


Computer science → Programming languages → Java → Working with data → Multithreading → Synchronization

Shared data

Theory

Practice

 0% completed, 0 problems solved

▼

Theory

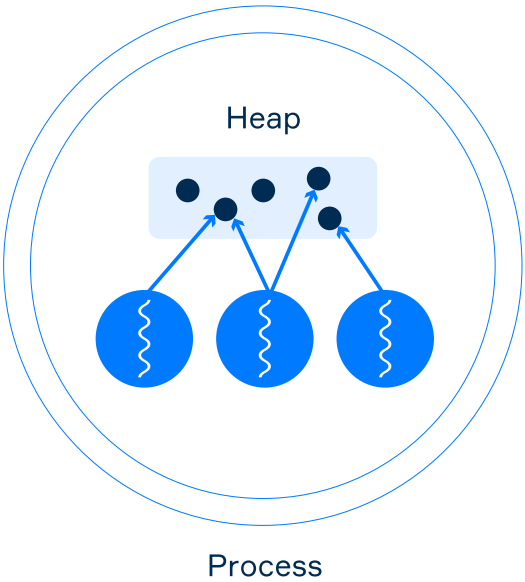
🕒 14 minutes reading

Verify to skip

Start practicing

§1. Sharing data between threads

Threads that belong to the same process share the common memory (that is called **Heap**). They may communicate by using shared data in memory. To be able to access the same data from multiple threads, each thread must have a reference to this data (by an object). The picture below demonstrates the idea.



Multiple threads of a single process have references to objects in the Heap

Let's consider an example. Here's a class named `Counter`.


```
1 class Counter {
2
3     private int value = 0;
4
5     public void increment() {
6         value++;
7     }
8
9     public int getValue() {
10
11         return value;
12     }
13 }
```

The class has two methods: `increment` and `getValue`. Each invocation of `increment` adds 1 to the field `value` and calling `getValue` returns the current value of the field.


And here's a class that extends `Thread`:

```
1 class MyThread extends Thread {
2
3     private final Counter counter;
4
5     public MyThread(Counter counter) {
6         this.counter = counter;
```

1 required topic

 `Exceptions in threads` ▼

2 dependent topics

 `Thread synchronization` ▼

`ThreadLocal` ▼

```
7     }
8
9     @Override
10
11     public void run() {
12
13         counter.increment();
14
15     }
16
17 }
```

The constructor of `MyThread` takes an instance of `Counter` and stores it to the field. The method `run` invokes the method `increment` of the `counter` object.

Let's now create an instance of `Counter` and two instances of `MyThread`. Both instances of `MyThread` have the same reference to the `counter` object.

```
1 Counter counter = new Counter();
2
3 MyThread thread1 = new MyThread(counter);
4 MyThread thread2 = new MyThread(counter);
```

Now let's see what happens when we start both threads one by one and print out the result of `counter.getValue()`.

```
1 thread1.start(); // start the first thread
2 thread1.join();  // wait for the first thread
3
4 thread2.start(); // start the second thread
5 thread2.join();  // wait for the second thread
6
7 System.out.println(counter.getValue()); // it prints 2
```

As you can see if you try it by yourself the result is 2, because both threads work with the same data by using a reference.

In this example, we started the first thread and waited until it has completed its work (by this time an increment happened), then we started the second thread and waited till it also has completed its work (increment's happened again). The result is exactly as we would've expected.

When you write your code in different threads that work with the same data concurrently, it is important to understand a few things:

- some operations are non-atomic;
- changes of a variable performed by one thread may be invisible to the other threads;
- if changes are visible, their order might not be (reordering).

Let's now learn more about those.

§2. Thread interference

A non-atomic operation is an operation that consists of multiple steps. A thread may operate on an intermediate value of non-atomic operation performed by another thread. This leads to a problem called **thread interference**: the sequences of steps of non-atomic operations performed by several threads may overlap.

Let's start by explaining why **increment** is a non-atomic operation and how exactly it works. As an example, consider the class `Counter` again.

```
1 class Counter {
2
3     private int value = 0;
```

```
4
5     public void increment() {
6         value++; // the same thing as value = value + 1
7     }
8
9     public int getValue() {
10
11         return value;
12     }
13 }
```

Note: in the previous example, the two threads did not work with the data at the same time. Before the start of the second thread, the first has already terminated.

The operation `value++` can be decomposed into three steps:

1. read the current value;
2. increment the value by 1;
3. write the incremented value back in the field;

Since the increment operation is non-atomic and takes 3 steps to work the **thread interference** may occur in case two threads call the method `increment` of the same instance of `Counter`,

In the same way, the operation `value--` may be decomposed into three steps.

Suppose that we have an instance of the `Counter` class:

```
1 Counter counter = new Counter();
```

The initial value of the field is 0.

Now if `Thread A` invokes the method `increment` of this instance and `Thread B` also invokes the method at the same time, the following happens:

1. **Thread A:** read value from the variable.
2. **Thread A:** increment the read value by 1.
3. **Thread B:** read value from the variable (it reads an intermediate value 0).
4. **Thread A:** write the result in the variable (now, the current value of the field is 1).
5. **Thread B:** increment the read value by 1.
6. **Thread B:** write the result in the variable (now, the current value of the field is 1).

In this case after calling the method `increment` from two threads we may obtain the unexpected result (1 instead of 2). That means that the result of `Thread A` was lost, overwritten by `Thread B`. Although sometimes the result may be correct, this particular interleaving is possible.

We've just seen how increment and decrement are non-atomic operations. In this topic, we will not discuss how this problem may be solved, just keep it in mind for now.

Let's consider another case: an assignment of 64-bit values. It may be surprising, but even the reading and writing fields of `double` and `long` types (64-bits) may not be atomic on some platforms.

```
1 class MyClass {
2
3     long longVal; // reading and writing may be not atomic
4
5     double doubleVal; // reading and writing may be not atomic
6 }
```

It means while a thread writes a value to a variable, another thread can access an intermediate result (for example, only 32 written bits). To make these operations atomic, fields should be declared using the `volatile` keyword.

```
1  class MyClass {
2
3      volatile long longVal; // reading and writing are atomic now
4
5      volatile double doubleVal; // reading and writing are atomic now
6  }
```

The reading of and writing to the fields of other primitive types (boolean, byte, short, int, char, float) are guaranteed to be **atomic**.

In large applications, **thread interference** bugs can be difficult to detect.

§3. Visibility between threads

Sometimes, when a thread changes shared data, another thread may not notice these changes or obtain them in a different order. It means different threads may have inconsistent views of the same data.

The reasons are different, including caching values for threads, compiler optimization, and more. Fortunately, most programmers do not need a detailed understanding of the causes. All that is needed is a strategy for avoiding them in the first place.

Example. Here's an `int` field, defined and initialized:

```
1  int number = 0;
```

The field is shared between two threads: `Thread A` and `Thread B`.

`Thread A` increments the `number` by 5.

```
1  number += 5;
```

Right after it, `Thread B` prints `number` in the standard output:

```
1  System.out.println(number);
```

The output may be either 0 or 5, because there is no guarantee that the change performed by `Thread A` is visible to `Thread B`.

As we've already mentioned, the keyword `volatile` is used for **visibility**. To make visible changes of a value made by one thread to other threads, we should declare the field with the keyword `volatile`.

```
1  volatile int number = 0;
```

When the field is declared as **volatile** all changes made to this field by a thread are guaranteed to be visible for another thread when it's reading the value from this field.

The `volatile` keyword may be written in an instance and static fields declaration.

§4. Other cases of visibility

Sometimes we don't need to write the `volatile` keyword. The following procedures will also guarantee visibility:

- changes of variables performed by a thread **before starting** a new thread are always visible to the new thread;
- changes of variables inside a thread are always visible to any other threads after it successfully returns from a `join` on the thread (we used this one at the beginning of this topic).

We will not consider all existing ways to guarantee visibility now. They are formalized using a special relationship named **"Happens-before"**. For now, keep in mind the use of the `volatile` and two cases above.

§5. More on volatile keyword

Again, the `volatile` keyword allows us to make visible changes of a field made by one thread to other threads. This keyword also makes writing to `double` and `long` fields atomic. But the keyword doesn't make the increment/decrement and similar operations atomic.

In fact, there's more abstract and complex things about `volatile` , but we'll skip this information for now.

 Report a typo

302 users liked this piece of theory. 6 didn't like it. What about you?



Start practicing

Verify to skip

Table of contents:

- [↑ Shared data](#)
- [§1. Sharing data between threads](#)
- [§2. Thread interference](#)
- [§3. Visibility between threads](#)
- [§4. Other cases of visibility](#)
- [§5. More on volatile keyword](#)
- [Discussion](#)


Comments (28)

Useful links (4)

Show discussion



- Tracks
- Pricing
- For organizations
- About
- Contribute
- Careers

 Become beta tester

Be the first to see what's new

Terms

Support





Made with  by Hyperskill and JetBrains