JAVA SE

Quiz yourself: Happens-before thread synchronization in Java with CyclicBarrier

The CyclicBarrier class provides timing synchronization among threads, while also ensuring that data written by those threads prior to the synchronization is visible among those threads.

by Simon Roberts and Mikalai Zaikin

September 13, 2021

Given the following `CBTest` class

```
import static java.lang.System.out;
public class CBTest {

    private List<Integer> results =
      Collections.synchronizedList(new ArrayList<>

    class Calculator extends Thread {
        CyclicBarrier cb;
        int param;
        Calculator(CyclicBarrier cb, int param) {
            this.cb = cb;
            this.param = param;
        }

        public void run() {
            try {
                results.add(param * param);
                cb.await();
            } catch (Exception e) {
            }
        }
    }

    void doCalculation() {
        // add your code here
    }

    public static void main(String[] args) {
        new CBTest().doCalculation();
    }
}
```

Which code fragment, when added to the doCalculation method independently, will make the code reliably print 13 to the console?
Choose one.

A.

```
CyclicBarrier cb = new CyclicBarrier(2, () -> {
    out.print(results.stream().mapToInt(v ->  v.ir
    });
new Calculator(cb, 2).start();
new Calculator(cb, 3).start();
```

The answer is A.

B.

```
CyclicBarrier cb = new CyclicBarrier(2);
out.print(results.stream().mapToInt(v -> v.intValu
new Calculator(cb, 2).start();
new Calculator(cb, 3).start();
```

The answer is B.

C.

```
CyclicBarrier cb = new CyclicBarrier(3);
new Calculator(cb, 2).start();
new Calculator(cb, 3).start();
cb.await();
out.print(results.stream().mapToInt(v -> v.intValu
```

The answer is C.

D.

```
CyclicBarrier cb = new CyclicBarrier(2);
new Calculator(cb, 2).start();
new Calculator(cb, 3).start();
out.print(results.stream().mapToInt(v -> v.intValu
```

The answer is D.

Answer. The `CyclicBarrier` class is a feature of the `java.util.concurrent` package, and it provides timing synchronization among threads while also ensuring that data written by those threads prior to the synchronization is visible among those

threads (this is the so-called “happens-before” relationship). These problems might otherwise be addressed using the `synchronized`, `wait`, and `notify` mechanisms, but those are generally considered low-level mechanisms that are harder to use correctly.

If multiple threads are cooperating on a task, at least two problems must commonly be addressed.

- The data written by one thread must be read correctly by another thread when the data is needed.
- The other thread must have an efficient means of knowing when the necessary data has been prepared and is ready to be read.

The `CyclicBarrier` addresses these problems by providing timing synchronization using a *barrier point* and a *barrier action*.

The operation of the `CyclicBarrier` might be likened to a group of colleagues at a conference preparing to go to a presentation together. They get up in the morning and go about their routines individually, getting ready for the presentation and their day. When they’re ready, they go to the lobby of the hotel they’re staying in and wait for the others. When all the colleagues are in the lobby, they all leave at once to walk over to the conference room.

Similarly, a `CyclicBarrier` is constructed with a count of “parties” as an argument. In the analogy, this represents the number of colleagues who plan to go to the presentation. In the real system, this is the number of threads that need to synchronize their activities.

When a thread is ready, it calls the `await()` method on the `CyclicBarrier` (in the analogy, this is arriving in the lobby). At this point, one of two behaviors occurs. Suppose the `CyclicBarrier` was constructed with a “parties” count of `3`. The first and second threads that call `await()` will be *blocked*, meaning their execution is suspended, using no CPU time, until some other occurrence causes the blocking to end. When the third thread calls `await()`, the blocking of the two threads that called `await()` before is ended, and all three threads are permitted to continue execution. (This is the second behavior mentioned earlier.)

After the `CyclicBarrier` thread-execution block is ended, data written by *any* of the threads prior to calling `await()` will be visible (unless it is perhaps subsequently altered, which can confuse the issue) by all the threads that called `await()` on this blocking cycle of this `CyclicBarrier`. This is the *happens-before* relationship, and it addresses the visibility problem.

After the threads are released, the `CyclicBarrier` can be reused for another synchronizing operation—that’s why the class has *cyclic* in the name. Note that some of the other synchronization tools in the `java.util.concurrent` API cannot be reused in this way.

The `CyclicBarrier` provides two constructors. Both require the number of parties (threads) they are to control, but the second also introduces the barrier action.

```
public CyclicBarrier(int parties, Runnable barrier
```

The barrier action defines an action that is executed when the barrier is tripped, that is, when the last thread enters the barrier. This barrier action will be able to see the data written by the awaiting threads, and any data written by the barrier action will be visible to the threads after they resume.

The API documentation states, “Memory consistency effects: Actions in a thread prior to calling `await()` happen-before actions that are part of the barrier action, which in turn happen-before actions following a successful return from the corresponding `await()` in other threads.”

Each of the quiz options creates a `CyclicBarrier` and passes it to a thread (created from the `Calculator` class). Each thread—or each calculator, if you prefer—performs a calculation and then adds the result of that calculation to a thread-safe `List` that’s shared between the two threads. (Note that the `ArrayList` itself isn’t thread-safe, but the `Collections.synchronizedList` method creates a thread-safe wrapper around it.)

After adding the result to the `List`, the calculator thread calls the `await()` method on the `CyclicBarrier`. Subsequently, the intention is to pick up the data items that have been added to the `List` and print the sum.

For this to work correctly, the summing operation must see *all* the data written by the calculators—and that must not occur until after the calculated values have been written. The code should achieve this using the `CyclicBarrier`.

In option B, the attempt to calculate the sum and print the result precedes the construction and start of the two calculator threads. As a result, option B is incorrect.

Option B might *occasionally* print the right answer; the situation is what’s called a *race condition*. Although unlikely, it’s not impossible that the JVM might happen to schedule its threads in a way that the calculations are completed before the summing and printing starts. It’s also possible that the data written in this situation might become visible to the thread that performs the summing and printing. However, such circumstances are unlikely at best and certainly not reliable. With option B, the output is most likely to be `0` (because the list is empty), but the values `4`, `9`, and `13` are all possible. There’s no way to predict what the results will be.

Option D is a variation of option B with swapped lines. Although this looks like the calculations might be executed before the summing and printing operations, the same race-condition uncertainty exists. So, although this option is more likely than option B to print `13` on any given run, for the same reasons, all the values are possible. Therefore, option D is incorrect.

Option C is almost correct but not quite. The `CyclicBarrier` is created with a `parties` count of `3`. Three calls to `await()` are made, so the main thread would not proceed until the two calculations are complete. The timing would be correct, and the visibility issue would be correctly addressed such that the last line of the option—the line that computes and prints the sum—would work reliably if it were not for one remaining problem with the implementation shown.

Blocking behaviors in the Java APIs are generally interruptible, and if they are interrupted, they break out of their blocked state and throw an `InterruptedException`, which is a checked exception. Because neither the code of option C nor the body of the `doCalculation` method into which the code is inserted includes code to address this exception, the code for option C fails to compile.

In fact, the `await()` method throws another checked exception, `BrokenBarrierException`, and this is also unhandled. However, while you might not know about the `BrokenBarrierException`, you should know about the `InterruptedException` because it's a fundamental and pervasive feature of Java's thread-management model. Because these checked exceptions are unhandled and the code does not compile, option C is incorrect.

In option A, the `CyclicBarrier` is created with a `parties` count of `2` (which will be the two calculator threads) and a barrier action. The barrier action is the lambda expression that aggregates results from working parties. The two `Calculator` threads invoke `await()` after they have written the result of their calculations to the list. This behavior ensures that both writes have occurred before—and the data is visible to—the barrier action. Then, the barrier action is invoked to perform the summing and printing. As a result, it's guaranteed that both `4` and `9` are in the list before the summing and printing *and* that they are visible to that operation. Consequently, the output must be `13`, and you know that option A is correct.

Conclusion. The correct answer is option A.

Related quizzes

- Quiz yourself: Rules about throwing checked exceptions in Java
- Quiz yourself: Using core functional interfaces
- Quiz yourself: Create worker threads using Runnable and Callable

---

Simon Roberts

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

Mikalai Zaikin

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous *Sun*

Certified Programmer for Java study guides by Kathy Sierra and Bert Bates.

Share this Page

Contact
US Sales: +1.800.633.0738
Global Contacts
Support Directory
Subscribe to Emails

About Us
Careers
Communities
Company Information
Social Responsibility Emails

Downloads and Trials
Java for Developers
Java Runtime Download
Software Downloads
Try Oracle Cloud

News and Events
Acquisitions
Blogs
Events
Newsroom

ORACLE | Integrated Cloud
Applications & Platform Services

© Oracle | Site Map | Terms of Use & Privacy | Cookie Preferences | Ad Choices