

CHAPTER 2



Synchronization

Developing multithreaded applications is much easier when threads don't interact, typically via shared variables. When interaction occurs, various problems can arise that make an application *thread-unsafe* (incorrect in a multithreaded context). In this chapter, you'll learn about these problems and also learn how to overcome them through the correct use of Java's synchronization-oriented language features.

The Problems with Threads

Java's support for threads facilitates the development of responsive and scalable applications. However, this support comes at the price of increased complexity. Without care, your code can become riddled with hard-to-find bugs related to race conditions, data races, and cached variables.

Race Conditions

A *race condition* occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the scheduler. Consider the following code fragment, which performs a computation as long as a certain precondition holds:

```
if (a == 10.0)
    b = a / 2.0;
```

There is no problem with this code fragment in a single-threaded context, and there is no problem in a multithreaded context when *a* and *b* are local variables. However, assume that *a* and *b* identify instance or class (static) field variables and that two threads simultaneously access this code.

Suppose that one thread has executed `if (a == 10.0)` and is about to execute `b = a / 2.0` when suspended by the scheduler, which resumes another thread that changes *a*. Variable *b* will not equal 5.0 when the former thread resumes its execution. (If *a* and *b* were local variables, this race condition wouldn't occur because each thread would have its own copy of these local variables.)

The code fragment is an example of a common type of race condition that's known as *check-then-act*, in which a potentially stale observation is used to decide on what to do next. In the previous code fragment, the “check” is performed by `if (a == 10.0)` and the “act” is performed by `b = a / 2.0;`

Another type of race condition is *read-modify-write*, in which new state is derived from previous state. The previous state is read, then modified, and finally updated to reflect the modified result via three indivisible operations. However, the combination of these operations isn't indivisible.

A common example of read-modify-write involves a variable that's incremented to generate a unique numeric identifier. For example, in the following code fragment, suppose that `counter` is an instance field of type `int` (initialized to 1) and that two threads simultaneously access this code:

```
public int getID()
{
    return counter++;
}
```

Although it might look like a single operation, expression `counter++` is actually three separate operations: read `counter`'s value, add 1 to this value, and store the updated value in `counter`. The read value becomes the value of the expression.

Suppose thread 1 calls `getID()` and reads `counter`'s value, which happens to be 1, before it's suspended by the scheduler. Now suppose that thread 2 runs, calls `getID()`, reads `counter`'s value (1), adds 1 to this value, stores the result (2) in `counter`, and returns 1 to the caller.

At this point, assume that thread 2 resumes, adds 1 to the previously read value (1), stores the result (2) in `counter`, and returns 1 to the caller. Because thread 1 undoes thread 2, we have lost an increment and a non-unique ID has been generated. This method is useless.

Data Races

A race condition is often confused with a *data race* in which two or more threads (in a single application) access the same memory location concurrently, at least one of the accesses is for writing, and these threads don't coordinate their accesses to that memory. When these conditions hold, access order is non-deterministic. Different results may be generated from run to run, depending on that order. Consider the following example:

```
private static Parser parser;

public static Parser getInstance()
{
    if (parser == null)
        parser = new Parser();
    return parser;
}
```

Assume that thread 1 invokes `getInstance()` first. Because it observes a null value in the `parser` field, thread 1 instantiates `Parser` and assigns its reference to `parser`. When thread 2 subsequently calls `getInstance()`, it could observe that `parser` contains a non-null reference and simply return `parser`'s value. Alternatively, thread 2 could observe a null value in `parser` and create a new `Parser` object. Because there is no *happens-before ordering* (one action must precede another action) between thread 1's write of `parser` and thread 2's read of `parser` (because there is no coordinated access to `parser`), a data race has occurred.

Cached Variables

To boost performance, the compiler, the Java virtual machine (JVM), and the operating system can collaborate to cache a variable in a register or a processor-local cache, rather than rely on main memory. Each thread has its own copy of the variable. When one thread writes to this variable, it's writing to its copy; other threads are unlikely to see the update in their copies.

Chapter 1 presented a `ThreadDemo` application (see Listing 1-3) that exhibits this problem. For reference, I repeat part of the source code here:

```
private static BigDecimal result;

public static void main(String[] args)
{
    Runnable r = () ->
        {
            result = computePi(50000);
        };
    Thread t = new Thread(r);
    t.start();
    try
    {
        t.join();
    }
    catch (InterruptedException ie)
    {
        // Should never arrive here because interrupt() is never
        // called.
    }
    System.out.println(result);
}
```

The class field named `result` demonstrates the cached variable problem. This field is accessed by a worker thread that executes `result = computePi(50000)`; in a lambda context, and by the default main thread when it executes `System.out.println(result)`.

The worker thread could store `computePi()`'s return value in its copy of `result`, whereas the default main thread could print the value of its copy. The default main thread might not see the `result = computePi(50000)`; assignment and its copy would remain at the null default. This value would output instead of `result`'s string representation (the computed pi value).

Synchronizing Access to Critical Sections

You can use synchronization to solve the previous thread problems. *Synchronization* is a JVM feature that ensures that two or more concurrent threads don't simultaneously execute a *critical section*, which is a code section that must be accessed in a *serial* (one thread at a time) manner.

This property of synchronization is known as *mutual exclusion* because each thread is mutually excluded from executing in a critical section when another thread is inside the critical section. For this reason, the lock that the thread acquires is often referred to as a *mutex lock*.

Synchronization also exhibits the property of *visibility* in which it ensures that a thread executing in a critical section always sees the most recent changes to shared variables. It reads these variables from main memory on entry to the critical section and writes their values to main memory on exit.

Synchronization is implemented in terms of *monitors*, which are concurrency constructs for controlling access to critical sections, which must execute indivisibly. Each Java object is associated with a monitor, which a thread can *lock* or *unlock* by acquiring and releasing the monitor's *lock* (a token).

■ **Note** A thread that has acquired a lock doesn't release this lock when it calls one of Thread's `sleep()` methods.

Only one thread can hold a monitor's lock. Any other thread trying to lock that monitor blocks until it can obtain the lock. When a thread exits a critical section, it unlocks the monitor by releasing the lock.

Locks are designed to be reentrant to prevent deadlock (discussed later). When a thread attempts to acquire a lock that it's already holding, the request succeeds.

■ **Tip** The `java.lang.Thread` class declares a static `boolean holdsLock(Object o)` method that returns `true` when the calling thread holds the lock on object `o`. You will find this method handy in assertion statements, such as `assert Thread.holdsLock(o);`

Java provides the `synchronized` keyword to serialize thread access to a method or a block of statements (the critical section).

Using Synchronized Methods

A *synchronized method* includes the `synchronized` keyword in its header. For example, you can use this keyword to synchronize the former `getID()` method and overcome its read-modify-write race condition as follows:

```
public synchronized int getID()
{
    return counter++;
}
```

When synchronizing on an instance method, the lock is associated with the object on which the method is called. For example, consider the following `ID` class:

```
public class ID
{
    private int counter; // initialized to 0 by default

    public synchronized int getID()
    {
        return counter++;
    }
}
```

Suppose you specify the following code sequence:

```
ID id = new ID();
System.out.println(id.getID());
```

The lock is associated with the `ID` object whose reference is stored in `id`. If another thread called `id.getID()` while this method was executing, the other thread would have to wait until the executing thread released the lock.

When synchronizing on a class method, the lock is associated with the `java.lang.Class` object corresponding to the class whose class method is called. For example, consider the following `ID` class:

```
public class ID
{
    private static int counter; // initialized to 0 by default

    public static synchronized int getID()
    {
        return counter++;
    }
}
```

Suppose you specify the following code sequence:

```
System.out.println(ID.getID());
```

The lock is associated with `ID.class`, the `Class` object associated with `ID`. If another thread called `ID.getID()` while this method was executing, the other thread would have to wait until the executing thread released the lock.

Using Synchronized Blocks

A *synchronized block* of statements is prefixed by a header that identifies the object whose lock is to be acquired. It has the following syntax:

```
synchronized(object)
{
    /* statements */
}
```

According to this syntax, *object* is an arbitrary object reference. The lock is associated with this object.

I previously excerpted a Chapter 1 application that suffers from the cached variable problem. You can solve this problem with two synchronized blocks:

```
Runnable r = () ->
{
    synchronized(FOUR)
    {
        result = computePi(50000);
    }
};
// ...
synchronized(FOUR)
{
    System.out.println(result);
}
```

These two blocks identify a pair of critical sections. Each block is guarded by the same object so that only one thread can execute in one of these blocks at a time. Each thread must acquire the lock associated with the object referenced by constant `FOUR` before it can enter its critical section.

This code fragment brings up an important point about synchronized blocks and synchronized methods. Two or more threads that access the same code sequence *must* acquire the same lock or there will be no synchronization. This implies that the same object must be accessed. In the previous example, `FOUR` is specified in two places so that only one thread can be in either critical section. If I specified `synchronized(FOUR)` in one place and `synchronized("ABC")` in another, there would be no synchronization because two different locks would be involved.

Beware of Liveness Problems

The term *liveness* refers to something beneficial happening eventually. A liveness failure occurs when an application reaches a state in which it can make no further progress. In a single-threaded application, an infinite loop would be an example. Multithreaded applications face the additional liveness challenges of deadlock, livelock, and starvation:

- *Deadlock*: Thread 1 waits for a resource that thread 2 is holding exclusively and thread 2 is waiting for a resource that thread 1 is holding exclusively. Neither thread can make progress.
- *Livelock*: Thread *x* keeps retrying an operation that will always fail. It cannot make progress for this reason.
- *Starvation*: Thread *x* is continually denied (by the scheduler) access to a needed resource in order to make progress. Perhaps the scheduler executes higher-priority threads before lower-priority threads and there is always a higher-priority thread available for execution. Starvation is also commonly referred to as *indefinite postponement*.

Consider deadlock. This pathological problem occurs because of too much synchronization via the `synchronized` keyword. If you're not careful, you might encounter a situation where locks are acquired by multiple threads, neither thread holds its own lock but holds the lock needed by some other thread, and neither thread can enter and later exit its critical section to release its held lock because another thread holds the lock to that critical section. Listing 2-1's atypical example demonstrates this scenario.

Listing 2-1. A Pathological Case of Deadlock

```
public class DeadlockDemo
{
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void instanceMethod1()
    {
        synchronized(lock1)
        {
            synchronized(lock2)
            {
                System.out.println("first thread in instanceMethod1");
                // critical section guarded first by
                // lock1 and then by lock2
            }
        }
    }
}
```

```

public void instanceMethod2()
{
    synchronized(lock2)
    {
        synchronized(lock1)
        {
            System.out.println("second thread in instanceMethod2");
            // critical section guarded first by
            // lock2 and then by lock1
        }
    }
}

public static void main(String[] args)
{
    final DeadlockDemo dld = new DeadlockDemo();
    Runnable r1 = new Runnable()
    {
        @Override
        public void run()
        {
            while(true)
            {
                dld.instanceMethod1();
                try
                {
                    Thread.sleep(50);
                }
                catch (InterruptedException ie)
                {
                }
            }
        }
    };
    Thread thdA = new Thread(r1);
    Runnable r2 = new Runnable()
    {
        @Override
        public void run()
        {
            while(true)
            {
                dld.instanceMethod2();
                try
                {
                    Thread.sleep(50);
                }
            }
        }
    };
}

```

```

        catch (InterruptedException ie)
        {
        }
    }
};
Thread thdB = new Thread(r2);
thdA.start();
thdB.start();
}
}
}

```

Listing 2-1's thread A and thread B call `instanceMethod1()` and `instanceMethod2()`, respectively, at different times. Consider the following execution sequence:

1. Thread A calls `instanceMethod1()`, obtains the lock assigned to the `lock1`-referenced object, and enters its outer critical section (but has not yet acquired the lock assigned to the `lock2`-referenced object).
2. Thread B calls `instanceMethod2()`, obtains the lock assigned to the `lock2`-referenced object, and enters its outer critical section (but has not yet acquired the lock assigned to the `lock1`-referenced object).
3. Thread A attempts to acquire the lock associated with `lock2`. The JVM forces the thread to wait outside of the inner critical section because thread B holds that lock.
4. Thread B attempts to acquire the lock associated with `lock1`. The JVM forces the thread to wait outside of the inner critical section because thread A holds that lock.
5. Neither thread can proceed because the other thread holds the needed lock. You have a deadlock situation and the program (at least in the context of the two threads) freezes up.

Compile Listing 2-1 as follows:

```
javac DeadlockDemo.java
```

Run the resulting application as follows:

```
java DeadlockDemo
```

You should observe interleaved first thread in `instanceMethod1` and second thread in `instanceMethod2` messages on the standard output stream until the application freezes up because of deadlock.

Although the previous example clearly identifies a deadlock state, it's often not that easy to detect deadlock. For example, your code might contain the following circular relationship among various classes (in several source files):

- Class A's synchronized method calls class B's synchronized method.
- Class B's synchronized method calls class C's synchronized method.
- Class C's synchronized method calls class A's synchronized method.

If thread A calls class A's synchronized method and thread B calls class C's synchronized method, thread B will block when it attempts to call class A's synchronized method and thread A is still inside of that method. Thread A will continue to execute until it calls class C's synchronized method, and then block. Deadlock is the result.

■ **Note** Neither the Java language nor the JVM provides a way to prevent deadlock, and so the burden falls on you. The simplest way to prevent deadlock is to avoid having either a synchronized method or a synchronized block call another synchronized method/block. Although this advice prevents deadlock from happening, it's impractical because one of your synchronized methods/blocks might need to call a synchronized method in a Java API, and the advice is overkill because the synchronized method/block being called might not call any other synchronized method/block, so deadlock would not occur.

Volatile and Final Variables

You previously learned that synchronization exhibits two properties: mutual exclusion and visibility. The `synchronized` keyword is associated with both properties. Java also provides a weaker form of synchronization involving visibility only, and associates only this property with the `volatile` keyword.

Suppose you design your own mechanism for stopping a thread (because you cannot use Thread's unsafe `stop()` methods for this task). Listing 2-2 presents the source code to a `ThreadStopping` application that shows how you might accomplish this task.

Listing 2-2. Attempting to Stop a Thread

```
public class ThreadStopping
{
    public static void main(String[] args)
    {
        class StoppableThread extends Thread
        {
            private boolean stopped; // defaults to false
```

```

@Override
public void run()
{
    while(!stopped)
        System.out.println("running");
}

void stopThread()
{
    stopped = true;
}
}
StoppableThread thd = new StoppableThread();
thd.start();
try
{
    Thread.sleep(1000); // sleep for 1 second
}
catch (InterruptedException ie)
{
}
thd.stopThread();
}
}

```

Listing 2-2's `main()` method declares a local class named `StoppableThread` that subclasses `Thread`. After instantiating `StoppableThread`, the default main thread starts the thread associated with this `Thread` object. It then sleeps for one second and calls `StoppableThread`'s `stop()` method before dying.

`StoppableThread` declares a `stopped` instance field variable that's initialized to `false`, a `stopThread()` method that sets this variable to `true`, and a `run()` method whose `while` loop checks `stopped` on each loop iteration to see if its value has changed to `true`.

Compile Listing 2-2 as follows:

```
javac ThreadStopping.java
```

Run the resulting application as follows:

```
java ThreadStopping
```

You should observe a sequence of running messages.

When you run this application on a single-processor/single-core machine, you'll probably observe the application stopping. You might not see this stoppage on a multiprocessor machine or a uniprocessor machine with multiple cores where each processor or core probably has its own cache with its own copy of `stopped`. When one thread modifies its copy of this field, the other thread's copy of `stopped` isn't changed.

You might decide to use the `synchronized` keyword to make sure that only the main memory copy of `stopped` is accessed. After some thought, you end up synchronizing access to a pair of critical sections in the source code that's presented in Listing 2-3.

Listing 2-3. Attempting to Stop a Thread via the `synchronized` Keyword

```
public class ThreadStopping
{
    public static void main(String[] args)
    {
        class StoppableThread extends Thread
        {
            private boolean stopped; // defaults to false

            @Override
            public void run()
            {
                synchronized(this)
                {
                    while(!stopped)
                        System.out.println("running");
                }
            }

            synchronized void stopThread()
            {
                stopped = true;
            }
        }
        StoppableThread thd = new StoppableThread();
        thd.start();
        try
        {
            Thread.sleep(1000); // sleep for 1 second
        }
        catch (InterruptedException ie)
        {
        }
        thd.stopThread();
    }
}
```

Listing 2-3 is a bad idea for two reasons. First, although you only need to solve the visibility problem, `synchronized` also solves the mutual exclusion problem (which isn't an issue in this application). More importantly, you've introduced a serious problem into the application.

You've correctly synchronized access to `stopped`, but take a closer look at the synchronized block in the `run()` method. Notice the `while` loop. This loop is unending because the thread executing the loop has acquired the lock to the current `StoppableThread` object (via `synchronized(this)`), and any attempt by the default main thread to call `stopThread()` on this object will cause the default main thread to block because the default main thread needs to acquire the same lock.

You can overcome this problem by using a local variable and assigning `stopped`'s value to this variable in a synchronized block, as follows:

```
public void run()
{
    boolean _stopped = false;
    while (!_stopped)
    {
        synchronized(this)
        {
            _stopped = stopped;
        }
        System.out.println("running");
    }
}
```

However, this solution is messy and wasteful because there is a performance cost (which is not as great as it used to be) when attempting to acquire the lock, and this task is being done for every loop iteration. Listing 2-4 reveals a more efficient and cleaner approach.

Listing 2-4. Attempting to Stop a Thread via the `volatile` Keyword

```
public class ThreadStopping
{
    public static void main(String[] args)
    {
        class StoppableThread extends Thread
        {
            private volatile boolean stopped; // defaults to false

            @Override
            public void run()
            {
                while(!stopped)
                    System.out.println("running");
            }

            void stopThread()
            {
                stopped = true;
            }
        }
    }
}
```

```

    StoppableThread thd = new StoppableThread();
    thd.start();
    try
    {
        Thread.sleep(1000); // sleep for 1 second
    }
    catch (InterruptedException ie)
    {
    }
    thd.stopThread();
}
}
}

```

Because `stopped` has been marked `volatile`, each thread will access the main memory copy of this variable and not access a cached copy. The application will stop, even on a multiprocessor-based or a multicore-based machine.

■ **Caution** Use `volatile` only where visibility is an issue. Also, you can only use this reserved word in the context of field declarations (you'll receive an error if you try to make a local variable `volatile`). Finally, you can declare `double` and `long` fields `volatile`, but should avoid doing so on 32-bit JVMs because it takes two operations to access a `double` or `long` variable's value, and mutual exclusion (via `synchronized`) is required to access their values safely.

When a field variable is declared `volatile`, it cannot also be declared `final`. However, this isn't a problem because Java also lets you safely access a `final` field without the need for synchronization. To overcome the cached variable problem in `DeadlockDemo`, I marked both `lock1` and `lock2` `final`, although I could have marked them `volatile`.

You will often use `final` to help ensure thread safety in the context of an *immutable* (unchangeable) class. Consider Listing 2-5.

Listing 2-5. Creating an Immutable and Thread-Safe Class with Help from `final`

```

import java.util.Set;
import java.util.TreeSet;

public final class Planets
{
    private final Set<String> planets = new TreeSet<>();

    public Planets()
    {
        planets.add("Mercury");
        planets.add("Venus");
    }
}

```

```

        planets.add("Earth");
        planets.add("Mars");
        planets.add("Jupiter");
        planets.add("Saturn");
        planets.add("Uranus");
        planets.add("Neptune");
    }

    public boolean isPlanet(String planetName)
    {
        return planets.contains(planetName);
    }
}

```

Listing 2-5 presents an immutable `Planets` class whose objects store sets of planet names. Although the set is mutable, the design of this class prevents the set from being modified after the constructor exits. By declaring `planets` `final`, the reference stored in this field cannot be modified. Furthermore, this reference will not be cached so the cached variable problem goes away.

Java provides a special thread-safety guarantee concerning immutable objects. These objects can be safely accessed from multiple threads, even when synchronization isn't used to *publish* (expose) their references provided that the following rules are observed:

- Immutable objects must not allow state to be modified.
- All fields must be declared `final`.
- Objects must be properly constructed so that “this” references don't escape from constructors.

The last point is probably confusing, so here is a simple example where `this` explicitly escapes from the constructor:

```

public class ThisEscapeDemo
{
    private static ThisEscapeDemo lastCreatedInstance;

    public ThisEscapeDemo()
    {
        lastCreatedInstance = this;
    }
}

```

Check out “Java theory and practice: Safe construction techniques” at www.ibm.com/developerworks/library/j-jtp0618/ to learn more about this common threading hazard.

EXERCISES

The following exercises are designed to test your understanding of Chapter 2's content:

1. Identify the three problems with threads.
2. True or false: When the correctness of a computation depends on the relative timing or interleaving of multiple threads by the scheduler, you have a data race.
3. Define synchronization.
4. Identify the two properties of synchronization.
5. How is synchronization implemented?
6. True or false: A thread that has acquired a lock doesn't release this lock when it calls one of Thread's `sleep()` methods.
7. How do you specify a synchronized method?
8. How do you specify a synchronized block?
9. Define liveness.
10. Identify the three liveness challenges.
11. How does the `volatile` keyword differ from `synchronized`?
12. True or false: Java also lets you safely access a `final` field without the need for synchronization.
13. Identify the thread problems with the following `CheckingAccount` class:

```
public class CheckingAccount
{
    private int balance;
    public CheckingAccount(int initialBalance)
    {
        balance = initialBalance;
    }
    public boolean withdraw(int amount)
    {
        if (amount <= balance)
        {
            try
```

```

        {
            Thread.sleep((int) (Math.random() * 200));
        }
        catch (InterruptedException ie)
        {
        }
        balance -= amount;
        return true;
    }
    return false;
}
public static void main(String[] args)
{
    final CheckingAccount ca = new CheckingAccount(100);
    Runnable r = new Runnable()
    {
        @Override
        public void run()
        {
            String name = Thread.currentThread().
                getName();
            for (int i = 0; i < 10; i++)
                System.out.println (name + "
                    withdraws $10: " +
                        ca.withdraw(10));
        }
    };
    Thread thdHusband = new Thread(r);
    thdHusband.setName("Husband");
    Thread thdWife = new Thread(r);
    thdWife.setName("Wife");
    thdHusband.start();
    thdWife.start();
}
}

```

14. Fix the thread problems in the previous `CheckingAccount` class.

Summary

Developing multithreaded applications is much easier when threads don't interact, typically via shared variables. When interaction occurs, race conditions, data races, and cached variable problems can arise that make an application thread-unsafe.

You can use synchronization to solve race conditions, data races, and cached variable problems. Synchronization is a JVM feature that ensures that two or more concurrent **threads** don't simultaneously execute a critical section that must be accessed in a serial manner.

Liveness refers to something beneficial happening eventually. A liveness failure occurs when an application reaches a state in which it can make no further progress. Multithreaded applications face the liveness challenges of deadlock, livelock, and starvation.

Synchronization exhibits two properties: mutual exclusion and visibility. The `synchronized` keyword is associated with both properties. Java also provides a weaker form of synchronization involving visibility only, and associates only this property with the `volatile` keyword.

When a field variable is declared `volatile`, it cannot also be declared `final`. However, this isn't a problem because Java also lets you safely access a `final` field without the need for synchronization. You will often use `final` to help ensure thread safety in the context of an immutable class.

Chapter 3 presents waiting and notification.