instead of Unsafe

- **VarHandles remove biggest temptation to use Unsafe**
  - **Java 9: https://bugs.openjdk.java.net/browse/JDK-8080588**
- **As fast as Unsafe**
- **Can read and write fields of class**
  - **getVolatile() / setVolatile()**
  - **compareAndSet(), returning boolean**
  - **compareAndExchangeVolatile(), returning found value always**

Javaspecialists.eu

JavaSpecialists.EU

Using VarHandles

- **First step is to set up the VarHandle**

```
public class PositionVarHandlesNonBlocking {
  private volatile double[] xy = {0, 0};

  private static final VarHandle XY_HANDLE;
  static {
    try {
      XY_HANDLE = MethodHandles.lookup().findVarHandle(
        PositionVarHandlesNonBlocking.class,
        "xy", double[].class);
    } catch (ReflectiveOperationException e) {
      throw new ExceptionInInitializerError(e);
    }
  }
```

Note: Exact API might still change

Javaspecialists.eu

JavaSpecialists.EU

CompareAndSet with VarHandle

- **do-while until we finally manage to move**

```
public void move(double deltaX, double deltaY) {
  double[] current, next = new double[2];
  do {
    current = xy;
    next[0] = current[0] + deltaX;
    next[1] = current[1] + deltaY;
  } while (!XY_HANDLE.compareAndSet(this, current, next));
}

public double distanceFromOrigin() {
  double[] current = xy;
  return Math.hypot(current[0], current[1]);
}
```

Javaspecialists.eu

JavaSpecialists.EU

compareAndExchangeVolatile() Proper CAS

- **Instead of having to read the volatile field, get it from CAS**

```
public void move(double deltaX, double deltaY) {
  double[] current, next = new double[2], swapResult = xy;
  do {
    current = swapResult;
    next[0] = current[0] + deltaX;
    next[1] = current[1] + deltaY;
  }
  while ((swapResult =
    (double[]) XY_HANDLE.compareAndExchangeVolatile(
      this, current, next)) != current);
}
```

Javaspecialists.eu

JavaSpecialists.EU

Shared Cache Lines

- Our modern architectures consists of several layers of memory caches
- Memory is stored in units called "cache lines"
  - These can be anything from 32 to 256 bytes large
- When a Java object is too small, it might have to share the same cache line with another object
- Trick by Martin Thompson is to pad the Java object with dummy data, thus forcing it to be in its own cache line
  - http://mechanical-sympathy.blogspot.com/2011/07/false-sharing.html

Javaspecialists.eu

JavaSpecialists.EU

@sun.misc.Contended

- **Java 8 annotation for padding fields**
  - **e.g. in Thread we see the random number seeds fields padded:**

```
@sun.misc.Contended("tlr") long threadLocalRandomSeed;
@sun.misc.Contended("tlr") int threadLocalRandomProbe;
@sun.misc.Contended("tlr") int threadLocalRandomSecondarySeed;
```

  - **Pads the fields with 128 bytes by default**
    - **Configurable with -XX:ContendedPaddingWidth=num**
  - **By default only enabled for system classes**
    - **Use in our classes with -XX:-RestrictContended**
  - **Creates dependency to a sun.misc.* class**

Javaspecialists.eu

JavaSpecialists.EU

13.3: Atomic Variable Classes

Atomic Variables and Nonblocking Synchronization

JavaSpecialists.EU

13.3: Atomic Variable Classes

- **Atomics are optimistic about clashes**
  - **They assume there will be no clash (optimism)**
  - **Compare-and-swap is used to set the field, if possible**
  - **If there was a clash, we simply try again**
- **With not too much contention, atomics are blazingly fast**
  - **But with heavy contention, atomics can sometimes be slower**
    - **They might clash so much that the "try again" is more than locks**
- **Useful for writing high-performant concurrent code**
  - **Especially for fine-grained fields like counters**

Javaspecialists.eu

JavaSpecialists.EU

Atomics As "Better Volatiles"

- **Atomics store the value internally as a *volatile* field**
  - **We thus have the same visibility semantics**
- **There is little reason to use volatile directly**
  - **Atomic classes also have other advantages in that they solve some read-modify-write race conditions**
    - **Though check-then-act would usually require a loop**

Javaspecialists.eu

JavaSpecialists.EU

Types Of Atomic Classes

- **The following types have atomics built in**
  - **AtomicBoolean**
  - **AtomicInteger**
    - **Use for int, short, byte and float (use Float.floatToIntBits(float))**
  - **AtomicLong**
    - **Use for long and double**
  - **AtomicReference**
- **There are also atomic array classes**
    - **Necessary as you can never make values of an array volatile!**
  - **AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray**

Javaspecialists.eu

JavaSpecialists.EU

How Do Atomics Work?

- **Thread safe without explicit locking**
  - **Tries to update the value repeatedly until success**
  - **Only works if there are no invariants across fields**
- **Here is how addAndGet() used to be written in the JDK**

```
public final int addAndGet(int delta) {
  for (;;) {
    int current = get();
    int next = current + delta;
    if (compareAndSet(current, next))
      return next;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Atomic Bank Account

```
public class BankAccount {
  private final AtomicInteger balance =
    new AtomicInteger();

  public BankAccount(int balance) {
    this.balance.set(balance);
  }
  public void deposit(int amount) {
    balance.addAndGet(amount);
  }
  public void withdraw(int amount) {
    deposit(-amount);
  }
  public int getBalance() {
    return balance.intValue();
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

LongAdder (Java 8)

- **Fast counter when we have high contention**
  - **Stripes the values into different cells to reduce contention**

```
public class BankAccount {
  private final LongAdder balance = new LongAdder();
  public BankAccount(long balance) {
    this.balance.add(balance);
  }
  public void deposit(long amount) {
    balance.add(amount);
  }
  public void withdraw(long amount) {
    deposit(-amount);
  }
  public long getBalance() {
    return balance.longValue();
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

LongAccumulator (Java 8)

- **More flexible highly concurrent class**
  - **For example, to find maximum values**

```
LongAccumulator acc =
    new LongAccumulator(
      (left, right) -> Math.max(left, right),
      Long.MIN_VALUE);
acc.accumulate(30);
```

  - **Or even simpler, we can use method references**

```
LongAccumulator acc =
    new LongAccumulator(Math::max, Long.MIN_VALUE);
acc.accumulate(30);
```

Javaspecialists.eu

JavaSpecialists.EU

Performance of LongAdder vs Traditional Add

- **In our experiment we added 200 000 000 numbers in parallel using LongStream on 1-4-2 MacBookPro Retina**
  - **LongAdder took 376ms (stdev 16)**
  - **LongAccumulator took 405ms (stdev 25)**
  - **AtomicLong took 5107ms (stdev 211)**
  - **synchronized field took 9005ms (stdev 174)**

Javaspecialists.eu

JavaSpecialists.EU

13.4: Nonblocking Algorithms

Atomic Variables and Nonblocking Synchronization

JavaSpecialists.EU

13.4: Nonblocking Algorithms

- **Lock-based algorithms can cause scalability issues**
  - **If a thread holds a lock and is swapped out, no one can progress**
- **Definitions of types of algorithms**
  - ***Lock-free*: at each step, some thread can make progress**
    - **e.g. ConcurrentLinkedQueue offer() and poll()**
  - ***Wait-free*: every call finishes in a finite number of steps**
- **We use Compare-And-Swap (CAS) to write lock-free algorithms**

Javaspecialists.eu

JavaSpecialists.EU

A Nonblocking Stack

- **In our stack, AtomicReference points to the top**
  - **We don't have method for telling us the size, just push() and pop()**

```
@ThreadSafe
public class ConcurrentStack<E> {
  private final AtomicReference<Node<E>> top
      = new AtomicReference<>();

  @Immutable
  private static class Node<E> {
    private final E item;
    private final Node<E> next;
    private Node(E item, Node<E> next) {
      this.item = item;
      this.next = next;
    }
  }
```

Javaspecialists.eu

JavaSpecialists.EU

Stack "push()" And "pop()" Methods

```
public void push(E item) {
  Node<E> oldHead, newHead;
  do {
    oldHead = top.get();
    newHead = new Node<>(item, oldHead);
  } while (!top.compareAndSet(oldHead, newHead));
}

public E pop() {
  Node<E> oldHead, newHead;
  do {
    oldHead = top.get();
    if (oldHead == null)
      return null;
    newHead = oldHead.next;
  } while (!top.compareAndSet(oldHead, newHead));
  return oldHead.item;
}
}
```

Javaspecialists.eu

JavaSpecialists.EU

Speculative Work

- **In algorithms using CAS, we usually need to be optimistic and do work that might need to be redone**
  - **We construct a new Node in push() method, hoping we win race**
  - **We might lose repeatedly**
  - **This is a characteristic of lock-free algorithms**

```
public void push(E item) {
  Node<E> oldHead, newHead;
  do {
    oldHead = top.get();
    newHead = new Node<>(item, oldHead);
  } while (!top.compareAndSet(oldHead, newHead));
}
```

Javaspecialists.eu

JavaSpecialists.EU

Java 8 accumulateAndGet()

```
public class ConcurrentStackJava8<E> {
  private final AtomicReference<Node<E>> top
      = new AtomicReference<>();

  private static class Node<E> {
    private final E item;
    private Node<E> next;
    private Node(E item) {
      this.item = item;
    }
  }

  public void push(E item) {
    top.accumulateAndGet(new Node<>(item), (oldHead, newHead) -> {
      newHead.next = oldHead;
      return newHead;
    });
  }
// etc.
```

Javaspecialists.eu

JavaSpecialists.EU

13: Exercises

Atomic Variables and Nonblocking Synchronization

JavaSpecialists.EU

Exercise 13.1: Make an Atomic BankAccount

- **Change this to use AtomicInteger instead**

```
public class BankAccount {
  private int balance;

  public BankAccount(int balance) { this.balance = balance; }

  public synchronized void deposit(int amount) {
    if (balance + amount < 0)
      throw new IllegalStateException();
    balance += amount;
  }
  public void withdraw(int amount) { deposit(-amount); }
  public synchronized int getBalance() { return balance; }
  public synchronized boolean transferTo(
      BankAccount other, int amount) {
    if (amount < 0) throw new IllegalArgumentException();
    if (balance < amount) return false;
    withdraw(amount);
    other.deposit(amount);
    return true;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

13: Atomic Variables And Nonblocking Synchronization

Questions?

Performance

14: Java Memory

JavaSpecialists.EU

4: Java Memory

- In this module, we will look at how to manage Java memory better
- OutOfMemoryErrors can dog our system
  - Can cause individual threads to die
  - Can make the system unstable
  - Not always easy to detect and remove
- Too much GC activity can cause scalability issues
- We will start by looking at Garbage Collection. We then look at ways to tune the JVM. Lastly we look at References

Javaspecialists.eu

JavaSpecialists.EU

14.1: Garbage Collection

JavaSpecialists.EU

14.1: Garbage Collection

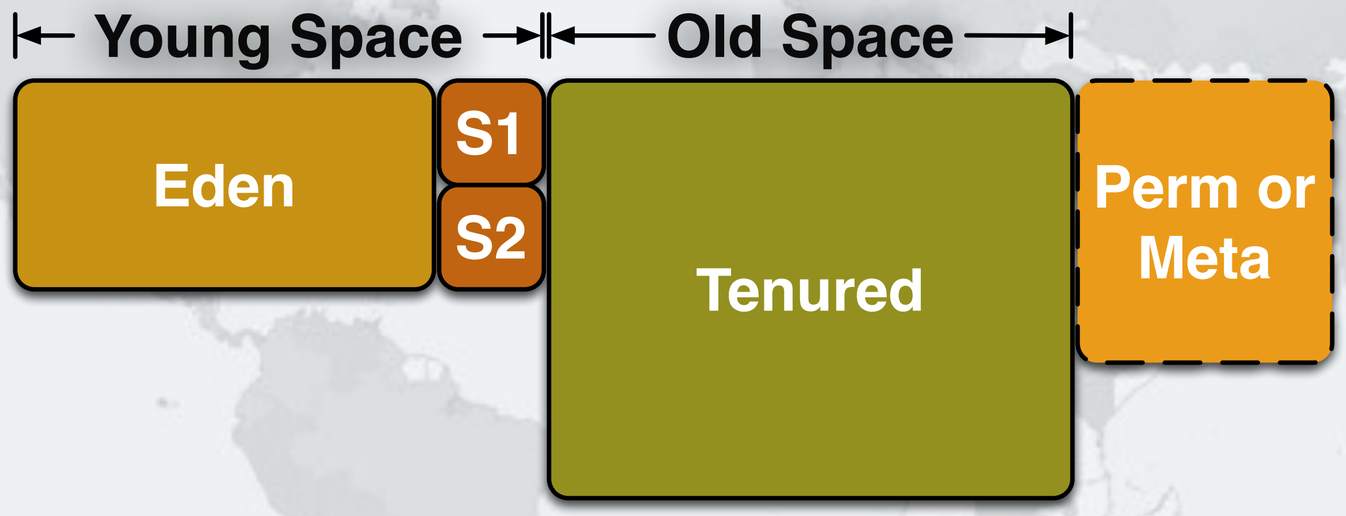
- In Java, we never have to explicitly release memory
  - The Garbage Collector figures out when objects are not needed
  - Even though this takes time, memory management in C++ also does
    - But it is not so obvious, as the pauses are not explicitly seen
- We have different Garbage Collectors in OpenJDK:
  - Throughput Collector
    - Uses multiple threads for collecting objects in parallel
  - Concurrent Collector
    - Collects the “old generation” concurrently with our threads
  - Garbage First (G1) Collector
    - Hybrid regional collector for heaps larger than 4 GB

Javaspecialists.eu

JavaSpecialists.EU

Throughput And Concurrent Collectors

- **Memory is split into spaces (drawn to scale)**

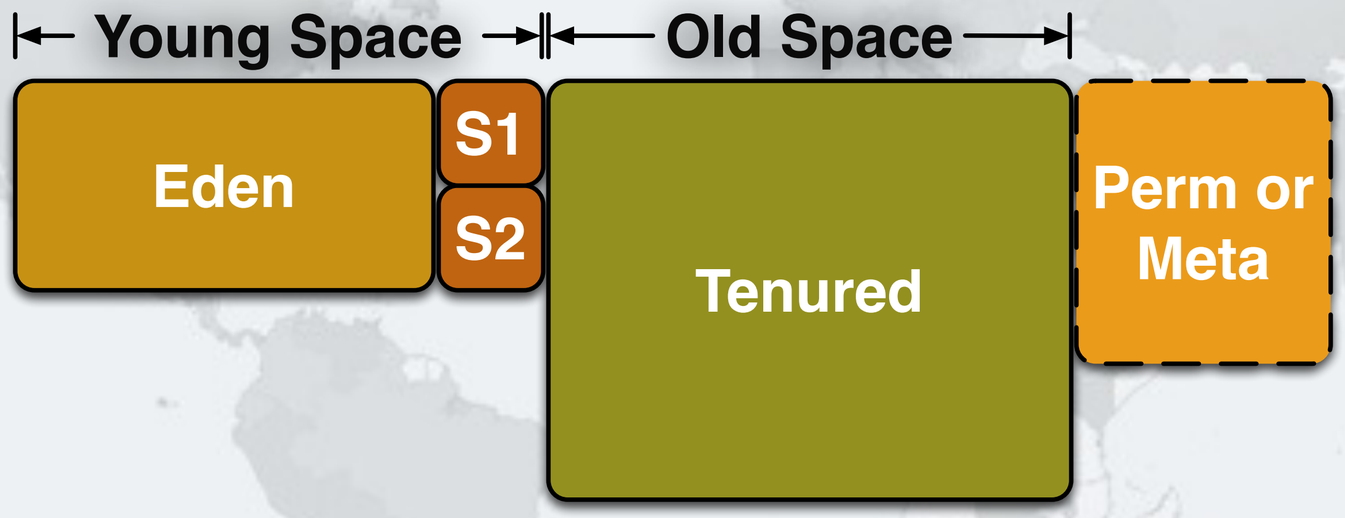


- **Objects are usually created in Eden**
  - **Unless they are too large, then they are created directly in Tenured**
  - **Or if they are considered almost permanent**
    - **Perm Space in Java 7, includes classes, Strings, etc.**
    - **Meta Space in Java 8, includes only classes and meta data**

Javaspecialists.eu
JavaSpecialists.EU

Young vs Old GC

- **Young GC (also called Partial or Scavenge GC) is fast**

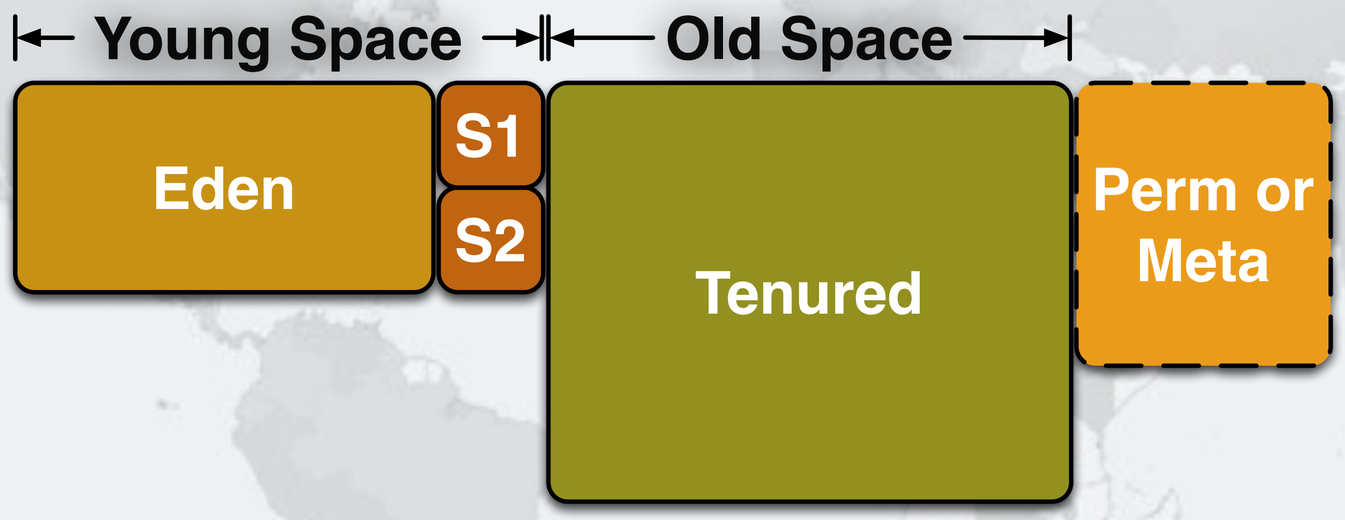


  - Typically takes a few milliseconds
  - Moves all live objects in Eden and active survivor to the inactive survivor space
  - Clears Eden and active survivor in one operation
  - Inactive and active survivors now swap around

Javaspecialists.eu
JavaSpecialists.EU

Young vs Old GC

- **Old GC is slower**

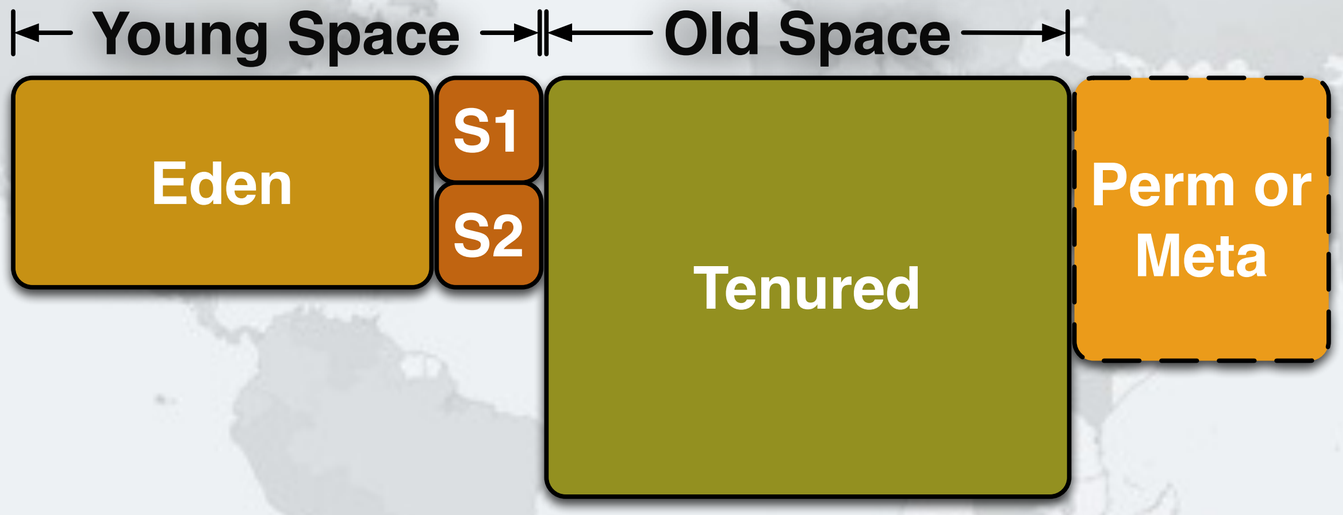


  - Can take seconds or minutes
  - Often needs to compact the Tenured space
  - Might also collect Perm or Meta Spaces

Javaspecialists.eu
JavaSpecialists.EU

Promotion From Young to Old

- **Objects are tenured (moved to old space) after some time**



  - **Depends on object age: number of collections it has survived**
  - **Tenuring age can change dynamically or can be hardcoded**
  - **Premature Tenuring: Young is full and object is tenured too early**
    - **Results in lots of expensive Old GCs**

Javaspecialists.eu
JavaSpecialists.EU

Stop-The-World (STW)

- **With GC, we try to avoid Stop-The-World events**
  - **This is when our *application threads* are stopped whilst GC happens**
  - **Happens in both young and old GCs**
  - **Remember Amdahl's Law**
    - **Even a small serial section (in our case STW) has a significant impact on scalability**

Javaspecialists.eu

JavaSpecialists.EU

Throughput Collector (Parallel)

- **Objects in Young and Old are collected in parallel**
  - **Total number of threads set relative to number of CPUs**
- **Stop-the-World pauses can be shorter, due to parallelism**
  - **Chunks must be large enough -XX:ParGCCardsPerStrideChunk**
- **Suitable for applications that utilize all cores and where the occasional longer delay does not matter**
  - **e.g. number crunching with Fork/Join or parallel streams**
- **Enable with flag**
  - **-XX:+UseParallelOldGC (Default in Java 7 and 8)**
  - **"PS Scavenge" for Young and "PS MarkSweep" for Old**

Javaspecialists.eu

JavaSpecialists.EU

Tuning the Throughput Collector

- **Adaptive Sizing (enabled by default)**
  - The JVM attempts to find the best sizes for the spaces
    - View with -XX:+PrintAdaptiveSizePolicy
    - Can make GC less efficient as it tries to resize automatically
      - Turn it off with -XX:-UseAdaptiveSizePolicy
- **Number of worker threads controlled by**
  - -XX:ParallelGCThreads=n
  - Default value is 8 + ((N - 8) * 5 / 8)
    - Thus a 64-core machine would use 8 + ((64 - 8) * 5 / 8) = 43 threads
  - Needs to be tuned if yours is not the only JVM running on the server

Javaspecialists.eu

JavaSpecialists.EU

Tuning the Throughput Collector

- We can set the maximum GC pauses for young and old
  - -XX:MaxGCMinorPauseMillis=n
  - -XX:MaxGCPauseMillis=n
  - But be realistic - if you make it too small, you will have more GCs!
- GC Time Ratio
  - % of time in GC is 1/(1+GCTimeRatio)
  - Default is 1% (GCTimeRatio=99), but can be more lenient (up to 5%)

% in GC	GCTimeRati
1%	99
2%	49
3%	32
4%	24
5%	19

Javaspecialists.eu

JavaSpecialists.EU

Concurrent Collector (CMS)

- **Parallel GC for objects in Young and concurrent for Old**
- **Shorter Stop-the-World makes application responsive**
  - **Uses spare cores to run Old GC in the background**
  - **e.g. your single-threaded application runs on a multi-core machine**
- **Increased overhead due to extra book keeping**
  - **Very slow single-threaded Stop-The-World if it cannot keep up**
- **Enable with flag**
  - **-XX:+UseConcMarkSweepGC**
  - **Uses “ParNew” for Young and “ConcurrentMarkSweep” for Old**

Javaspecialists.eu

JavaSpecialists.EU

Concurrent Collector

- **Splits collection into 7 phases**
  - ***CMS-initial-mark: stop-the-world***
    - **usually quite fast**
  - **CMS-concurrent-mark: concurrent**
  - **CMS-concurrent-preclean: concurrent**
  - **CMS-concurrent-abortable-preclean: concurrent (optional)**
  - ***CMS-remark: stop-the-world***
    - **should be fast if all goes well**
  - **CMS-concurrent-sweep: concurrent**
  - **CMS-concurrent-reset: concurrent**
- **preclean starts once young space is 50% full or 2 minutes are over**
  - **This avoids costly back-to-back Stop-The-World events**

Javaspecialists.eu

JavaSpecialists.EU

Costly CMS Stop-the-World Events

- **Concurrent Mode Failure**
  - During a Young GC it is discovered that the Old Space is too small
    - Similar to a Full GC in the Throughput Collector
- **Promotion Failed**
  - The Old Space is large enough, but too fragmented
  - Forces a full Mark Sweep Compact with a single thread
- **Perm/Meta GC**
  - CMS does not collect perm spaces in Java 6 and 7 by default
    - Enable with -XX:+CMSClassUnloadingEnabled, but it is STW
  - A full GC is triggered when we run out of space

Javaspecialists.eu

JavaSpecialists.EU

Tuning the Concurrent Collector

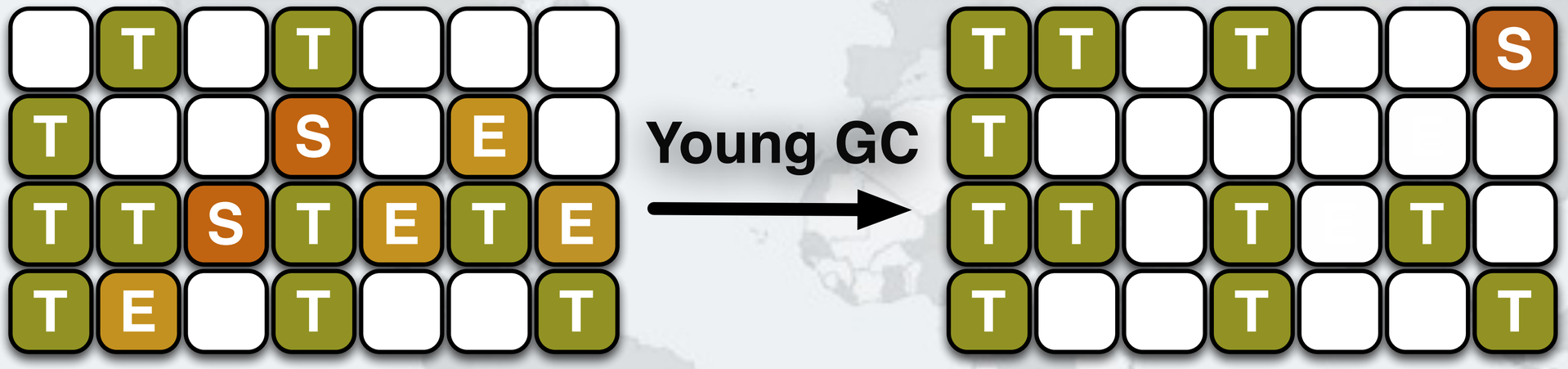
- **Number of CMS worker threads**
  - **Default is 1/4 of ParallelGCThreads**
  - **Can be overridden with -XX:ConcGCThreads=n**
    - **Should be related to number of *idle* cores on your machine**
- **Allocate more heap and start background thread sooner**
  - **-XX:+UseCMSInitiatingOccupancyOnly to use occupancy as trigger**
  - **-XX:CMSInitiatingOccupancyFraction=n fraction when GC happens**
    - **e.g. -XX:CMSInitiatingOccupancyFraction=50 - GC happens at 50%**

Javaspecialists.eu

JavaSpecialists.EU

Garbage-First Collector (G1)

- **Concurrent collector, optimized for large memory spaces**



  - **The heap is typically divided into 2048 regions**
    - **Regions are 1, 2, 4, 8, 16, or 32M**
    - **Each can be either Eden (E), Survivor (S) or Tenured (T)**
    - **We again have a ratio or young to old, like the other collectors**
  - **When a Young GC occurs, objects in Eden regions are moved to a new Survivor region**
    - **Older survivor objects are "Tenured"**

Javaspecialists.eu
JavaSpecialists.EU

Garbage-First Collector (G1)

- **Concurrent GC of Tenured spaces**

- During "Marking", we find tenured regions that are mostly garbage
  - Marked with the letter "G"
    - These can be efficiently collected using a copy collector
    - Reduces the need for compaction later
- Since this is concurrent, young GCs could happen simultaneously

Javaspecialists.eu

JavaSpecialists.EU

Tuning the G1 Collector

- **Avoiding concurrent mode failures causing full GCs**
  - Make sure that the maximum memory is large enough
- **Set maximum pause time**
  - G1 has a maximum GC pause time of 200 ms
  - Collects all young regions and as many old as fit in the time budget
    - It is ignored if we have more young regions than our budget allows
  - Maybe reduce it with -XX:MaxGCPauseMillis=n
    - But not too small, otherwise GC frequency increases
- **Increase G1 background threads**
  - Similar to CMS tuning, concurrent threads are roughly the same
  - Can be overridden with -XX:ConcGCThreads=n

Javaspecialists.eu

JavaSpecialists.EU

Heap Sizing

- "Adaptive sizing" adjusts heap automatically to minimize Stop-the-World events
- Heap too large
  - GC events happen seldom, but each can take a long time
- Heap too small
  - GC events happen frequently, causing contention
  - We might see OutOfMemoryErrors
- Somewhere in between is good
  - Typically between 30% and 60% more than the stable size
  - CMS usually needs a bit more, as it does not compact

Javaspecialists.eu

JavaSpecialists.EU

Young Generation Sizing

- To minimize full GC we collect objects in young space
  - Short-lived objects should remain in young and be collected
  - Long-lived objects should be evacuated as quickly as possible
- We can enlarge Eden and the Survivor Spaces
  - Object stay in survivor spaces longer
- We can also set Tenuring Threshold
  - This says at what age objects are evacuated to old space
  - Is usually modified dynamically be the JVM, based on GC conditions

Javaspecialists.eu

JavaSpecialists.EU

Young Size Switches

- We can use these JVM settings to configure young
  - -XX:NewRatio=n
    - Ratio between new and old generation
    - NewRatio=4 would mean New : Old = 1 : 4
      - Heap size of 1GB, would lead to a 200MB new generation.
  - -XX:SurvivorRatio=n
    - Ratio between one survivor space and eden
    - SurvivorRatio=8 would mean Survivor : Eden = 1 : 8
      - With 200MB new space, we would have two 20MB survivor spaces and one 160 MB eden space

Javaspecialists.eu

JavaSpecialists.EU

Tenuring Distribution

- The age at which objects are promoted to Tenured space
  - A number between 1 and 15, # of Young GCs object has survived
  - Usually starts at 7 (-XX:InitialTenuringThreshold=n) with a maximum of 15 (-XX:MaxTenuringThreshold=n)
  - Adaptive sizing changes it automatically to avoid premature tenuring
- These flags are all-or-nothing (not recommended)
  - -XX:+AlwaysTenure
    - Any live objects in Eden are promoted to Tenured
  - -XX:+NeverTenure
    - Live objects are stuffed into Survivor until it is full and then promoted

Javaspecialists.eu

JavaSpecialists.EU

A Large Fibonacci Number

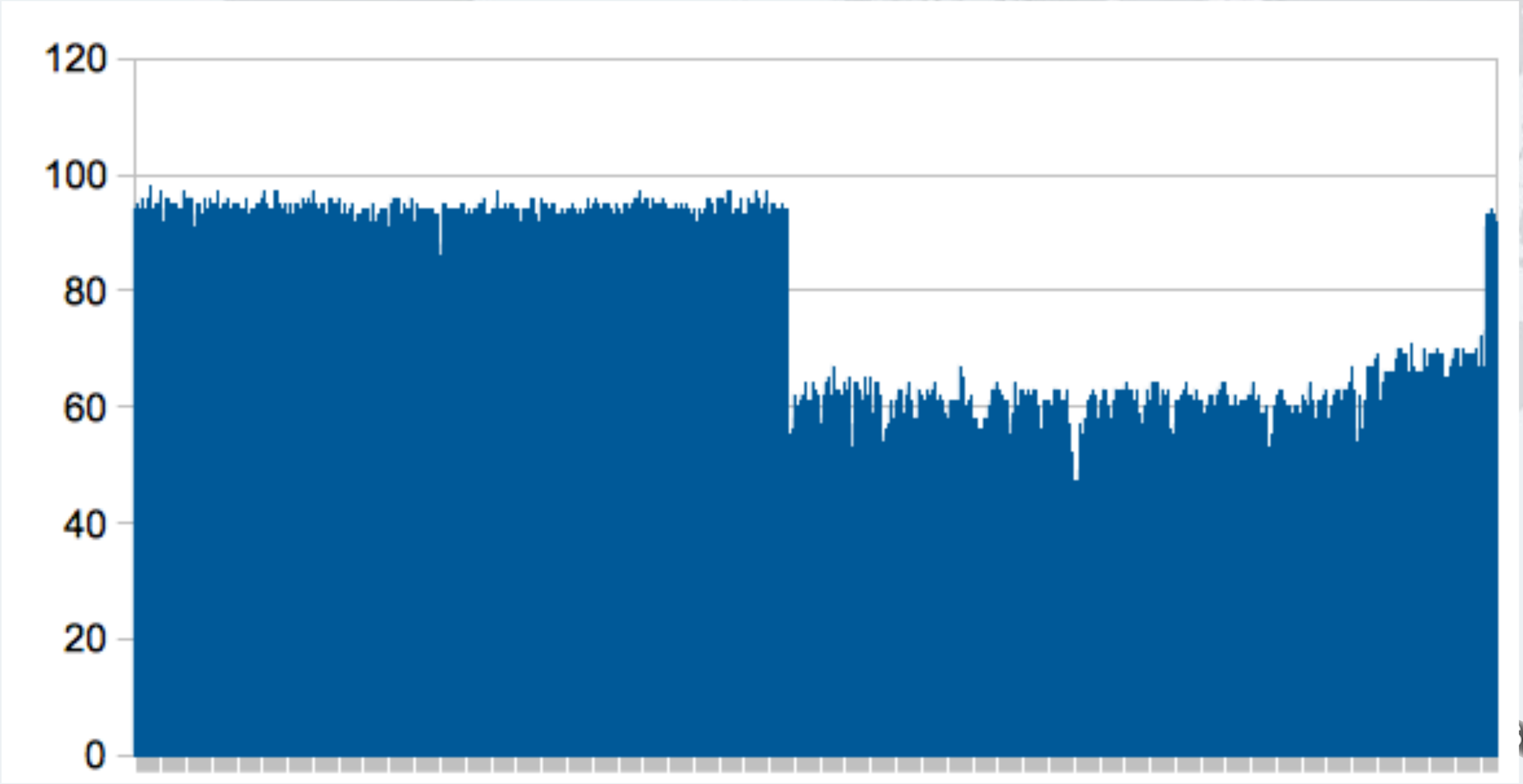
- We wanted to calculate Fibonacci of one billion
- Algorithm was based on Dijkstra's sum of squares
- Parallelized with fork/join
- We used BigInteger, but with a parallel Karatsuba multiplication

Javaspecialists.eu

JavaSpecialists.EU

Cpu Graph

- **8-core machine, we never reached 100% CPU and for half the time, it was at 60%**



Javaspecialists.eu
Specialists.EU

GC Log Summary HP JMeter (not Apache)

Heap Capacity

	Eden	Survivor	Old	Perm	Total
Initial Capacity	N/A	N/A	N/A	N/A	119.75 (MB)
Final Capacity	N/A	N/A	N/A	N/A	1.772 (GB)
Peak Capacity	N/A	N/A	N/A	N/A	1.937 (GB)
Peak Usage of Capacity	N/A	N/A	N/A	N/A	99.468%

GC Activity Summary

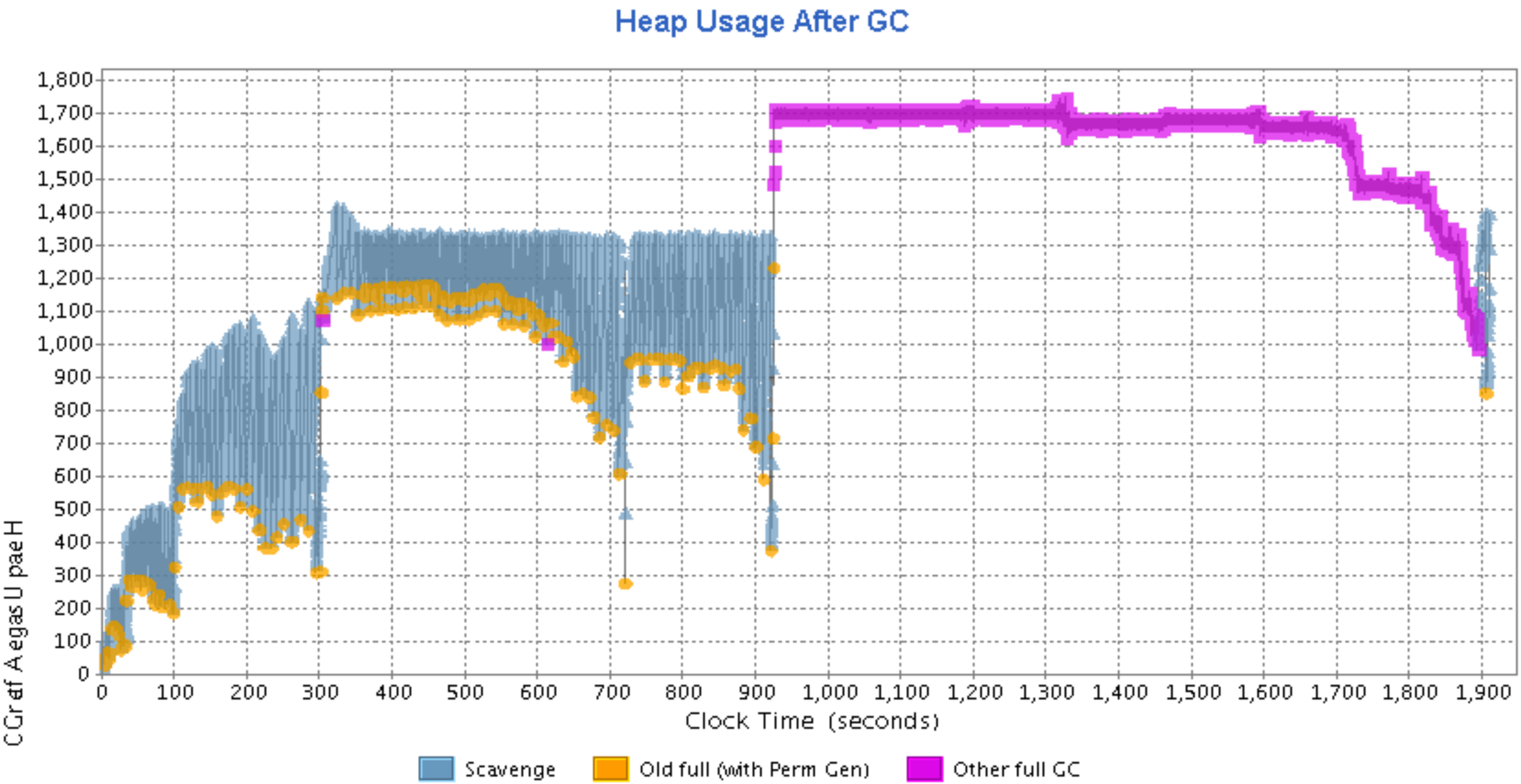
	Last occurrence (s)	Count	Average interval (s)	Average duration (s)	Average rate of collection
Scavenge	1,910.579 (s)	6,200	0.308 (s)	0.009 (s)	62.442 (GB/s)
Old full (with Perm Gen)	1,905.872 (s)	193	9.909 (s)	0.087 (s)	9.902 (GB/s)
Other full GC	1,894.055 (s)	9,728	0.163 (s)	0.04 (s)	6.12 (GB/s)

Overall Statistics

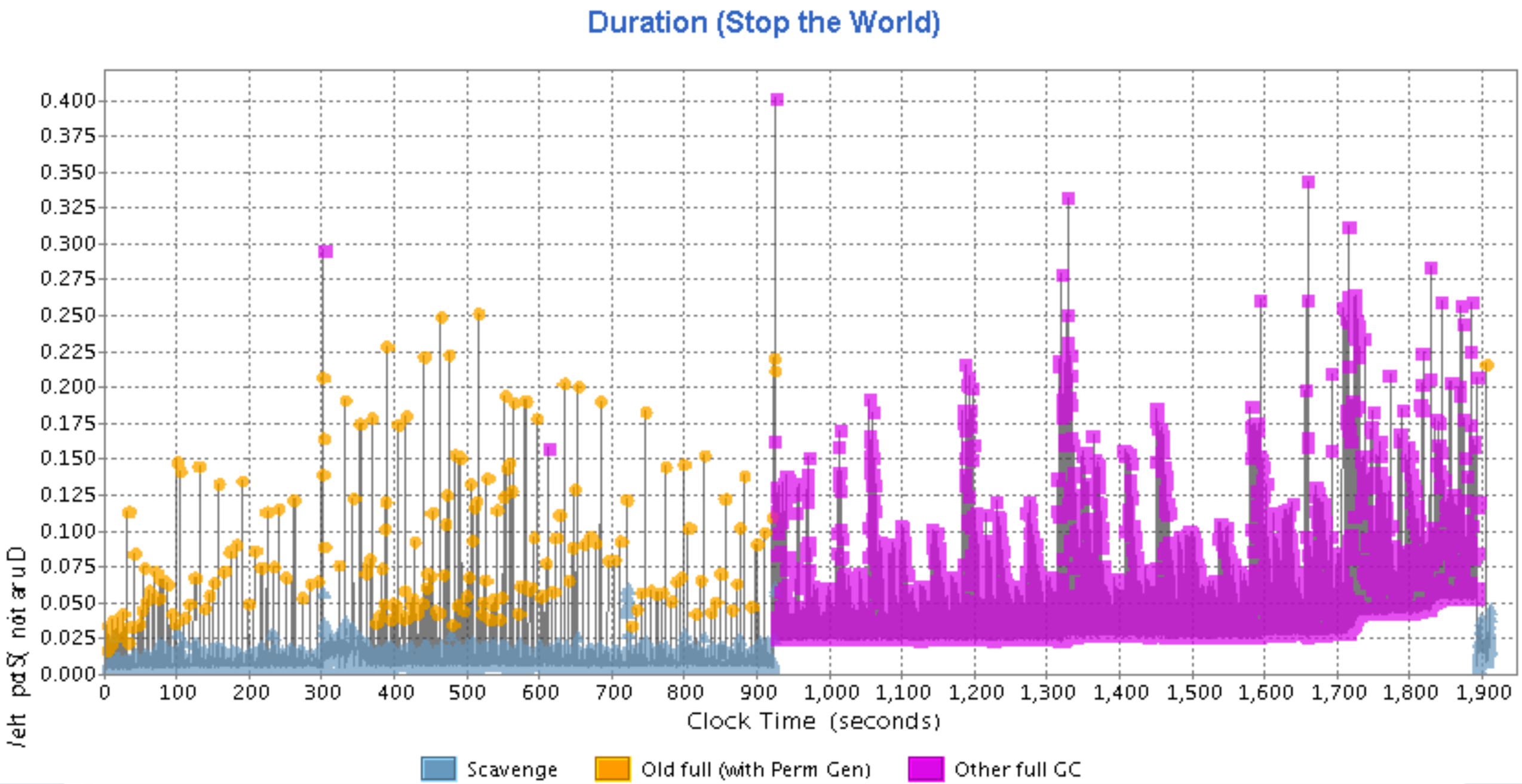
Name	Value	Name	Value
Duration of the measurement	1,948.787 (s)	Time spent in GC	464.772 (s)
Total bytes allocated	5.983(TB)	Percentage of time in GC	**23.849%**
Number of GC events	16,121	Time spent in Full GC	407.652 (s)
Average bytes allocated per GC	389.157 (MB)	Percentage of time in Full GC	**20.918%**
Avg. ideal allocation rate	4.128 (GB/s)	Avg. allocation rate	3.144 (GB/s)
Residual bytes	1.362 (GB)		

Time in GC: 23.85% Time in Full GC: 20.92% Full GC to GC: 87.71%

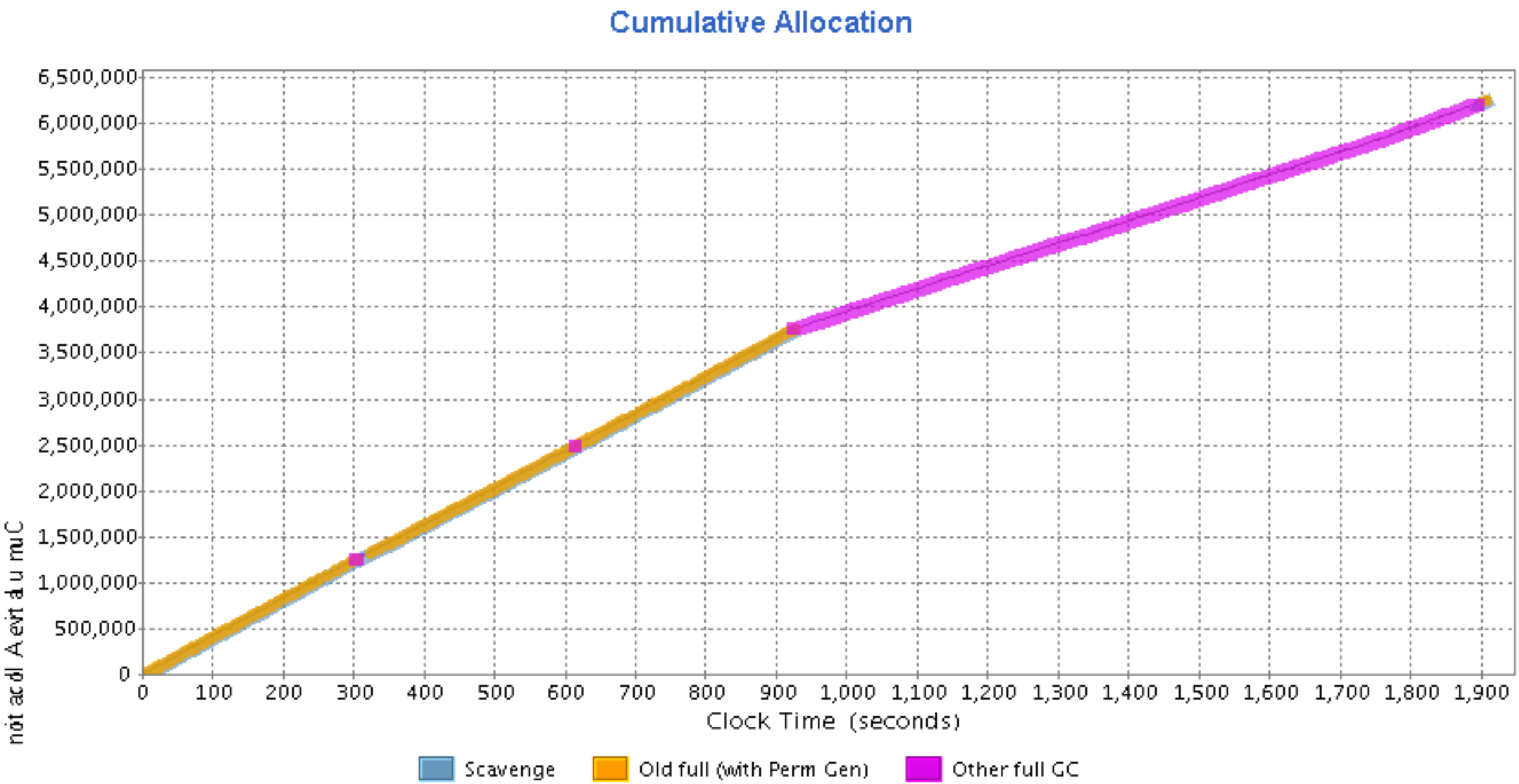
Heap Usage After Gc



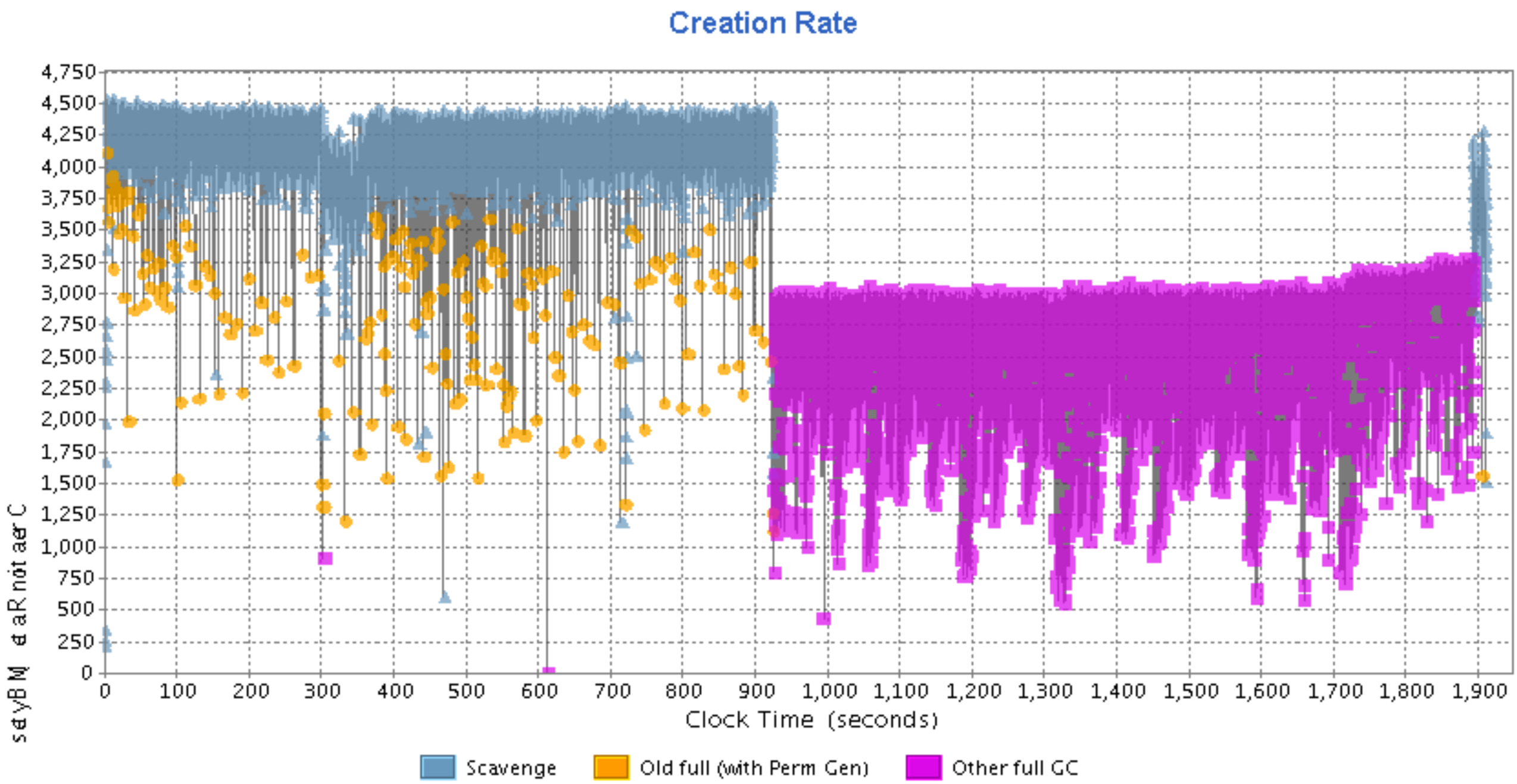
Duration (Stop The World)



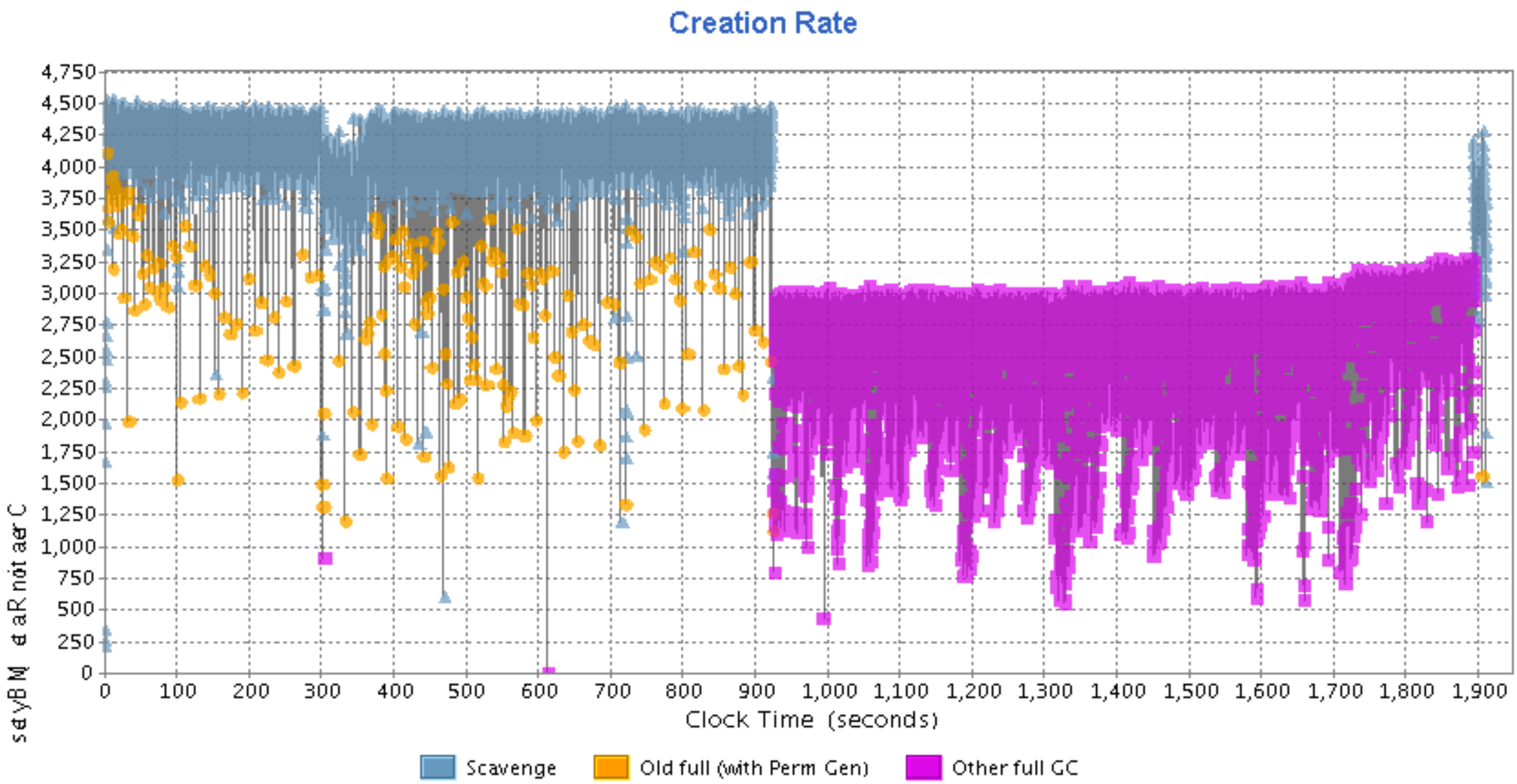
Cumulative Allocation



Creation Rate

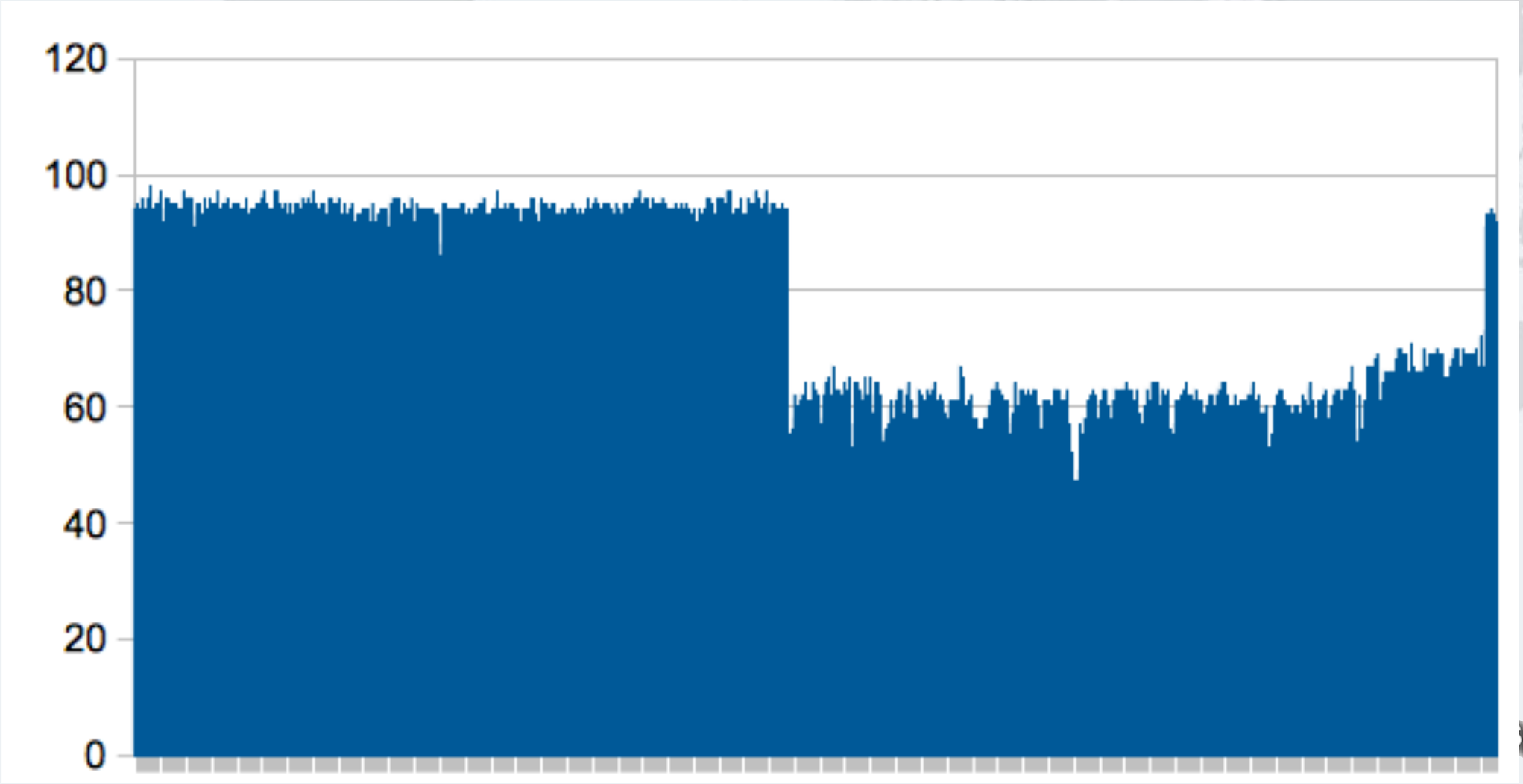


Where Did We See That Before?



Cpu Graph Mirrors Creation Rate

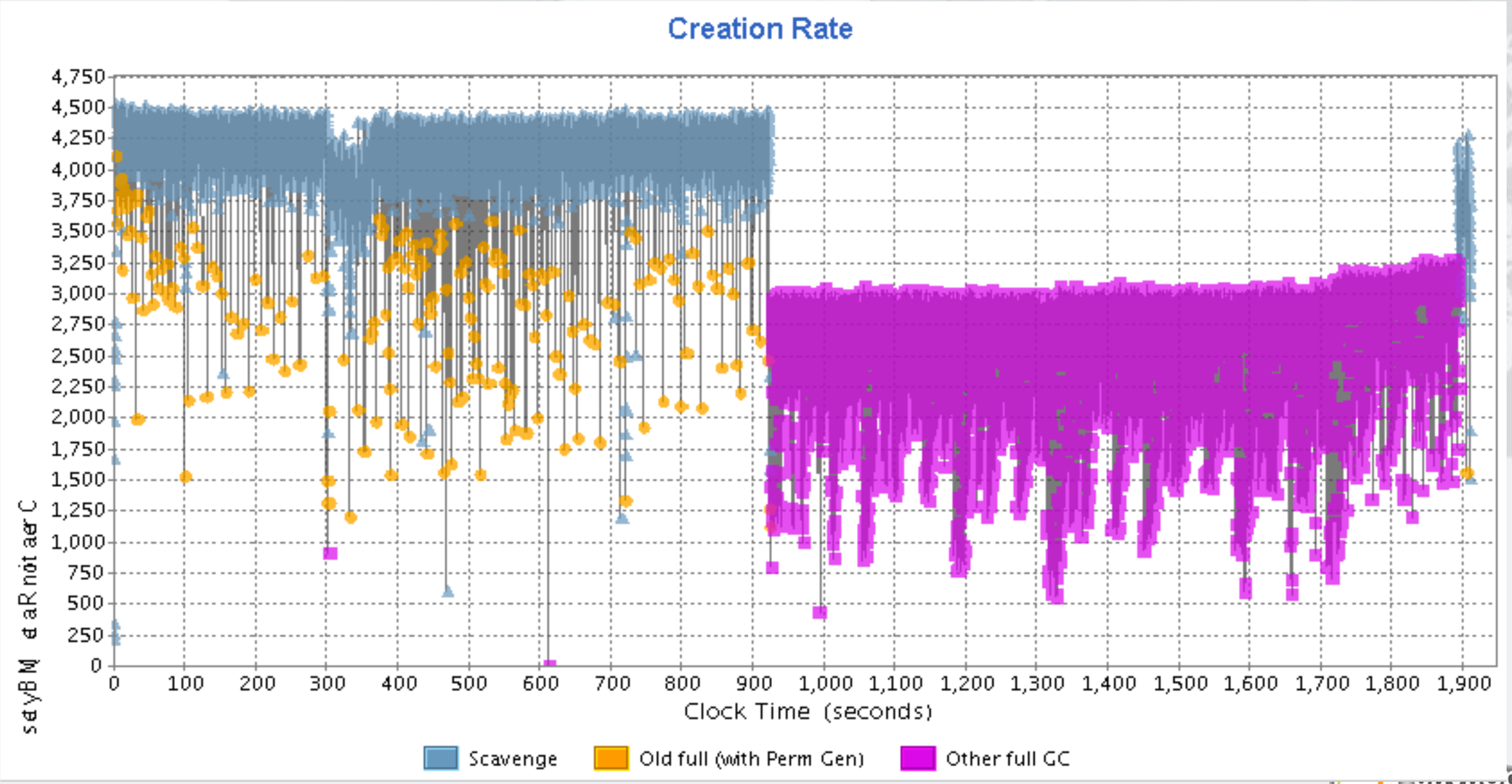
- **Almost the same as creation rate graph!**



Javaspecialists.eu
Specialists.EU

What Caused This?

- **Note Scavenge GCs vs Full GCs**



Javaspecialists.eu

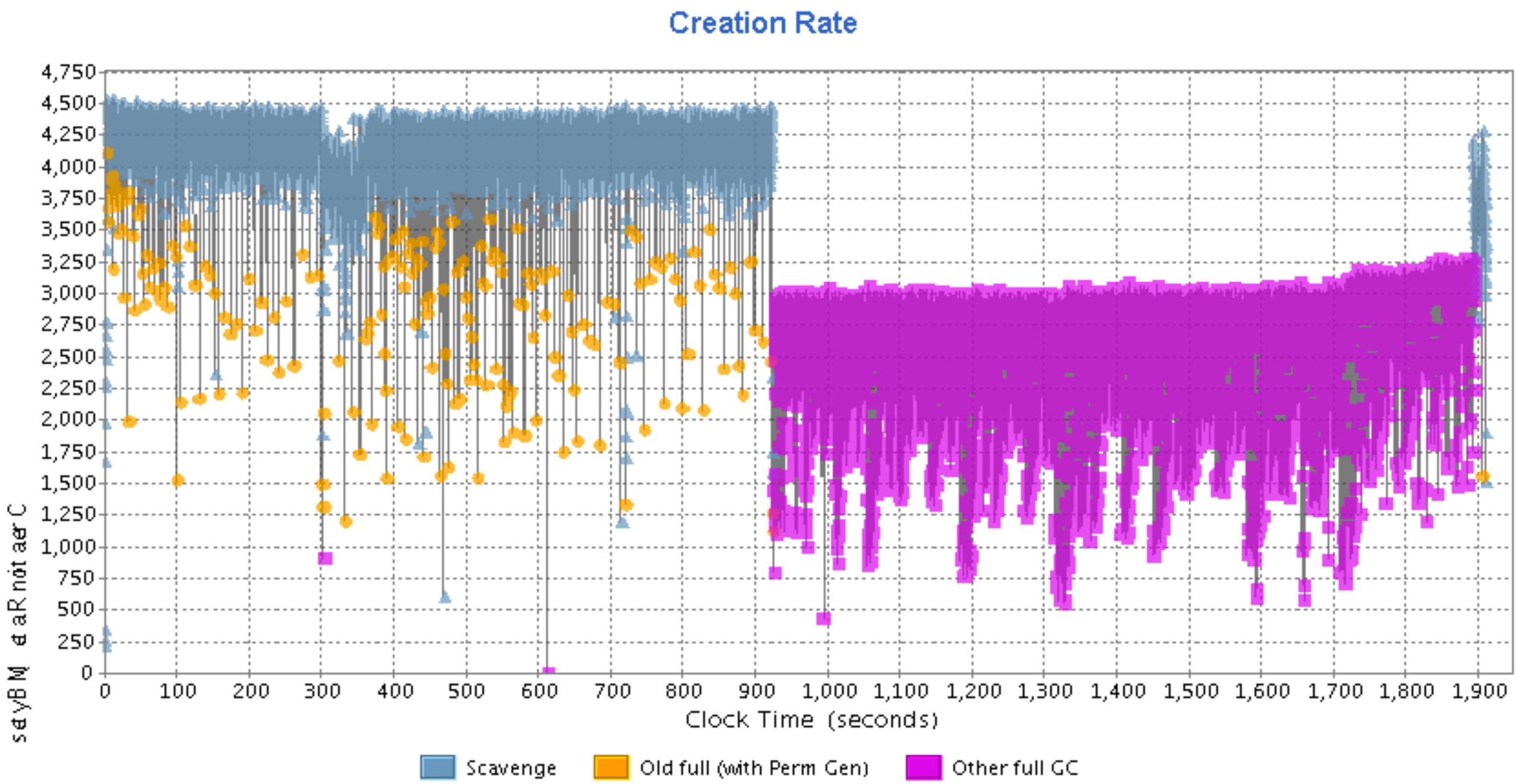
Cause

- **BigInteger contains a large int[]**
  - **For large numbers, this did not fit into young space**
  - **Created directly in old**
  - **Hence the many full GCs**

Javaspecialists.eu

JavaSpecialists.EU

How Can We Solve This?



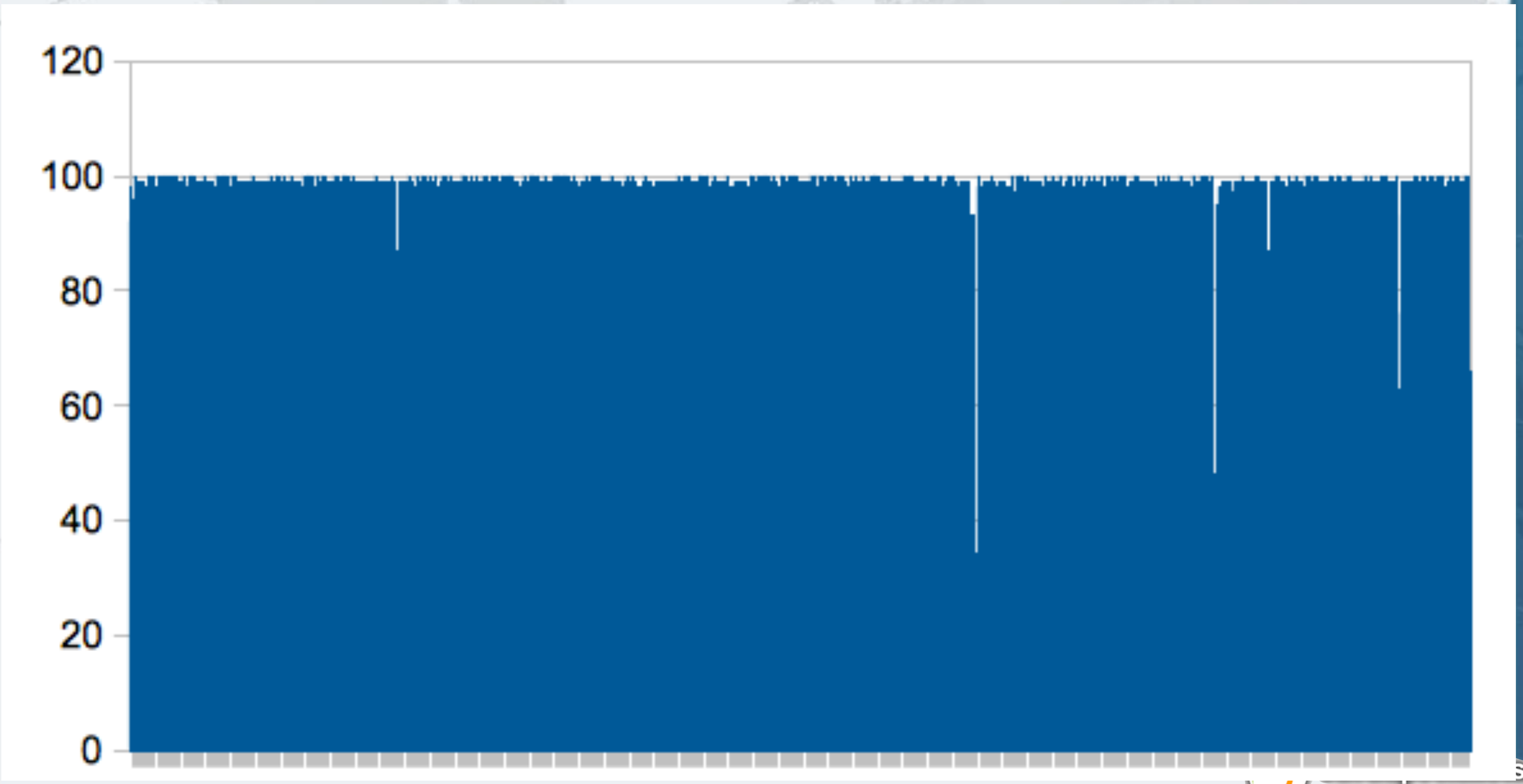
Settings For Large Eden And Survivor

- **Used settings -Xmx8g -XX:NewSize=4g -XX:SurvivorRatio=1**
- **Resulted in**
  - **8 GB total heap**
  - **4 GB young generation**
  - **1.3 GB survivor spaces**

Javaspecialists.eu

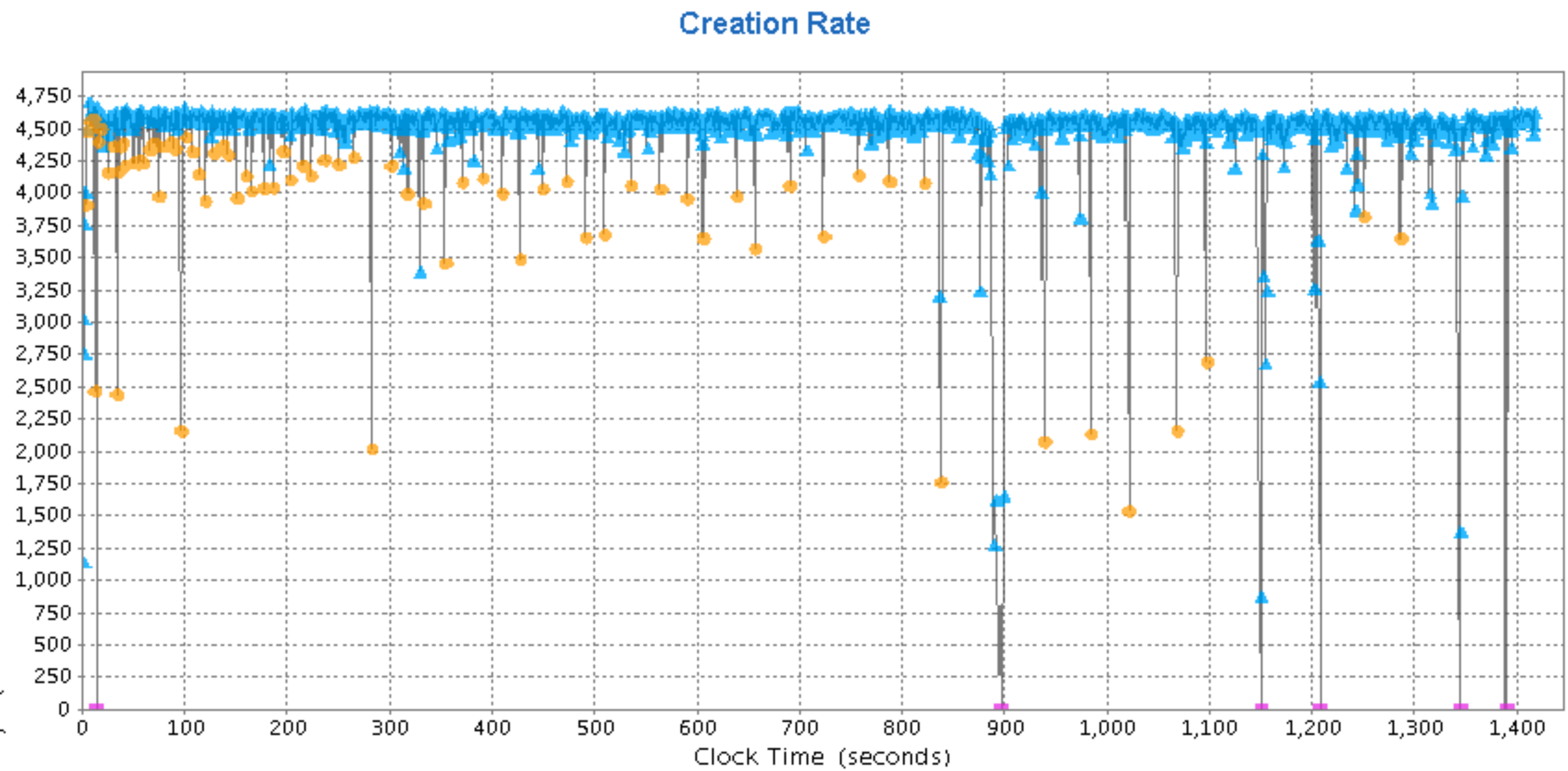
JavaSpecialists.EU

New Cpu Graph



Javaspecialists.eu

New Creation Rate



Performance Improvements

- **By simply changing the configuration, we went from 1910 to 1417 seconds**
  - **26% performance improvement**

Javaspecialists.eu

JavaSpecialists.EU

Consequence Of Bad Settings

- **We also tried with a large eden and tiny survivor spaces**
- **Completely flattened the server**
  - **Had to force a hard reset**

Javaspecialists.eu

JavaSpecialists.EU

Summary Page

Heap Capacity

	Eden	Survivor	Old	Perm	Total
Initial Capacity	N/A	N/A	42.188 (MB)	20.75 (MB)	4.041 (GB)
Final Capacity	N/A	N/A	N/A	N/A	8 (GB)
Peak Capacity	N/A	N/A	4 (GB)	20.75 (MB)	8 (GB)
Peak Usage of Capacity	N/A	N/A	136.241%	13.234%	99.425%

GC Activity Summary

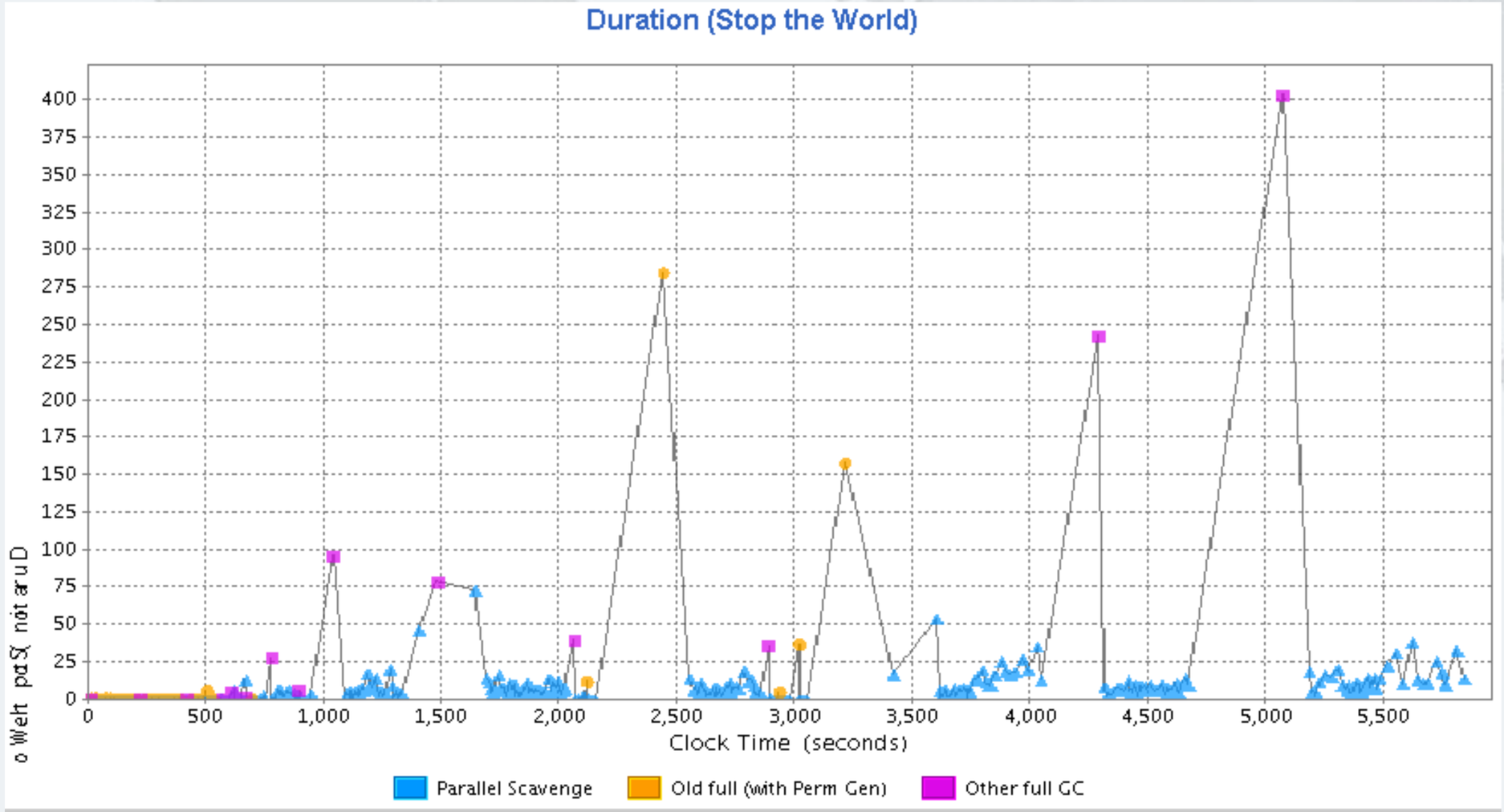
	Last occurrence (s)	Count	Average interval (s)	Average duration (s)	Average rate of collection
Parallel Scavenge	5,844.247 (s)	1,265	4.615 (s)	1.704 (s)	2.314 (GB/s)
Old full (with Perm Gen)	3,215.734 (s)	60	54.467 (s)	8.479 (s)	557.492 (MB/s)
Other full GC	5,076.196 (s)	14	389.131 (s)	66.704 (s)	32.534 (MB/s)

Overall Statistics

Name	Value	Name	Value
Duration of the measurement	5,961.088 (s)	Time spent in GC	3,598.302 (s)
Total bytes allocated	5.176(TB)	Percentage of time in GC	**60.363%**
Number of GC events	1,339	Time spent in Full GC	1,442.594 (s)
Average bytes allocated per GC	3.958 (GB)	Percentage of time in Full GC	**24.2%**
Avg. ideal allocation rate	2.243 (GB/s)	Avg. allocation rate	910.408 (MB/s)
Residual bytes	3.958 (GB)		

Javaspecialists.eu
JavaSpecialists.EU

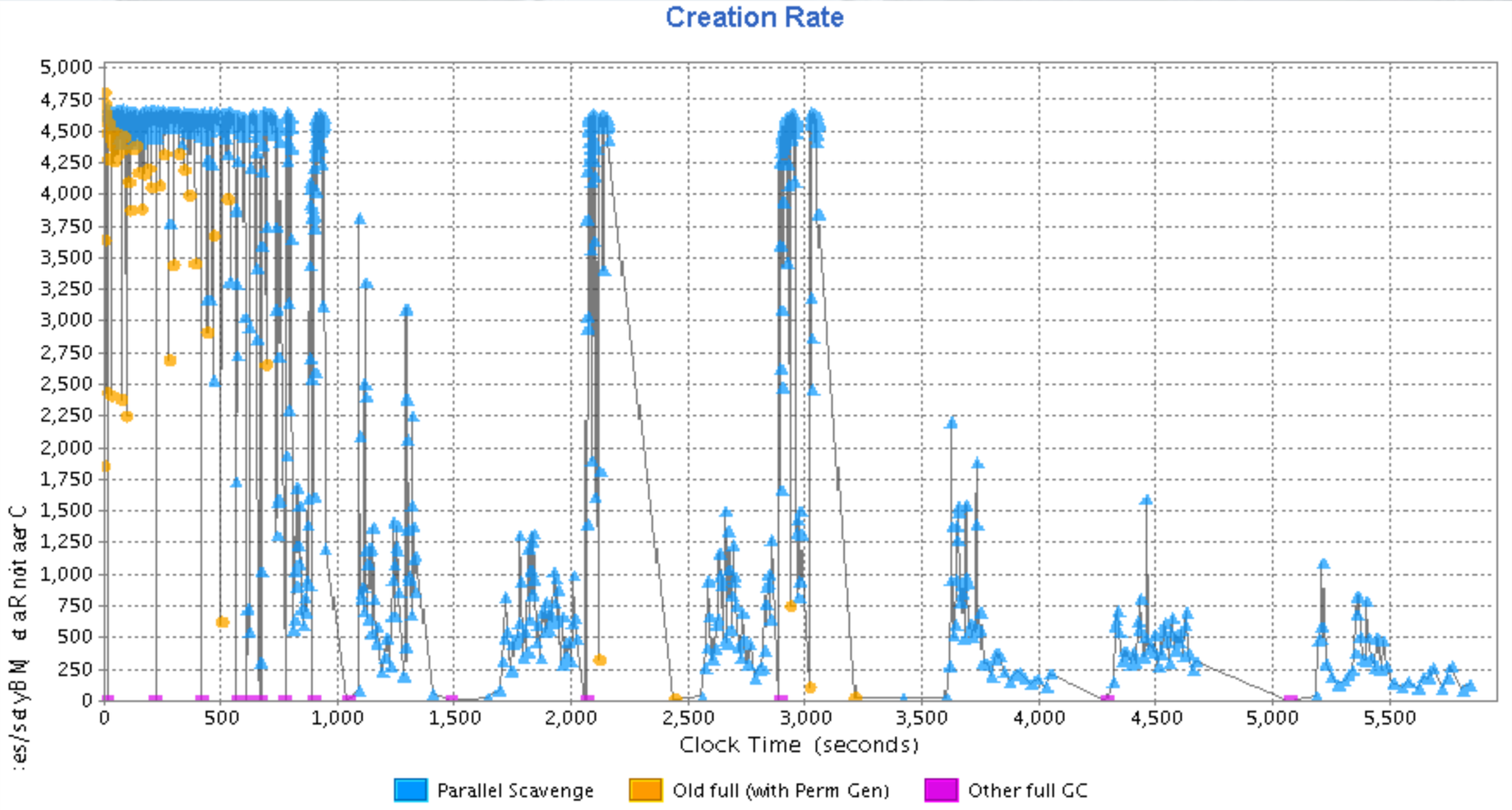
Duration - Over 400S!



Javaspecialists.eu

JavaSpecialists.EU

Creation Rate Is Down



Javaspecialists.eu

JavaSpecialists.EU

Garbage Collection – Measuring GC Activity

- **First step in any tuning – *measure!***
- ***Always* log the GC activity to a file**
  - **java -Xloggc:filename <application>**
  - **E.g. java -Xloggc:java2demo.gc -jar Java2Demo.jar**
- **Inside java2demo.gc:**

```
626.479: [GC 8364K->7420K(14276K), 0.0042340 secs]
626.853: [GC 8443K->7555K(14276K), 0.0057060 secs]
627.360: [GC 8579K->7698K(14276K), 0.0059210 secs]
628.052: [GC 8722K->7826K(14276K), 0.0039920 secs]
629.060: [GC 8850K->7864K(14276K), 0.0023520 secs]
629.799: [GC 8888K->7891K(14276K), 0.0022830 secs]
629.835: [Full GC 8080K->7329K(14276K), 0.1772550 secs]
632.841: [GC 8353K->7364K(14276K), 0.0017530 secs]
636.482: [GC 8388K->7365K(14276K), 0.0010690 secs]
640.172: [GC 8389K->7390K(14276K), 0.0010820 secs]
```

Javaspecialists.eu

JavaSpecialists.EU

Reading GC Logs

- **GC logs are neither "human" nor machine readable**
  - **Format changes all the time, so parsers break**
    - **HP JMeter can read this output, so maybe fix yours manually**

```
626.479: [GC 8364K->7420K(14276K), 0.0042340 secs]
```

  - **Output is generated by different threads, so it is jumbled up**
  - **But it is also very difficult to read, for example**

```
0.187: [GC (Allocation Failure)
Desired survivor size 16777216 bytes, new threshold 7 (max 15)
[PSYoungGen: 98304K->16352K(114688K)] 98304K->86785K(245760K), 0.0345901 secs] [Times: user=0.04 sys=0.19,
real=0.04 secs]
0.222: [Full GC (Ergonomics) [PSYoungGen: 16352K->0K(114688K)] [ParOldGen: 70433K->77212K(275456K)] 86785K-
>77212K(390144K), [Metaspace: 4208K->4206K(1056768K)], 0.0158416 secs] [Times: user=0.08 sys=0.01, real=0.02
secs]
```

    - **(Now imagine reading 10000 lines of that output!)**

- **The only tool that can parse them is Censum**
  - **By Kirk Pepperdine: www.jclarity.com**

Javaspecialists.eu

JavaSpecialists.EU

Generating GC Logs

- We have lots of flags for generating GC logs
  - -Xloggc:filename.vgc - writes the log to the specified file
  - Rotate GC logs with:
    - -XX:GCLogFileSize=n (minimum 8k) -XX:NumberOfGCLogFiles=n
    - -XX:+UseGCLogFileRotation
  - -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps
  - -XX:+PrintGCDetails - gives more information about the spaces
  - -XX:+PrintGCCause - shows why the GC event happened
  - -XX:+PrintTenuringDistribution - shows premature tenuring
  - -XX:+PrintGCApplicationStoppedTime - shows STW times
- We usually leave GC logging on in production

Javaspecialists.eu

JavaSpecialists.EU

14.1: Exercises

Garbage Collection

JavaSpecialists.EU

14.1: Tune Different Garbage Collectors

- **Run the ObjectCreation class with -XX:+UseParallelOldGC -Xmx1g to get a baseline speed**
  - **Check the logs for full GCs (-Xloggc:parold.vgc). See if you can reduce the full GCs by setting the "New Ratio" (-XX:NewRatio=n) or the "Survivor Ratio" (-XX:SurvivorRatio=n). See also what happens when you set the maximum GC pause time lower or higher (-XX:MaxGCPauseMillis=num).**
- **Next run it with -XX:+UseConcMarkSweepGC -Xmx1g**
  - **Again, try improve by setting the new and survivor ratios**
- **We then run it with -XX:+UseG1GC -Xmx1g**
- **Lastly, we run the tests again with -Xmx4g**

Javaspecialists.eu

JavaSpecialists.EU

14.2: References

14.2: References

- **Typically, an object will not be garbage collected if it is somehow reachable**
- **Java allows us to define different types of references**
  - **Soft**
  - **Weak**
  - **Phantom**

Javaspecialists.eu

JavaSpecialists.EU

References – Object Reachability

- **Reachability strength**
  - "Strongly reachable" is visible via a normal pointer
  - "Softly reachable" is only visible via a soft reference
  - "Weakly reachable" is only visible via weak reference
  - "Finalizer reachable" is only visible via the Finalizer
  - "Phantom reachable" is only visible via a phantom reference
  - "Not reachable" object can now be garbage collected

Javaspecialists.eu

JavaSpecialists.EU

Using References – SoftReference

- **Soft references are released when no strong reference points to the object and the memory is needed**

```
public class SoftReferenceTest {
  public static void main(String... args) {
    String name = new String("Heinz"); // dynamic
    Reference<String> refStr =
      new SoftReference<>(name);
    name = null;
    System.out.println(refStr.get());
    System.gc();
    System.out.println(refStr.get());
    try { byte[][] b = new byte[1024][1024 * 1024 * 1024]; }
    catch(OutOfMemoryError e) {}
    System.out.println(refStr.get());
  }
}
```

```
Heinz
Heinz
null
```

Javaspecialists.eu

JavaSpecialists.EU

Use Case for SoftReference

- **Lazy Initialization in java.lang.Class**
  - **Class reflection information is costly to calculate**
    - **We thus only want to create it as needed**
  - **However, it also occupies a lot of memory**
    - **We thus want to clear this information when not needed**
- **Issues with SoftReference**
  - **Does not know how costly the referenced object is**
    - **Both in memory usage and in creation cost**
  - **Only looks at time of last access and kilobytes free**
  - **Collection is quadratic - do not overuse!**

Javaspecialists.eu

JavaSpecialists.EU

Using References – WeakReference

- **Weak references are released when no strong reference points to the object and the garbage collector has run**

```
public class WeakReferenceTest {
  public static void main(String... args) {
    // need to make dynamic String
    String name = new String("Heinz");
    Reference<String> refStr =
      new WeakReference<>(name);
    name = null;
    System.out.println(refStr.get());
    System.gc();
    System.out.println(refStr.get());
  }
}
```

```
Heinz
null
```

Javaspecialists.eu

JavaSpecialists.EU

Use Case for WeakReference

- **Prevent memory leaks**
  - **FailFastCollection holds weak references to its iterators**
    - **http://www.javaspecialists.eu/archive/Issue171.html**
  - **Dynamic proxy cache holds weak references to class loaders**
  - **java.util.logging uses them to prevent memory leaks between hierarchies of loggers**
  - **Threads hold weak references to its ThreadLocals**
  - **WeakHashMap uses**
    - **Weak references for the keys**
    - **Strong references for the values**
    - **Any method on the WeakHashMap might be delayed as it cleans up the map of expired entries**

Javaspecialists.eu

JavaSpecialists.EU

Phantom References

- Object is not strongly, softly, weakly or finalizer reachable
- All objects after finalization will be Phantom reachable
- Applications using Phantom must call clear() before the reference will be collected

Javaspecialists.eu

JavaSpecialists.EU

Use Case for PhantomReference

- **Phantom references are more flexible than finalizers**
  - **Not used much**
  - **MemoryMappedBuffers are *unmapped* with PhantomReferences**
    - **Not predictable when they will be unmapped**
    - **Causes issues on Windows as you run out of file handles**

Javaspecialists.eu

JavaSpecialists.EU

Viewing GC Costs with References

- **We can see the costs of reference collecting with**
  - **-XX:+PrintReferenceGC -XX:+PrintGCDetails**

```
11.653:
  [Full GC (Ergonomics)
    13.238: [SoftReference, 0 refs, 0.0000338 secs]
    13.238: [WeakReference, 55 refs, 0.0000080 secs]
    13.238: [FinalReference, 51 refs, 0.0000167 secs]
    13.238: [PhantomReference, 0 refs, 0.0000025 secs]
    13.238: [JNI Weak Reference, 0.0000058 secs]
    [PSYoungGen: 32K->0K(555008K)]
    [ParOldGen: 654472K->564443K(860160K)]
    654504K->564443K(1415168K),
    [Metaspace: 5074K->5074K(1056768K)],
    1.9708780 secs]
  [Times: user=15.14 sys=0.05, real=1.98 secs]
```

Javaspecialists.eu

JavaSpecialists.EU

14.2: Exercises

References

Exercise 14.2.1: WeakReferences (Opt)

- **The FailFastCollection causes an exception when you modify a collection where another thread is iterating**
- **In the FailFastCollection, we need to implement a mechanism to check for active iterators:**

```
private void checkForCurrentIteration() {
  // check whether we have any cleared weak
  // references as keys in the iterators map.
  // If we do, remove them.  Otherwise, force a
  // full GC (System.gc()) and then check again
  // for cleared weak references.  If we still
  // have uncleared weak references to iterators,
  // throw a ConcurrentModificationException
}
```

- **Why can we not use finalize() or ReferenceQueue?**

Javaspecialists.eu

JavaSpecialists.EU

14: Java Memory

Questions?

15: Java Optimizations

JavaSpecialists.EU

15: Java Optimizations

- **We start by looking at the tuning process**
  1. **Find the bottlenecks quickly**
  2. **Verify them**
  3. **Fix them**
  4. **Check that it has improved things and still works**
  5. **If we have not reached our target, GOTO 1**
- **Next we look at how we can control JIT and HotSpot**
- **Lastly we look at how to fix some typical CPU bottlenecks**

Javaspecialists.eu

JavaSpecialists.EU

15.1: Tuning Process

JavaSpecialists.EU

15.1: Tuning Process

- **Best practices get us big performance gains fast**
- **Performance can stress good design practices**
  - **Good design should win over performance**
  - **May require us to de-normalize the design**
  - **Resist temptation to optimize everything**
- **Know where to spend your efforts**
  - **Measure and benchmark**

Javaspecialists.eu

JavaSpecialists.EU

Big Gains Quickly

- **Amdahl's law applies**
  - Consider an 4 layered application
    - Servlet takes 10%
    - Business component takes 11%
    - EJB takes 23%
    - SQL takes 56%
  - Scenario 1, tuning Servlet gives 20x improvement
    - "Google" says that servlets are slow
    - 0.10/20 + 0.11/1 + 0.23/1 + 0.56 /1 = 0.905
  - Scenario 2, tuning SQL give 2x improvement
    - We *measure* and discover SQL is the bottleneck
    - 0.10/1 + 0.11/1 + 0.23/1 + 0.56/2 = 0.72

Javaspecialists.eu

JavaSpecialists.EU

Specify The Required Performance

- Specify the required performance
- Specify the required performance
- Specify the required performance
- Specify the required performance
- Specify the required performance
- Specify the required performance
- Specify the required performance
- Specify the required performance

Javaspecialists.eu

JavaSpecialists.EU

Javaspecialists.eu

Specify the required performance

JavaSpecialists.EU

Optimization Methodology

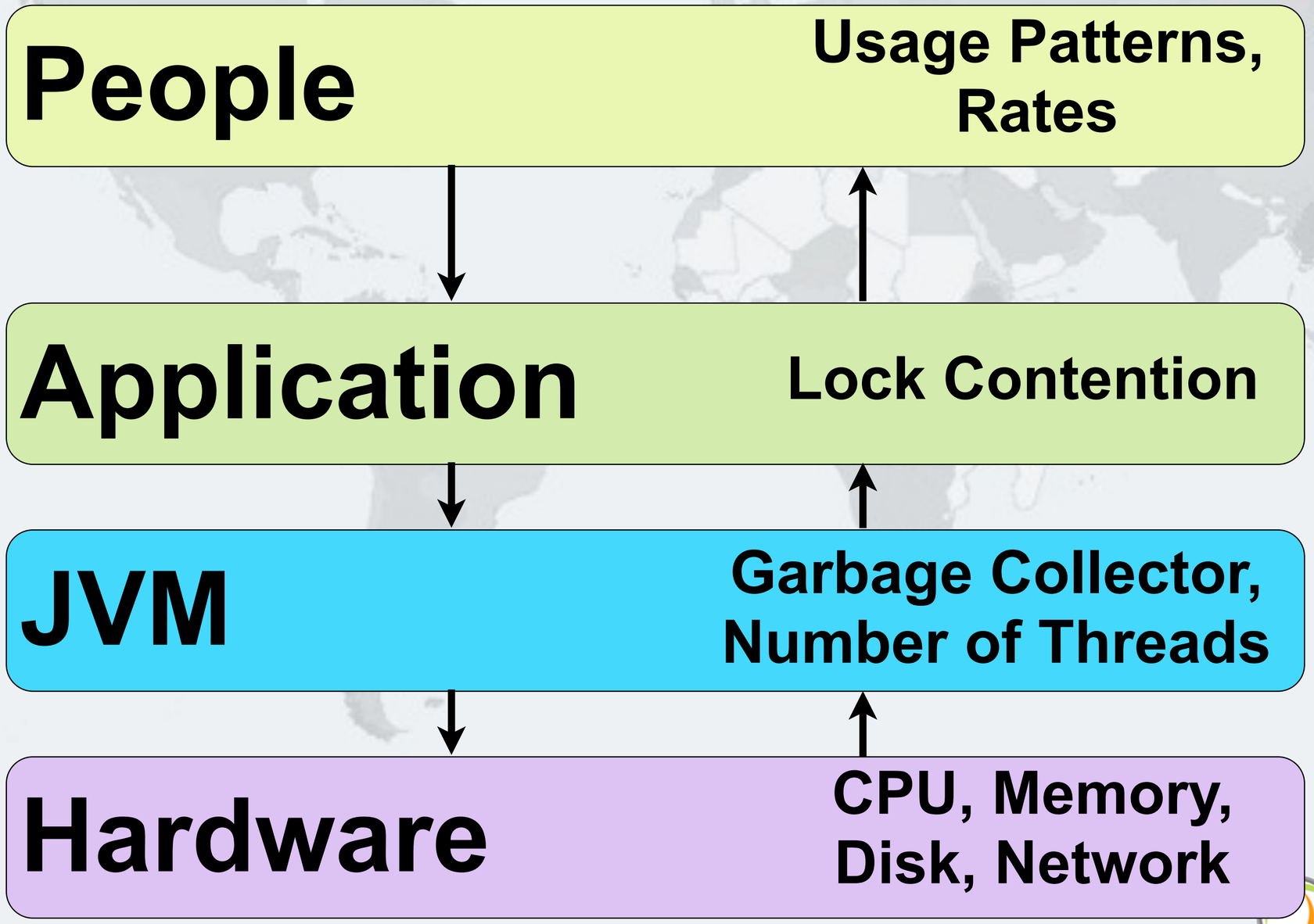
1. **Load test to identify bottlenecks**
   - **Identify the easiest to fix**
2. **Derive a hypothesis for the cause of the bottleneck**
   - **Create a test to isolate the factor identified by the hypothesis**
     - **This is important, we have often been fooled by profilers!**
3. **Alter the application or configuration**
4. **Test that the change improves the situation**
   - **Also make sure the system still works correctly**

- **Repeat process until targets are met**

Javaspecialists.eu

JavaSpecialists.EU

System Overview - The Box



Javaspecialists.eu
JavaSpecialists.EU

CPU Bottlenecks

- **Most performance problems appear as CPU bottleneck**
  - **Not necessarily a bottleneck in your application code**
    - **Garbage collection consumes 100% CPU when running**
  - **Need to identify who is keeping the CPU busy**
    - **Try to make them as silent as possible during tuning**

Javaspecialists.eu

JavaSpecialists.EU

Consumers of CPU

- **Operating system (OS) activities**
  - **Thread scheduling/context switching**
    - **Look for large runnable queues or high context switch number**
    - **Typically you would have high system time**
      - **But not always, this can be hidden from view by CAS**
  - **Interrupt handling**
  - **System starved for memory (page scan rates)**

Javaspecialists.eu

JavaSpecialists.EU

When Not To Use Clever Optimizations

- **Low level optimizations should only be applied in identified bottlenecks**
- **Heavily optimized (clever) code can be harder for the HotSpot compiler to identify and make faster**
- **Don't use clever optimizations throughout your code**
  - **You are wasting coding time and making it more difficult for future maintainers of your code**
  - **In addition, you might make the code slower overall!**

Javaspecialists.eu

JavaSpecialists.EU

Microbenchmarking

- **A microbenchmark should be used only to verify an identified bottleneck**
  - **It can be incredibly misleading**
  - **Need to understand the system context**
    - **Method f() takes one second, but is only called once a year!**
    - **Method g() takes 50ms and is called 10x a second!**
- **Limit the scope of what is being tested**
  - **Ask the right question**
    - **Ideally focus on a single effect or feature**

Javaspecialists.eu

JavaSpecialists.EU

Microbenchmarks Difficulties

- **Difficulties include:**
  - **Every layer may be caching**
  - **There may not be real contention**
  - **Even the hardware layer might be optimising our benchmark**
- **With Java, we have the added effects of**
  - **HotSpot profiler optimising code**
  - **Garbage collector**

Javaspecialists.eu

JavaSpecialists.EU

Microbenchmarking Harness

- **Java Microbenchmarking Harness (JMH) developed by Aleksey Shipilev of Oracle**
  - **http://openjdk.java.net/projects/code-tools/jmh/**
  - **http://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf**
  - **Helps in setting up a harness that**
    - **Gathers accurate statistics**
    - **Has low impact on the tests**
    - **Does good warmup of the HotSpot engine**
    - **Avoids dead-code elimination using “black holes”**
  - **Double-check results, remember Garbage-In-Garbage-Out effect**

Javaspecialists.eu

JavaSpecialists.EU

15.1: Exercises

Tuning Process

JavaSpecialists.EU

Exercise 15.1.1: Microbenchmark Example

- **The following sets of specifications define a function check on string data input. The function must check that a given string is**
  1. **an integer**
  2. **not the empty string or null**
  3. **an integer greater than 10**
  4. **an integer between 2 and 100000 inclusive**
  5. **That the first digit is 3.**
- **Required Performance: Should be 6x faster than current results**

Javaspecialists.eu

JavaSpecialists.EU

Programmer Produced This (*It Does Work*)

```
public boolean checkInteger(String testInteger) {
  try {
    //fails if not a number
    Integer theInteger = new Integer(testInteger);
    return
        (!theInteger.toString().equals("")) && //not empty
            (theInteger > 10) && //greater than ten
            ((theInteger >= 2) &&
                (theInteger <= 100000)) && //2>=X<=100000
            //first digit is 3
            (theInteger.toString().charAt(0) == '3');
  } catch (NumberFormatException err) {
    return false;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Run A Microbenchmark

- **Run the ValidatorTestHarness**
  - **This creates a dataset**
  - **It then reads in the data set and calls the testInteger method**
  - **Record the average and standard deviation**
- **Look at the CPU, Memory, GC, Disk and Network**
- **Do not modify the checkInteger() method**
  - **You may improve the ValidatorTestHarness**
- **You may need these settings to connect with JVisualVM**

```
-Dcom.sun.management.jmxremote.port=21000
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
```

Javaspecialists.eu
JavaSpecialists.EU

Windows Task Manager

File Options View Help

Applications | Processes | Performance

Select Columns

Select the columns that will appear on the Process page of the Task Manager.

- [x] Image Name
- [x] PID (Process Identifier)
- [x] CPU Usage
- [x] CPU Time
- [x] Memory Usage
- [ ] Memory Usage Delta
- [ ] Peak Memory Usage
- [ ] Page Faults
- [ ] USER Objects
- [x] I/O Reads
- [ ] I/O Read Bytes
- [ ] Page Faults Delta
- [x] Virtual Memory Size
- [ ] Paged Pool
- [ ] Non-paged Pool
- [ ] Base Priority
- [ ] Handle Count
- [x] Thread Count
- [ ] GDI Objects
- [x] I/O Writes
- [ ] I/O Write Bytes
- [ ] I/O Other
- [ ] I/O Other Bytes

OK | Cancel

sage	VM Size	Threads	I/O Reads	I/O Writes
16 K	0 K	1	0	0
12 K	24 K	34	2	868
68 K	1,096 K	6	179	48
56 K	1,168 K	10	7,209	0
28 K	5,340 K	17	177	79
88 K	2,280 K	27	31,104	29,120
44 K	1,920 K	14	1,939	1,368
08 K	720 K	3	0	0
00 K	1,180 K	9	469	470
08 K	3,156 K	10	138	138
88 K	1,940 K	13	1,519	58
20 K	520 K	4	124	25
24 K	276 K	2	3	3
76 K	1,024 K	6	79	22
92 K	4,132 K	6	206	201
60 K	948 K	3	33	10
64 K	848 K	5	308	12
68 K	808 K	3	8	8
08 K	2,712 K	11	600	51
08 K	2,140 K	2	497	496
68 K	2,988 K	7	263	2
60 K	1,468 K	9	5	2
24 K	328 K	1	0	88

End Process

Processes: 26 | CPU Usage: 0% | Mem Usage: 84360K / 260456K

Javaspecialists.eu

15.2: JIT And Hotspot

JavaSpecialists.EU

15.2: JIT and HotSpot

- Java used to be a strictly interpreted language (1.0, 1.1)
- The Java Virtual Machine would act like a simple CPU, taking the byte code instructions and executing them
- Instead of compiling the byte codes to native machine code, Sun developed a Just-In-Time compiler that would compile all the classes that were used (not everything) into machine code
  - This often did unnecessary work, making startup slower
  - On the positive side, the performance improvement was dramatic

Javaspecialists.eu

JavaSpecialists.EU

The Hotspot Profiler

- **HotSpot profiles the running application**
  - **Builds an execution profile**
  - **Uses the profile to build an execution plan**
    - **Directs the JIT on how to compile**
  - **Uses threshold values and other conditions to trigger the JIT**
    - **JIT compiles byte code to native code**
    - **Uses On Stack Replacement (OSR) to switch from byte code to native execution**
      - **Must pause the thread**
    - **May re-optimize or de-optimize based on further profiling**

Javaspecialists.eu

JavaSpecialists.EU

Client Hotspot (C1)

- **Client assumptions**
  - **Desktop**
    - **single CPU, small amount of memory**
  - **Shorter running processes**
  - **Startup and response times are critical**
- **Client settings**
  - **Shorter profiling times before optimization**
  - **Quicker optimization but maybe not as good**

Javaspecialists.eu

JavaSpecialists.EU

Server Hotspot (C2)

- **Server assumptions**
  - **Server class machine**
    - **Many CPUs, large amounts of memory**
  - **Longer running processes**
  - **Pause times and throughput are critical**
- **Server settings**
  - **Longer profiling times before optimization**
  - **Better optimizations**

Javaspecialists.eu

JavaSpecialists.EU

Tiered Compilation Hotspot

- On by default in Java 8
  - Control with -XX:+/-TieredCompilation
- Does a first-pass compile with the client HotSpot compiler
  - After a small threshold of calls, typically in the hundreds
  - First-pass compile is fast, but not as thorough
- Does second-pass compile with server HotSpot compiler
  - After a larger threshold, typically about 10000
  - Better optimizations, but take longer
- Print events with -XX:+PrintTieredEvents

Javaspecialists.eu

JavaSpecialists.EU

Vm Switches

- **Several switches affect HotSpot**
  - -server - Default on 64-bit or unix based JVMs
  - -client - Default on 32-bit non-unix JVMs
  - -XX:+/-TieredCompilation - turns on/off tiered compilation
  - -d32 or -d64
    - Uses either a 32-bit or 64-bit JVM
  - -Xmixed (default)
  - -Xint
  - -Xcomp
  - -showversion - Tells you what JVM version you are using
  - -XX:+PrintFlagsFinal - Prints all possible switches and their values

Javaspecialists.eu

JavaSpecialists.EU

Hotspot

- **Optimizations**
  - Code is run for a period of time before being compiled
  - Methods maybe inlined into the callers block
  - Methods may have to be dynamically de-optimized to account for code from a different class
  - Code maybe replaced whenever an optimization is recognized

Javaspecialists.eu

JavaSpecialists.EU

Hotspot

- **Effects on Benchmarking?**
  - **The execution of the code will stop when On Stack Replacement (OSR) takes place**
  - **Execution modal switches from interpreted to native**
- **How can we deal with this?**
  - **Pre-heat the method cache before taking any measurements**

Javaspecialists.eu

JavaSpecialists.EU

15.2: Exercises

JIT and HotSpot

JavaSpecialists.EU

Exercise 15.2.1:

- **Compare the performance using the JIT settings:**
  - **-Xmixed**
  - **-Xcomp**
  - **-Xint**
  - **(Thus three different runs)**
- **You will run it three times and record the time that it ran**
  - **You don't have to run the -Xint tests to completion**

Javaspecialists.eu

JavaSpecialists.EU

Exercise 15.2.2:

- **Pick the fastest run in exercise 15.2.1**
- **Increase the DATA_POINTS constant to 100**
- **Now use the "Sampler" in JVisualVM with the settings:**
  - **"Do not profile packages" - remove "java.*, "**
  - **Sampling Frequency - change to 20ms**
  - **Now click on CPU and let it run for a few minutes**
  - **Click on Snapshot, then Hot Spots, then select "main" thread**

Profiler Snapshot

View: Methods | main

Hot Spots – Method	Self time [%]	Self time	Self time (CPU)
java.lang.Throwable.**fillInStackTrace[native]** ()		6,837 ms (81%)	6,837 ms
java.lang.Integer.**getChars** ()		1,326 ms (15.7%)	1,326 ms
xj_conc.ch15_optimizations.exercise_15_2_1.Val		156 ms (1.9%)	156 ms
java.lang.Integer.**toString** ()		119 ms (1.4%)	119 ms

Javaspecialists.eu

JavaSpecialists.EU

15.3: Typical Problem Areas

JavaSpecialists.EU

15.3: Typical Problem Areas

- **Object creation often shows up as a CPU problem**
  - **Look at object creation rates in the GC Logs**
  - **Use memory or CPU profiler to find the culprit**
- **Bad algorithms cause excessive looping and repetition**
  - **Can best be repaired by better algorithm**
  - **Find this using a CPU profiler**

Javaspecialists.eu

JavaSpecialists.EU

Array Creation

- **An array object can be of virtually any size**
  - **Can easily fragment the heap**
  - **Ordinary objects are relatively constant in size**
- **Array creation usually causes the largest CPU bottlenecks**
  - **String (backed by an array)**
  - **BigDecimal / BigInteger (backed by arrays)**
  - **Collections (many backed by arrays)**

Javaspecialists.eu

JavaSpecialists.EU

Temporary Objects

- **Reduce the number of temporary objects being used**
  - **Especially in loops**
  - **Avoid creating temporary objects in frequently called methods**
  - **Reusing objects helps reduce object churn**
- **Escape analysis can let these be allocated on stack**

Javaspecialists.eu

JavaSpecialists.EU

Lazy Initializing Of Expensive Objects

- **Eliminate object-creation bottlenecks by moving object creation to an alternative time**
  - **Use lazy initialization when objects or variables may never be used, or when you need to distribute the load of creating objects**
    - **Be careful of concurrency though**
- **e.g. ArrayList**

```
public ArrayList() { // Java 6
  this(10);
}
```

```
public ArrayList() { // Java 7+
  super();
  this.elementData = EMPTY_ELEMENTDATA;
}
```

Javaspecialists.eu

JavaSpecialists.EU

Strings And Constants

- **String**
  - **Is widely used**
    - **Generally shows up near the top of any memory profile**
  - **Receives special treatment from the compiler**
    - **Has a literal form**
    - **Special operators '+' and '+='**
    - **Shared strings, used by `intern()`**
      - **View events with -XX:+PrintStringTableStatistics**
      - **Extend table with -XX:StringTableSize=n**

Javaspecialists.eu

JavaSpecialists.EU

Strings

- **String Objects:**
  - **Are usually created in large numbers**
  - **Are typically short lived and hence stress GC**
- **Often makes the tuning process a lot more interesting**
  - **In fact, it is often so difficult to deal with the performance problems that String usage can present, it is worth the effort to deal with the problems up front**

Javaspecialists.eu

JavaSpecialists.EU

String Types

- **String declaration:**

```
public final class String implements Serializable,
  Comparable, CharSequence
```

- **StringBuffer declaration:**

```
public final class StringBuffer implements Serializable,
  CharSequence, Appendable
```

- **StringBuilder declaration**
  - **Unsynchronized form of StringBuffer**

```
public final class StringBuilder implements Serializable,
  CharSequence, Appendable
```

Javaspecialists.eu

JavaSpecialists.EU

Strings Can Be Merged By Static Compiler

- **Consider:** `String a = "x" + "y" + 3;`
  - **Is merged to** `String a = "xy3";` **at static compile time**
  - **Only works if we are adding constants together**
    - **Constant Strings, primitives**

Javaspecialists.eu

JavaSpecialists.EU

String Causing Object Creation

- **Strings are unmodifiable**
  - **Any change will force a copy of the underlying** `char[]`
  - **The + and += operators will result in "hidden" object creation**

Javaspecialists.eu

JavaSpecialists.EU

Complexity of += vs. StringBuilder.append()

- **Complexity cost:**
  - **String += is O(n)**
  - **StringBuilder and StringBuffer append() is O(1)**
- **Thus with a loop of size n that adds chars together**
  - **String += costs $O(n^2)$, quadratic performance**
  - **StringBuilder.append() costs O(n), linear performance**

Javaspecialists.eu

JavaSpecialists.EU

String Vs. Stringbuilder

- **String and StringBuilder are tightly coupled**
  - **Designed to complement each other**
  - **Concatenation code inserts StringBuilders into byte code**
    - `"hello " + name + " how are you?"` **is compiled to:**

```
new StringBuilder().append("hello ").append(name).
    append(" how are you?").toString()
```

- **StringBuilder is mutable**
- **Easily Convertible to String**

Javaspecialists.eu

JavaSpecialists.EU

String Performance Conclusions

- **String performance is reasonable for query and poor for mutator operations**
- **Use String when the mutation can be calculated at compile time**
- **Use StringBuilder when the calculation can only be calculated at runtime**
  - **The String + operator generates StringBuilder code automatically**
    - **No need to change "Hello " + s + " world" to use StringBuilder**
  - **But, the += String operator makes new StringBuilders every iteration**

Javaspecialists.eu

JavaSpecialists.EU

Which Do You Think Is Fastest?

```
public class StringAppender {
  public static String appendBasic(
      String s1, String s2, String s3) {
    return "SELECT " + s1 + " FROM " + s2 + " WHERE " + s3;
  }
  public static String appendStringBuilder(
      String s1, String s2, String s3) {
    return new StringBuilder().append("SELECT ").append(s1)
        .append(" FROM ").append(s2).append(" WHERE ")
        .append(s3).toString();
  }
  public static String appendStringBuilderSize(
      String s1, String s2, String s3) {
    int len = 20 + s1.length() + s2.length() + s3.length();
    return new StringBuilder(len).append("SELECT ").append(s1)
        .append(" FROM ").append(s2).append(" WHERE ")
        .append(s3).toString();
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Performance Output

```
...

    long time = System.currentTimeMillis();
    for (int i = 0; i < REPEATS; i++) {
      StringAppender.appendBasic("Name,Phone", "Employee,Salary",
          "Sex='F' AND income > 100000 AND eyes='blue'");
    }
    time = System.currentTimeMillis() - time;
    System.out.println("appendBasic                    " + time);

...
```

- **appendBasic 1405**
- **appendStringBuilder 1351**
  - **Same byte code as appendBasic, so it will produce same result**
- **appendStringBuilderSize 672**

Javaspecialists.eu

JavaSpecialists.EU

But with -XX:+OptimizeStringConcat

- appendBasic 378
- appendStringBuilder 372
- appendStringBuilderSize 282
- And since Java 7, they perform about the same
  - String intrinsics

Javaspecialists.eu

JavaSpecialists.EU

Converting int to String

- **Logically, Integer.toString(i) should be faster than ""+i**
  - **Internally they look very similar**
    - **Calculate how long the int will be in String form**
    - **Make a char array or expand if necessary**
      - **StringBuilder starts with size 16; large enough**
    - **Use Integer.getChars() to write digits into char[]**
  - **However, -XX:+OptimizeStringConcat has more aggressive optimizations for the ""+i case**

Javaspecialists.eu

JavaSpecialists.EU

Converting String To Object

- **Converting a String to an Object is getting better but is fairly inefficient**
- **StreamTokenizer is highly un-optimized**
  - **Creates many temporary objects**
- ***Custom Tokenizer may result in significant performance improvements***

Javaspecialists.eu

JavaSpecialists.EU

String.substring()

- **String used to have an efficient method of returning a substring without copying the underlying array**
  - **Used to return a new String object sharing underlying** `char[]`
  - **Since Java 7, a new char[] is created**

Javaspecialists.eu

JavaSpecialists.EU

String Deduplication

- **Java 1.8.0_20 can replace char[]s of duplicate strings**
  - **Only works for the G1 collector -XX:+UseStringDeduplication**
  - **Threshold when deduplicated -XX:StringDeduplicationAgeThreshold**

```
public class DeduplicationDemo {
  public static void main(String... args) throws Exception {
    char[] heinz = {'h', 'e', 'i', 'n', 'z'};
    String[] s = {new String(heinz), new String(heinz),};
    Field value = String.class.getDeclaredField("value");
    value.setAccessible(true);
    System.out.println("Before GC");
    System.out.println(value.get(s[0]));
    System.out.println(value.get(s[1]));
    System.gc(); Thread.sleep(100);
    System.out.println("After GC");
    System.out.println(value.get(s[0]));
    System.out.println(value.get(s[1]));
  }
}
```

```
Before GC
[C@76ed5528
[C@2c7b84de
After GC
[C@2c7b84de
[C@2c7b84de
```

Javaspecialists.eu

Strings in Java 9

- **Strings in Java 9 are backed by byte[]**
  - Flag -XX:+CompactStrings is true by default
  - Uses one byte per char if characters are all Latin1 (ISO-8859-1)
- **Java 6 had flag -XX:+UseCompressedStrings**
  - String could contain either byte[] of char[]
  - Compression only worked with 7-bit ASCII characters
- **Any Java reflection code that assumes String is backed by char[] is now broken!**
  - String internals changed in Java 1.2, 1.3, 1.5, 1.7, 1.8 and 1.9!
  - Cannot assume anything about the gory internals

Javaspecialists.eu

JavaSpecialists.EU

Regular Expression Performance

- **Regular expressions (java.util.regex) are very useful**
  - **Flexible and powerful, but might create lots of objects**
  - **Best used in methods that are not on critical path**
  - **Can usually get better performance parsing Strings character-by-character yourself with charAt()**

Javaspecialists.eu

JavaSpecialists.EU

Exception Performance

- **Exceptions have two very expensive operations**
  - **fillInStackTrace() - called during Throwable construction**
  - **getStackTrace() - called when printStackTrace() is invoked**
- **Avoid constructing exceptions**
  - **When you *cause* NullPointerException, ClassCastException and ArrayIndexOutOfBoundsException, stack trace is optimized away**
    - **Turn off with -XX:-OmitStackTraceInFastThrow**
  - **Turn off *all* exception stack traces with -XX:-StackTraceInThrowable**
    - **This is only an emergency measure to circumvent stack traces**

Javaspecialists.eu

JavaSpecialists.EU

Loops

- Usually a CPU bottleneck contains a loop
- First step is to decrease complexity of algorithm
  - Temporarily store old results for faster calculations
  - Come up with mathematically better algorithms
- Code in loop should do as little as possible
  - Only do what is necessary inside loop
  - Move everything else outside the loop
  - HotSpot might move code out of the loop for you

Javaspecialists.eu

JavaSpecialists.EU

Caching Previous Results

- **Algorithm is exponential $O(k^n)$**
- **We are working out the values over and over**

```
public interface Fibonacci {
  BigInteger f(int n);
}

public class FibonacciPlain implements Fibonacci {
  public BigInteger f(int n) {
    if (n == 0) return BigInteger.ZERO;
    if (n == 1) return BigInteger.ONE;
    return f(n - 1).add(f(n - 2));
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Static Cache Of Previous Results

- **What issues do you see?**

```
public class FibonacciStaticCache implements Fibonacci {
  private final static Map<Integer, BigInteger> cache;
  static {
    cache = new ConcurrentHashMap<>();
    cache.put(0, BigInteger.ZERO); cache.put(1, BigInteger.ONE);
  }
  public BigInteger f(int n) {
    BigInteger result = cache.get(n);
    if (result == null) {
      BigInteger newVal = f(n-1).add(f(n-2));
      result = cache.putIfAbsent(n, newVal);
      if (result == null) result = newVal;
    } // unfortunately computeIfAbsent() won't work here
    return result;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Stack Cache Of Previous Results

- **Can still overflow stack, but no more memory leak**

```
public class FibonacciStackCache implements Fibonacci {
  public BigInteger f(int n) {
    Map<Integer, BigInteger> cache = new HashMap<>();
    cache.put(0, BigInteger.ZERO);
    cache.put(1, BigInteger.ONE);
    return f(n, cache);
  }
  private BigInteger f(int n, Map<Integer, BigInteger> cache) {
    BigInteger result = cache.get(n);
    if (result == null) {
      BigInteger result = f(n-1, cache).add(f(n-2, cache));
      cache.put(n, newVal);
   }
    return result;
  }
}
```

Javaspecialists.eu

JavaSpecialists.EU

Lessons From Caching

- **Can speed up code significantly**
- **Be careful of memory leaks**
  - **One of the most common causes in real systems**
- **Thread safety if cache is shared by multiple threads**
- **Benchmark might be faster than speed of light (caching)**

Javaspecialists.eu

JavaSpecialists.EU

Moving Code Outside Of Loops

- **Move out of the loop any code that is repeatedly executed with the same result**

```
for (...) {
  x[i] = h[i]*Math.cos(radians);
```

  - **Math.cos(radians) is invariant and can be moved out of the loop**

```
double cos = Math.cos(radians);
for (...) {
  x[i] = h[i]*cos;
```

Javaspecialists.eu

JavaSpecialists.EU

Method Calls In Tight Loops

- Avoid method calls in loops when acceptable
- For tightest loop tuning, when the loop has been identified as a bottleneck, eliminate method calls even if this requires rewriting or in lining
  - Check though whether the HotSpot compiler is doing this for you
  - See with -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining
- Even polymorphic calls can sometimes be inlined

Javaspecialists.eu

JavaSpecialists.EU

Arrays And Loops

- **Avoid unnecessary array access**

```
for (int i..) {
  for (int j..) {
    tot[i] += x[j];
```

  - **Use a temporary variable and assign back later**

```
for (int i..) {
  tot_i = tot[i];
  for (int j..) {
    tot_i += x[j];
  }
  tot[i] = tot_i;
```

  - **Remember that array elements are loaded into cache lines**
    - **Typically 64 bytes at a time**

Javaspecialists.eu

JavaSpecialists.EU

Loop Calls To Jni

- **Make sure that any loops using native method calls are converted so that the native call includes the loop instead of running the loop in Java**
- **Any loop iteration parameters should be passed to the native call.**
- **Replace**

```
for (int i=0; i<count; i++)
    native_call(x,y,...);
```

- **With**

```
native_call_looped(count, x,y,...);
```

Javaspecialists.eu

JavaSpecialists.EU

Problem Areas – Other Areas

- **Short-circuiting**
  - **Provides a fast fail mechanism for boolean expressions**

```
if (bool1 && bool2)
  this.execute();
```

  - **The first boolean should be the one that would cause the expression to fail most often**
  - **calls should avoid side-effects**
    - **queries should not mutate**
- **Boolean comparisons**
  - **|| and && are "short-circuit" boolean operators**
  - **& and | are bitwise operators and are never short-circuited**

Javaspecialists.eu
JavaSpecialists.EU

Other Areas – Final

- **Modifier that is used with**
  - **Local Variables and Parameters**
    - **Only necessary pre-Java 8 when accessing from inner classes**
  - **Fields**
  - **Methods**
  - **Classes**
- **Marks the "things" as unchangeable**
  - **Good for thread safety**
  - **Bad for extensibility**
- **Compile and run time optimizations**

Javaspecialists.eu

JavaSpecialists.EU

15.3: Exercises

Typical Problem Areas

JavaSpecialists.EU

Exercise 15.3.1: Tune Method

- **Find a faster way to write the FastValidator.checkInteger() method, using the techniques in this section**
- **Remember our target is to be 6x faster**
  - **We look mainly at "User Time" in our output as a guide**

Javaspecialists.eu

JavaSpecialists.EU

15: Java Optimizations

Questions?

16: Conclusion

Where to next?

JavaSpecialists.EU

16: Conclusion

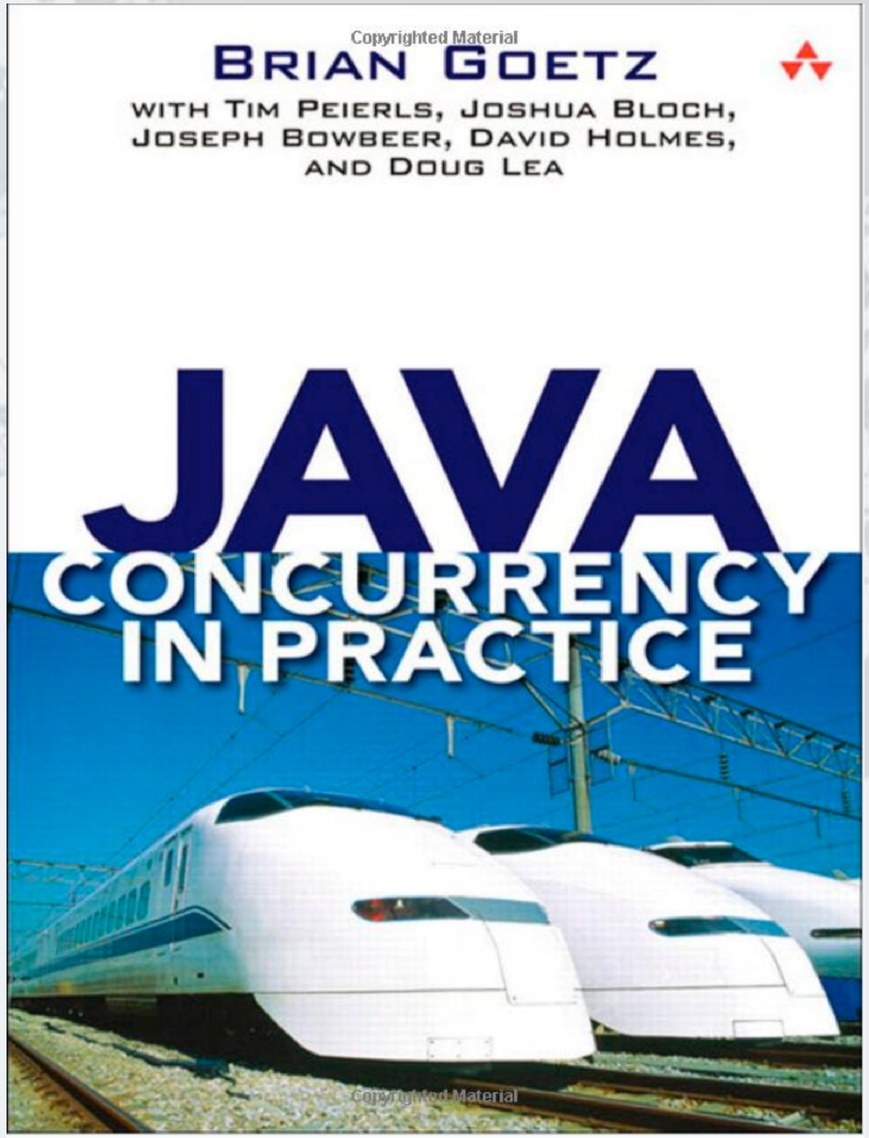
- **Congratulations if you have endured this far**
  - **Concurrency is a hard nut to crack**
  - **We covered a lot of material and some complex topics**
    - **Well done and <u>thank you!</u>**

Javaspecialists.eu

JavaSpecialists.EU

Java Concurrency In Practice

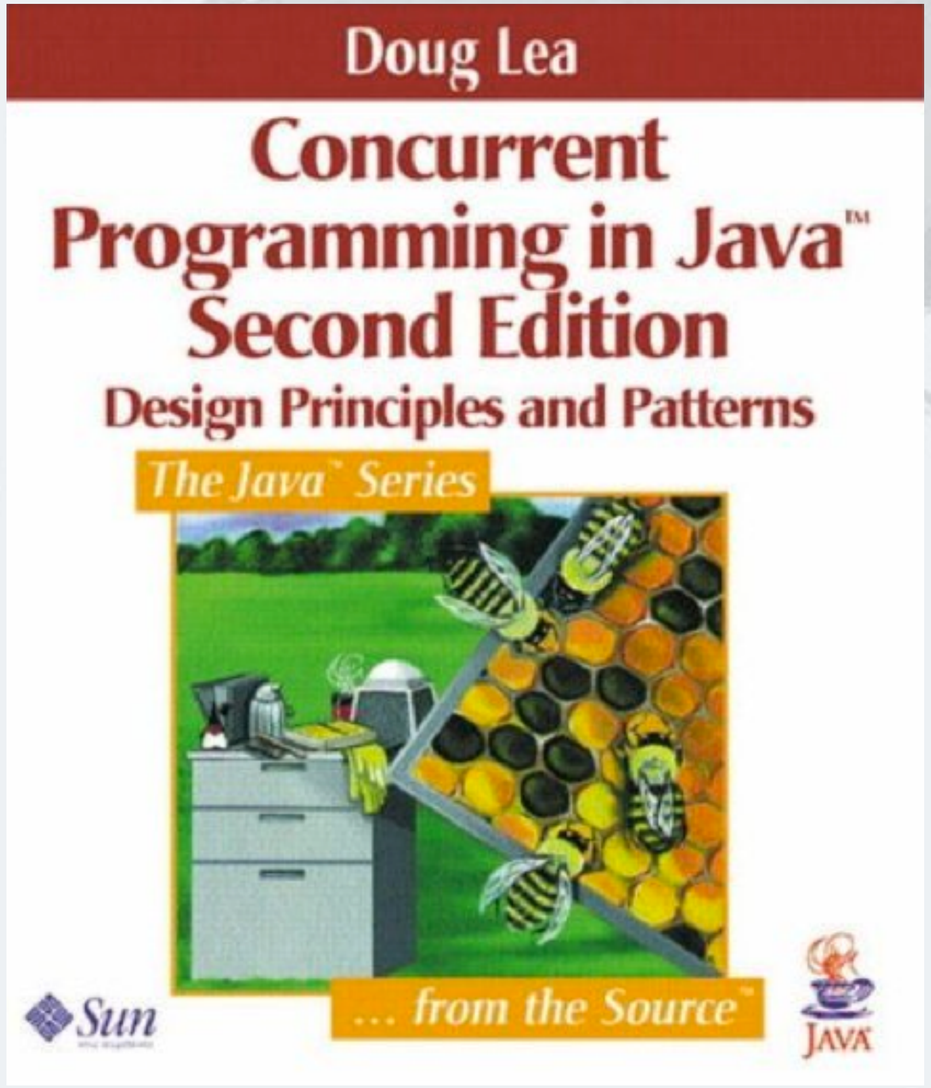
- **Brian Goetz**
  - **With Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes and Doug Lea**
- **Excellent resource to read**
- **Course is based on this book**



Javaspecialists.eu
JavaSpecialists.EU

Concurrent Programming In Java (2Nd Edition)

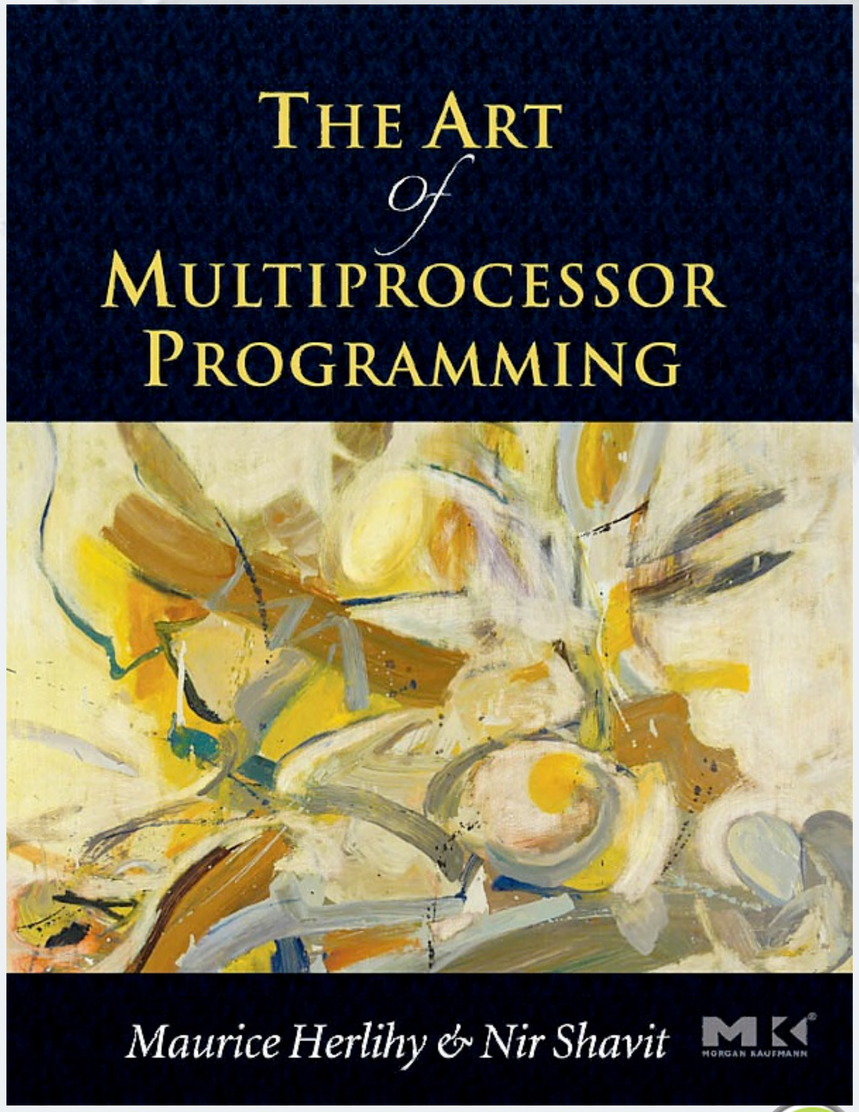
- **Doug Lea**
  - **The foundation of almost all concurrent classes in Java**
  - **A bit dated, but nice as background information**



Javaspecialists.eu
JavaSpecialists.EU

The Art Of Multiprocessor Programming

- **Herlihy & Shavit**
  - **Theoretical book on how things work "under the hood"**
  - **Good as background reading**



Javaspecialists.eu
JavaSpecialists.EU

Java Memory Model

- **Explains the foundation of how memory is managed**
- **Java Memory Model (JSR133)**
  - **http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf**

Javaspecialists.eu

JavaSpecialists.EU

Other Concurrency Related Resources

- **Discussions about Java concurrency by the experts**
  - **http://g.oswego.edu/dl/concurrency-interest**
- **Cliff Click's Highly Scalable Utilities**
  - **http://sourceforge.net/projects/high-scale-lib**
- **Writing concurrent code based on underlying hardware**
  - **https://groups.google.com/forum/#!forum/mechanical-sympathy**
- **JClarity Censum**
  - **https://groups.google.com/a/jclarity.com/forum/#!forum/friends**

Javaspecialists.eu

JavaSpecialists.EU

Get to Know Your IDE

- **Spend a couple hours learning the shortcuts to your IDE**
  - **You will win back the time quickly due to accelerated coding**
- **Learn some unix system tools**
  - **grep / awk / sed / wc / uniq / etc.**
- **Figure out your system monitoring tools**

Javaspecialists.eu

JavaSpecialists.EU

Never Stop Learning New Things

- **Software Engineering knowledge has 3 year half-life**
  - **Half of what you know today will be obsolete in 3 years time**
  - **Some things last longer, for example, Design Patterns**
- **Read books, magazines, blogs**
- **Subscribe to The Java Specialists' Newsletter**
  - **http://www.javaspecialists.eu/archive**

Javaspecialists.eu

JavaSpecialists.EU

The End – Thank You!

Please feel at liberty to contact us on

http://www.javaspecialists.eu/contact.html