JavaSpecialists

Mastering Platform Threads in Java 1.3

Dr Heinz M. Kabutz

Last updated 2023-11-22

© 2007-2023 Heinz Kabutz – All Rights Reserved

Copyright Notice

- **© 2007-2022 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.
  - No part of this course material may be used for internal company training
- **Please contact heinz@javaspecialists.eu if you are in any way uncertain as to your rights and obligations.**

JavaSpecialists

1: Welcome

Comfort and Learning

- **We need**
  - oxygen
  - short breaks every 45 minutes
  - physical exercise after class
    - Run, walk, gym, cycle, etc.

Questions

- **There are some stupid questions**
  - They are the ones we did not ask
  - Once we have asked them, they are not stupid anymore
- **The more we ask, the more everyone learns**

Exercises

- **We don’t learn how to cycle from lectures**
- **After each lecture, we do exercises**
  - During exercises, real learning takes place
  - 50% of time spent applying our knowledge

Hints and Solutions

- **Our exercises also have solutions**
  - masteringthreads.ch2_basics_of_threads.exercise_2_1
  - masteringthreads.ch2_basics_of_threads.solution_2_1
- **Don't look at solutions until you have tried yourself**

Java Version

- **For this course, we use Java 21**
  - Includes virtual threads, revised threading library
    - Where we use Java 21 concepts, they are marked as ㉑

JavaSpecialists

2: Basics of Threads

Multiple Processes

- **In the past, machines had a single CPU**
  - Programs seemed to run at the same time
    - Word, Excel, Firefox, Chrome, Limewire
- **Illusion; our O/S quickly swaps between processes**
- **Each process typically ran in its own memory space**
  - Inter-process communication expensive

Why use threads?

- **Threads are like lightweight processes**
  - share the same memory space
- **Scheduler fast at swapping between threads**
- **In a multi-core machine, threads can improve performance if algorithms are structured correctly**
- **In a single-core, single processor machine, threads are useful to simplify coding**

Threading Models

- **Preemptive multithreading (Native Threads)**
  - Operating system forces a context switch
  - Threads can be swapped out at inconvenient time
  - 1-to-1 mapping from Java to platform thread

Threading Models

- **Cooperative multithreading (Green Threads)**
  - Threads give up control at a stopping point
    - yield, sleep, wait
  - Infinite loops might never give up control
  - m-to-1 mapping from Java to platform thread

Threading Models

- **Which One?**
  - Preemptive (native) is safer
  - In modern JDKs, preemptive is used
  - Project Loom moving back to more cooperative model
    - Virtual threads ㉑
      - m-to-n mapping from Java thread to platform thread

Parallel Computing

- **Solving a problem on many CPUs in parallel**
  - Large problem is broken into smaller ones
  - These are then solved in parallel on multiple cores
  - Focus is on solving problems faster
  - Communication overhead reduces speedup possibilities
- **Typically used on large number of cores**
  - E.g. few threads per core
- **Examples:**
  - Weather prediction, financial trend analysis, code cracking

Concurrent Computing

- **Interacting tasks may execute in parallel**
  - Independent tasks simplify architecture
  - Usually not processor intensive
  - Do something useful during wait time (IO, Locks, etc.)
  - Focus on interaction between tasks (memory integrity, progress)
  - Does not always scale well
- **Can be used on any number of cores**
- **Examples:**
  - Blocking IO, Swing progress bars, background tasks

Java Memory Model (JSR 133)

- **Describes shared Java memory behaviour**
- **Minimum requirement of what must happen**
- **Allows JVM implementors some freedom**
- **A correctly written multi-threaded Java application will be correct on every available Java VM**
  - A correctly running Java application on one JVM could still be incorrect if it breaks JMM laws
- **More detail in next section**

Creating New Thread

- **Simply override the Thread class**

```
public class MyThread extends Thread {
    public void run() {
        // our concurrent task here
    }
}
```

Creating New Thread

- **Make instance of MyThread and call start()**

```
var thread = new MyThread();
thread.start();
```

  - The start( ) method performs initialization, then calls run( )

Creating New Thread

- **When run( ) exits, thread automatically dies**
  - Threads can only be stopped by them leaving run()

New Thread with Runnable

- We can also pass in a Runnable lambda

```
var thread2 = new Thread(() -> {
    // our concurrent task here
});
thread2.start();
```

New Thread with Builder ㉑

- Create a thread with Thread.ofPlatform() builder

```
var thread3 = Thread.ofPlatform()
        .start(() -> {
            // our concurrent task here
        });
```

Shared Memory with Multithreading

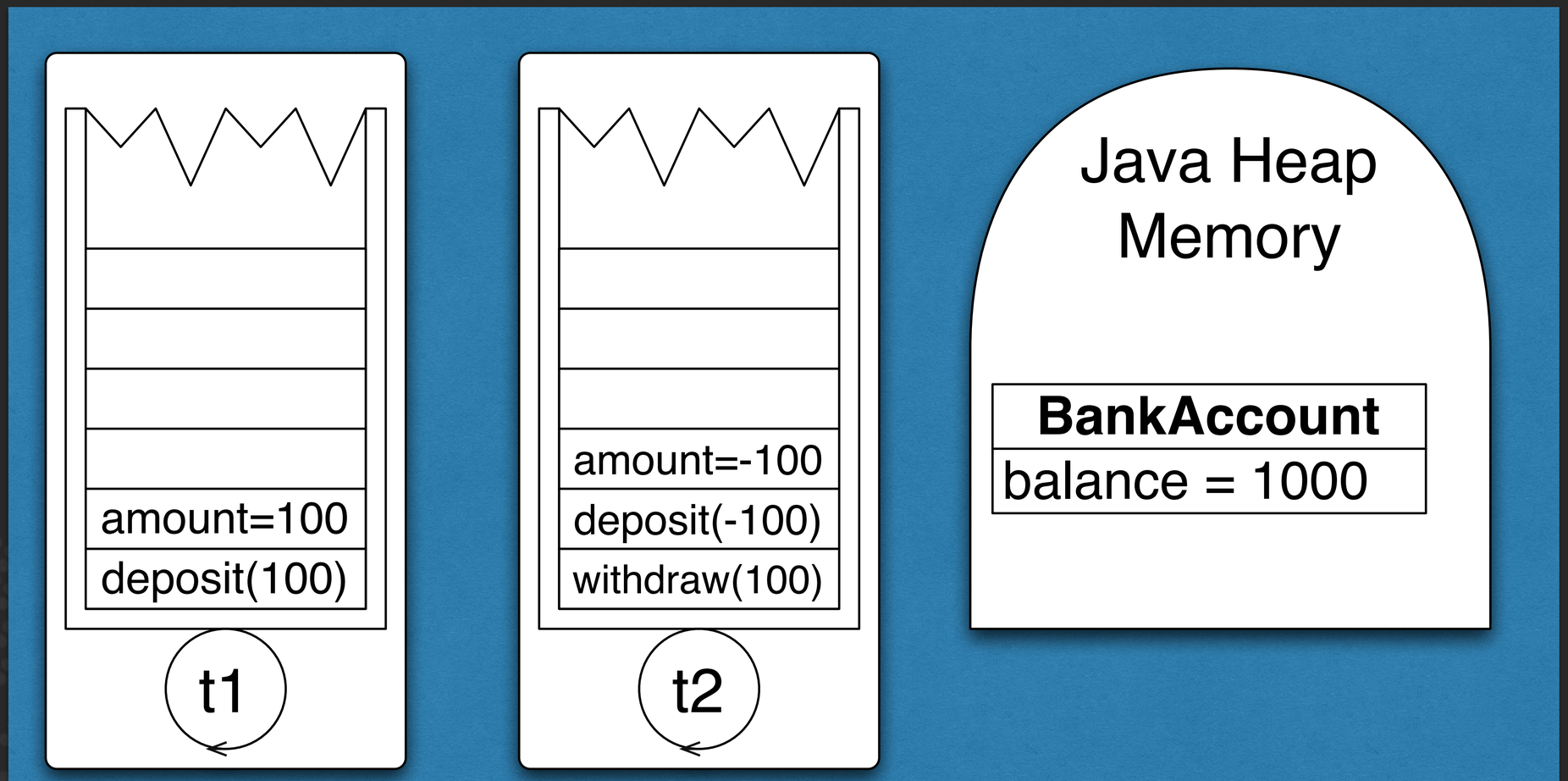
- **How can two threads access one object?**
  - Add & withdraw from bank account
  - IO at the same time, etc.
- **If several threads access a memory location at the same time, we will get *race conditions***

BankAccount Example

```
public class BankAccount {
    private double balance;
    public BankAccount(double initial) {
        balance = initial;
    }
    public void deposit(double amount) {
        balance = balance + amount;
    }
    public void withdraw(double amount) {
        deposit(-amount);
    }
}
```

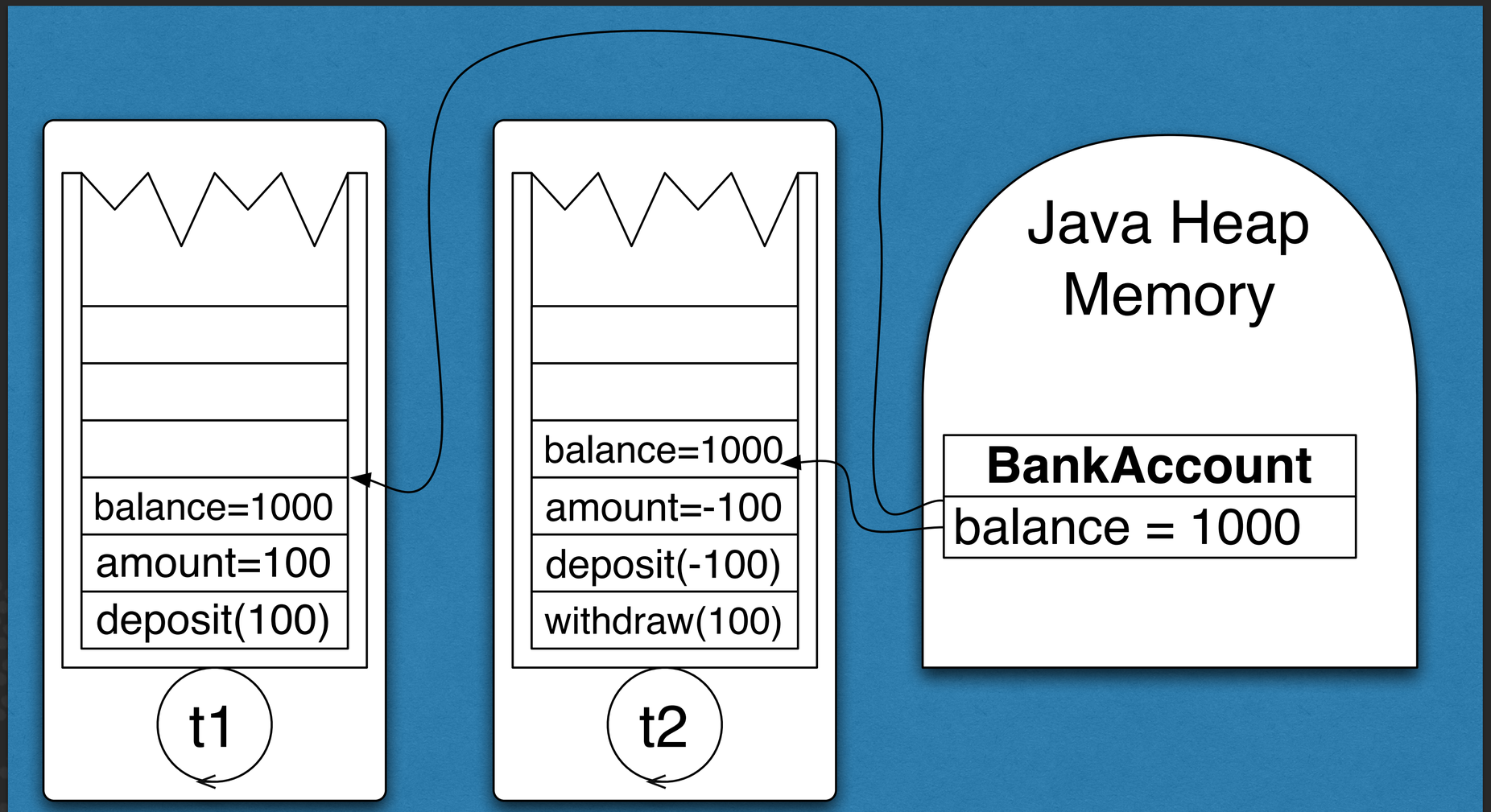
Demo of Race Condition

- **t1 calls deposit(100), t2 calls withdraw(100)**
  - This happens at the same time



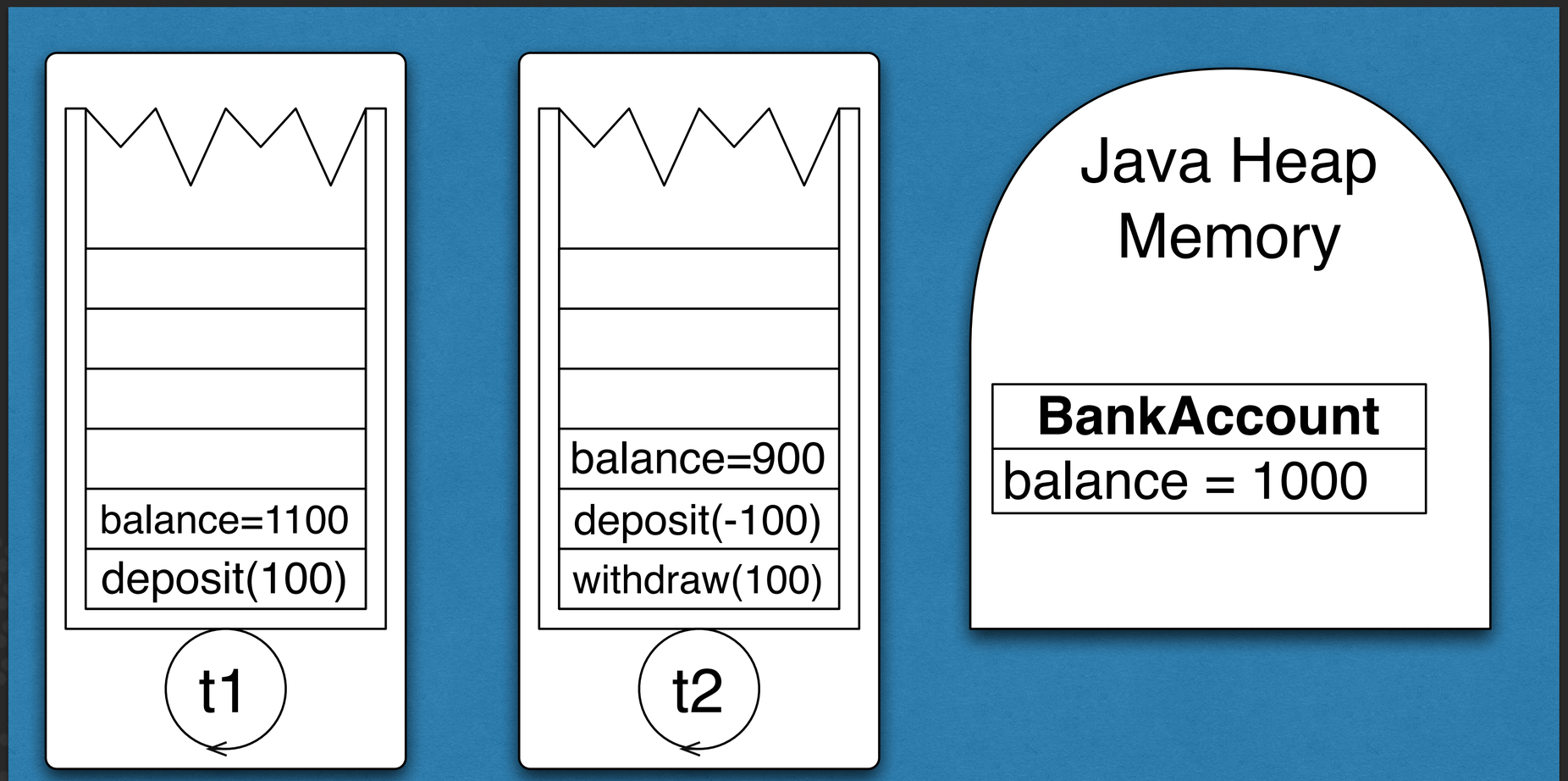
Demo of Race Condition

- Both t1 and t2 read the current balance



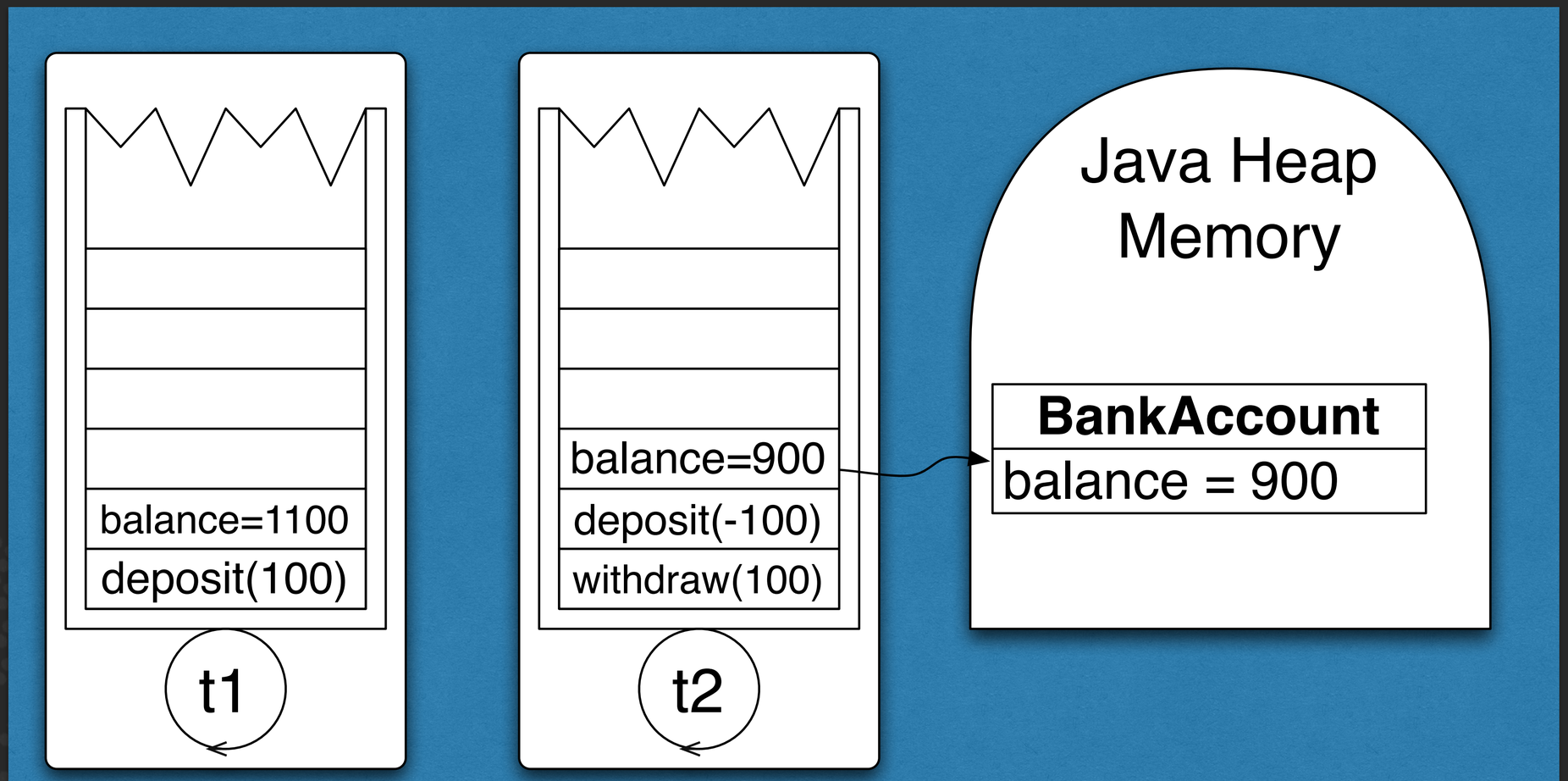
Demo of Race Condition

- **Both sum the last two values on the stack**
  - t1 now sees balance as 1100, t2 sees it as 900



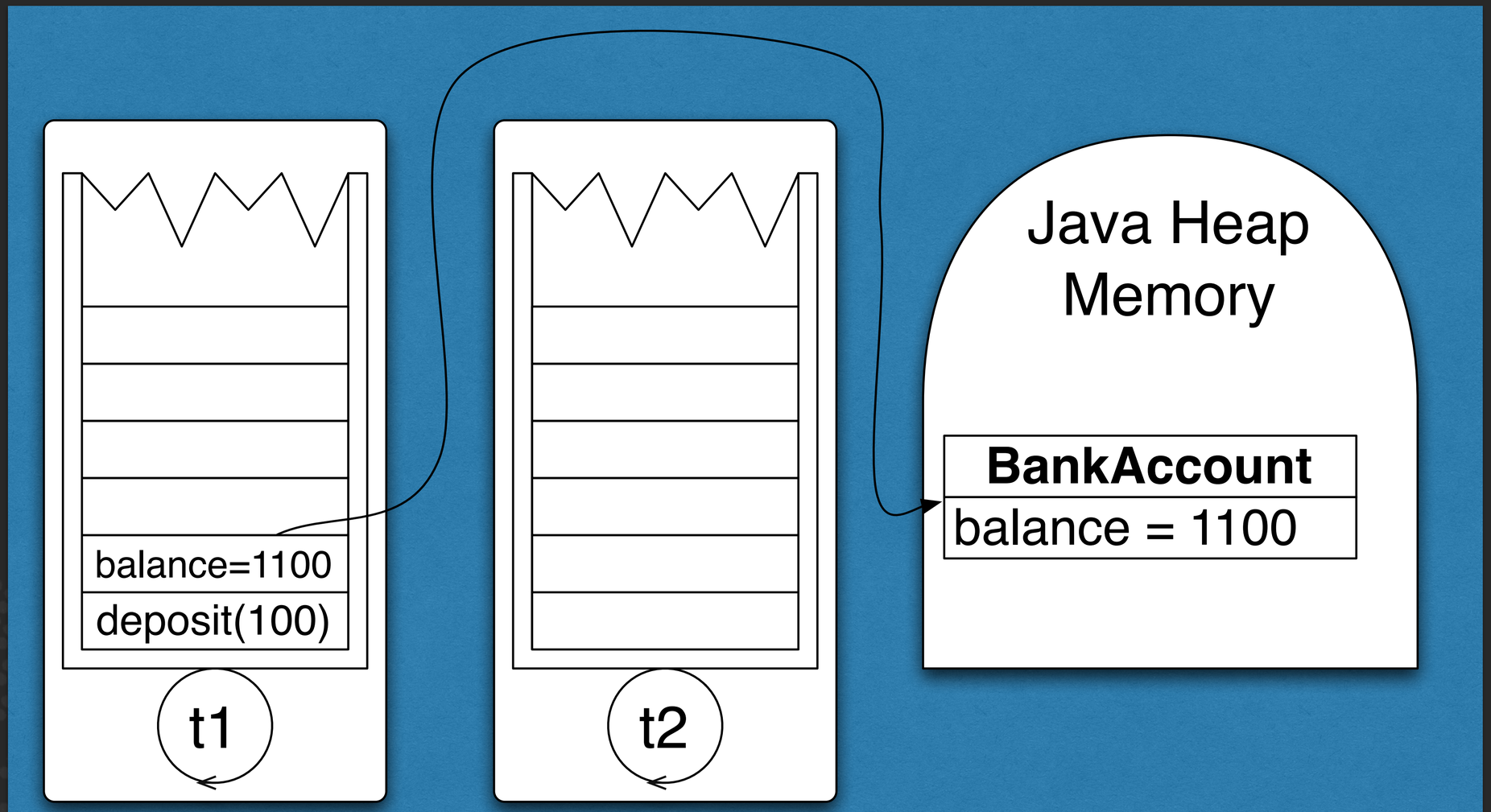
Demo of Race Condition

- **Result is unpredictable - race condition**
  - For example, t2 writes the balance field first



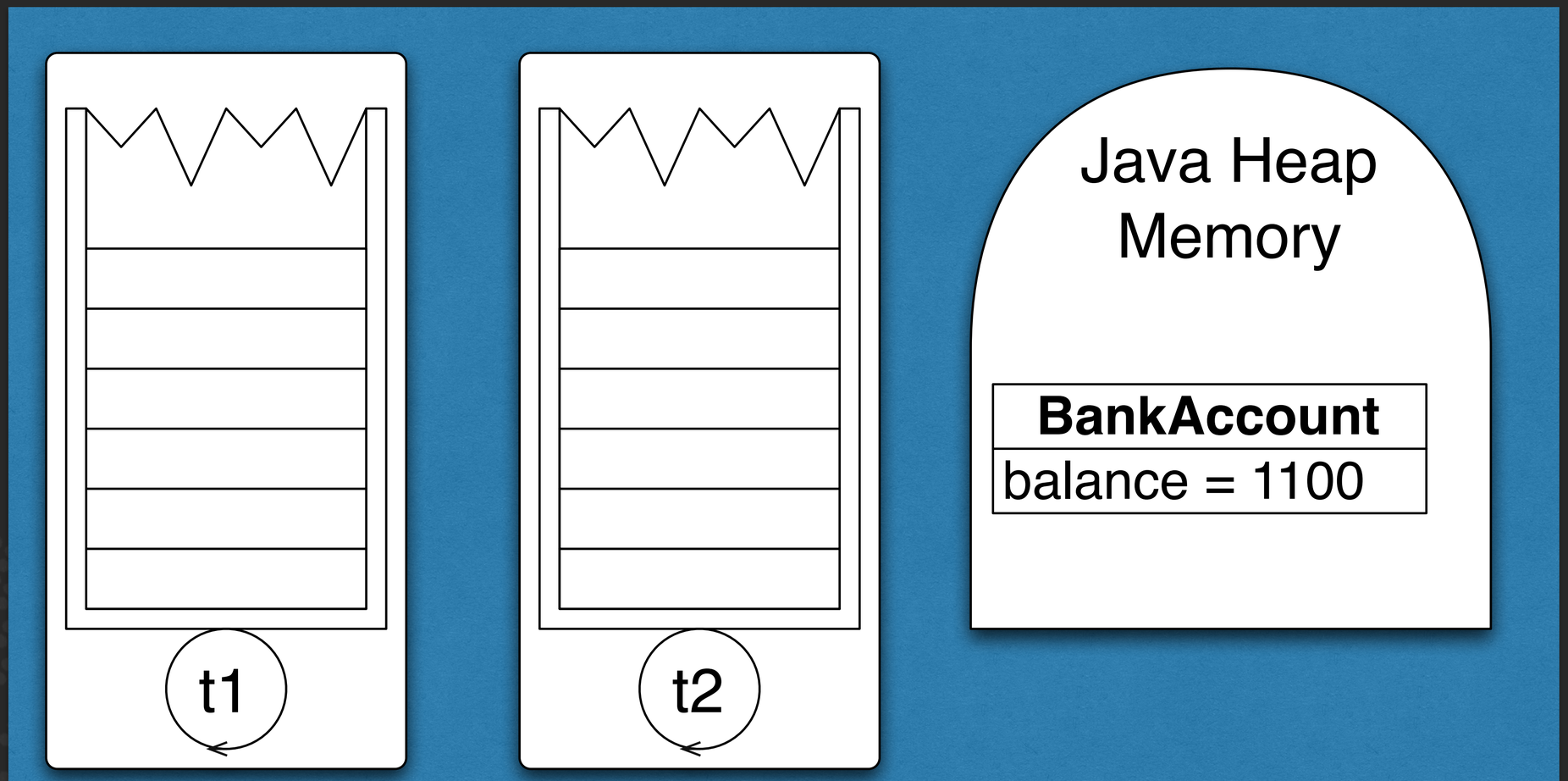
Demo of Race Condition

- Balance is overwritten by t1, so is 1100!



Result of Demo

- **BankAccount balance now is 1100**
  - We called deposit(100) *and* withdraw(100)



Race Conditions are Unpredictable

- **(Almost) anything can happen**
- **Without synchronization, balance can be:**
  - 1000 - what we want
  - 1100 - the withdraw() result was overwritten by deposit()
  - 900 - the deposit() result was overwritten by withdraw()
  - Both 900 and 1100 - each thread sees his own result
  - Neither - if field is a numeric 64-bit type
    - More in next section

Synchronized

- **Built-in mechanism for thread safety**
- **Most Java objects can be used as a monitor**
  - Except for those annotated with @ValueBased
    - Primitive wrappers, Optional, java.time, immutable collections
- **Only one thread at a time can acquire the monitor with** synchronized
- **No back-off strategy to give up waiting for** synchronized

Synchronized BankAccount

```java
public class BankAccount {
    private final Object monitor = new Object();
    private double balance;
    public BankAccount(double initial) {
        balance = initial;
    }
    public void deposit(double amount) {
        synchronized (monitor) {
            balance = balance + amount;
        }
    }
    public void withdraw(double amount) {
        deposit(-amount);
    }
}
```

Synchronized Methods

- **We can also synchronize the entire method**
  - Same as synchronizing on "this"

```
public synchronized void deposit(double amount) {
    balance = balance + amount;
}
```

Synchronized Methods vs "this"

- What are the differences and similarities?

```
public void deposit(double amount) {
    synchronized (this) {
        balance = balance + amount;
    }
}

public synchronized void deposit(double amount) {
    balance = balance + amount;
}
```

Synchronized Static Methods

- Synchronized static method locks on class

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static synchronized Singleton getInstance() {
        if (instance == null) instance = new Singleton();
        return instance;
    }
}
```

- getInstance() could also be written like this

```
public static Singleton getInstance() {
    synchronized (Singleton.class) {
        if (instance == null) instance = new Singleton();
        return instance;
    }
}
```

Locking on this vs. private lock

- **Synchronized methods break encapsulation**
  - Impossible to change locking strategy later
  - Other code could use our object as a lock

Locking on this vs. private lock

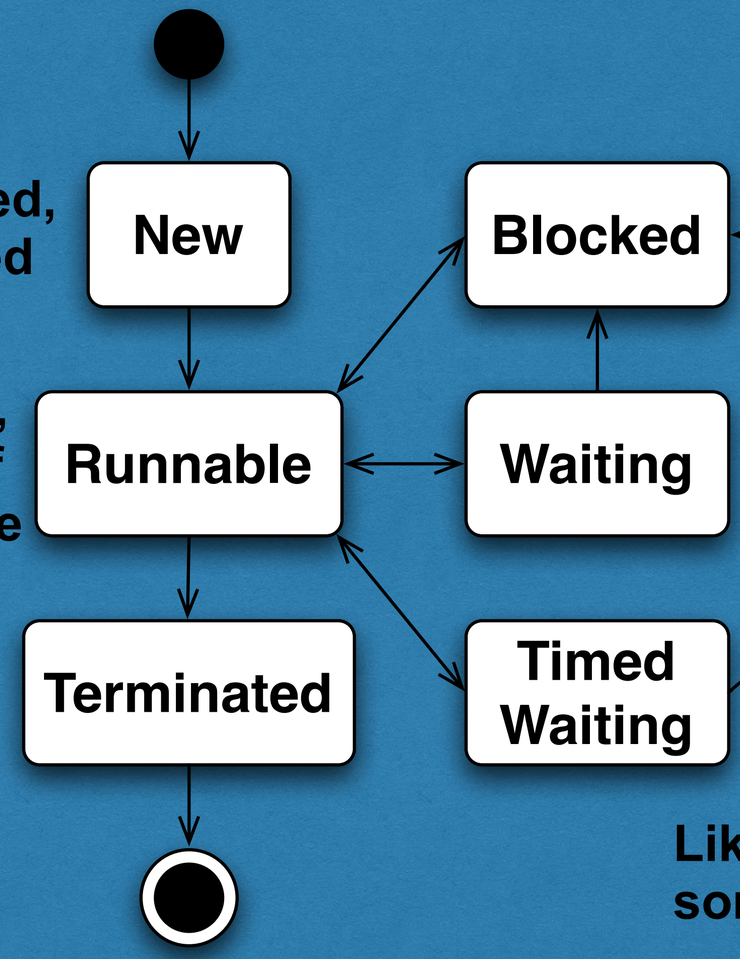
- **Lock splitting**
  - We use different locks for unrelated fields
  - e.g. methods changeAddress and printAddress in a class BankAccount should be synchronized but don't need to be synchronized in respect to balance updating.

Locking on this vs. private lock

- **Ease of understanding**
  - Locking on a specific object instance might be easier
  - "Sprinkling" synchronized like pixie dust causes deadlocks

Thread States

Threads can go directly from Waiting to Blocked if exiting from wait()



Thread created, but not started

Thread started, can run code if OS gives it time

Thread exited run() method

Trying to get monitor lock (via synchronized)

Waiting for some condition (wait, join, take, etc.)

Like Waiting, but gives up after some timeout (wait, sleep)

Thread in Waiting and Timed Waiting states can usually be interrupted.

Inter-thread Communication

- **Threads often need to wait for other threads**
  - For example, a queue might be empty
  - Instead of giving up we tell the thread to wait()
- **When the resource becomes available, we notify() waiting thread**

Wait & Notify

- **Basic inter-thread communication**
  - lock.wait() releases the object lock and waits
  - lock.notify() sends a message to one thread
  - lock.notifyAll() sends a message to all waiting threads

Which Should We Prefer?

- **wait() code should always also work with notifyAll()**
  - while(!precondition) lock.wait();
  - Also caters for case where we lose the race for the lock
- **notifyAll() is easier to code than notify()**
  - Don't have to cater for case where interrupt and notify clash

```java
public class ProducerConsumer {
    private final List<Runnable> tasks = new LinkedList<>();
    public ProducerConsumer() {
        var thread = new Thread(() -> {
            try {
                while (true) take().run();
            } catch (InterruptedException ignore) { }
        });
        thread.setDaemon(true);
        thread.start();
    }
    private Runnable take() throws InterruptedException {
        synchronized (tasks) {
            while (tasks.isEmpty()) tasks.wait();
            return tasks.remove(0);
        }
    }
    public void submit(Runnable task) {
        synchronized (tasks) {
            tasks.add(task);
            tasks.notifyAll();
        }
    }
}
```

ReentrantLock and Condition

- **wait() is not suitable for virtual threads**
- **For new code, use ReentrantLock and Condition**
  - Condition has methods await() and signal() / signalAll()
  - Or even better, a higher-level class such as BlockingQueue
    - More in chapter 4

ProducerConsumer using ReentrantLock

```
public class ProducerConsumerLock {
    private final Lock lock = new ReentrantLock();
    private final Condition notEmpty = lock.newCondition();
    private final List<Runnable> tasks = new LinkedList<>();

    public ProducerConsumerLock() {
        var thread = new Thread(() -> {
            while (true) take().run();
        });
        thread.setDaemon(true);
        thread.start();
    }
```

```java
    private Runnable take() {
        lock.lock();
        try {
            while (tasks.isEmpty())
                notEmpty.awaitUninterruptibly();
            return tasks.remove(0);
        } finally {
            lock.unlock();
        }
    }
    public void submit(Runnable task) {
        lock.lock();
        try {
            tasks.add(task);
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

Priorities

- **Threads can have different priorities**
  - Thread.MIN_PRIORITY = 1
  - Thread.NORM_PRIORITY = 5
  - Thread.MAX_PRIORITY = 10
- **Caveat: Thread priority cannot be relied on**
  - It might be ignored by JVM or underlying operating system

Timers

- **We want to minimize number of threads**
- **For periodic tasks use java.util.Timer**
  - One Timer thread executes lots of tasks
  - Uses less threads in system
- **Don't let exceptions bubble up past our run()**

```
var task = new TimerTask() {
    public void run() {
        System.out.println(new Date());
    }
};
var timer = new Timer();
timer.schedule(task, 1000, 1000);
```

Daemon Threads

- JVM runs until all non-daemon threads exit
- Call setDaemon(true) before start()

```
var t = new Thread(task);
t.setDaemon(true);
t.start();
```

- Alternatively, use Thread Builder ㉑

```
var t = Thread.ofPlatform()
        .daemon()
        .start(task);
```

Techniques for Shutting Down

- **State Machines**
  - Controls states such as RUNNING, SHUTDOWN, TERMINATED
  - Allows most flexibility in describing state transitions
- **Poison Pill**
  - When we want to shut down, we submit a poison pill token
    - Making sure that all our threads see the poison pill
    - When they do, they shut down, passing the poison pill along
- **Which to use?**
  - Use State machines for more complex system
  - Poison pills usually easier to implement for simple cases

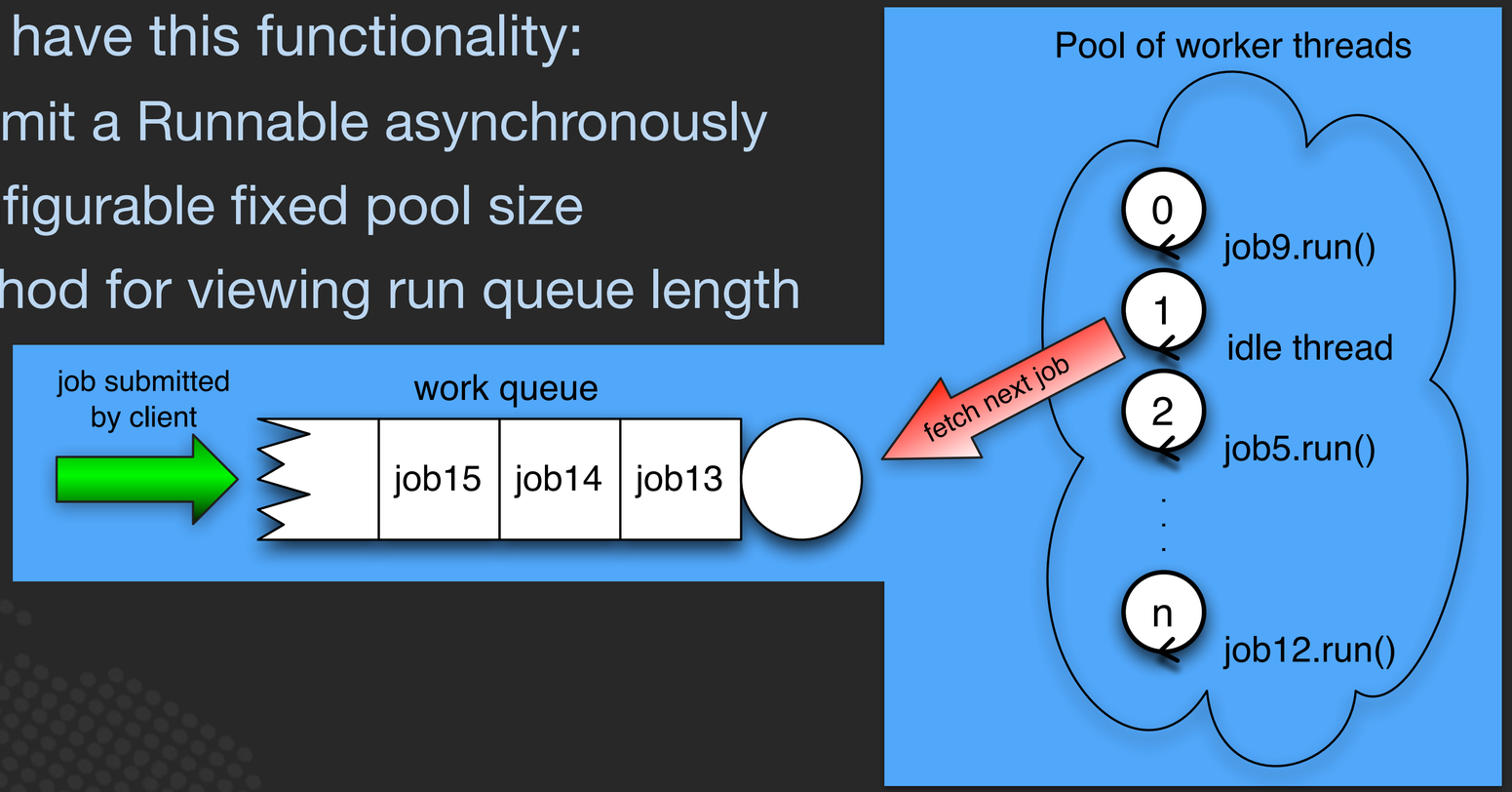
JavaSpecialists

2: Exercises

Basics of Threads

Exercise 2.1: Thread Pool

- **Create our own ThreadPool implementation, using**
  - LinkedList for the Runnable tasks, with monitor locking
  - wait/notifyAll for take()/submit()
  - Must have this functionality:
    - Submit a Runnable asynchronously
    - Configurable fixed pool size
    - Method for viewing run queue length



Exercise 2.2: Using ReentrantLock

- **Change monitor to "owned" locks**
  - Remember to always use try/finally to unlock reliably

Exercise 2.3: Priorities

- **Construct one thread for each priority**
  - This experiment works well on Windows
  - Mac OS X and Linux give all threads same priority
- **Wait until they have all been started, then start them all off at the same time**
  - Or as closely to the same time as possible
- **Give them each equal amount of work to do and measure how fast they complete**

JavaSpecialists

3: The Secrets of Concurrency

3: The Secrets of Concurrency

- **Writing correct concurrent code is hard**
  - Only *perfect* is good enough
- **We need to synchronize in correct places**
  - Too much synchronization we risk deadlock and contention
  - Too little synchronization we risk early writes, corrupt data, race conditions and stale local copies of fields
- **In this section, we will look at ten laws that will make it easier for us to write correct thread-safe code.**

The Secrets of Concurrency

- **Ten laws to help us write thread-safe code**
  - Law 1: The Law of the Sabotaged Doorbell
  - Law 2: The Law of the Distracted Spearfisherman
  - Law 3: The Law of the Overstocked Haberdashery
  - Law 4: The Law of the Blind Spot
  - Law 5: The Law of the Leaked Memo
  - Law 6: The Law of the Corrupt Politician
  - Law 7: The Law of the Micromanager
  - Law 8: The Law of Cretan Driving
  - Law 9: The Law of Sudden Riches
  - Law 10: The Law of the Uneaten Lutefisk

JavaSpecialists

1. The Law of the Sabotaged Doorbell

Instead of arbitrarily suppressing interruptions, manage them better.

* Removing batteries from doorbell to avoid hawkers also shuts out wanted visitors

Law 1: Sabotaged Doorbell

- **Have we ever seen code like this?**

```
try {
    Thread.sleep(1000);
} catch(InterruptedException ex) {
    // this won't happen here
}
```

- **We will answer the following questions:**
  - What does InterruptedException mean?
  - How should we handle it?

Shutting Down Threads

- **Shutdown threads when they are inactive**
  - In WAITING or TIMED_WAITING states:
    - Thread.sleep()
    - BlockingQueue.take()
    - Semaphore.acquire()
    - wait()
    - join()

Thread “interrupted” Status

- **We can interrupt a thread with:**
  - someThread.interrupt();
  - Sets the “interrupted” status to true
  - What else?
    - If thread is in state WAITING or TIMED_WAITING, the thread immediately returns by throwing InterruptedException and sets “interrupted” status back to false
    - Else, the thread does nothing else. In this case, someThread.isInterrupted() will return true
- **Beware of Thread.interrupted() side effect**

InterruptedException Approaches

- **1: Simply re-throw exception**
  - Approach used by java.util.concurrent
  - Not always possible if we are overriding a method
- **2: Catch and return without InterruptedException**
  - Best to return in "interrupted" state
  - boolean volatile field as backup if calling alien code

```
while (running) {
    // do something
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt(); break;
    }
}
```

Thread.interrupted()

- **Start waiting methods with**

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

  - Now we immediately leave if the thread state is interrupted

JavaSpecialists

2. The Law of the Distracted Spearfisherman

Focus on one thread at a time. The school of threads will blind us.

* The best defence for a fish is to swim next to a bigger, better fish.

Law 2: Distracted Spearfisherman

- **We should know each thread in our JVM**
  - Good reason to have fewer threads!
- **Don’t jump from thread to thread**
  - Systematically work through thread dump

Generating Thread Dumps

- **The jstack tool dumps threads of process**
  - Similar to CTRL+Break or CTRL+\
- **jstack -l prints additional info about locks**
- **jstack -e extended listing, memory allocated per thread**
- **jstack will be replaced with jcmd <pid> Thread.print**
  - And jcmd <pid> Thread.dump_to_file filename

JavaSpecialists

3. The Law of the Overstocked Haberdashery

Having too many threads is bad for our application. Performance will degrade and debugging will become difficult.

* Haberdashery: A shop selling sewing wares, e.g. threads and needles.

TARMAK

Law 3: Overstocked Haberdashery

- **Story: Client-side library running on server**
- **We will answer the following questions:**
  - How many threads can we create?
  - What is the limiting factor?
  - How can we create more threads?

```java
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.LockSupport;

public class ThreadCreationTest {
    public static void main(String... args) {
        var threads = new AtomicInteger(0);
        while (true) {
            Thread.ofPlatform().start(() -> {
                System.out.println("threads created: " +
                        threads.incrementAndGet());
                LockSupport.park();
            });
        }
    }
}
```

JVM Crashes

```
threads created: 2012
threads created: 2013
threads created: 2014
threads created: 2015
threads created: 2016
threads created: 2017
threads created: 2018
threads created: 2019
threads created: 2020
threads created: 2021
threads created: 2022
[1.452s][warning][os,thread] Failed to start thread - pthread_create failed
   (EAGAIN) for attributes: stacksize: 1024k, guardsize: 4k, detached.
threads created: 2023
Exception in thread "main" java.lang.OutOfMemoryError: unable to create
   native thread: possibly out of memory or process/resource limits reached
   at java.base/java.lang.Thread.start0(Native Method)
   at java.base/java.lang.Thread.start(Thread.java:803)
   at ThreadCreationTest.main(ThreadCreationTest.java:14)
```

How to Create More Threads?

- **We created only a few thousand threads**
- **A smaller stack size can help on 32-bit**
  - java -Xss200k ThreadCreationTest
  - Can cause other problems
    - See The Law of the Distracted Spearfisherman
- **Loom allows millions of blocking virtual threads ㉑**
  - Thread.sleep(), Condition.await(), Socket.read
  - However, synchronized/wait not fully supported yet

How Many Threads is Healthy?

- **How long is a piece of string?**
- **Platform threads**
  - Active threads are usually same as hardware threads
  - Inactive threads try keep in the hundreds
- **Virtual threads ㉑**
  - Should not be used for number crunching, i.e. parallelism
  - Limited by amount of memory, can be millions
    - Unlike platform threads, virtual threads are not GC roots

JavaSpecialists

4. The Law of the Blind Spot

It is not always possible to
see what other threads (cars) are doing
with shared data (road)

Law 4: Blind Spot

- **Threads can keep local copies of fields**
  - Deliberately allowed by Java Memory Model
- **We might not see another thread's changes**
- **Usually happens when we try avoid synchronization**

shutdown() might have no effect

```
public class Runner {
    private boolean running = true;

    public void doJob() {
        while (running) {
            // do something
        }
    }

    public void shutdown() {
        running = false;
    }
}
```

Why?

- **Thread1 calls doJob()**
  - makes a local copy of running
- **Thread2 calls shutdown()**
  - modifies field running
- **Thread1 does not see the changed value of** running **and continues reading the local stale value**

Making Field Changes Visible

- **Three ways of preventing this**
  - Make field volatile
  - Make field final puts a “freeze” on value
    - Won't work if we need to change the field
  - Make read and writes to field synchronized
    - Also includes new locks

Runner with "running" as volatile

```
public class RunnerVolatile {
    private volatile boolean running = true;

    public void doJob() {
        while (running) {
            // do something
        }
    }

    public void shutdown() {
        running = false;
    }
}
```

JavaSpecialists

5. The Law of the Leaked Memo

The JVM is allowed to reorder our statements resulting in seemingly impossible states (seen from the outside)

* Memo about hostile takeover bid left lying in photocopy machine

Law 5: Leaked Memo

- **Two threads call f() and g()**
  - What are the possible values of a and b ?

```
public class EarlyWrites {
    private int x;
    private int y;

    public void f() {
        int a = x;
        y = 3;
    }

    public void g() {
        int b = y;
        x = 4;
    }
}
```

```
Obvious
answers:
a=4, b=0
a=0, b=3
```

```
Non-obvious
answer:
a=0, b=0
```

```
Extremely non-obvious answer:
a=4, b=3
```

The order of Things

- JMM allows reordering of statements
- Includes writing of fields
- To the writing thread, statements appear in order

How to Prevent This?

- **Java can't move writes out of synchronized**
  - Can increase the scope of synchronized
    - Called "lock coarsening"
- **Keyword** volatile **prevents early writes**
  - From the Java Memory Model:
    - There is a happens-before edge from a write to a volatile variable v to all subsequent reads of v by any thread (where subsequent is defined according to the synchronization order)

JavaSpecialists

6. The Law of the Corrupt Politician

In the absence of proper controls, corruption is unavoidable.

* Power tends to corrupt, and absolute power corrupts absolutely

Law 6: Corrupt Politician

- **Without controls, the best code can go bad**

```
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; }
}
```

What happens?

- **The += operation is not atomic**
  - Thread 1
    - Reads balance = 1000
    - Locally adds 100 = 1100
    - Before the balance written, Thread 1 is swapped out
  - Thread 2
    - Reads balance=1000
    - Locally subtracts 100 = 900
    - Writes 900 to the balance field
  - Thread 1
    - Writes 1100 to the balance field

JavaSpecialists

7. The Law of the Micromanager

Even in life, it wastes effort and frustrates the other threads.

* mi·cro·man·age: to manage or control with excessive attention to minor details.

Law 7: Micromanager

- **Thread contention is difficult to spot**
- **Performance does not scale**
- **None of the usual suspects**
  - CPU
  - Disk
  - Network
  - Garbage collection
- **Points to thread contention**

Real Example – *Don't Do This!*

- **"How to add contention 101"**

```
private static final String WRITE_LOCK_OBJECT =
        "WRITE_LOCK_OBJECT";
```

- **Later on in the class**

```
synchronized (WRITE_LOCK_OBJECT) { /*...*/ }
```

- **Constant Strings are flyweights!**
  - Multiple parts of code locking on one object

AtomicInteger

- **Thread safe without explicit locking**
- **Tries to update the value until success**
  - AtomicInteger.equals() is not overridden

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

Atomic BankAccount

```
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

8. The Law of Cretan Driving

The JVM does not enforce all the rules.

Our code is probably wrong, even if it works.

* Don’t stop dead at a stop sign if you treasure your car!

Kolymbari Intersection

Too many signs - my head hurts!

Seems to work on my machine

Law 8: Cretan Driving

- **Learn the JVM Rules !**
  - And obey them, even when no one else does
- **From JSR 133 – Java Memory Model**
  - VM implementers are encouraged to avoid splitting their 64-bit values where possible. Programmers are encouraged to declare shared 64-bit values as volatile or synchronize their programs correctly to avoid this.

JSR 133 allows this – NOT a Bug

- **Method set() called by two threads with**
  - 0x12345678ABCD0000L
  - 0x1111111111111111L

```
public class LongFields {
    private long value;
    public void set(long v) { value = v; }
    public long get()       { return value; }
}
```

- **Besides obvious answers, “value” could also be**
  - 0x11111111ABCD0000L or 0x1234567811111111L

Java Virtual Machine Specification

- **Gives great freedom to JVM writers**
  - For running on disparate hardware
- **Makes it difficult to write 100% correct Java**
  - It might work on all JVMs to date, it might still not be correct!
- **Theory vs Practice clash**

Synchronize at the Right Places

- **Too much synchronization gives contention**
  - With more CPUs, performance does not improve
  - The Law of the Micromanager

Synchronize at the Right Places

- **Lack of synchronization leads to corrupt data**
  - The Law of the Corrupt Politician

Synchronize at the Right Places

- **Fields might be written early**
  - The Law of the Leaked Memo

Synchronize at the Right Places

- **Changes to shared fields might not be visible**
  - The Law of the Blind Spot

JavaSpecialists

9. The Law of Sudden Riches

Additional resources (faster CPU, disk or network, more memory) for seemingly stable system can make it unstable.

* Sudden inheritance or lottery win …

Law 9: Sudden Riches

- **Better hardware can break system**
  - Old system: Dual processor
  - New system: Dual core, dual processor

Faster Hardware

- **Latent defects show up more quickly**
  - Instead of once a year, now once a week
  - Bug might have always been there
- **Faster hardware often coincides with higher customer utilization**
  - More contention
- **E.g. DOM tree becomes corrupted**
  - Detected problem by synchronizing all subsystem access
  - Fixed by copying the nodes whenever they were read

JavaSpecialists

10. The Law of the Uneaten Lutefisk

A deadlock in Java can only be resolved by restarting the Java Virtual Machine.

* Imagine a Viking father insisting that his stubborn child eat its lutefisk before going to bed

Lutefisk

- It is not as delicious as it looks

Law 10: Uneaten Lutefisk

- **Part of program stops responding**
- **GUI does not repaint**
  - Under Swing
- **Users cannot log in anymore**
  - Could also be The Law of the Corrupt Politician
- **Two threads want what the other has**
  - And are not willing to part with what they already have

Using Multiple Locks

```
public class HappyLocker {
    private final Object lock = new Object();

    public synchronized void f() {
        synchronized(lock) {
            // do something ...
        }
    }

    public void g() {
        synchronized(lock) {
            f();
        }
    }
}
```

Finding the Deadlock

- **Press CTRL+Break, CTRL+\ or use jstack**

```
Full thread dump:
Found one Java-level deadlock:
=============================
"g()":
  waiting to lock monitor 0x0023e274 (object 0x22ac5808, a
   HappyLocker),
  which is held by "f()"
"f()":
  waiting to lock monitor 0x0023e294 (object 0x22ac5818, a
   java.lang.Object),
  which is held by "g()"
```

Deadlock Means We Are Dead ! ! !

- **Deadlock can be found with jconsole**
- **However, there is no way to resolve it**
- **Better to automatically raise critical error**
  - Newsletter 130 – Deadlock Detection with new Lock
  - http://www.javaspecialists.eu/archive/Issue130.html

JavaSpecialists

3: Exercises

Secrets of Concurrency

Exercise 3.1: Repair the Doorbell

- **Let's add a shutdownNow() method to ThreadPool**
  - Shuts down the thread pool abruptly, interrupting the worker threads currently executing tasks, and returning a list of unstarted tasks.

Exercise 3.2: Corrupt Politicians

- **This exercise should be easy!**
  - Just don't get caught ...
- **Taking our BankAccount**
  - Create two threads that deposit and withdraw from the same account. See if we can corrupt the Bank Account.

Ex 3.3: Which Laws Are Violated?

```
public class CASCounter implements Counter {
    private long count = 0;
    private Thread owner;

    public long getCount() {return count;}

    public void increment() {
        do {
            while (this.owner != null) ;   // wait
            this.owner = Thread.currentThread();
            for (int i = 0; i < 6; i++) ;  // delay
        } while (this.owner != Thread.currentThread());
        this.count++;
        this.owner = null;
    }
}
```

Write code to verify some of your findings

JavaSpecialists

4: Applied Threading Techniques

Executors

- **Creating threads for small tasks is costly**
  - The Law of the Overstocked Haberdashery
- **Surge of traffic can make everybody slow**
- **Better approach is to use thread pools**
- **In Java we have**
  - Executor and ExecutorService
  - ForkJoinPool (not covered)

ExecutorService Interface

- **Tasks can be submitted as Callables**
  - submit() returns a Future
  - Tasks can be canceled through interruption
- **Lifecycle management for threads in pool**
  - We can shut it down with shutdown() or shutdownNow()
  - We can wait for it to complete shut down with awaitTermination()
    - And ExecutorService implements AutoCloseable ㉑
      - Shuts down the pool and waits for it to terminate

ExecutorService

- Callable<V> more useful than Runnable

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

- submit(Callable<V>) returns a Future<V>

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

```java
public class ExecutorTest {
    public static void main(String... args) throws Exception {
        try (var pool = Executors.newFixedThreadPool(4)) {
            var futures = new ArrayList<Future<Long>>();
            for (int i = 0; i < 16; i++) {
                int submitOrder = i;
                Callable<Long> task = () -> {
                    long total = ThreadLocalRandom.current()
                            .ints(100_000_000, 0, 5).sum();
                    return total * 10000 + submitOrder;
                };
                futures.add(pool.submit(task));
            }
            for (Future<Long> future : futures) {
                System.out.println(future.get());
            }
        }
    }
}
```

```
2000048490000
1999901860001
1999902890002
2000262080003
2000050590004
2000274120005
1999859840006
...
```

Future results are retrieved in the order jobs were submitted, not in the order they were completed

Types of ExecutorService

- **Executors.newFixedThreadPool()**
  - Keeps a permanent total number of threads
  - Threads are not started until we start submitting jobs - why?

Types of ExecutorService

- **Executors.newSingleThreadExecutor()**
  - Like a FixedThreadPool, but with only one thread

Types of ExecutorService

- **Executors.newCachedThreadPool()**
  - Creates new threads as needed
  - Destroys unused threads after a minute

Types of ExecutorService

- **Executors.newScheduledThreadPool()**
  - Should be used instead of Timer

Types of ExecutorService

- **Executors.newSingleThreadScheduledExecutor()**
  - Very similar to the java.util.Timer class

Types of ExecutorService

- **Executors.newWorkStealingPool()**
  - Creates a ForkJoinPool for parallel programming

Types of ExecutorService

- **Executors.newVirtualThreadPerTaskExecutor() ㉑**
  - Makes a new virtual thread per task
    - So cheap to create, no point pooling virtual threads

CompletionService

- **Returns results in order that they complete**
- **CompletionService wraps ExecutorService**
- **CompletionService is parameterized with generics**
  - Should only be used with one type of Callable

```java
public class CompletionServiceTest {
    public static void main(String... args) throws Exception {
        try (var pool = Executors.newFixedThreadPool(16)) {
            var service = new ExecutorCompletionService<Long>(pool);
            for (int i = 0; i < 128; i++) {
                int submitOrder = i;
                service.submit(() -> {
                    long total = ThreadLocalRandom.current()
                            .ints(100_000_000, 0, 5).sum();
                    return total * 10000 + submitOrder;
                });
            }
            for (int i = 0; i < 128; i++) {
                System.out.println(service.take().get());
            }
        }
    }
}
```

```
1999859180013
2000065570000
2000163270011
1999863930006
2000098410015
1999977600007
...
```

Count Down Latch

- Set up with an initial size

```
var latch = new CountDownLatch(10);
```

- latch.await() waits until count == 0
- latch.countDown() reduces the latch count

```java
public class CountDownTest {
    public static void main(String... args) throws Exception {
        try (var pool = Executors.newFixedThreadPool(5)) {
            var latch = new CountDownLatch(3);
            for (int i = 0; i < 5; i++) {
                pool.submit(() -> {
                    System.out.println("Waiting ...");
                    latch.await();
                    System.out.println("... finished");
                    return null;
                });
                Thread.sleep(1000);
                latch.countDown();
            }
        }
    }
}
```

```
Waiting ...
Waiting ...
Waiting ...
... finished
... finished
... finished
Waiting ...
... finished
Waiting ...
... finished
```

Semaphore

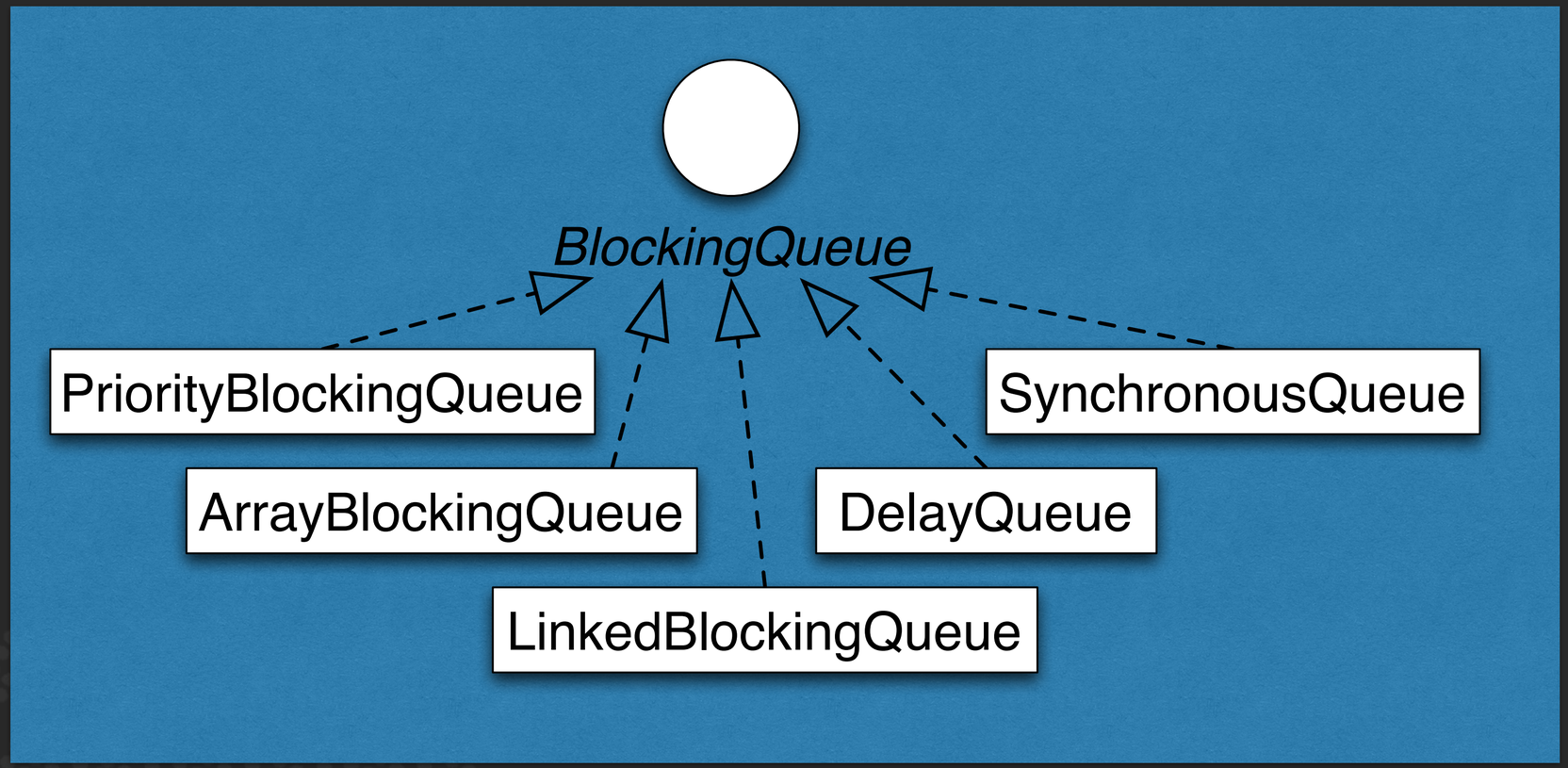
- **The “bouncer” of Java concurrency**
  - Limits resources to specified number
  - Has no knowledge of who has acquired permits
- **We acquire the semaphore with acquire()**
- **We release it again with release()**

```java
public class SemaphoreTest {
    public static void main(String... args) {
        var time = System.nanoTime();
        try (var pool = Executors.newCachedThreadPool()) {
            var bouncer = new Semaphore(25);
            for (int i = 0; i < 100; i++) {
                pool.submit(() -> {
                    bouncer.acquireUninterruptibly();
                    try {
                        Thread.sleep(1000);
                        return null;
                    } finally {
                        bouncer.release();
                    }
                });
            }
        }
        time = System.nanoTime() - time;
        System.out.println(time / 1_000_000 + "ms");
    }
}
```

```
4020ms
```

BlockingQueues

- Extension of Queue interface
- Can block on removing or adding



BlockingQueue Methods

- **put() adds elements into queue**
  - Blocks if queue is full
- **take() removes front of queue**
  - Blocks if queue is empty
- **Functions above throw InterruptedException**
- **This is used in Producer/Consumer scenarios**

ArrayBlockingQueue

- **Blocking queue based on an array**
  - Circular structure with head that moves
- **Fixed length – size of array**
- **Most memory efficient blocking queue**
- **Single lock for put() and take()**
  - SPSC usage can be slower than LinkedBlockingQueue

LinkedBlockingQueue

- **Blocking queue based on a linked node structure**
- **Maximum length is unbounded by default**
  - We can specify a maximum length
- **One lock for put() and another for take()**
  - But memory layout and node sizes makes it slower than ArrayBlockingQueue
- **Used for fixed thread pool**

PriorityBlockingQueue

- **Returns elements using natural order**
  - Can be overridden with java.util.Comparator
- **Smallest element returned by take()**
- **Iterator does not return ordered list**
  - We should not iterate over queues anyway
- **Sort order is not stable**

DelayQueue

- Similar to PriorityBlockingQueue
- take() returns items once delay is elapsed
- Takes java.util.concurrent.Delayed objects
- Used for scheduled thread pools

SynchronousQueue

- **Has a size of zero!**
- **put() and take() both block**
- **Don’t confuse synchronous with synchronized**
  - Synchronous as opposed to asynchronous
- **Used with the cached thread pool**

```java
public class ProducerConsumerBlockingQueue {
    private final BlockingQueue<Runnable> tasks =
            new LinkedBlockingQueue<>();

    public ProducerConsumerBlockingQueue() {
        Thread.startVirtualThread(() -> {
            while (true) {
                try {
                    tasks.take().run();
                } catch (InterruptedException e) {
                    throw new CancellationException("interrupt");
                }
            }
        });
    }

    public void submit(Runnable task) {
        tasks.add(task);
    }
}
```

Exceptions

- **Pre-Java 5, uncaught exceptions got lost**
  - www.javaspecialists.eu/archive/Issue089.html
- **For example:**

```
public class ExceptionHandling {
    public static void main(String... args) {
        Thread.ofPlatform().start(() -> {
            throw new RuntimeException();
        });
    }
}
```

```
Exception in thread "Thread-0" java.lang.RuntimeException
        at ExceptionHandling.lambda$main$0(ExceptionHandling.java:6)
```

Handling Uncaught Exceptions

- **Thread.UncaughtExceptionHandler**
  - Can be per-Thread or global
- **Implement method uncaughtException()**
- **From Exception Handler, we can send error report**
  - Better to know we have a problem

```
public class SimpleExceptionHandler
        implements Thread.UncaughtExceptionHandler {
    public void uncaughtException(Thread t, Throwable e) {
        StackTraceElement ste = e.getStackTrace()[0];
        System.err.printf("%s: %s at line %d of %s%n",
                t.getName(), e, ste.getLineNumber(),
                ste.getFileName());
    }
}
```

Setting the Uncaught Handler

- Use thread.setUncaughtExceptionHandler()

```
public class ExceptionHandling2 {
    public static void main(String... args) {
        Thread.ofPlatform()
                .uncaughtExceptionHandler(
                        new SimpleExceptionHandler())
                .start(() -> {
                    throw new RuntimeException();
                });
    }
}
```

```
Thread-0: java.lang.RuntimeException at line 10 of ExceptionHandling2.java
```

ThreadGroup Exception Handling

- Pre-Java 5 approach
- Subclass ThreadGroup and override

```
public void uncaughtException(Thread t, Throwable e)
```

- Make the new ThreadGroup the container for all of our threads

ThreadPoolExecutor afterExecute()

- **We can override the ThreadPoolExecutor afterExecute(Runnable r, Throwable t) method**
  - A bit odd:
    - The Throwable refers to exceptions within the ThreadPoolExecutor
      - Not the exceptions in our tasks
    - The Runnable is usually a FutureTask wrapping our task
      - If it is, we can downcast it and if isDone() is true call get()
        - » If the task was cancelled, then t = the CancellationException
        - » If an exception occurred, then t = the cause
        - » If we get interrupted, we self-interrupt
- **That's enough, let's look at the recommended code**

ThreadPoolExecutor afterExecute()

```
protected void afterExecute(Runnable r, Throwable t) {
    super.afterExecute(r, t);
    if (t == null && r instanceof Future<?> f && f.isDone()) {
        try {
            Object result = f.get();
        } catch (CancellationException ce) {
            t = ce;
        } catch (ExecutionException ee) {
            t = ee.getCause();
        } catch (InterruptedException ie) {
            // ignore/reset
            Thread.currentThread().interrupt();
        }
    }
    if (t != null) handler.uncaughtException(
            Thread.currentThread(), t);
}
```

ThreadLocal

- **Associate state with a thread**
  - e.g. user or Transaction ID
- **Each thread has a map of its ThreadLocals**
  - Once thread stops, all its ThreadLocals are released

Example of ThreadLocal

- Create a per-thread instance of SimpleDateFormat

```
public class DateFormatter {
    private static final ThreadLocal<DateFormat> formatter =
            ThreadLocal.withInitial(() ->
                    new SimpleDateFormat("yyyy-MM-dd"));

    public static String format(Date date) {
        return formatter.get().format(date);
    }

    public static Date parse(String date) throws ParseException {
        return formatter.get().parse(date);
    }
}
```

Stopping Threads

- **Thread.stop() was disabled in Java 20**
  - Has always been dangerous to use
    - Could result in broken invariants
    - Inconsistent state
    - May cause deadlocks
- **stop() causes an asynchronous exception**
  - Could stop us in the middle of a critical piece of code
- **Instead, we use interrupt and cooperative shutdown**

Shutdown Hooks

- **Code to be executed when JVM shuts down**
  - When the last non-daemon thread finishes
  - Or when System.exit() is called

Shutdown Hooks

- **Shutdown hook tasks are specified as threads**
  - Runtime.getRuntime().addShutdownHook(Thread)
  - Can be executed in any order

Shutdown Hooks

- **halt() is called after shutdown hooks are done**
  - Bypass shutdown hooks by calling halt() directly
    - Runtime.getRuntime().halt(0);

Shutdown Hook Example

```
public class ShutdownHookExample {
    public static void main(String... args)
            throws IOException {
        var out1 = new PrintWriter(
                new FileWriter("test1.txt"),
                false); // auto-flush off
        out1.println("This will not be in test1.txt file!");

        var out2 = new PrintWriter(
                new FileWriter("test2.txt"),
                false); // auto-flush off
        Runtime.getRuntime().addShutdownHook(
                new Thread(out2::close));
        out2.println("This will be in the test2.txt file!");
    }
}
```

JavaSpecialists

4: Exercises

Applied Threading Techniques

Exercise 4.1: Simplify Thread Pool

- **Use BlockingQueue to simplify thread pool**
  - Replace LinkedList with LinkedBlockingQueue and add an AtomicBoolean "running" to state whether the pool has been shut down or not. The reason that we need a separate field for the state is that we cannot lock the queue for exclusive access. We will not need the ReentrantLock and Condition once we have moved to the LinkedBlockingQueue.
- **Implement the awaitTermination() method**
  - Use a CountDownLatch to keep track of the live worker threads

Exercise 4.2: ExecutorService

- **Delegate to ExecutorService in ThreadPool**
  - Should be very simple code now

Exercise 4.3: Dining Philosophers

- **Problem description**
  - Thanks Marco Tedone for letting us use his code
  - We know the typical "dining philosopher" problem
    - Each philosopher has two forks, each of which is shared with an adjacent philosopher. The philosopher is either thinking, hungry or eating. In order to eat, he has to pick up both forks.
  - The challenge is to avoid a deadlock. If every philosopher first picks up the right fork, then they could all sit there holding a single fork since the left fork has been picked up by the person on his left.
  - This code appears to be correct, but there is one little problem that causes two adjacent philosophers to eat at the same time. Fix it. (Good luck!)

Exercise 4.4: ThreadLocal

- **We can call methods inside constructors**
  - Even *abstract* methods!
  - The most derived method is invoked
  - However, the subclass constructor has not been called yet
- **Try and work around this by using ThreadLocal in the subclass**
  - Hint: only works if the superclass constructor takes at least one parameter

JavaSpecialists

5: Threading Problems

Race Conditions

- **Output is dependent on timing of events**
  - Occurs in poorly-designed electronics systems
  - Also in ***multithreaded or distributed programs***

Example of Simple Race Condition

```java
// result is unpredictable
public class RaceConditionExample {
    private static long count = 0;
    public static void main(String... args) throws Exception {
        Runnable task = () -> {
            for (int i = 0; i < 100_000_000; i++) count++;
        };
        var threads = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(task);
            threads[i].start();
        }
        for (var thread : threads) thread.join();
        System.out.println("count = " + count);
    }
}
```

204068911

Fixed with LongAdder

```
public class RaceConditionExampleFixed {
  private static LongAdder count = new LongAdder();
  public static void main(String... args) throws Exception {
    Runnable task = () -> {
      for (int i = 0; i < 100_000_000; i++)
        count.increment();
    };
    var threads = new Thread[5];
    for (int i = 0; i < threads.length; i++) {
      threads[i] = new Thread(task);
      threads[i].start();
    }
    for (var thread : threads) thread.join();
    System.out.println("count = " + count);
  }
}
```

500000000

Lazy Initialization Race Conditions

- **Be careful when creating objects lazily**
  - Might create several map objects
  - Also, HashMap is not thread-safe

```
public class LazyInitializationRaceCondition {
    private Map<String, String> map;

    public String getName(String id) {
        if (map == null) map = new HashMap<>();
        return map.get(id);
    }

    public void putName(String id, String name) {
        if (map == null) map = new HashMap<>();
        map.put(id, name);
    }
}
```

ConcurrentHashMap

- **Never use plain HashMap**
  - Even if we do not need it, use the concurrent map

```
public class LazyInitializationRaceConditionFixed {
    private final Map<String, String> map =
            new ConcurrentHashMap<>();

    public String getName(String id) {
        return map.get(id);
    }

    public void putName(String id, String name) {
        map.put(id, name);
    }
}
```

How to Detect Race Conditions?

- **Can be extremely difficult to detect**
  - Happens with *check-then-act* constructs
    - Out-of-date information leads to wrong decisions
  - Often undetected until the system runs on faster server (The Law of Sudden Riches) or meets production load
- **Look for clues: corrupt data, threads stuck waiting for signals**
- **Make sure that synchronization is correct!**

Starvation

- **Thread perpetually denied resources**
  - Can cause OutOfMemoryError
  - Can prevent program from ever completing
- **Most common situation is when some low priority thread is ignored for long periods of time, preventing it from ever finishing work**
- **In Java, thread priorities are just a hint for the operating system. The mapping to system priorities is system dependent**
- **Tweaking thread priorities might result in starvation**

ReadWriteLock Starvation

- **Starvation can happen with ReadWriteLock**
  - In Java 5, readers could tag-team each other
    - Lots of readers would starve the writers
  - Since Java 6, readers can be starved if we have a single reader thread and lots of writers
- **Similar starvation may occur with StampedLock**
  - Best to use optimistic reads to avoid starving the writers
- **Only use ReadWriteLock when we are sure that we will not continuously be acquiring locks**

Detecting Thread Starvation

- **Part of the system is not performing well**
- **Look for threads with different priorities**
  - Thread dump with jstack
  - Note that Linux / Mac OS X gives all threads the same OS priority
- **Look for Thread.yield() and Thread.sleep() in odd places**

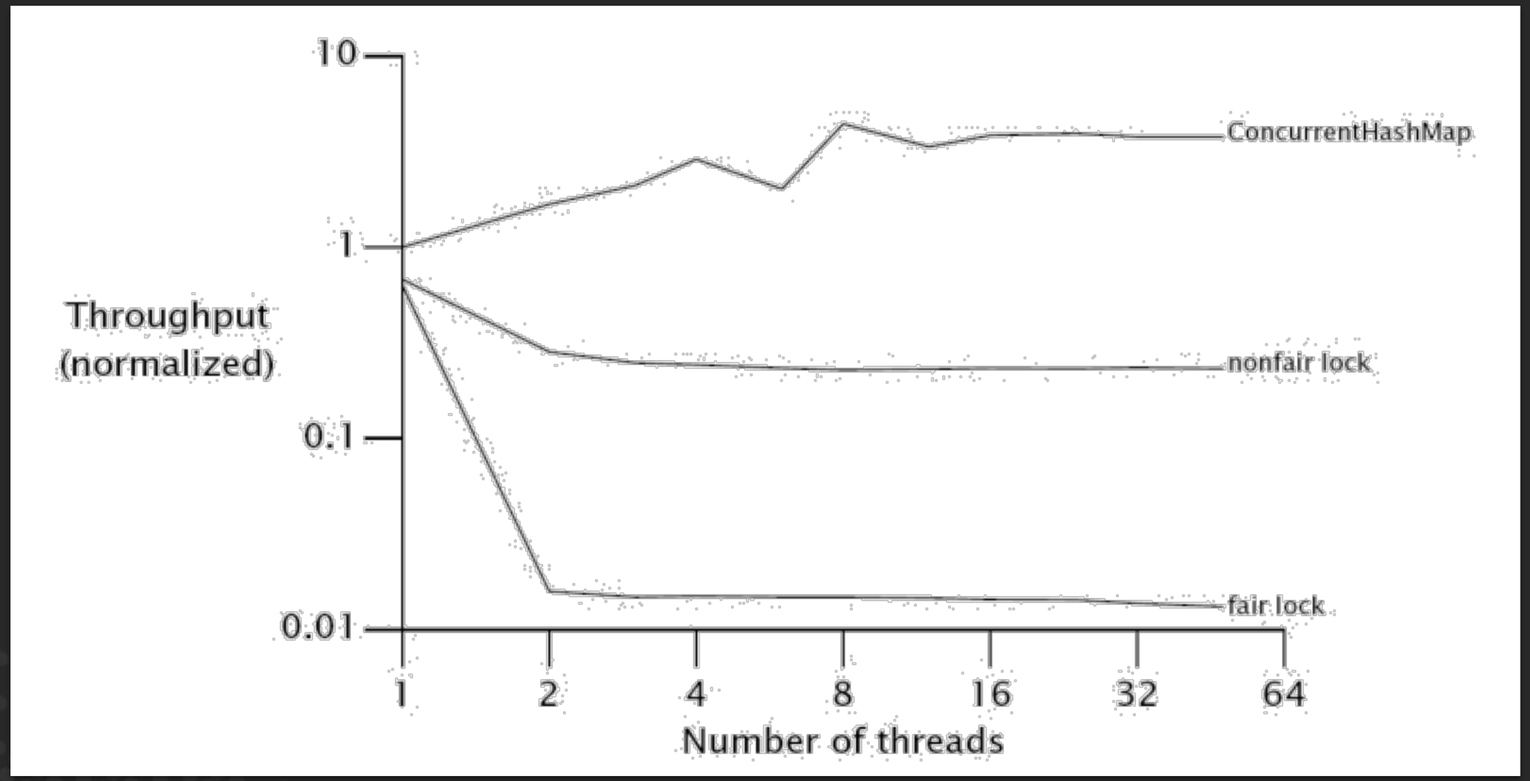
```
2023-06-17 08:56:55
"G1 Conc#0" os_prio=31 cpu=15.32ms elapsed=70244.37s
"G1 Main Marker" os_prio=31 cpu=1.68ms elapsed=70244.37s
"G1 Refine#0" os_prio=31 cpu=2.55ms elapsed=70244.37s
"G1 Refine#1" os_prio=31 cpu=0.44ms elapsed=70243.34s
"GC Thread#0" os_prio=31 cpu=590.96ms elapsed=70244.37s
"GC Thread#1" os_prio=31 cpu=585.07ms elapsed=70243.34s
"GC Thread#2" os_prio=31 cpu=588.50ms elapsed=70243.34s
"GC Thread#3" os_prio=31 cpu=578.39ms elapsed=70243.05s
"G1 Young RemSet Sampling" os_prio=31 cpu=5120.03ms elapsed=70244.37s
"VM Periodic Task Thread" os_prio=31 cpu=19056.95ms elapsed=70244.16s
"VM Thread" os_prio=31 cpu=2735.19ms elapsed=70244.33s
"Reference Handler" #2 daemon prio=10 os_prio=31 cpu=1.23ms elapsed=70244.31s
"C1 CompilerThread0" #7 daemon prio=9 os_prio=31 cpu=1035.64ms
elapsed=70244.26s
"C2 CompilerThread0" #5 daemon prio=9 os_prio=31 cpu=1400.57ms
elapsed=70244.27s
"Service Thread" #9 daemon prio=9 os_prio=31 cpu=0.06ms elapsed=70244.16s
"Signal Dispatcher" #4 daemon prio=9 os_prio=31 cpu=10.49ms elapsed=70244.27s
"Sweeper thread" #8 daemon prio=9 os_prio=31 cpu=207.54ms elapsed=70244.26s
"Common-Cleaner" #10 daemon prio=8 os_prio=31 cpu=43.67ms elapsed=70244.15s
"Finalizer" #3 daemon prio=8 os_prio=31 cpu=0.83ms elapsed=70244.31s
"JPS event loop" #12 prio=5 os_prio=31 cpu=1735.27ms elapsed=70243.12s
"DestroyJavaVM" #14 prio=5 os_prio=31 cpu=1842.43ms elapsed=70241.88s
"ObjectCleanerThread" #11 daemon prio=1 os_prio=31 cpu=203.76ms
elapsed=70243.12s
```

Fairness

- **Java locking mechanisms are non-fair**
  - One thread could get a lock repeatedly
  - Not the same as unfair
- **The operating system allocates time round-robin, thus fairness is not necessary to enforce in Java**
- **Critical sections should always be as short as possible**

Making Java Locks Fair

- **ReentrantLocks should not be made fair**
  - There is significant cost in managing fairness



  - Fair vs non-fair lock performance (Goetz JCiP 2006)

Deadlock

- **Usually several threads blocking each other**
  - But deadlock can happen with just one thread
- **Happens in these scenarios**
  - Two threads acquiring two locks in different orders
  - Alien method calls whilst holding a lock
  - ReentrantLock that we do not unlock
  - Upgrades from ReadLock to WriteLock
  - Threads trying to access static methods of classes being loaded
  - Bounded resources deadlocking due to capacity being reached

```java
// this class can cause deadlock if f() and g()
// are called by two threads
public class HappyLocker {
    private final Object lock = new Object();
    // a Thread that calls f() first acquires this
    // and then lock
    public synchronized void f() {
        synchronized (lock) {
            // do something ...
        }
    }
    // a Thread that calls g() first acquires lock
    // and then this through f()
    public void g() {
        synchronized (lock) {
            f();
        }
    }
}
```

```java
public class DeadlockExample {
    public static void main(String... args) {
        int UPTO = Integer.parseInt(args[0]);
        var pc = new HappyLocker();
        new Thread(() -> {
            for (int i = 0; i < UPTO; i++) pc.f();
            System.out.println("No deadlock");
        }, "f()").start();
        new Thread(() -> {
            for (int i = 0; i < UPTO; i++) pc.g();
        }, "g()").start();
    }
}
```

On a Single-Core Machine

```
> java DeadlockExample 10000

No deadlock

> java DeadlockExample 1000000

No deadlock

> java DeadlockExample 10000000
```

(no output – we suspect a deadlock)

How can we find deadlocks?

- **Always visible in the thread dump**
  - Generate with Ctrl+Break or Ctrl+\ or jstack
  - Some deadlocks are explicitly shown in dump
- **When we suspect a deadlock:**
  - Try and have the deadlock reproduced in a short period of time, i.e. program should not run 5 days before deadlock appears
  - When deadlock seems to appear, press CTRL+Break
  - Follow the thread stack trace to see if more than one lock is involved

jstack Output

```
Full thread dump OpenJDK 64-Bit Server VM (11+28 mixed mode):

Found one Java-level deadlock:
=============================
"f()":
  waiting to lock monitor 0x80d100 (object 0xee7c40, a java.lang.Object),
  which is held by "g()"
"g()":
  waiting to lock monitor 0x80d000 (object 0xee7c30, a HappyLocker),
  which is held by "f()"
Java stack information for the threads listed above:
===================================================
"f()":
  at HappyLocker.f(HappyLocker.java:10)
  - waiting to lock <0x0000000787ee7c40> (a java.lang.Object)
  - locked <0x0000000787ee7c30> (a HappyLocker)
"g()":
  at HappyLocker.f(HappyLocker.java:10)
  - waiting to lock <0x0000000787ee7c30> (a HappyLocker)
  at HappyLocker.g(HappyLocker.java:18)
  - locked <0x0000000787ee7c40> (a java.lang.Object)
```

Livelock

- **Thread keeps on retrying a failing operation**
  - Usually due to an unrecoverable error
  - Exceptions not handled correctly, etc.

Example – Spot the LiveLock

```
public void assign(Container container) {
    synchronized (lock) {
        PooledThread thread = null;
        do {
            while (this.idle.isEmpty()) {
                try {
                    lock.wait();
                } catch (InterruptedException ie) {
                    Thread.currentThread().interrupt();
                }
            }
            if (!this.idle.isEmpty())
                thread = this.idle.getFirst();
        } while ((thread == null) || (!thread.isRunning()));
        this.moveToRunning(thread);
        thread.start(container);
        lock.notify();
    }
}
```

JavaSpecialists

5: Exercises

Threading Problems

Exercise 5.1: What Can Happen?

```java
public class PrinterClass {
    private static final boolean OUTPUT_TO_SCREEN = false;
    private boolean printingEnabled = OUTPUT_TO_SCREEN;

    public synchronized boolean isPrintingEnabled() {
        return printingEnabled;
    }
    public void print(String s) {
        print(this, s);
    }
    private synchronized static void print(PrinterClass pc, String s) {
        if (pc.isPrintingEnabled())
            System.out.println("Printing: " + s);
    }
    public synchronized void setPrintingEnabled(boolean printingEnabled) {
        if (!printingEnabled)
            print(this, "Printing turned off!");
        this.printingEnabled = printingEnabled;
    }
}
```

Write code to demonstrate the problem

Exercise 5.2: Respond to Bala

- **Here is the email that I received from Bala**
  - Discovered a bug in ThreadLocal
  - Hi Heinz,
    I am writing this mail as i am getting null values randomly in ThreadLocal get when using HashMap as threadlocal value. The code and results are attached and i used Java1.4 and 1.5 (Sun Java). I taught you might have an answer already or you will be interested in finding one.

    I guess i need not explain about the code much as these are simple classes and the main class is ProcessTaskCaller

    I am setting a for a threadlocal in a thread and accessing the value in a different class (I am trying to access additional values in the ProcessTaskProxy class that are not passed via method)
    Thanks and Regards
    Bala

JavaSpecialists

6: Conclusion

Where to next?

6: Conclusion

- **There is more to learn about threading**
  - Extreme Java - Concurrency Performance Course
    - More detailed look at this subject, also performance
      - Includes: VarHandles, CompareAndSwap, Lock Splitting, Lock Striping, ThreadPoolExecutor deep dive, CompletableFuture, Parallel streams, Virtual Threads, ForkJoinPool, RecursiveTask, ReentrantLock, StampedLock, Finding and solving contention, memory management, and much more
  - Java Memory Model - JSR 133
  - Concurrency Interest Mailing List
  - Brian Goetz - Java Concurrency in Practice

JavaSpecialists

The End – Thank You!

Contact me on
heinz@javaspecialists.eu