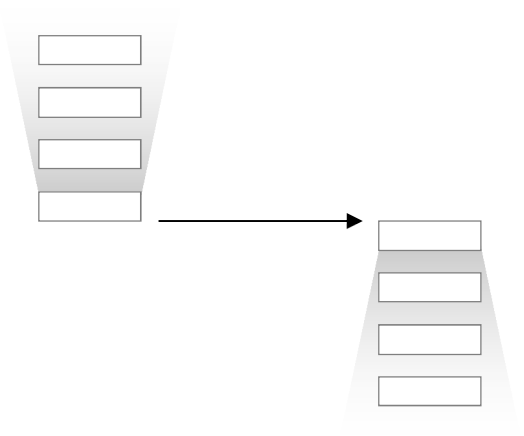
Java Memory Model

Into the core of concurrent programming



March 2013

Before we start

Maybe you ask already…

Why should I know anything about that?

There is a plenty of answers – choose for yourself:

- Sadly, many developers lack even essential pieces of knowledge like this one – but you might not anymore… which makes you special and cool
- You can prevent yourself from making dumb and hard-to-find errors (which the others do)
- You could write better code then: not only correct, but also more efficient
- After all, it is an interesting topic ;-)

Outline of the lecture

Rather ask immediately than never. You might miss your best opportunity.

I. Introduction

II. Rules for programming in concurrent environment

III. Java Memory Model explained

IV. Sharing in concurrent environment

V. Summary for the basic topics

VI. Shortly about some advanced topics

A bit of theory: what is…

…a thread (of execution)

- The smallest sequence of instructions that can be managed independently by a (system) scheduler and execute independently as well
- It is an active entity, unlike a process, it actually “runs”

…a process

- An instance of a program that is being executed
- A collection of various resources (including memory) and a container for threads
- It’s a passive entity, a process without threads would do nothing

…a shared resource

- A resource accessible to multiple processes, or rather to multiple threads

…concurrency

- A property of systems in which several computations are executing in parallel (potentially interacting with each other)

Sharing a resource (or a state)

Thread A

```
int amount = …
balance = balance + amount;
```

```
int balance;
```

A shared piece of memory

Thread B

```
int amount = …
balance = balance + amount;
```

- When sharing a resource…
  …multiple threads may access it
- When accessing at the same time
  …a conflict may occur
- When a conflict occurs…
  …the outcome is indeterminate
- Preventing parallel access prevents the conflict and prevents threads from ruining the balance
- That's why *locks* were invented
  - To prevent unwanted access ;-)
  - To make an action appear atomic

About concurrent programming

> “Perhaps surprisingly, concurrent programming isn't so much about threads or locks, any more than civil engineering is about rivets and I-beams.
>
> **Writing thread-safe code is, at its core, about managing access to state, and in particular to shared, mutable state.**”

- Alright, this **is** important, but there is something more yet
  - We’ll have to deal with threads and locks for a while to find it out

Let’s write a lock – a very, very simple lock

Can you?

A very simple lock

```
Lock.lock():
int tid = currentThreadId();
while !compareAndSet(&owner, 0, tid);
```

Not Java

```
int owner;
```

A shared piece of memory – the instance of `Lock`

```
Lock.unlock():
int tid = currentThreadId();
if (!compareAndSet(&owner, tid, 0)) {
    throw new Error(…);
}
```

Not Java

The `compareAndSet()` operation must compare a value in a variable and set it to a given value **atomically**

That's trick, I admit… but:

- such an operation is supported by the hardware usually (so called CAS instructions)
- there are solutions that can work without that (Peterson's algorithm, Dekker's algorithm)

Sharing… revisited

Thread A

```
int amount = …
L.lock();
balance = balance + amount;
L.unlock();
```

```
int balance;
Lock L;
```

A shared piece of memory

Thread B

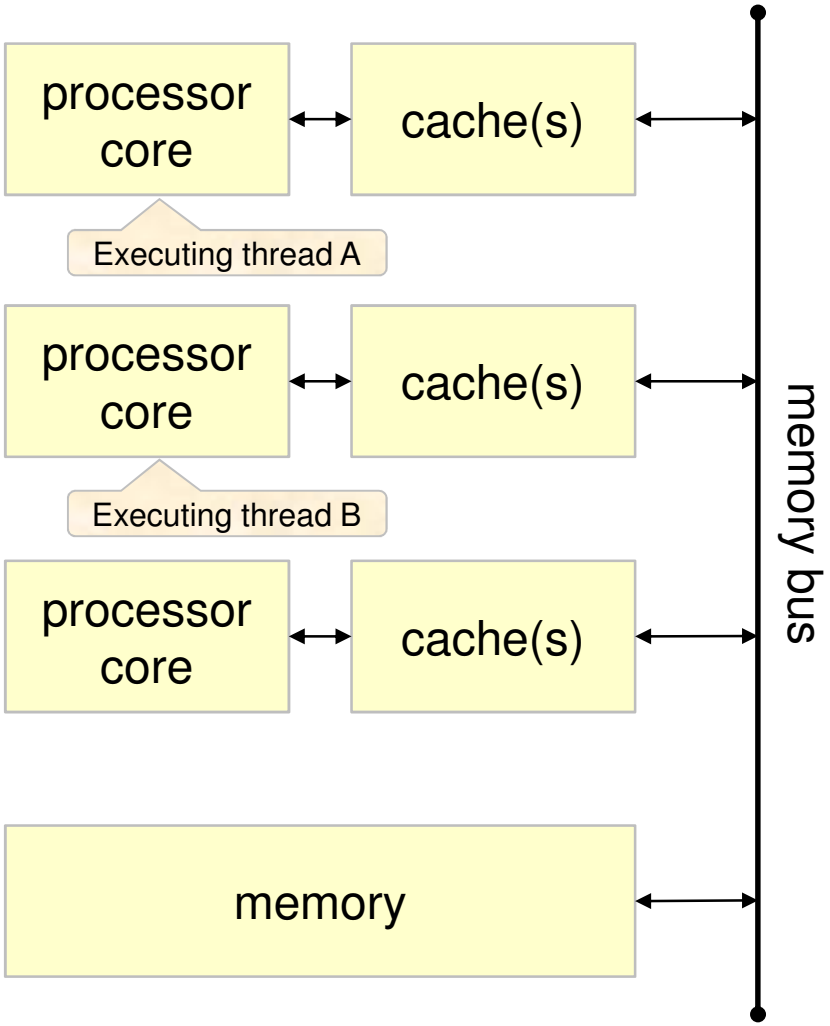
```
int amount = …
L.lock();
balance = balance + amount;
L.unlock();
```

If the lock works, the conflict is avoided…

…so we are safe, right?

Right?

About memory architecture



We would certainly be safe…

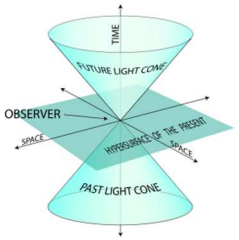
- if there is a single CPU
- or if all memory operations are synchronous

Otherwise thread A may **see** another value at the same memory address than thread B (because of the caches)

↳ We have to consider memory **visibility** for particular threads

Having all memory access synchronous is too harsh and inefficient, but it might be sufficient to synchronize just something…

Concurrent troubles



Here we come to the **basic rules of concurrent programming**:

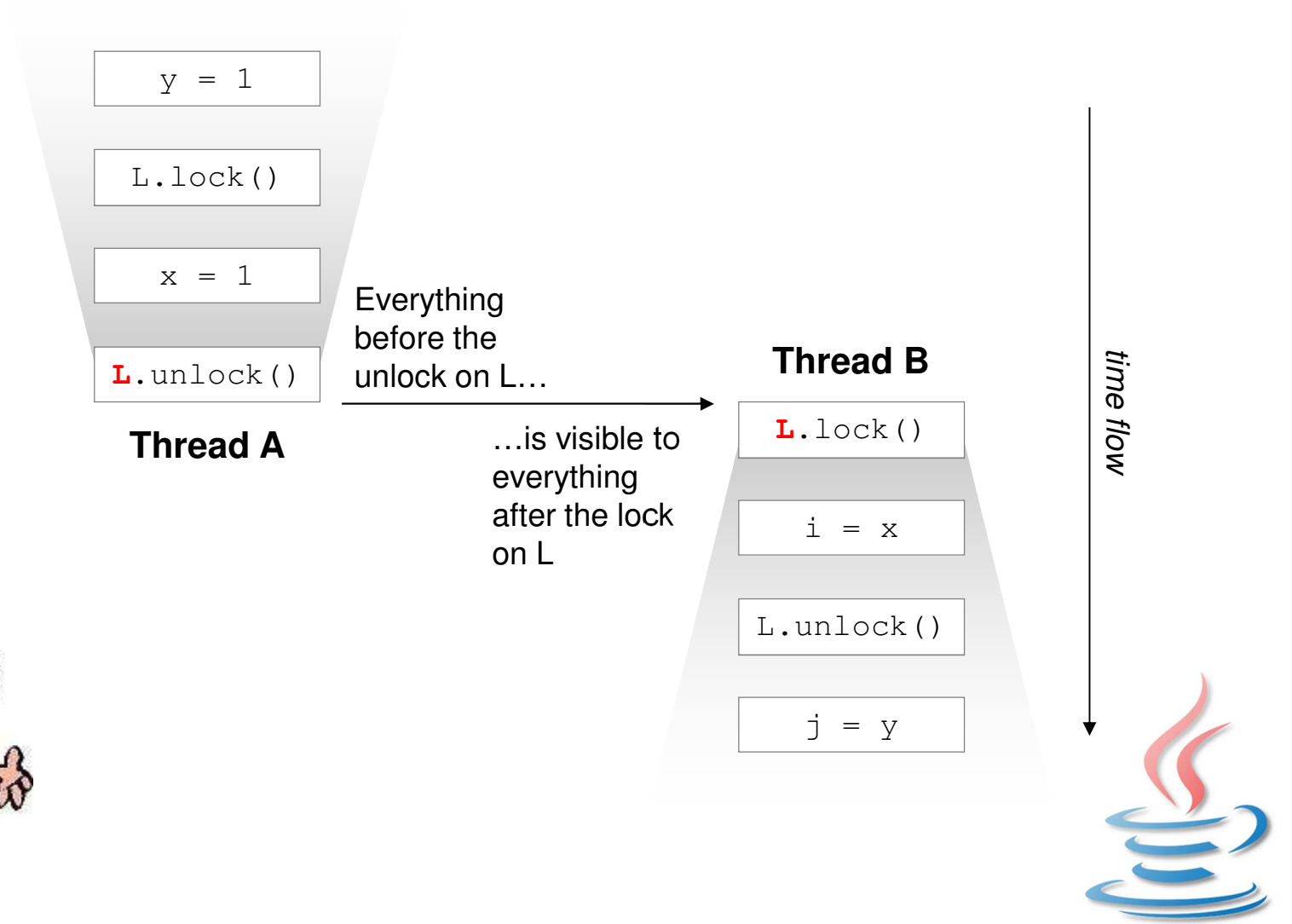
- **All actions must be considered with the respect to their mutual order**, i.e. with the respect to their executors and the observers
- **The visibility of the changes is essential**

For concurrent programming, **we need to ensure a well-defined ordering of all actions and the visibility of their effects**

"What *happens before* this and can we see the result here?"

- A total ordering is defined naturally when having a single thread only
  - All actions and their effects are ordered sequentially in the program order
- A partial ordering can be defined for multiple threads easily
  - Actions and their effects can be ordered sequentially, but just in a thread's local scope
  - An ordering with the respect to the other threads can be defined by using special actions which ensure visibility of the effects of the past local actions, which are atomic and which have guaranteed mutual order

Java Memory Model in a picture



Before and after… formally

- Java Memory Model defines a **partial ordering** called *happens-before*:

  "If action A *happens before* action B, then action B can see the results of action A (whether or not A and B occur in different threads)"

- The (basic) rules for the "happens-before" relation:
  - **Program order rule**: each action in a thread happens before every action in that thread that comes later in the program order
  - **Lock rule**: an unlock of a lock happens before every subsequent lock on that same lock (this applies on library locks as well as on intrinsic locks)
  - **Volatile variable rule**: a write to a volatile field happens before every subsequent read of that same field (this applies on atomic variables too)
  - **Thread start rule**: a request to start a thread happens before every action in the started thread
  - **Thread termination rule**: any action in a thread happens before any other thread detects that thread has terminated
  - **Interruption rule**: a thread requesting interruption on another thread happens before the interrupted thread detects the interrupt
  - **Finalizer rule**: the end of a constructor of an object happens before the start of the finalizer for that object

Benefits of Java Memory Model

- Java Memory Model implies directly:
  - Locks provide both atomicity and visibility
  - Volatile variables provide just the visibility
  - Atomic variables (see later) provide the visibility and limited atomicity

- "Piggybacking" technique
  - The strength of "happens-before" allows to piggyback sometimes on the visibility guarantees that are implied by the existing "happens-before" enforcement
  - This technique is quite fragile, but can gain some performance and allows e.g. safe using of various thread-safe collections in a natural way as a side-effect

- Reasonable price for achieving thread safety measured by the complexity and the imposed performance penalty

Stale data

- If a thread reads data from a memory place without ensuring the proper visibility guarantee, it may load *stale data* (the data which are not fresh because of a change performed by another thread in the memory place)

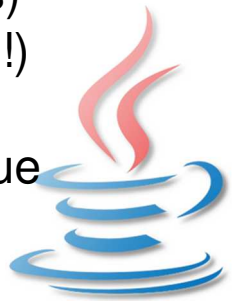
- What are the consequences?
  - Usually bad enough – it is the common cause of race conditions and other hazards… stale data can spoil anything

? But… there is something to know anyway: "out-of-thin-air" safety
  - ➢ This kind of safety tells that stale data can only contain the data written by a thread in the past (although not telling which thread and how old the data might be – you just know the data are not random)

  ☠ It does **not** apply on `double` and `long` values (see JVM internals)

  ± It applies on all other values (including references, but not objects!)

  ± This kind of safety has a use, e.g., for hash code caching technique



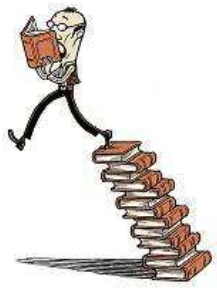
Escape… and publication

- *Publishing* an object means making it available to code outside of the object's current scope
  - Publishing an object may indirectly publish others
  - Publishing can compromise encapsulation, break invariants… (recall e.g. leaking references to an object's internals)

- When an object is published when it should not have been, it is said to have *escaped*
  - It brings many hazards: abusing or misusing the object, accessing an incomplete instance (if the object escapes during its construction), viewing an inconsistent state of the object (publishing without considering memory visibility) etc.

- Never allow an object to escape, always ensure safe publication

Safe publication: considerations

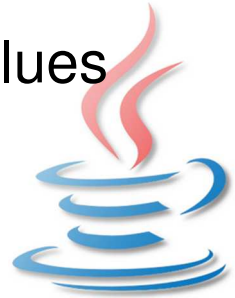


Initialization safety: `final` field semantics

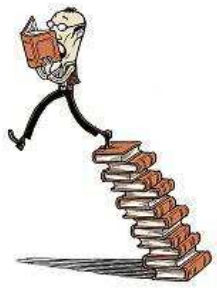
- **"A thread that can only see a reference to an object after that object has been completely initialized is guaranteed to see the correctly initialized values for that object's `final` fields"**
- This is another huge point in favor of (true) immutability and a help for other objects too (especially for achieving at least the effective immutability)

What to watch carefully?

- ☠ Construction time: leaking incomplete instance
- ⁉ Fields: inspect what is loaded and stored
- ⁉ Return values: the most common way for an escape
- ⁉ Method arguments: often not realized that they accept values
- ⁉ Inner (non-static) class instances: contain the implicit reference to the outer instance



Safe publication: techniques



What is safe then?

A properly constructed object can be published by…

- Initializing the object reference in a `static` initializer
- Storing a reference to it into a `volatile` field or an atomic reference instance (e.g. `AtomicReference`)
- Storing a reference to it into a `final` field of a properly constructed object
- Storing a reference to it into a properly synchronized field
  - Either using explicit lock or using some form of piggybacking
  - Various thread-safe library collections can be included here

⇘ Then reading from such a reference source provides a safely published object

Safe publication: summary

- ✓ Immutable objects can be published through any mechanism
- ± Effectively immutable objects must be safely published
- 😐 Mutable objects must be safely published and must be either thread-safe or guarded (see later)

- ⁉ Always document the publishing requirement and the policy how to deal with the instances – it's a part of the contract

Confinement: overview

Confining some data ~ preventing the data from sharing

- When an object is confined to a thread, its usage becomes thread-safe even if the confined object itself is not

- ✓ No sharing, no synchronization → this is the cheapest way
- ✓ It allows to use safely the code which is not thread-safe
- ✓ It can help to avoid deadlocks and overwhelming complexity
- ± It is just the matter of design and implementation
  - The compiler is not helpful at all

- Used very often, in many cases implicitly or “by the way”

Confinement: techniques

- Confining to dedicated privileged threads
  - Original and typical use, common for GUI frameworks, database connections…
- Stack confinement
  - The data are reachable through the local variables only → confined intrinsically
- Instance confinement
  - Encapsulating a thread-unsafe instance to provide thread-safe access to it
- Ad-hoc thread confinement
  - Delegating the execution of some single-threaded code to a single thread while sharing the data is avoided during the execution
  - Correct memory visibility is required!
- Thread locals
  - Every thread can use the same “global” way to access a private copy of some data

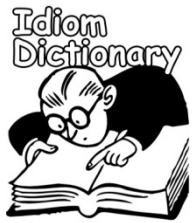
Common sharing policies

- Thread confinement
  - A thread-confined object is owned exclusively by a thread
  - Only the owning thread can access the object
- Shared read-only
  - A shared read-only object can be accessed by multiple threads without additional synchronization, but can't be modified by any thread
- Shared thread-safe
  - A thread-safe object performs synchronization internally
  - Multiple threads can freely access it through its public interface without further synchronization
- Guarded
  - A guarded object can be accessed only with a specific lock held
  - Guarded objects include those that are encapsulated within other thread-safe objects and published objects that are known to be guarded

Examples

Let’s explore some common idioms

Idioms: double-checked locking

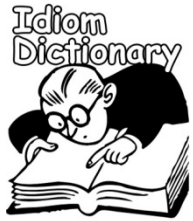


- Prior Java 5: broken (insufficient `volatile` support)
- Still often implemented wrong or misunderstood; this is a correct code snippet applied on a singleton (the original common use case):

```
public final class ResourceFactory {

    private static volatile Resource INSTANCE;

    public static Resource getInstance() {
        Resource result = INSTANCE;

        if (result == null) {
            synchronized (ResourceFactory.class) {
                result = INSTANCE;
                if (result == null) {
                    result = new Resource(…);
                    …
                    INSTANCE = result;
                }
            }
        }

        return result;
    }
}
```

Warning!

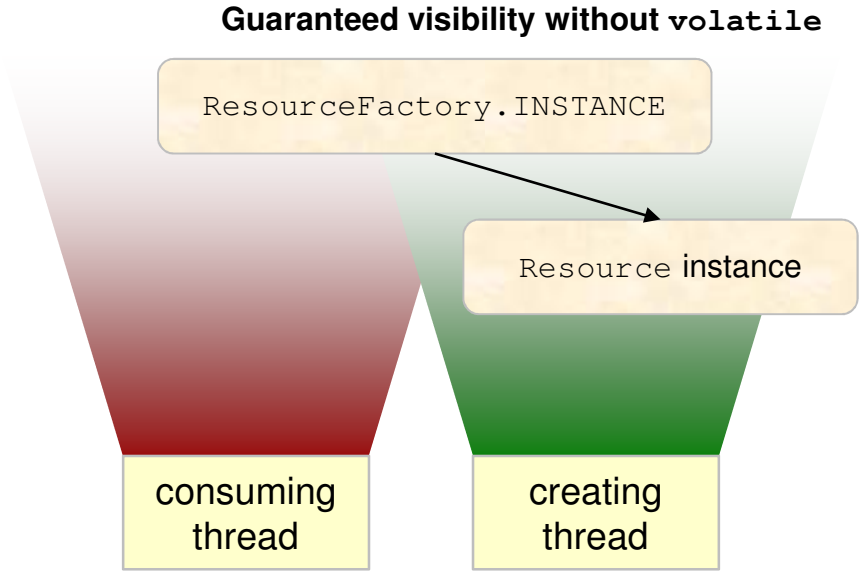
Idioms: double-checked locking



Why `volatile` is necessary?

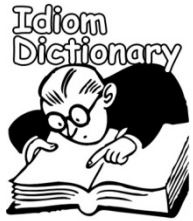
- Imagine `Resource` implementation like this:

```
public final class Resource {

    …                                   !
    private String data;

    public Resource(…) {
        …
        data = …
        …
    }

    public String getData() {
        return data;
    }
}
```



- The instance is **effective** immutable **only**
  - Such an instance requires already safe publication which does not occur if `INSTANCE` in the double-locking idiom has not volatile semantics

Idioms: lazy initialization holder

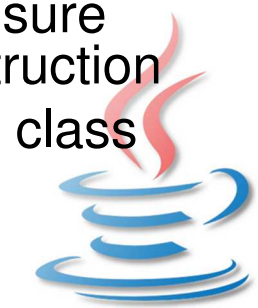


- Double-checked locking was broken before and implemented often incorrectly → for singletons (the common use case) a safer idiom was developed

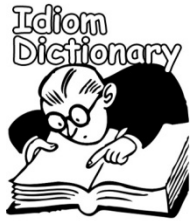
```
public final class ResourceFactory {

    private static class ResourceHolder {
        public static final Resource RESOURCE
            = new Resource();
    }

    public static Resource getResource() {
        return ResourceHolder.RESOURCE;
    }
}
```

- This idiom uses the class-loading & initialization process to ensure the safe publication and synchronization of the resource construction

😐 Unfortunately, it is usable for singletons only as it relies on the class instantiation (and `Class` instances are singletons)



Idioms: final wrapper

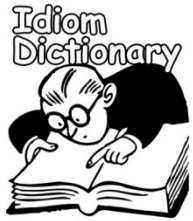


- Let's consider such a class:

```
public final class FinalValue<T> {

    private final T value;

    public FinalValue(T instance) {
        value = instance;
    }

    public T value() {
         return value;
    }
}
```

- Such a class can be used for an alternative to double-checked locking, thanks to the special semantics of `final` fields
  - …but it does not perform necessarily better than using a volatile variable
  - …and it is even less explicit and known, use with care!

Idioms: using final wrapper



- Using the `FinalValue` class which was sketched on the previous slide:

```
public final class ResourceFactory implements Factory<Resource> {

    private FinalValue<Resource> instance;

    @Override
    public Resource getInstance() {
        FinalValue<Resource> result = instance;

        if (result == null) {
            synchronized (this) {
                if (instance == null) {
                    instance = new FinalValue<>(new MyResource());
                }

                result = instance;
            }
        }

        return result.value();
    }
}
```

- ☠ The local variable `result` is required for the correctness!
- Not a well-established idiom, but a good example for `final` fields

Summary

What you should certainly remember

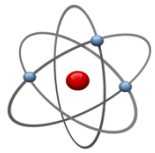
Summary of the essentials

- Actions' mutual order and visibility of their effects are essential
  - The order must be always proved using the memory model's rules
- Basic rules for *happens-before*
  - Especially "lock" and "volatile" rules
- Limit mutability, strive for true immutability – use `final`
  - Highly recommended, little known, frequently neglected
  - Limiting mutability is always a good idea anyway
- Always consider confinement
  - Often the cheapest and least complicated option
- Never allow an object to escape, publish safely + document
  - Watch out what becomes available outside an instance

Some extra stuff

For those who understand everything

About atomic variables



Classes in `java.util.concurrent.atomic`

- They have volatile read/write semantics (i.e. form *happens-before* relation as `volatile` does)
- Extra compound operations
  - Compare-and-set operations
  - Change-and-get for `AtomicInteger` and `AtomicLong`
  - Marking and stamping for `Atomic*Reference`

➢ The atomic variables are important means for light-weight synchronization and non-blocking techniques

To be continued...
See you next time

Questions?

Petr Doležal

petr.dolezal@atos.net