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

```java
public class Singleton {

    private static Singleton instance;

    private final Singleton() {}

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

```java
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!

1st 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


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

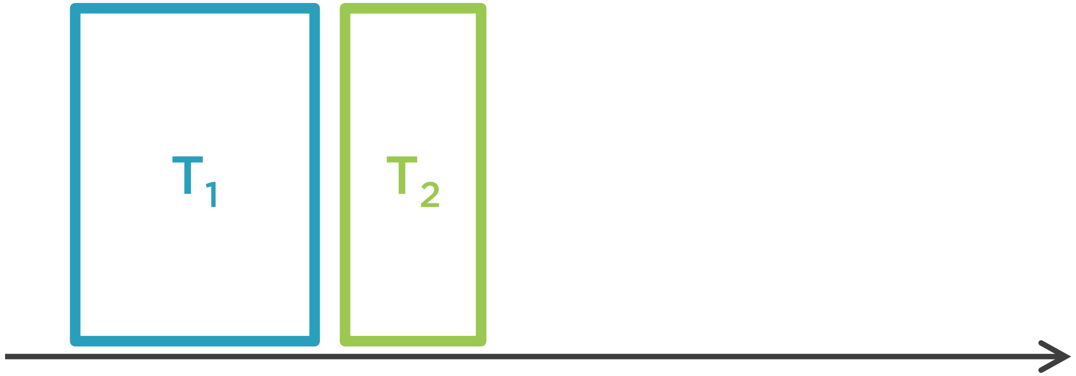
1) T_1 is entering `getInstance()`
2) T_1 executes the test

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

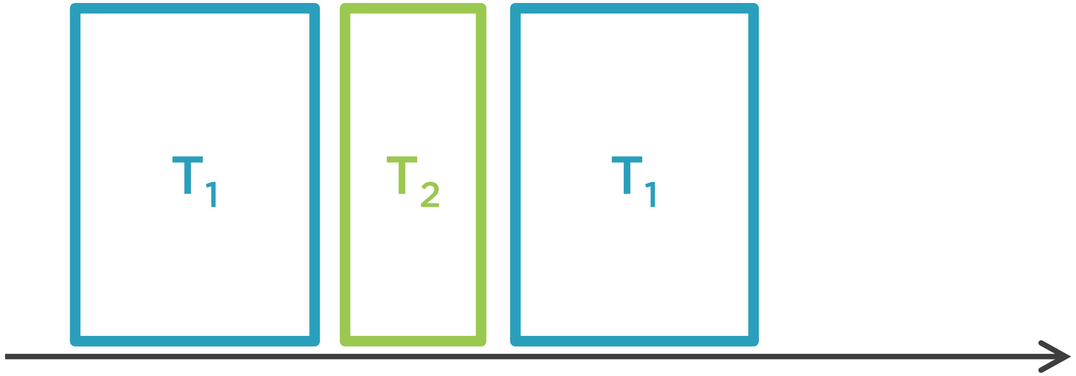
Singleton on a Single Core CPU




```
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()`

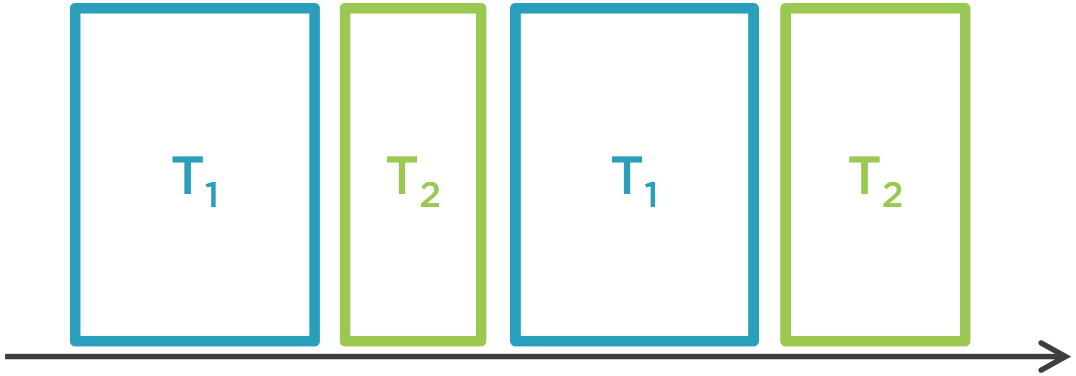
Singleton on a Single Core CPU




```
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




```java
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()`
6) T_2 can enter `getInstance()` and read `instance`

Singleton on a Two Cores CPU


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

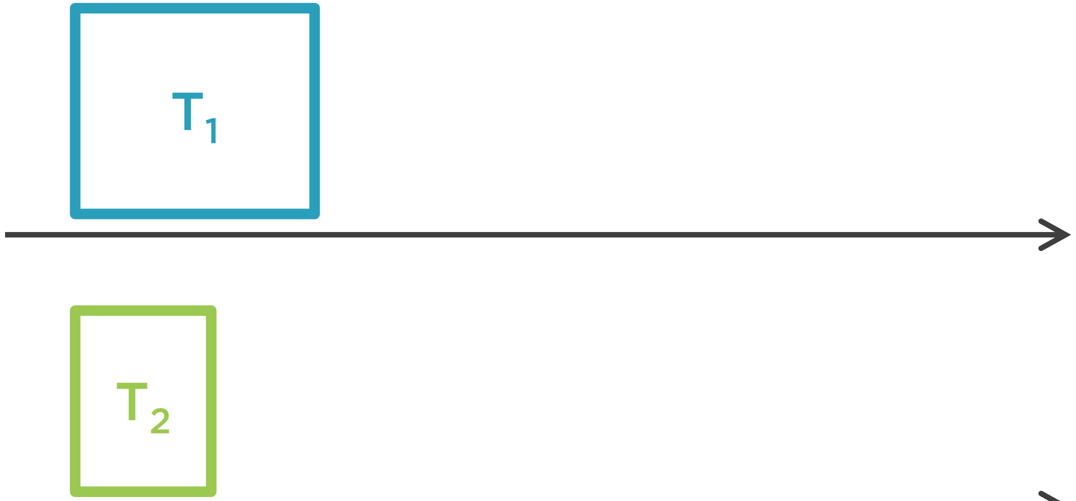
1) T_1 is entering `getInstance()`
2) T_1 executes the test

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*

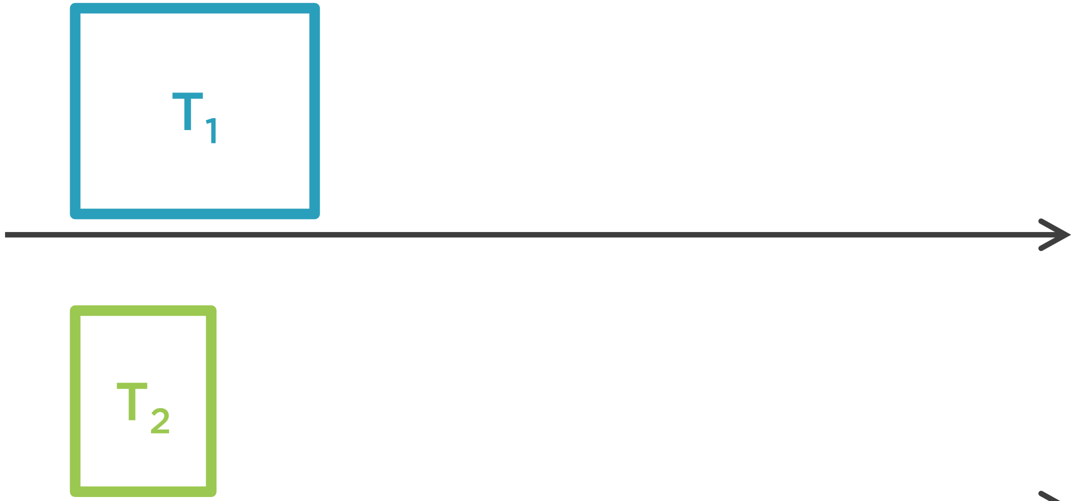
Singleton on a Two Cores CPU




```
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()`

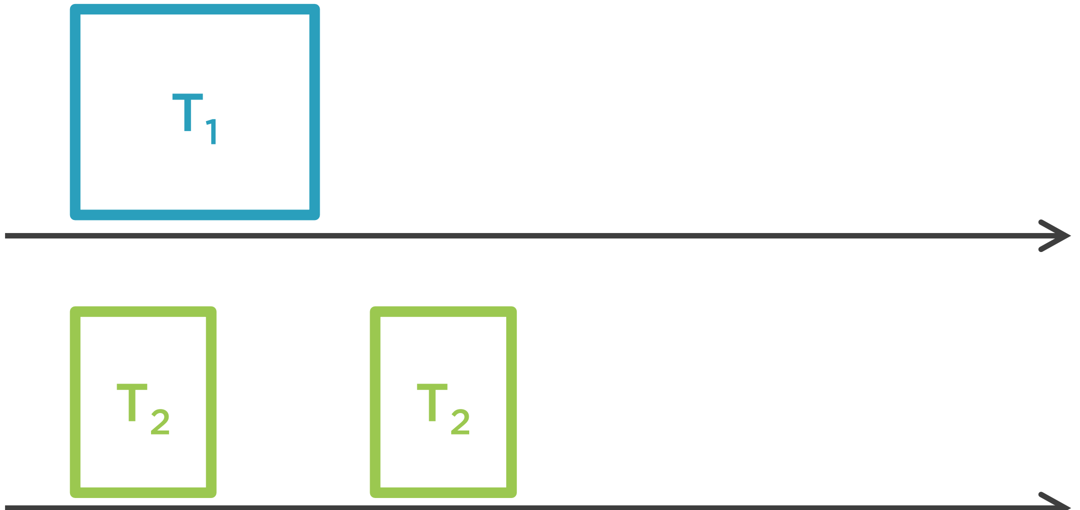
Singleton on a Two Cores CPU




```java
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()`

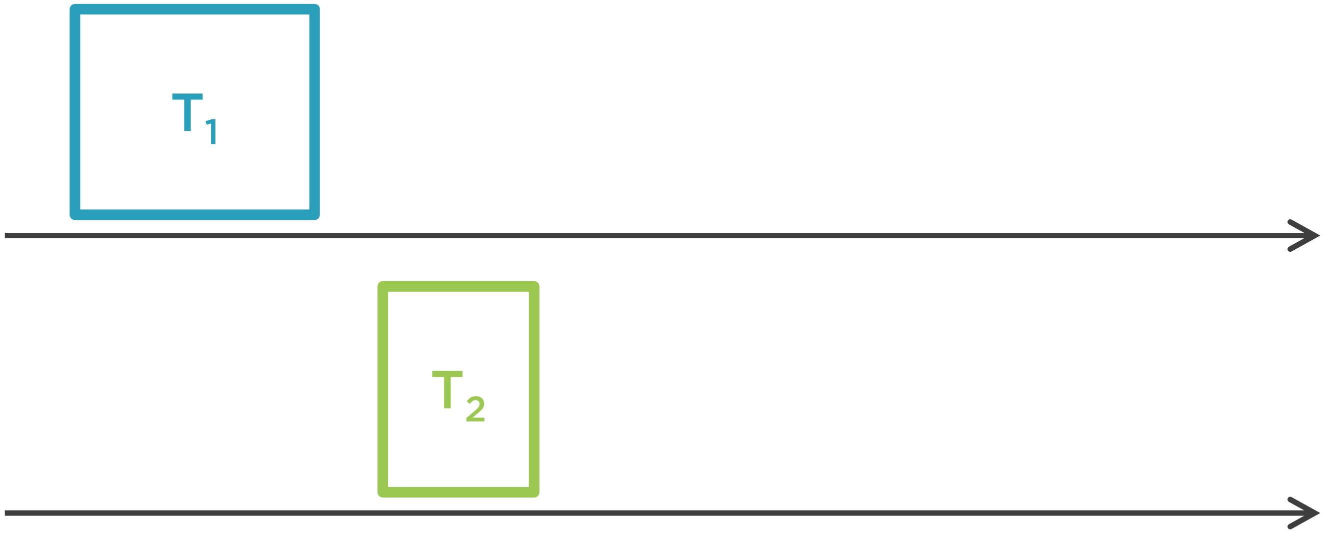
Singleton on a Two Cores CPU



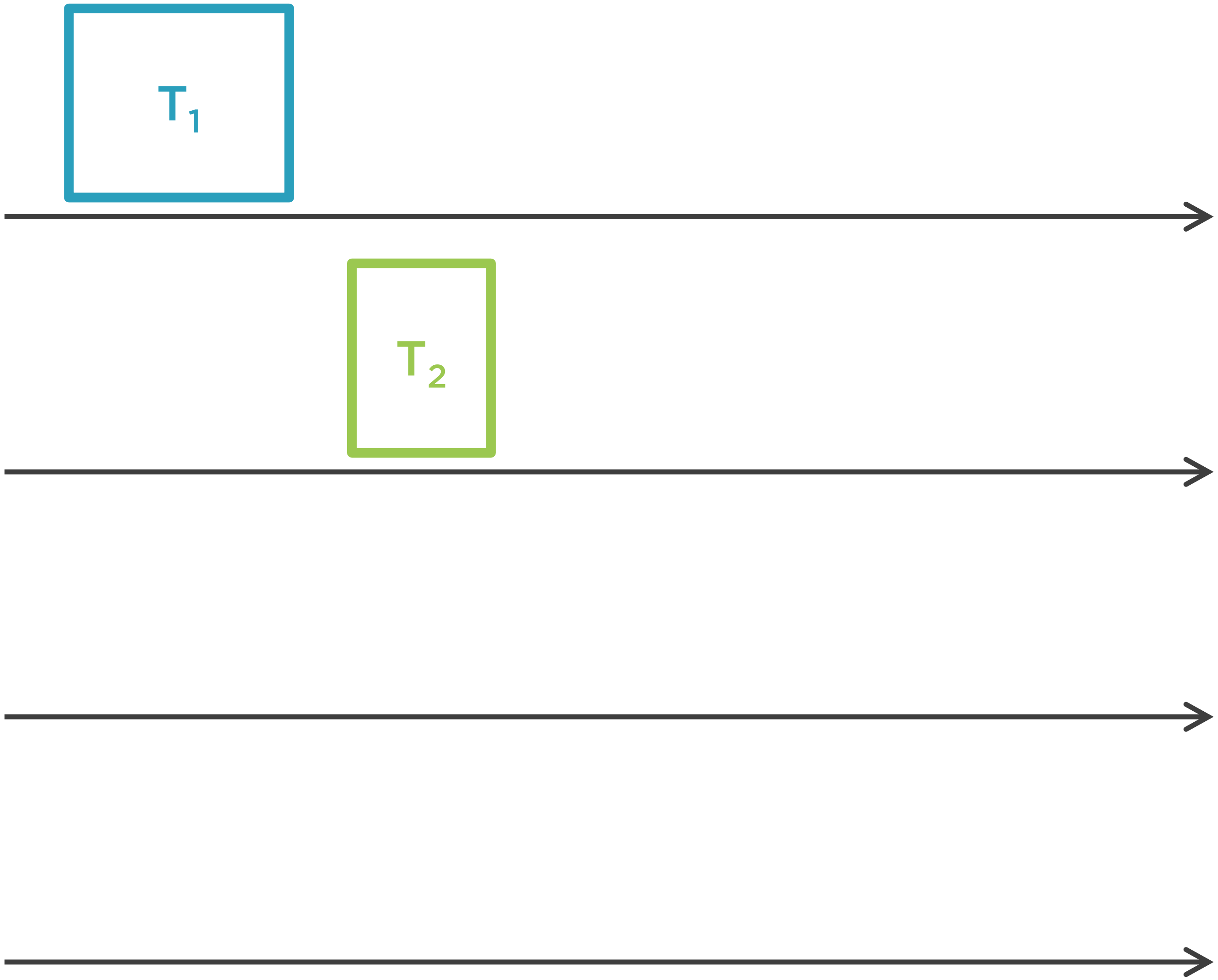

```
Singleton synchronized getInstance() {
   if (instance == null) {
      instance = new Singleton();
   }
   return instance;
}
```

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`

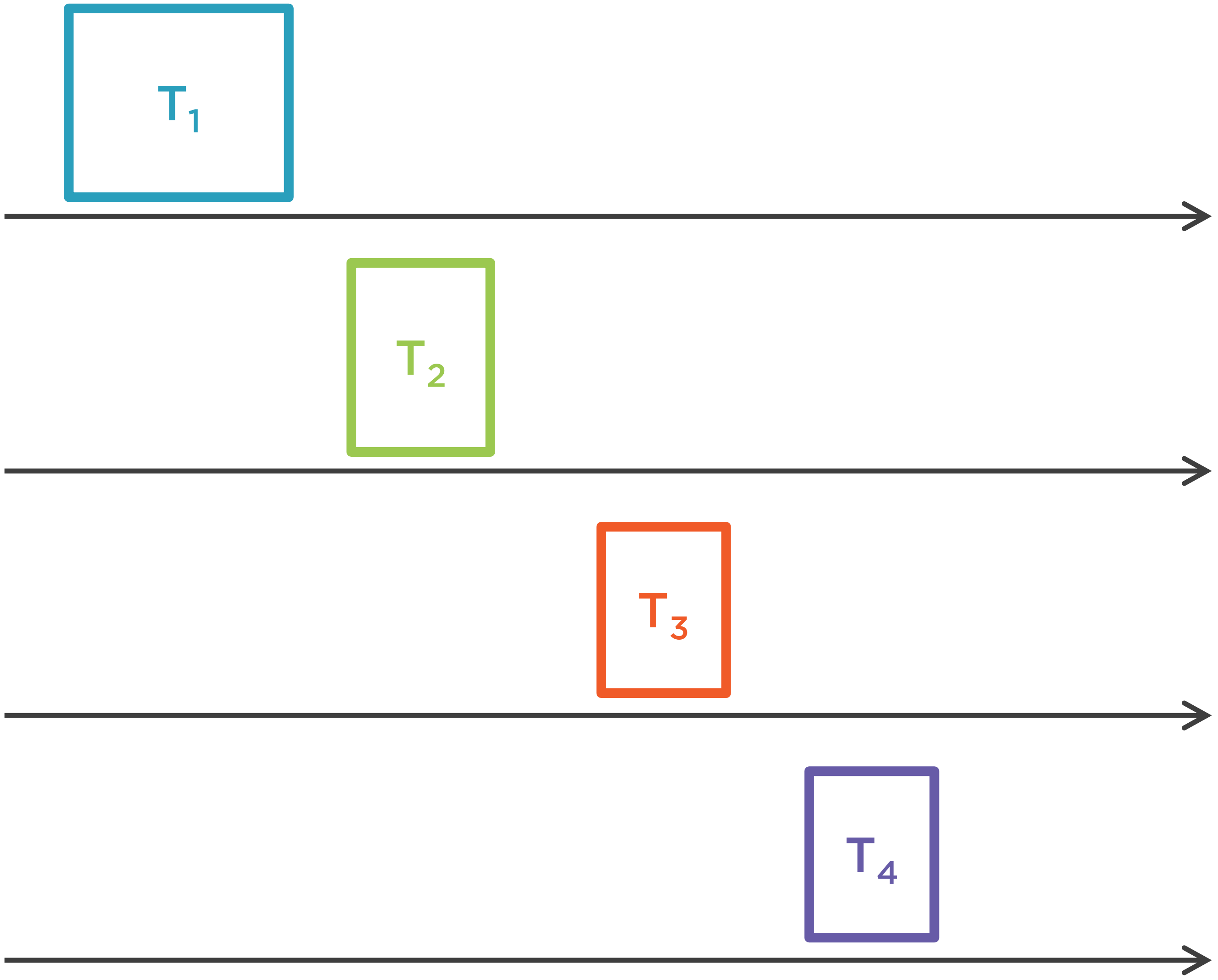
Singleton on a Multiple Cores CPU



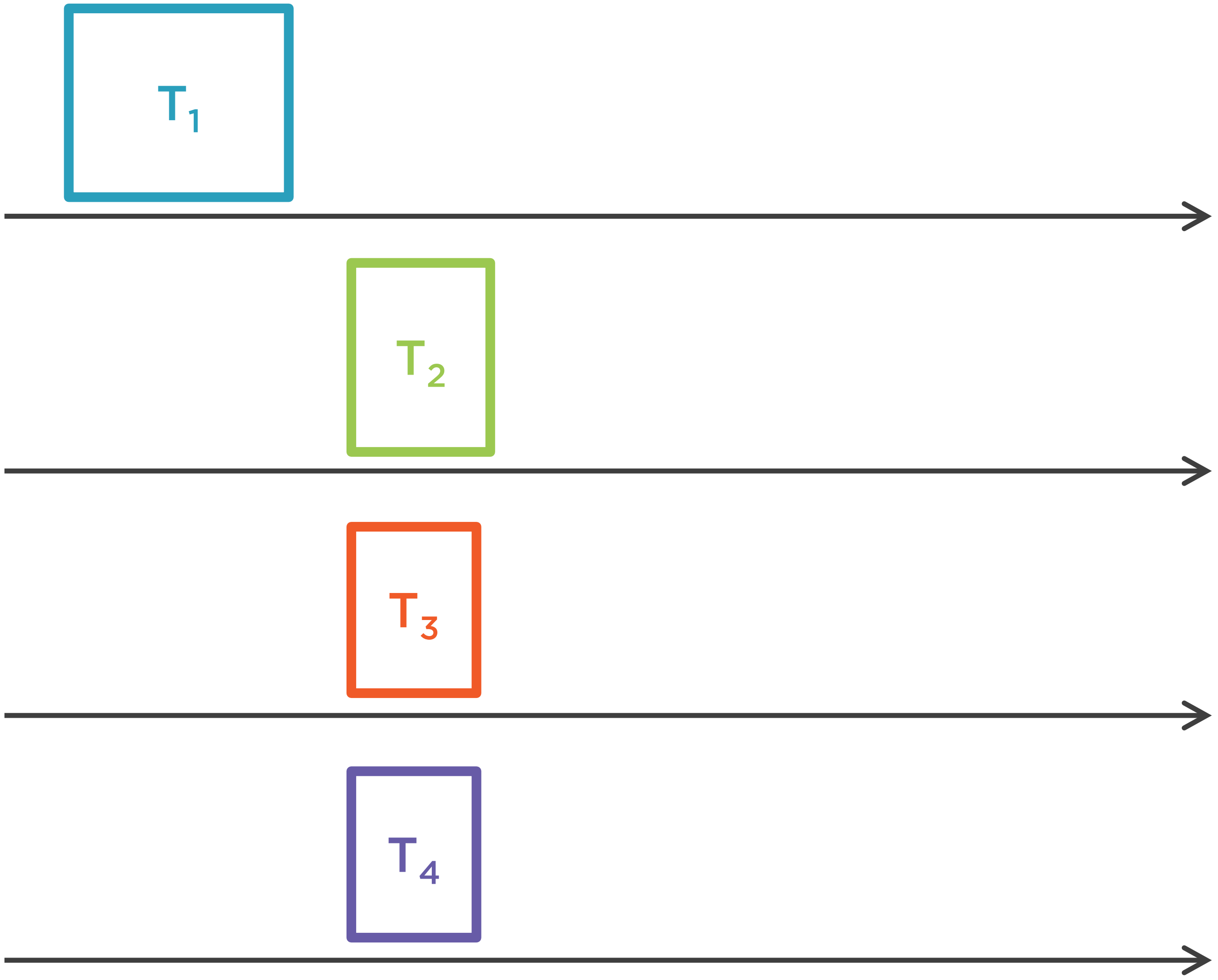
Singleton on a Multiple Cores CPU



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

```java
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…

```java
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…

```java
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!

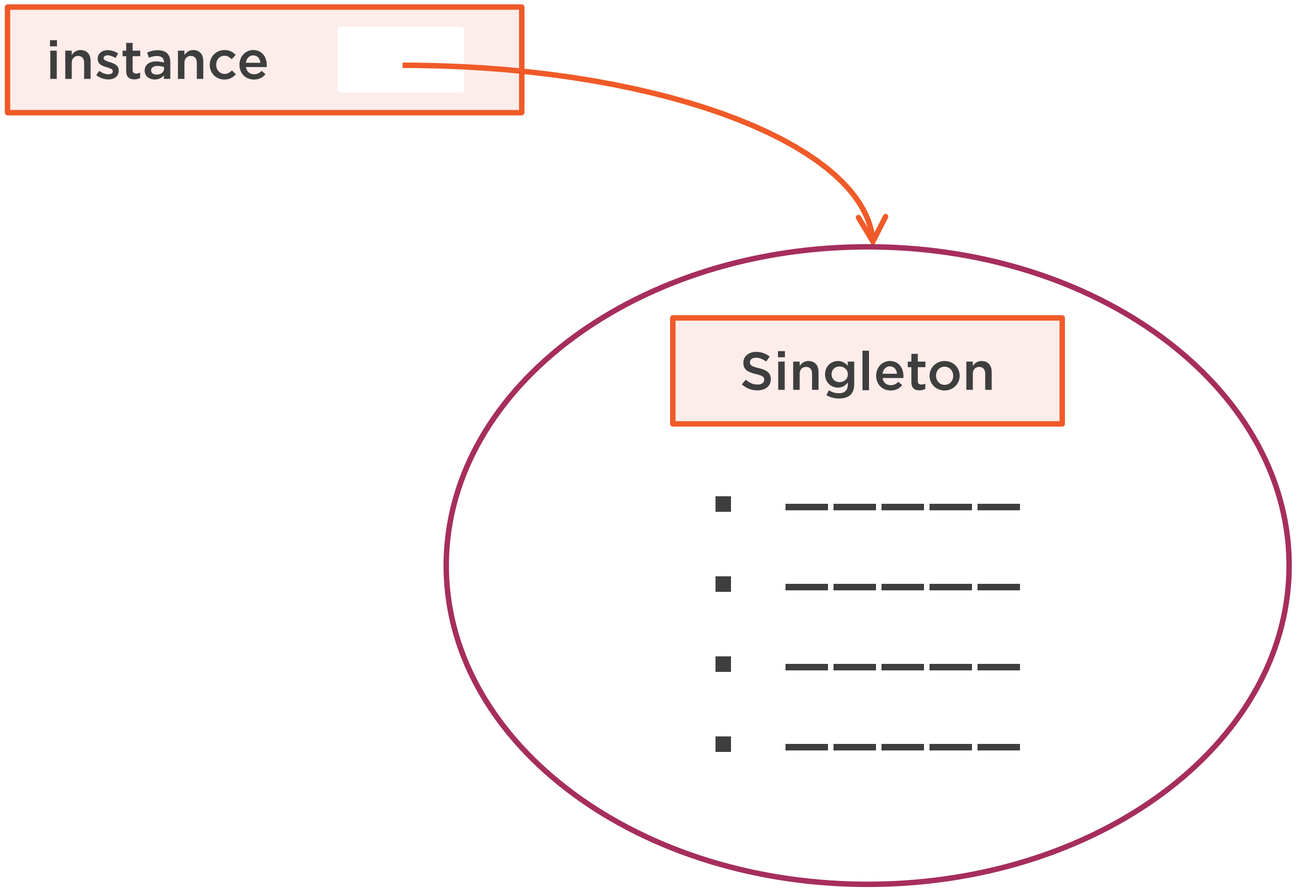
Buggy Double Check Locking

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

```java

    /**
     * Returns a comparator that compares {@link Comparable} objects in natural
     * order.
     *
     * <p>The returned comparator is serializable and throws {@link
     * NullPointerException} when comparing {@code null}.
     *
     * @param  <T> the {@link Comparable} type of element to be compared
     * @return a comparator that imposes the <i>natural ordering</i> on {@code
     *         Comparable} objects.
     * @see Comparable
     * @since 1.8
     */
    @SuppressWarnings("unchecked")
    public static <T extends Comparable<? super T>> Comparator<T> naturalOrder() {
        return (Comparator<T>) Comparators.NaturalOrderComparator.INSTANCE;
    }
```

Example of the Comparator Interface

Comparator.class

Comparator.class | Comparators$NaturalOrderComparator.class

```java
    /**
     * Compares {@link Comparable} objects in natural order.
     *
     * @see Comparable
     */
    enum NaturalOrderComparator implements Comparator<Comparable<Object>> {
        INSTANCE;

        @Override
        public int compare(Comparable<Object> c1, Comparable<Object> c2) {
            return c1.compareTo(c2);
        }

        @Override
        public Comparator<Comparable<Object>> reversed() {
            return Comparator.reverseOrder();
        }
    }
```

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