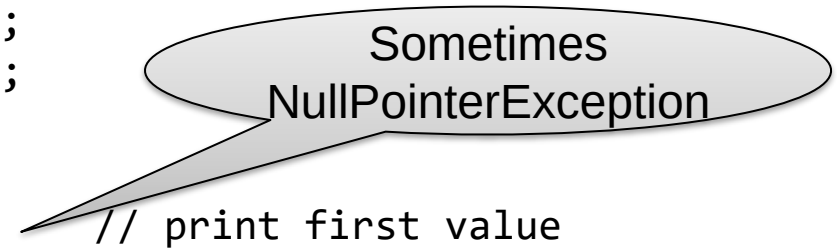
Race conditions

Bad example

```
class Element {
    int value;
    Element next = null;
    public Element(int v) { this.value=v; }
}
class List {
    Element head = null;
    void add(int value) {
        Element newElement=new Element(value);
        newElement.next=head;
        head=newElement;
    }
}
public static void main(String[] args) throws InterruptedException {
    List list=new List();
    Thread t1=new Thread(() -> list.add(3));
    Thread t2=new Thread(() -> list.add(4));
    t1.start(); t2.start();
    t1.join(); t2.join();
    System.out.println(list.head.value);      // print first value
    System.out.println(list.head.next.value); // print second value
}
```



Two threads working in parallel

- What is happening in the example with the List?
- After we have created the list, we have a list object with head=null:

```
list = {
    Element head = null;
}
```

- Both threads try to add a new element to the list:

Thread 1:

```
Element newElement=new Element(3);
newElement.next=head;
head=newElement;
```

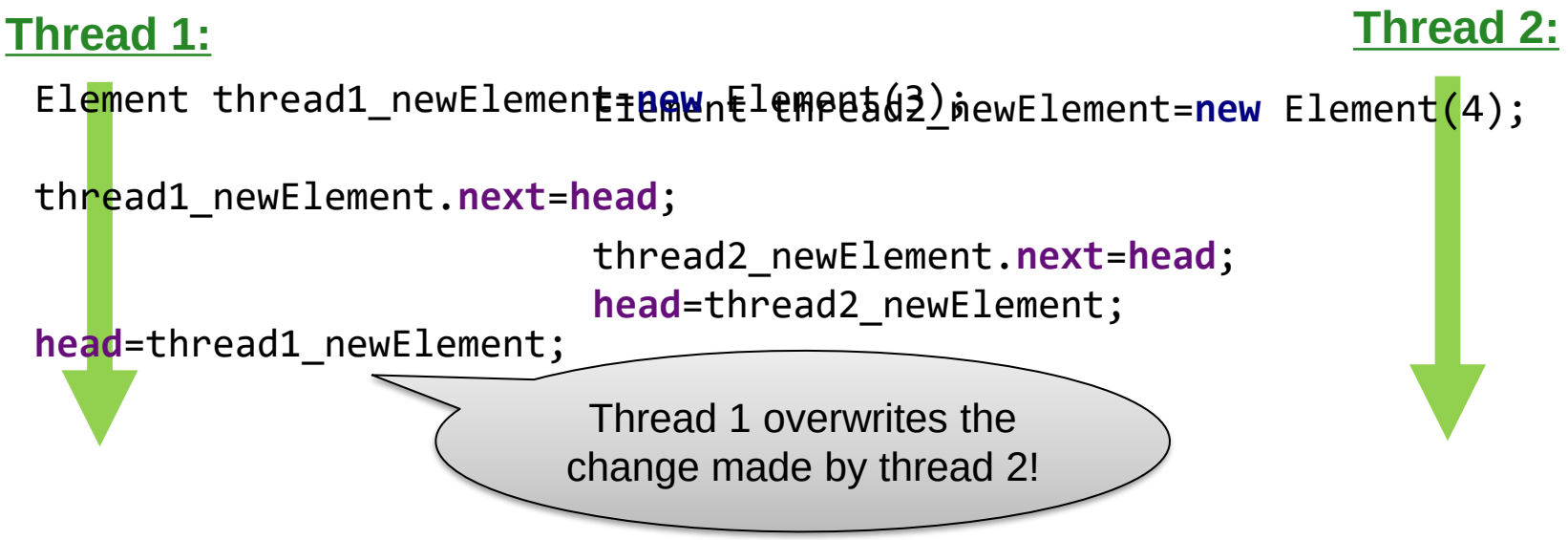
Thread 2:

```
Element newElement=new Element(4);
newElement.next=head;
head=newElement;
```

- What will happen?

Two threads working in parallel (2)

- Neither Java nor the operating system give any guarantees in what order the two threads are executed
- Since both threads are running in parallel, it can happen that the order of execution is overlapping or interleaved (*fr*. entrelacé). Example:



- Strange things happen if two threads work with the list head at the same time. This is called a *race condition*.

Two threads working in parallel (3)

- Of course, the same problem can also appear if the two threads call different methods:

```
class List {
    Element head;

    void add(int value) {
        Element newElement=new Element(value);
        newElement.next=head;
        head=newElement;
    }

    void remove() {
        if(head!=null) {
            head=head.next;
        }
    }
}
```

- Imagine what could happen if the add-method and the remove-method are executed in parallel by two threads

Another bad example

```
public class IncrementCounter {
    private int counter=0;       // both threads use the same counter

    private void increment() {
        for(int i=0;i<10000;i++) {
            counter++;
        }
    }

    public void test() throws InterruptedException {
        Thread t1=new Thread(()->increment());
        Thread t2=new Thread(()->increment());
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(counter);
    }

    public static void main(String[] args) throws InterruptedException {
        new IncrementCounter().test();
    }
}
```

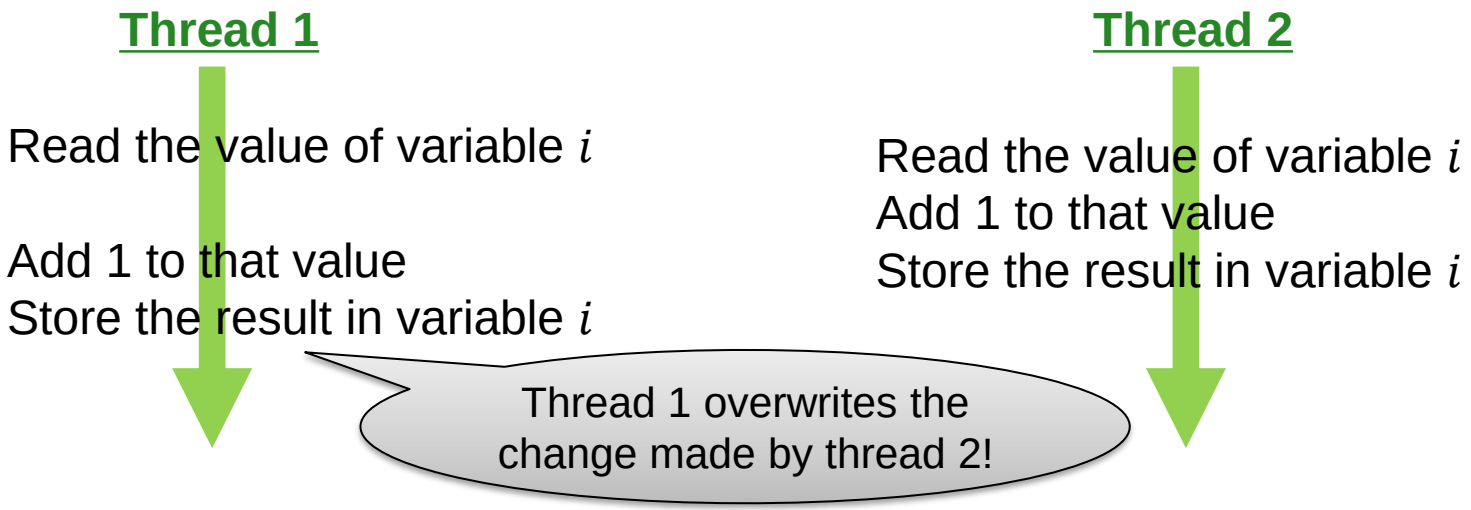
Race condition

- Be careful: Race conditions can even happen in a single line of code
- A line like

  i = i + 1; ("Bad example 1" from last week)

  consists of three low-level instructions for your computer:

  1. Read the value of variable i
  2. Add 1 to that value
  3. Store the result in variable i
- With two threads, the following can happen:



Monitors

Monitor

- We must prevent that a thread changes a variable or an object while another thread tries to use (or change) it
- In Java, every object can have a *monitor*. A monitor helps to prevent that threads execute a given section of code at the same time.
- Example:

```
synchronized(list) {

    list.add(3);

}
```

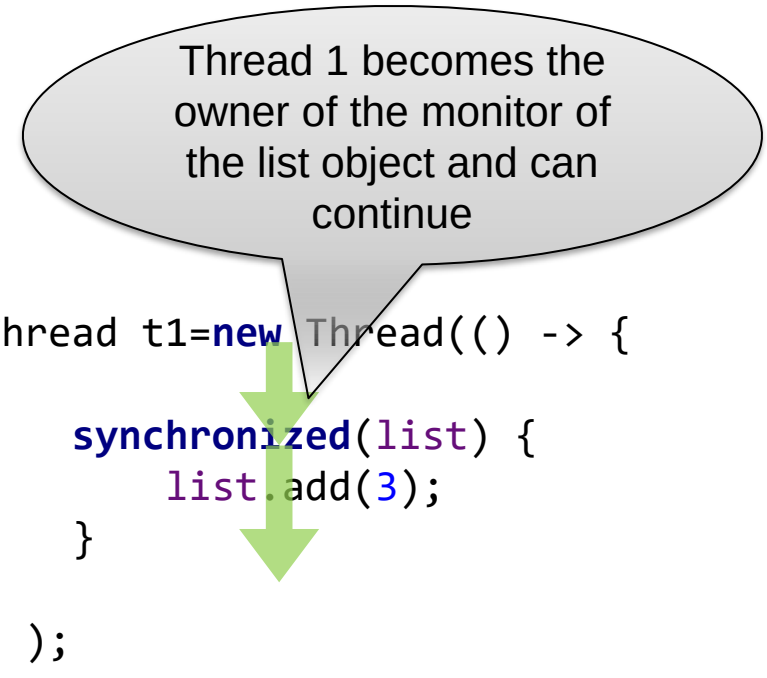
With the synchronized statement, a thread becomes the owner of the monitor of the list object. The thread can enter the block.

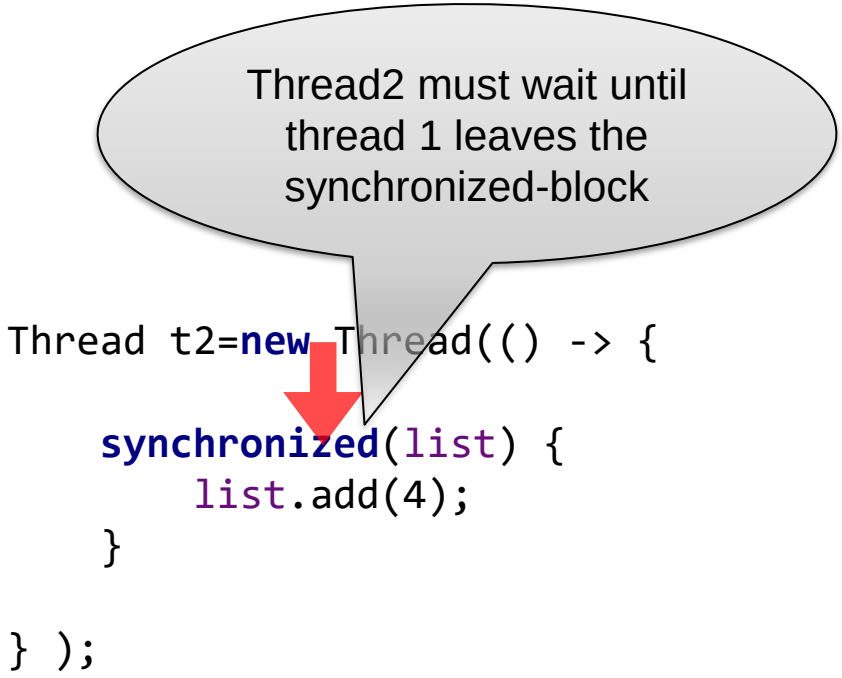
Only one thread can be owner of the monitor of an object at a given time. If another thread wants the monitor, it must wait until the monitor is free.

The block inside the synchronized statement is called a *critical section*.

At the end of the synchronized statement, the owner of the monitor releases it. A waiting thread can now get the monitor.

Synchronized execution





- Note: In this example, we have assumed that thread 1 first enters the critical section. It can also happen that thread 2 enters first. Then thread 1 would have to wait.

Objects for monitors

- Threads can use *any* object's monitor for synchronization. It can be even an object specifically created for that purpose. Of course, both threads must use the same object to synchronize:

```
List list=new List();
Object someObjectForSynchronization=new Object()

Thread t1=new Thread(() -> {
    synchronized(someObjectForSynchronization) {
        list.add(3);
    }
} );

Thread t2=new Thread(() -> {
    synchronized(someObjectForSynchronization) {
        list.add(4);
    }
} );
```

Where to put the synchronized statement

- Instead of using a synchronized statement at every caller of the add method, it's easier to put it directly inside the add method:

```
void add(int value) {
    Element newElement=new Element(value);
    synchronized(someObjectForSynchronization) {
        newElement.next=head;
        head=newElement;
    }
}
```

- Often, people simply use the object of the method for the synchronization:

```
void add(int value) {
    Element newElement=new Element(value);
    synchronized(this) {
        newElement.next=head;
        head=newElement;
    }
}
```

Synchronized method

- It's also possible to mark the entire method as "synchronized":

```
synchronized void add(int value) {
   Element newElement=new Element(value);
   newElement.next=head;
   head=newElement;
}
```

- That's (mostly) equivalent to:

```
void add(int value) {
   synchronized(this) {
       Element newElement=new Element(value);
       newElement.next=head;
       head=newElement;
   }
}
```

can be moved outside the critical section

- In a synchronized method, the *entire* method body is synchronized. This is often useful, but in our example it's not necessary to put the Element construction inside the critical section
- Only use synchronization where needed! If *everything* is synchronized, why using threads?

Classes in java.util.*

- Most data structures in java.util.* are not *thread-safe*: race conditions can happen!
  - ArrayList, LinkedList, HashSet, PriorityQueue, HashMap,...
- If you want to work with these classes from multiple threads, you have to use synchronized-statements in your code
- But there already a lot of helper classes and methods that you can use:

```
// creates a thread-safe map
Map m = Collections.synchronizedMap(new HashMap(...));


// creates a thread-safe list
List list = Collections.synchronizedList(new LinkedList(...));
```

- There are many other methods to create thread-safe sets, queues, etc.

How does `Collections.synchronizedList` work?

- The method `synchronizedList` in

  `List list = Collections.synchronizedList(new LinkedList(...));`

  returns an object of type SynchronizedList
- SynchronizedList is a *wrapper class* (a design pattern!). It doesn't contain any data. It just wraps thread-safe methods "around" a normal list object:

```
class SynchronizedList {
    final List list;
    final Object mutex = new Object();

    SynchronizedList(List list) {
        this.list = list;
    }

    public void add(int index, E element) {
        synchronized (mutex) {
           list.add(index, element);
        }
    }
    ...
}
```