

# Implementing a Thread Safe Singleton on a Multicore CPU

---



**José Paumard**

PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard <http://blog.paumard.org/>



# Agenda



The **Singleton** is a very common pattern

How to implement it in a **thread safe** way?

**Synchronization:** works but poor performances

**Double-check locking:** why is it buggy?

The right solution



# The Singleton Pattern

---



# Singleton

Well known pattern from the Gang of Four

The idea is: a Singleton class should have only one instance

Tricky to write in a concurrent environment



# First Implementation of the Singleton Pattern

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



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

Problems with this code?

Read operation

Write operation

If they occur in different threads, it is a race condition

No happens before link between them



# First Implementation

**Not thread safe!**

**1<sup>st</sup> solution: make the read and the write operations “synchronous”**



# The Synchronized Singleton Pattern

---





# Second Implementation of the Singleton

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



Execution on a  
Single Core CPU

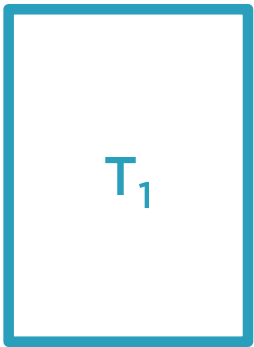
Suppose that two threads  $T_1$  and  $T_2$  are calling the `getInstance()` method

$T_1$  is the first to enter the synchronized block



# Singleton on a Single Core CPU

- 1)  $T_1$  is entering `getInstance()`
- 2)  $T_1$  executes the test

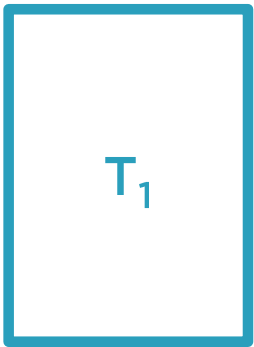


```
Singleton synchronized getInstance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}
```



# Singleton on a Single Core CPU

- 1)  $T_1$  is entering `getInstance()`
- 2)  $T_1$  executes the test
- 3) The thread scheduler gives the hand to  $T_2$



```
Singleton synchronized getInstance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}
```

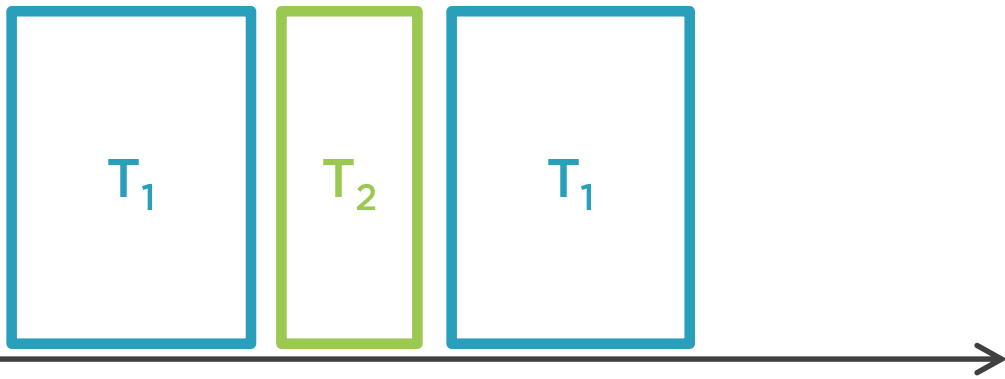
# Singleton on a Single Core CPU



```
Singleton synchronized getInstance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}
```

- 1)  $T_1$  is entering `getInstance()`
- 2)  $T_1$  executes the test
- 3) The thread scheduler gives the hand to  $T_2$
- 4)  $T_2$  tries to enter `getInstance()`

# Singleton on a Single Core CPU

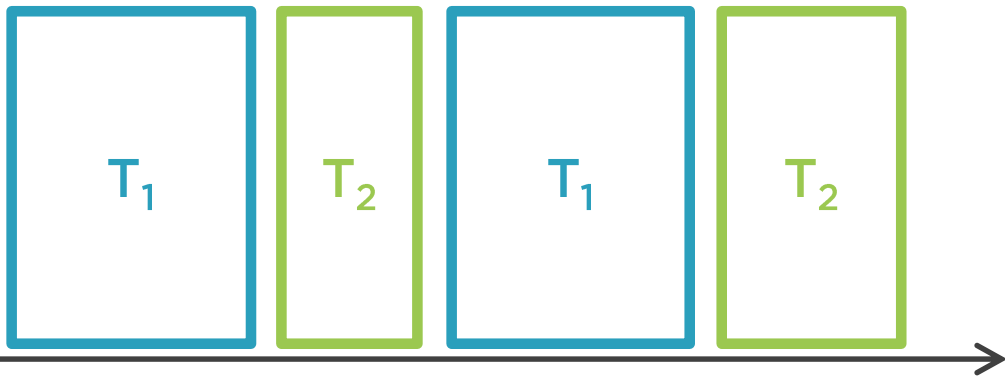


```
Singleton synchronized getInstance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}
```

- 1)  $T_1$  is entering `getInstance()`
- 2)  $T_1$  executes the test
- 3) The thread scheduler gives the hand to  $T_2$
- 4)  $T_2$  tries to enter `getInstance()`
- 5)  $T_1$  has the hand again, finishes `getInstance()`



# Singleton on a Single Core CPU

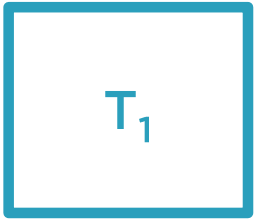


```
Singleton synchronized getInstance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}
```

- 1) T<sub>1</sub> is entering getInstance()
- 2) T<sub>1</sub> executes the test
- 3) The thread scheduler gives the hand to T<sub>2</sub>
- 4) T<sub>2</sub> tries to enter getInstance()
- 5) T<sub>1</sub> has the hand again, finishes getInstance()
- 6) T<sub>2</sub> can enter getInstance() and read instance



# Singleton on a Two Cores CPU



1)  $T_1$  is entering `getInstance()`

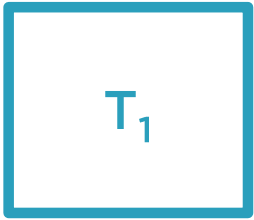
2)  $T_1$  executes the test

```
Singleton synchronized getInstance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}
```





# Singleton on a Two Cores CPU



- 1)  $T_1$  is entering `getInstance()`
- 2)  $T_1$  executes the test
- 3) The thread scheduler gives the hand to  $T_2$  *at the same time*

```
Singleton synchronized getInstance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}
```



# Singleton on a Two Cores CPU



```
Singleton synchronized getInstance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}
```

- 1)  $T_1$  is entering `getInstance()`
- 2)  $T_1$  executes the test
- 3) The thread scheduler gives the hand to  $T_2$  *at the same time*
- 4)  $T_2$  tries to enter `getInstance()`



# Singleton on a Two Cores CPU



```
Singleton synchronized getInstance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}
```

- 1)  $T_1$  is entering `getInstance()`
- 2)  $T_1$  executes the test
- 3) The thread scheduler gives the hand to  $T_2$  *at the same time*
- 4)  $T_2$  tries to enter `getInstance()`
- 5)  $T_1$  finishes `getInstance()`



# Singleton on a Two Cores CPU



```
Singleton synchronized getInstance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}
```

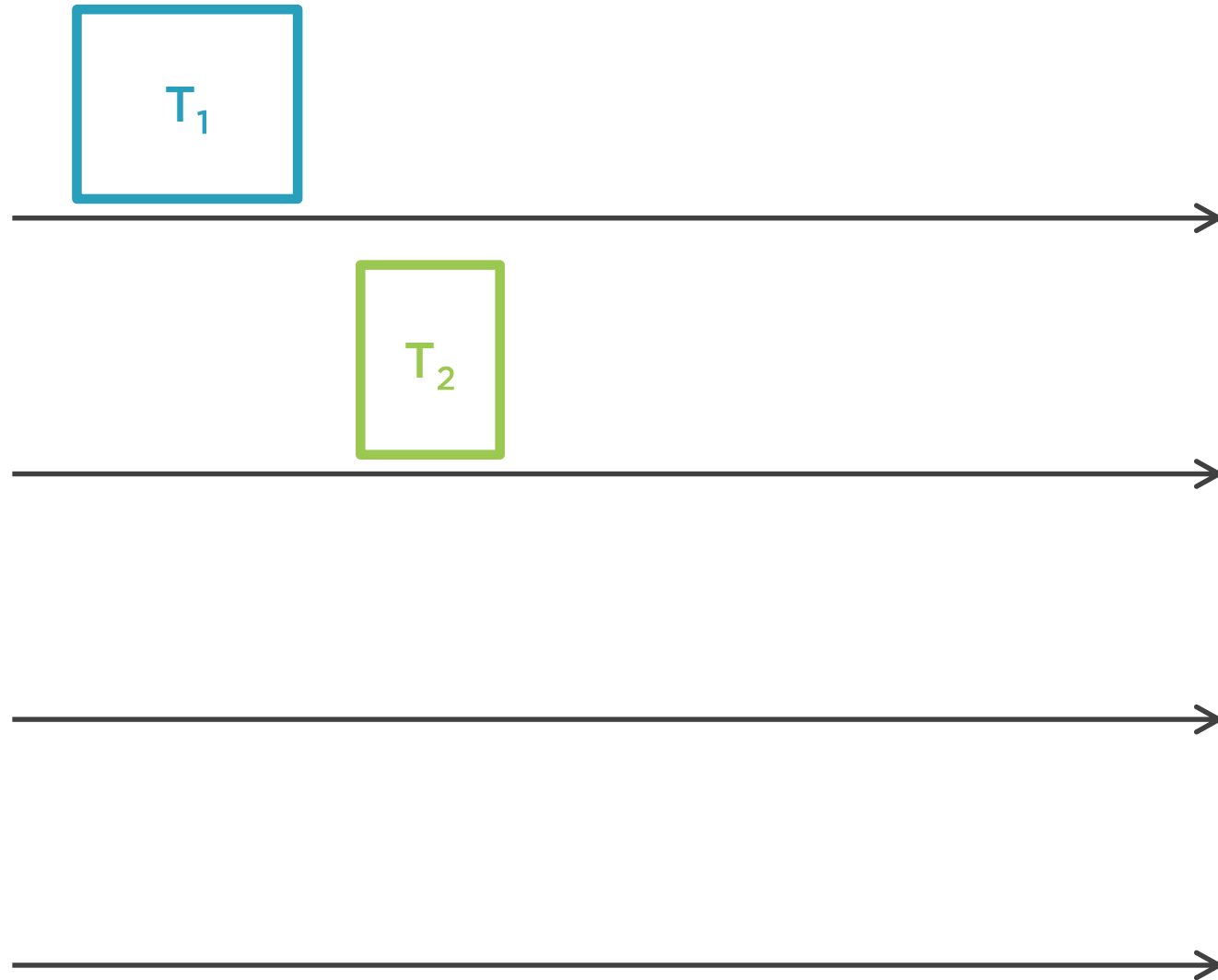
- 1)  $T_1$  is entering `getInstance()`
- 2)  $T_1$  executes the test
- 3) The thread scheduler gives the hand to  $T_2$  *at the same time*
- 4)  $T_2$  tries to enter `getInstance()`
- 5)  $T_1$  finishes `getInstance()`
- 6)  $T_2$  can enter `getInstance()` and read instance



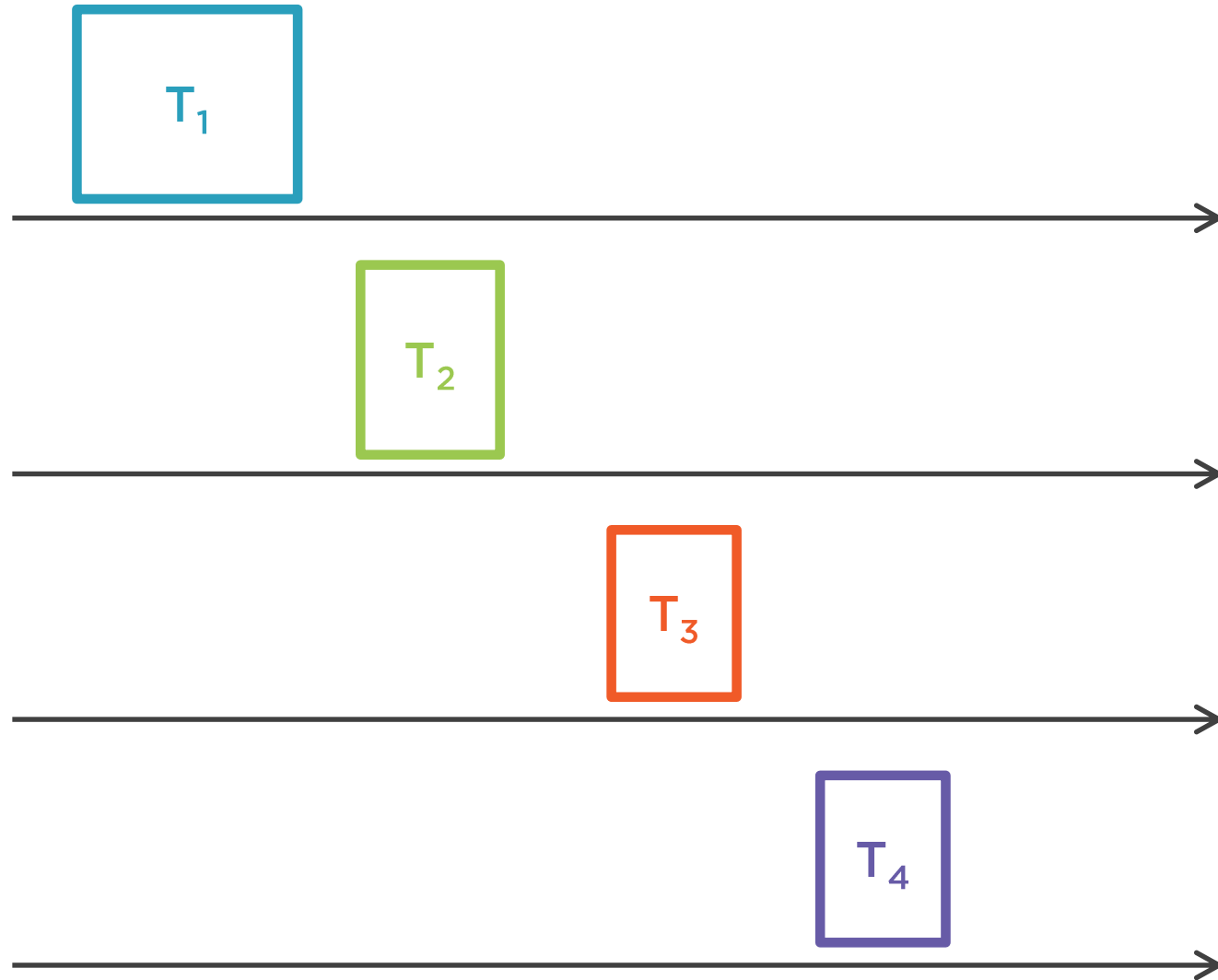
# Singleton on a Multiple Cores CPU



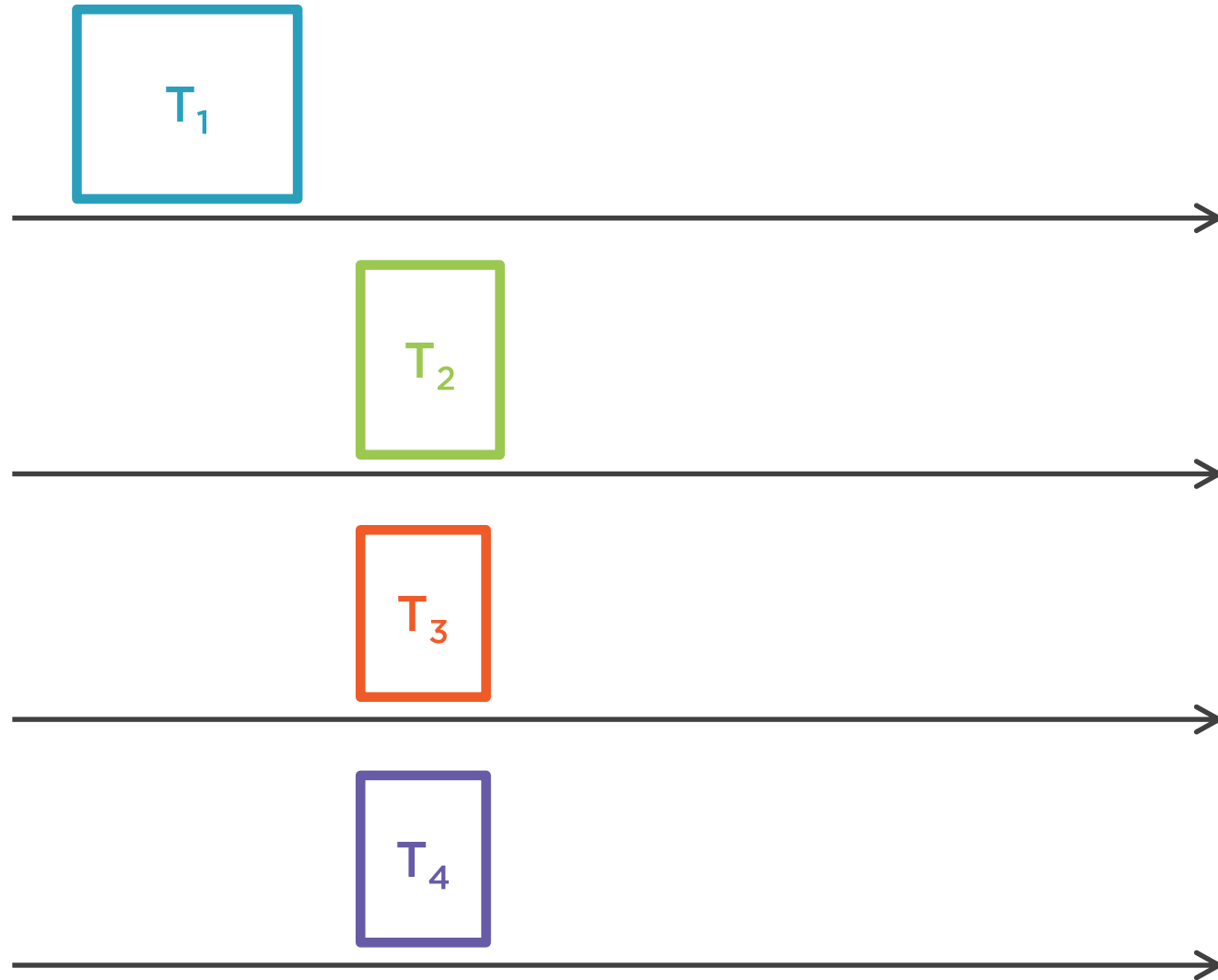
# Singleton on a Multiple Cores CPU



# Singleton on a Multiple Cores CPU



# Singleton on a Multiple Cores CPU





## Execution on a Two Cores CPU

Since the read is synchronized, it cannot be made in parallel

Once instance has been initialized, we want to be able to allow its reading in parallel



# The Double Check Locking Singleton Pattern

---



# Double Check Locking

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



# Double Check Locking

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



# Double Check Locking

It looks like it solves our reading problem...

Really?

Let us take a closer look at this code



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

If instance is not null, we read it and return it

Is it a synchronized or volatile read?

The answer is no...



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

If instance is null, we create it and return it

Is it a synchronized or volatile write?

The answer is yes...



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

So we have a non  
synchronized read

Supposed to return the value  
set by a synchronized write

Do we have the guarantee  
that the read will get the  
value set by the write?

For that we need a happens  
before link

And we do not have it!





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

So this code is buggy  
because there is no **happens  
before link** between the read  
returning the value and the  
write that sets it

A very subtle bug!



# Buggy Double Check Locking

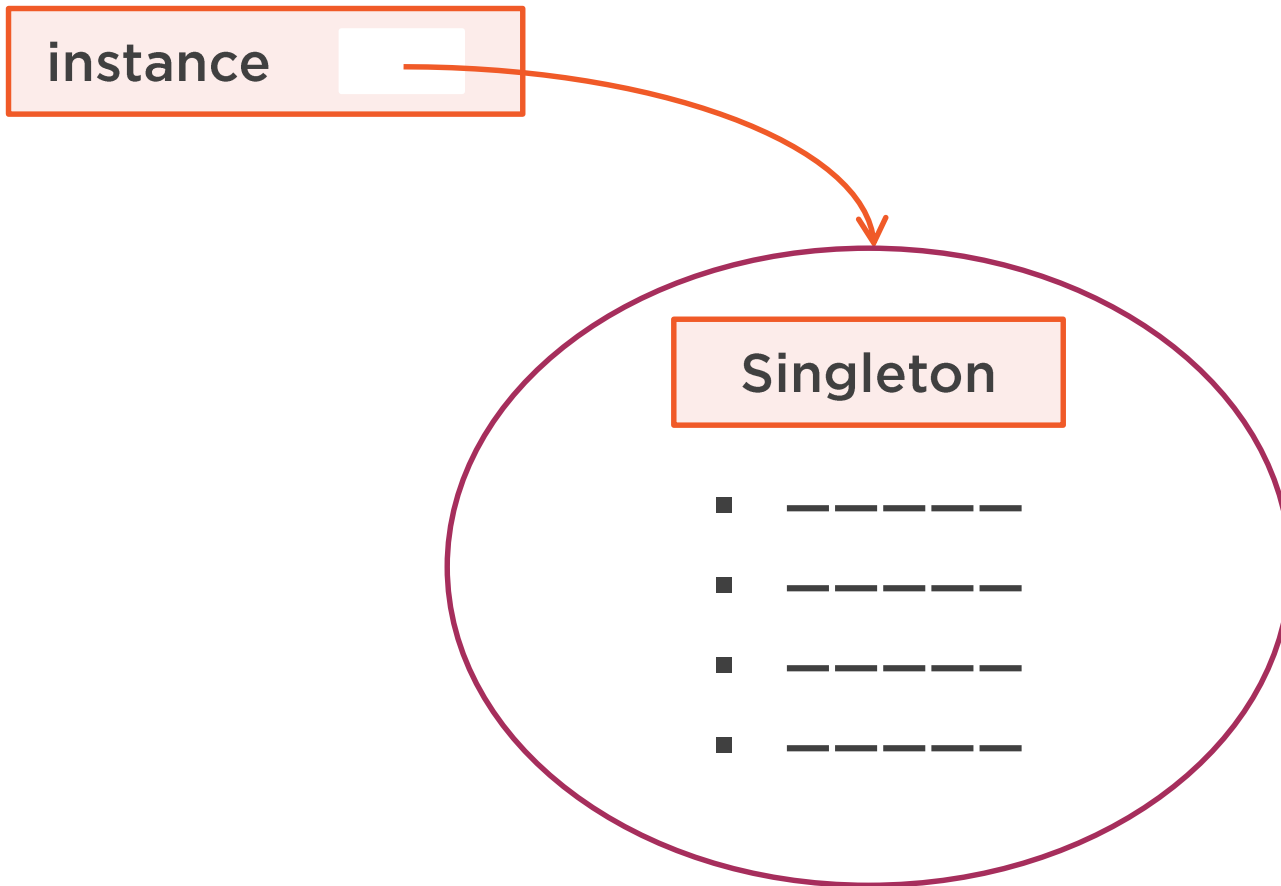
The bug of the double check locking is a concurrent bug

Cannot be observed on a single core CPU

The effect can be very weird: one can observe an object that is not fully built



# Singleton on a Multiple Cores CPU



- a) Memory allocation
- b) Copy of the pointer to the singleton field
- c) Construction process

## A Possible Solution

Since the problem comes from the non-synchronized / volatile read, let us make it volatile



# Double Check Locking (fixed)

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



## A Possible Solution

Indeed in this case, the double check  
locking is fixed

But with the same performance issues as in  
the synchronized case



# The Right Solution

```
public enum Singleton {  
    INSTANCE  
}
```



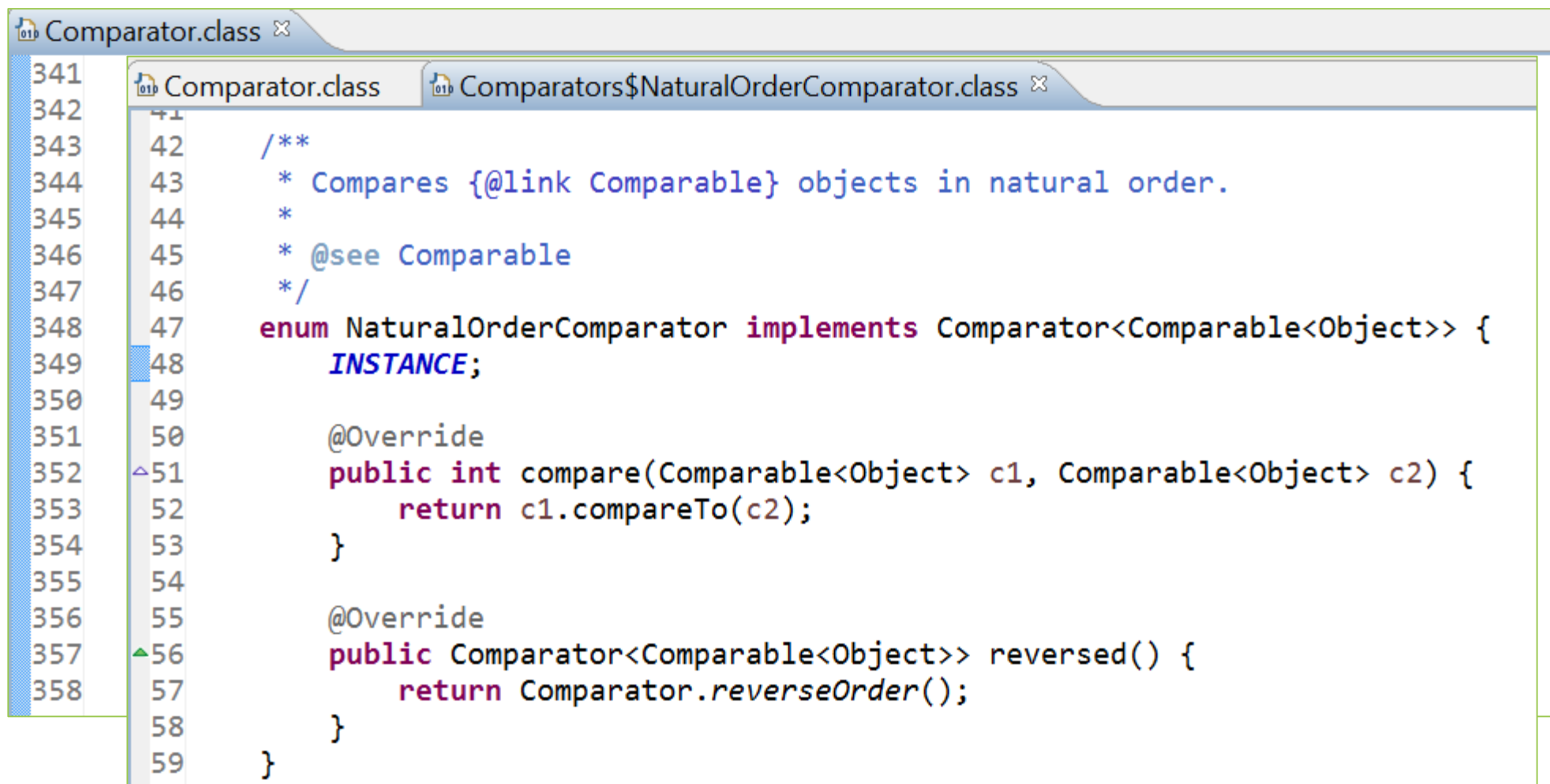
# Example of the Comparator Interface

```
Comparator.class ✕
341
342  /**
343   * Returns a comparator that compares {@link Comparable} objects in natural
344   * order.
345   *
346   * <p>The returned comparator is serializable and throws {@link
347   * NullPointerException} when comparing {@code null}.
348   *
349   * @param <T> the {@link Comparable} type of element to be compared
350   * @return a comparator that imposes the <i>natural ordering</i> on {@code
351   *         Comparable} objects.
352   * @see Comparable
353   * @since 1.8
354   */
355  @SuppressWarnings("unchecked")
356  public static <T extends Comparable<? super T>> Comparator<T> naturalOrder() {
357      return (Comparator<T>) Comparators.NaturalOrderComparator.INSTANCE;
358  }
```





# Example of the Comparator Interface



```
341
342
343 42 /**
344 43  * Compares {@link Comparable} objects in natural order.
345 44  *
346 45  * @see Comparable
347 46  */
348 47 enum NaturalOrderComparator implements Comparator<Comparable<Object>> {
349 48     INSTANCE;
350 49
351 50     @Override
352 51     public int compare(Comparable<Object> c1, Comparable<Object> c2) {
353 52         return c1.compareTo(c2);
354 53     }
355 54
356 55     @Override
357 56     public Comparator<Comparable<Object>> reversed() {
358 57         return Comparator.reverseOrder();
359 58     }
359 59 }
```



# Demo



Let us see some code!

Watching volatility in action

Fixing the example of the first module



# How to Write Correct Concurrent Code?

## 1) Check for race conditions

- They occur on fields (not variables / parameters)
- 2 threads are reading / writing a given field

## 2) Check for the happens-before link

- Are the read / write volatile?
- Are they synchronized?
- If not, there is a probably bug

## 3) Synchronized or volatile?

- Synchronized = atomicity
- Volatile = visibility



# Wrapup



What did we learn?

We saw how very subtle bugs can arise in concurrent programming

The only way to find them is to check the read and write operations

Are they synchronized or volatile?

If not, then race conditions can occur

