Atomic types

Denis Miginsky

Atomic types: atomic integer in java

```java
//java.util.concurrent.atomic.AtomicInteger
```

```java
//common operations
AtomicInteger(int value)

int get()

void set(int value)

double doubleValue()
```

```java
//"strange" operations
int getAndIncrement()

int incrementAndGet()

int addAndGet(int delta)

int getAndAccumulate(int x,
       IntBinaryOperator accumulatorFunction)

boolean compareAndSet(int expect, int update)

void lazySet()
```

Basic reference

Operations with a mutable reference:

- construction with an initial value
- dereference
- update a value
    - imperative way: dereference and modify the value itself
    - functional way: dereference, compute a new value using the previous one and replace it

Java: most variables and attributes are mutable references

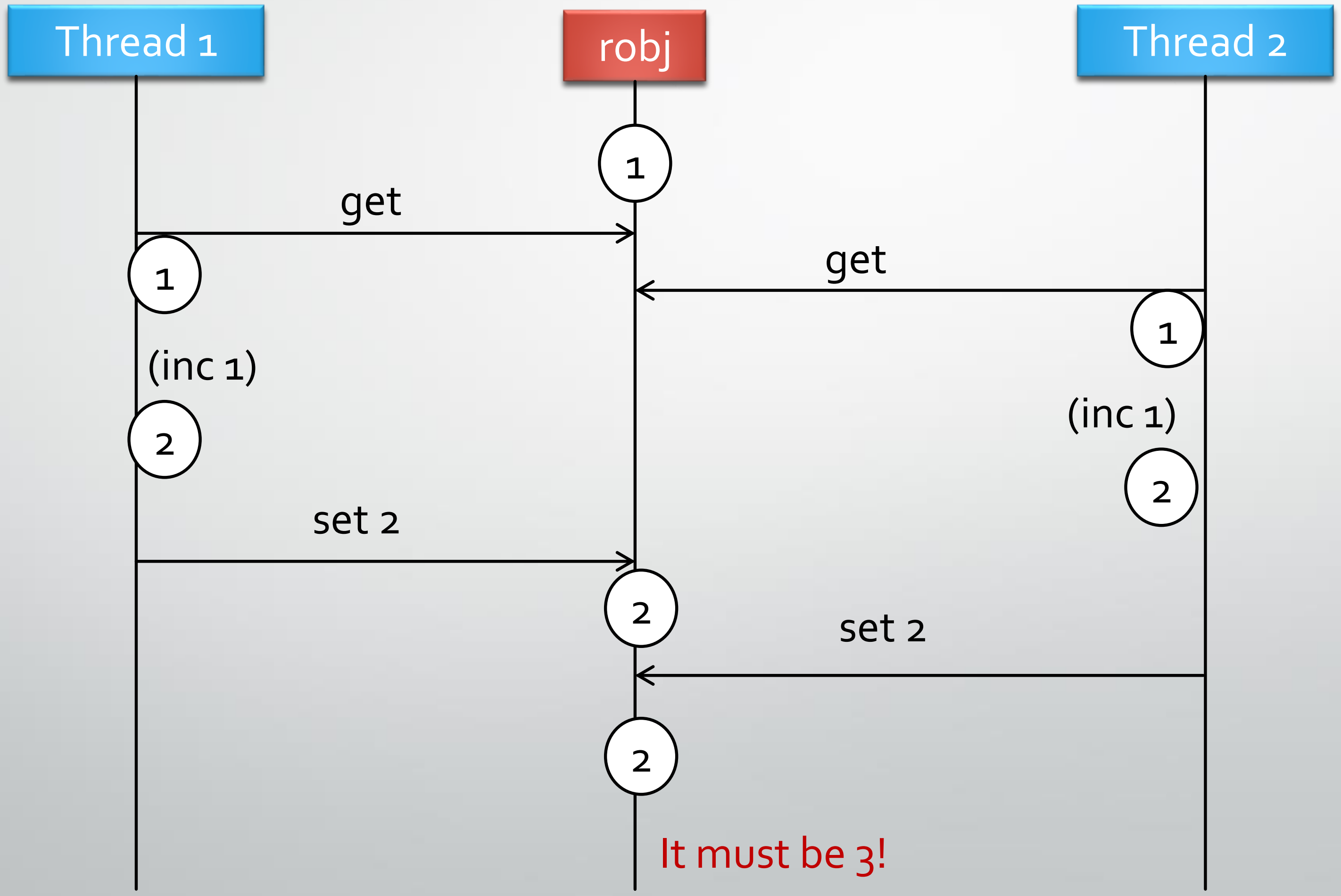
Clojure: most named objects are immutable references

Lost update

Scenario:
Two threads tries to update a reference simultaneously. The result depends on the particular race condition and a scenario type:

- the worst imperative scenario: both threads are modifying the reference at the same time; result is completely unpredictable including consistency lost
- the worst functional scenario: both threads read the reference simultaneously, then compute new values and approximately simultaneously set it back. Assuming that setting a reference is atomic operation, **one of updates will be lost**.

Lost update: illustration



Lost update: java

```java
private static int counter = 0;
private static void updateCounter() {
    for (int i=0; i < 10000; ++i) ++counter;
}
public static void lostUpdate() {
    counter = 0;
    Thread t1=new Thread(()->updateCounter());
    Thread t2=new Thread(()->updateCounter());
    t1.start();
    t2.start();
    try {
        t1.join();
        t2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Result: " + counter);
}
//>>Result: 15317
```

Lost update: java AtomicInteger

```java
private static AtomicInteger aCounter = new AtomicInteger(0);
private static void updateAtomicCounter() {
    for (int i=0; i < 10000; ++i) aCounter.incrementAndGet();
}
public static void foundUpdate() {
    aCounter.set(0);
    Thread t1=new Thread(()->updateAtomicCounter());
    Thread t2=new Thread(()->updateAtomicCounter());
    t1.start();
    t2.start();
    try {
        t1.join();
        t2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Result: " + aCounter.get());
}
//>>Result: 20000
//lost updates are fixed!
```

Atomic reference: Java

```java
private static AtomicReference<Integer> arCounter =
                          new AtomicReference<>(new Integer(0));
private static void updateARCounter() {
    for (int i=0; i < 10000; ++i)
        arCounter.updateAndGet(x->x+1);
}
public static void foundUpdate() {
    aCounter.set(0);
    Thread t1=new Thread(()->updateARCounter());
    Thread t2=new Thread(()->updateARCounter());
    t1.start();
    t2.start();
    try {
        t1.join();
        t2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Result: " + aCounter.get());
}
//>>Result: 20000
```

AtomicReference: compareAndSet

```java
//old-style (pre-Java 8) updateAndGet replacement
private static void updateARCounter() {
    for (int i=0; i < 10000; ++i) {
        while (true) {
            //subsequent two calls are not atomic together
            Integer old = arCounter.get();
            //but compareAndSet will check if the value remains unchanged
            if (arCounter.compareAndSet(old, old+1)) break;
        }
    }
}
```

Atomic reference: Clojure

```clojure
(let [robj (atom 0),                                   ;this is mutable counter
      multi-inc (fn []
                  (dotimes [n 10000]
                    (swap! robj inc))),  ;same as updateAndGet
      t1 (new Thread multi-inc),
      t2 (new Thread multi-inc)]
  (.start t1)
  (.start t2)
  (.join t1)
  (.join t2)
  (println "Result: " @robj))
;;>>20000
```

atom: API

```
;;create and initialise an atomic reference
(atom 1)
;;create an atom with additional validator
;;mutators will fail if the validator returns false
;;by throwing the java.lang.InvalidStateException
(atom 1 :validator (fn [state] (>= state 0)))
;;get the current value
(deref a)
@a
;;replace a value with a new one
(reset! a 2)
;;replace a value by passing a mutator function
(swap! a (fn [x] (* x 2)))
```

atom with validation

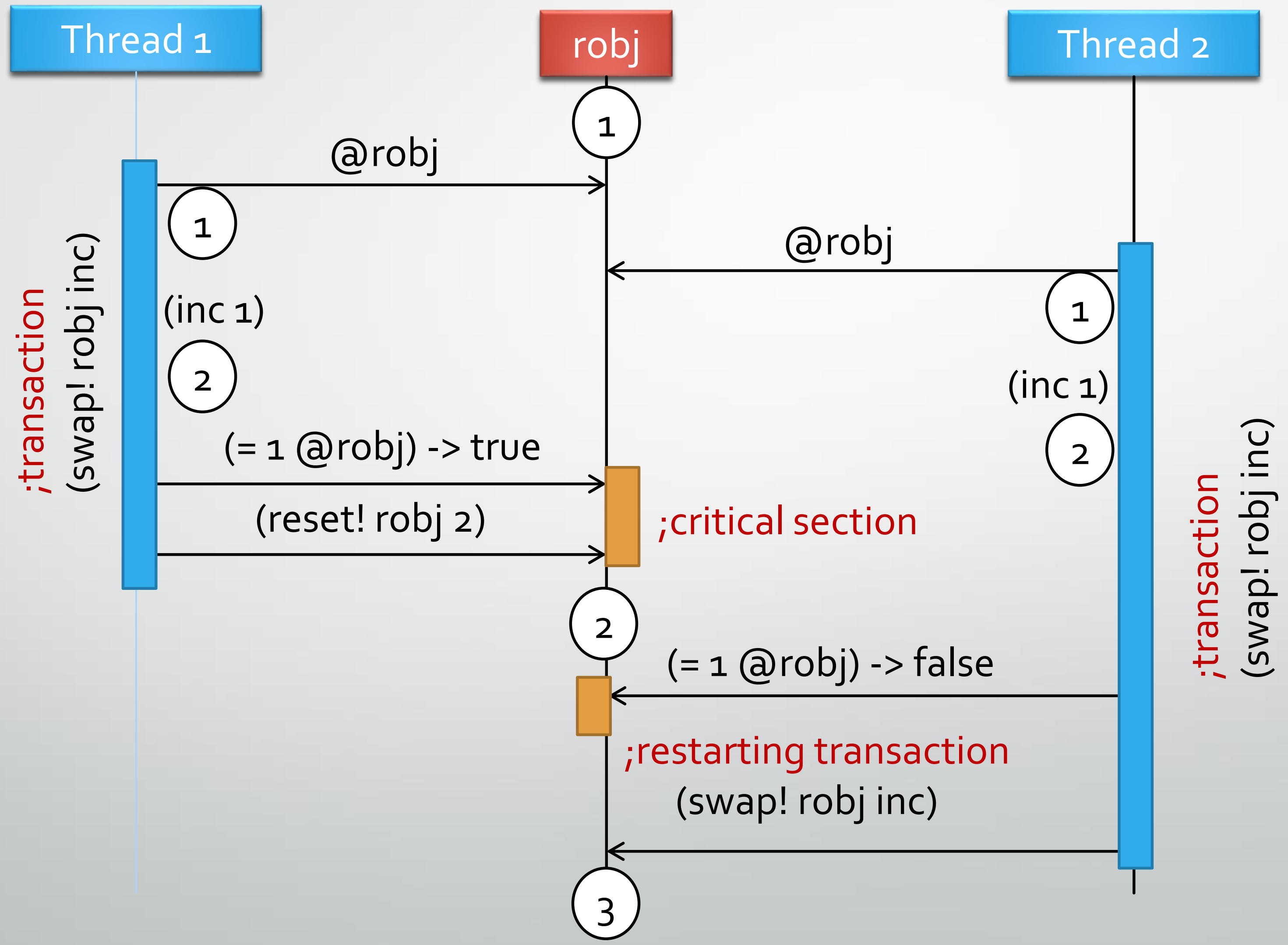
```
(let [cnt (atom 1 :validator (fn [state] (>= state 0)))]

  (println
    (try
      (swap! cnt dec)
      (catch IllegalStateException e “Error”)))
;;>> 0

  (println
    (try
      (swap! cnt dec)
      (catch IllegalStateException e “Error”)))
;;>> Error

  (println @cnt))
;;>> 0
```

swap! and optimistic lock



Optimistic and pessimistic locking

	Pessimistic	Optimistic
Paradigm	Everything is going to be bad	Everything will be alright
Errors & inconsistencies	Prevent at all costs	Identify and try to fix
Locking	Everything possibly needed and from the very beginning	What and when is absolutely needed
Cost to fix errors	Exhaustive locks	Possible restarts

Basic transaction definition

A transaction is a sequence of actions with side-effects that turns the system from one **valid state** to another one and that must be either **completed entirely or not completed at all**.

Unlike a critical section, a transaction could be reverted and restarted when necessary.

Message bus pattern

A systems is represented as a set of message processors. A processor could send a message to another processor.
A processor can either accept a message and react on it or ignore it.

Reaction on a message can include:

- changing the processor’s state (it could be mutable)
- sending new messages to other processors

A message processor is **active**, i.e. it occupies its own thread (effectively at least).
It can also make its own decisions, i.e. send messages outside the context of reaction on other messages.

Message bus pattern is useful for UI and other interactive systems as well as for parallel computing.

Similar concepts

Multi-agent system/multi-agent simulation – a system is represented as a set if interacting passive or active agents. Each agent is relatively independent and self-aware. There is not central agent who manages the entire system.

Original OOP (Smalltalk) – each object encapsulates behaviour that is a set of possible reactions on messages.

agent in Clojure

```
;;create and initialise an agent
(agent 1)
;;get the current value
(deref a)
@a
;;send a message to the given agent in form of function
;;this function will be applied to the agent's state and the
;;rest arguments eventually and the agent's state will be
;;replaced with its result
;;unlike swap! for atom it can possess side effects because it
;;will never restart and there could be no simultaneous
;;operations with the same agent
(send a + 2)
;;if a function sent to the agent threw an exception
;;the agent is considered as broken, and the exception can be
;;got by this
(agent-error a)
;;broken agent stops to accept messages
;;it can be restarted by this
(restart-agent a)
```

Agent usage example

```
(let [cnt (agent 1)]                    ;1 - initial value
  (send cnt inc)                        ;send message to modify the state
  (println @cnt)                        ;1 - still initial value
  (Thread/sleep 10)
  (println @cnt)                        ;2 - modified value after a while
  (send cnt (fn [_] (throw (new Exception)))) ;breaks the agent
  (Thread/sleep 10)
  (println @cnt)                        ;2 - value is the same
  (println (agent-error cnt))           ;stacktrace and other info
  (send cnt inc))                       ;will refuse to process a message
;>>RuntimeException: Agent is failed, needs restart
```

Task C4

Similar to the **task 14** (Java Concurrency) implement a production line for **safes** and **clocks** (code skeleton with line structure is provided).

Implementation must use **agents** for processors (factories) and **atoms** in combination with **agents** for storages.

The storage's agent must be notified by factory that the appropriate resource is produces, the agent puts it to the storage's atom and notifies all the engaged processors.

The task is to implement the processor's notification message (a function).

See requirements for the notification message implementation inside the code provided.

The production line configuration is also provided within code.

Note that there is shortage for metal. Make sure that both final wares (clocks and safes) are produced despite of this (at reduced rate of course).