Chapter 5. Thread Safety

With the introduction of the *java.util.concurrent* package in Java 5, threads became commonly used to improve the performance of complex applications. In graphical (or *headed*) applications, they improve responsiveness by reducing the load on the main thread that processes information to render *views*—programmed components the user can see and interact with on-screen. When a thread is created within a program that has a concept of a main or UI thread, it's referred to as a *background thread*. These background threads often receive and process user interaction events, like gestures and text input; or other forms of data retrieval, like reading from a server; or local stores, like a database or filesystem. On the server side, backend applications using threads have better throughput by leveraging the multiple cores of modern CPUs.

However, using threads has its own risks, as you will see in this chapter. Thread safety can be seen as a set of techniques and good practices to circumvent those risks. Those techniques include *synchronization*, *mutexes*, and *blocking* versus *nonblocking*. Higher-level concepts like thread confinement are also important.

The goal of this chapter is to introduce you to some important thread-safety concepts that will be used in the following chapters. However, we won't cover thread safety extensively. For example, we won't explain *object publication* or provide details about the Java memory model. These are advanced topics that we encourage you to learn after you understand the concepts explained in this chapter.

An Example of a Thread Issue

To understand what thread safety is, we'll pick a simple example of a thread-safety issue. When a program runs several threads concurrently, each thread has the potential to do things *at the same time* as other running threads. But it doesn't necessarily mean this will happen. When it does happen, you need to prevent one thread from accessing an object that is being mutated by another thread, because it could read an inconsistent state of the object. The same goes for simultaneous mutations.

Ensuring that only one thread at a time can access a block of code is called *mutual exclusion*. Take the following, for example:

```
class A {
    var aList: MutableList<Int> = ArrayList()
    private set

    fun add() {
        val last = aList.last()  // equivalent of aList[aList.size - 1]
        aList.add(last + 1)
    }

    init {
        aList.add(1)
    }
}
```

The `add()` method takes the last element of the list, adds 1, and appends the result into the list. What would be the expected behavior if two threads attempted to simultaneously execute `add()`?

When the first thread references the last element, the other thread might have had time to execute the entire `aList.add(last + 1)` line.[1] In this case, the first thread reads 2 for the last element and will append 3 to the list. The resulting list would be `[1, 2, 3]`. Another scenario is possible. If the second thread didn't have time to append a new value, then the two threads will read the same value for the last element. Assuming that the rest of the execution runs without hiccups, we get the result `[1, 2, 2]`. One more hazard may happen: if the two threads try to append the new element to the list at exactly the same time, an `ArrayIndexOutOfBoundsException` will be thrown.

Depending on the interleaving of the threads, the result may be different. There's no guarantee that we'll get a result at all. Those are symptoms of a class or function that's not thread-safe, which may not behave correctly when accessed from multiple threads.

So, what could we do to fix this potential misbehavior? We have three options:

1. Don't share state across threads.
2. Share immutable state across threads.

3. Change our implementation so that multiple threads can use our class and get predictable results.

There are multiple strategies for approaching some kind of thread safety, each with its own strengths and caveats, so it is important for a developer to be able to evaluate their options and choose one that best fits the needs of a threading issue.

The first option is relatively obvious. When threads can work on completely independent datasets, there's no risk of accessing the same memory addresses.

The second option is making use of immutable objects and collections. Immutability is a very effective way to design robust systems. If a thread can't mutate an object, there's simply no risk of reading inconsistent state from another thread. In our example, we could make the list immutable, but then threads wouldn't be able to append elements to it. This doesn't mean that this principle can't be applied here. In fact, it can—but we'll come back to it later in this chapter. We have to mention that there's a potential downside with using immutability. In essence, it requires more memory because of object copying. For example, whenever a thread needs to work with another thread's state, a copy of the state object is performed. When done repeatedly and at a high pace, immutability can increase the memory footprint—which may be an issue (especially on Android).

The third option could be described like so: "Any thread which executes the `add` method happens before any subsequent `add` accesses from other threads." In other words, `add` accesses happen serially, with no interleaving. If your implementation enforces the aforementioned statement, then there won't be thread-safety issues—the class is said to be thread-safe. In the world of concurrency, the previous statement is called an *invariant*.

Invariants

To properly make a class or a group of classes thread-safe, we have to define invariants. An invariant is an assertion that should always be true. No matter how threads are scheduled, the invariant shall not be violated.

In the case of our example, it could be expressed like this (from the standpoint of a thread):

> *When I'm executing the `add` method, I'm taking the last element of the list and when I'm appending it to the list, I'm sure that the inserted element is greater than the previous one by a difference of 1.*

Mathematically, we could write:

$$list[n] = list[n-1] + 1$$

We've seen from the beginning that our class wasn't thread-safe. Now we can say so because when executed in a multithreaded environment, the invariant is sometimes violated or our program just crashes.

So, what can we do to enforce our invariants? Actually, this is a complex matter, but we'll cover some of the most common techniques:

- Mutexes
- Thread-safe collections

Mutexes

Mutexes allow you to prevent concurrent access of a state—which can be a block of code or just an object. This mutual exclusion is also called *synchronization*. An `Object` called a *mutex* or *lock* guarantees that when taken from a thread, no other thread can enter the section guarded by this lock. When a thread attempts to acquire a lock held by another thread, it's blocked—it cannot proceed with its execution until the lock is released. This mechanism is relatively easy to use, which is why it's often the go-to response of developers when facing this situation. Unfortunately, this is also like opening a Pandora's box to problems like deadlocks, race conditions, etc. These problems that can arise from improper synchronization are so numerous that drawing a complete picture is way beyond the scope of this book. However, later in the book we will discuss some of them, like deadlocks in communicating sequential processes.

Thread-Safe Collections

Thread-safe collections are collections that can be accessed by multiple threads while keeping their state consistent. The `Collections.synchronizedList` is a useful way to make a `List` thread-safe. It returns a `List` that wraps access to the `List` passed as a parameter, and regulates concurrent access with an internal lock.

At first sight, it looks interesting. So you could be tempted to use it:

```
class A {
    var list =
        Collections.synchronizedList<Int>(object : ArrayList<Int?>() {
            init {
                add(1)
            }
        })

    fun add() {
        val last = list.last()
        list.add(last + 1)
    }
}
```

For the record, here is the equivalent in Java:

```
class A {
    List<Integer> list = Collections.synchronizedList(
        new ArrayList<Integer>() {{
           add(1);
        }}
    );

    void add() {
        Integer last = list.get(list.size() - 1);
        list.add(last + 1);
    }
}
```

There’s a problem with both implementations. Can you spot it?

NOTE

We could also have declared the list as:

```
var list: List<Int> = CopyOnWriteArrayList(listOf(1))
```

which, in Java, is the equivalent of:

```
List<Integer> list = new CopyOnWriteArrayList<>(Arrays.asList(1));
```

`CopyOnWriteArrayList` is a thread-safe implementation of `ArrayList` in which all mutative operations like `add` and `set` are implemented by making a fresh copy of the underlying array. Thread *A* can safely iterate through the list. If in the meantime, thread *B* adds an element to the list, a fresh copy will be created and only visible from thread *B*. This in itself doesn't make the class thread-safe—it is because `add` and `set` are guarded by a lock. This data structure is useful when we are iterating over it more often than we are modifying it, as copying the entire underlying array can be too costly. Note that there is also a `CopyOnWriteArraySet`, which is simply a `Set` implementation rather than a `List` implementation.

We've indeed fixed the concurrent access issue, although our class still doesn't conform to our invariant. In a test environment, we created two threads and started them. Each thread executes the `add()` method once, on the same instance of our class. The first time we ran our test, after the two threads finished their job, the resulting list was `[1, 2, 3]`. Curiously, we ran this same test multiple times, and the result was sometimes `[1, 2, 2]`. This is due to the exact same reason shown earlier: when a thread executes the first line inside `add()`, the other thread can execute the whole `add()` method before the first thread proceeds with the rest of its execution. See how pernicious a synchronization issue can be: it looks good, but our program is broken. And we can easily have it wrong, even on a trivial example.

A proper solution is:

```
class A {
    val list: MutableList<Int> = mutableListOf(1)

    @Synchronized
    fun add() {
        val last = list.last()
        list.add(last + 1)
    }
}
```

It may help to see the Java equivalent:

```
public class A {
    private List<Integer> list = new ArrayList<Integer>() {{
        add(1);
    }};

    synchronized void add() {
        Integer last = list.get(list.size() - 1);
        list.add(last + 1);
    }
}
```

As you can see, we actually didn't need to synchronize the list. Instead, the `add()` method should have been synchronized. Now when the `add()` method is first executed by a thread, the other one blocks when it tries to execute `add()`, until the first thread leaves the `add()` method. No two threads execute `add()` at the same time. The invariant is then honored.

This example demonstrates that a class can internally use thread-safe collections while not being thread-safe. A class or code is said to be thread-safe when its invariants are never violated. Those invariants, and how the class should be used according to their creators, define a policy that should be clearly expressed in the javadoc.

NOTE

This is Java's built-in mechanism to enforce mutual exclusion. A synchronized block is made of a lock and a block of code. In Java, every `Object` can be used as a lock. A synchronized method is a synchronized block whose lock is the instance of the class instance. When a thread enters a synchronized block, it acquires the lock. And when a thread leaves the block, it releases the lock.

Also note that the `add` method could have been declared as using a `synchronized` statement:

```
void add() {
    synchronized(this) {
        val last = list.last()
        list.add(last + 1)
    }
}
```

A thread cannot enter a synchronized block whose lock is already acquired by another thread. As a consequence, when a thread enters a synchronized method it prevents other threads from executing any synchronized method or any block of code guarded by this (also called *intrinsic* lock).

Thread Confinement

Another way to ensure thread safety is to ensure that only one thread owns the state. If the state isn't visible to other threads, there's simply no risk of having concurrency issues. For example, a public variable of a class (where usage is intended to be thread-confined to the main thread) is a potential source of bugs since a developer (unaware of this thread policy) could use the variable in another thread.

The immediate benefit of thread confinement is simplicity. For example, if we follow the convention that every class of type `View` should only be used from the main thread, then we can save ourselves from synchronizing our code all over the place. But this comes at a price. The correctness of the client code is now on the shoulders of the developer who uses our code. In Android, as we've seen in the previous chapter, manipulating views should only be done from the UI thread. This is a form of thread confinement—as long as you don't break the rules, you shouldn't have issues involving concurrent access to UI-related objects.

Another noteworthy form of thread confinement is `ThreadLocal`. A `ThreadLocal` instance can be seen as a provider to some object. This provider ensures that the given instance of the object is per-thread unique. In other words, each thread owns its own instance of the value. An example of usage is:

```
private val myConnection =
        object : ThreadLocal<Connection>() {
            override fun initialValue(): Connection? {
                return DriverManager.getConnection(connectionStr)
            }
        }
```

Often used in conjunction with JDBC connections, which aren't thread-safe, `ThreadLocal` ensures that each thread will use its own JDBC connection.

Thread Contention

Synchronization between threads is hard because a lot of problems can happen. We just saw potential thread-safety issues. There is another hazard that can affect performance: *thread contention*, which we encourage all programmers to familiarize themselves with. Consider this example:

```
class WorkerPool {
    private val workLock = Any() // In Java, we would have used `new Object
    
    fun work() {
        synchronized(workLock) {
            try {
                Thread.sleep(1000) // simulate CPU-intensive task
            } catch (e: Exception) {
                e.printStackTrace()
            }
        }
    }

    // other methods which may use the intrinsic lock
}
```

So, we have a `WorkerPool`, which controls the work done by worker threads in such a way that only one worker at a time can do the real work inside the `work` method. This is the kind of situation you may encounter when the actual work involves the use of non-thread-safe objects and the developer decided to solve this using this locking policy. A dedicated lock was created for the `work` method, instead of synchronizing on `this`, because other methods can now be called by workers without mutual exclusion. This is also the reason why the lock is named after the related method.

If several worker threads are started and call this `work` method, they will contend for the same lock. Eventually, depending on the interleaving of the threads, a worker is blocked because another one acquired the lock. This isn't a problem if the time spent waiting for the lock is significantly less than the rest of the execution time. If this isn't the case, then we have a thread contention. Threads spend most of their time waiting for each other. Then the operating system may preemptively stall some threads so that other threads in the wait state can resume their execution, which makes the situation even worse because context switches between threads aren't free. It can result in a performance impact when they occur frequently.

As a developer, you should always avoid thread contention as it can rapidly degrade throughput and have consequences beyond the affected threads, since the rate of context switches is likely to increase, which in itself impacts performance overall.

One of the most effective ways to avoid such a situation is to avoid blocking calls, which we explain in the next section.

Blocking Call Versus Nonblocking Call

So far, we know that a thread can be blocked when attempting to obtain a lock held by another thread. The function that led the thread to be blocked is then a *blocking call*. Even if the lock might be acquired immediately, the fact that the call may potentially block makes it a `blocking call`. But this is just a particular case. There are actually two other ways of blocking a thread. The first one is by running CPU-intensive computations—this is also called a *CPU-bound* task. The second one is by waiting for a hardware response. For example, it happens when a network request causes the calling thread to wait for the response from a remote server—we then talk about an IO-bound task.[2]

Everything else that makes the call return quickly is considered *nonblocking*.

When you're about to make a blocking call, you should avoid doing it from the main thread (also called the UI thread, on Android).[3] This is because this thread runs the event loop that processes touch events, and all UI-related tasks like animations. If the main thread gets blocked repeatedly and for durations exceeding a few milliseconds, the responsiveness is impacted and this is the cause of Android's *application not responding* (ANR) errors.

Nonblocking calls is one building block of a responsive app. You need now to recognize patterns which leverage this technique. Work queues is one of them, and we'll encounter various forms of them throughout this book.

NOTE

Most often, the terms *synchronous* and *asynchronous* are respectively used as synonyms for *blocking* and *nonblocking*. While they are conceptually close concepts, the usage of, for instance, asynchronous instead of nonblocking depends on the context. Asynchronous calls usually involve the idea of a callback, while this is not necessarily the case for nonblocking.

Work Queues

Communication between threads and, in particular, work submission from one thread to another is widely used in Android. It's an implementation of the *producer-consumer* design pattern. Applied to threads, the producer is in this context a thread which generates data that needs to be further processed by a consumer thread. Instead of having the producer directly interacting with the consumer through shared mutable state, a queue is used in between to enqueue the work generated by the producer. It decouples the producer from the consumer—but this isn't the only benefit, as we'll see. Often, the `Queue` works in a FIFO (first in, first out) manner.[4]

Semantically it can help to think of a `Queue` like a queue of moviegoers. As the first viewer arrives, they are put at the front of the queue. Each additional viewer is added behind the last. When the doors open and viewers are allowed to enter, the first person in line is let in first, then the next, and so on, until the entire `Queue` is empty.

The producer puts an object at the head of the queue, and the consumer pops an object at the tail of the queue. The `put` method might be a blocking call, but if it can be proven that most of the time it effectively doesn't block (and when it does, it's for a short time), then we have a very efficient way to offload work from the producer to the consumer in a nonblocking way (from the standpoint of the producer), as shown in Figure 5-1.

In practice, enqueued objects are often `Runnable` instances submitted by a background thread and processed by the main thread. Also, this isn't limited to one producer and one consumer. Multiple producers can submit work to the queue, concurrently with multiple consumers taking work out of the queue. This implies that the queue must be thread-safe.[5]

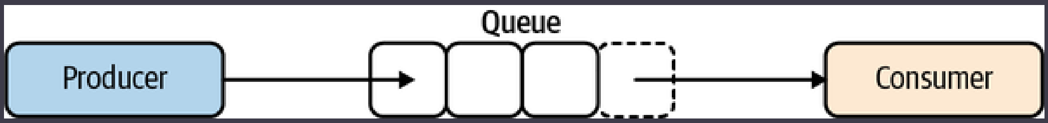

Figure 5-1. Producer-consumer.

NOTE

Don't confuse a `Queue` with a `Stack`, which uses LIFO (last in, first out) instead of FIFO.

Semantically, let's imagine a `Stack` as a stack of pancakes. When the kitchen makes more pancakes, they go on the top of the stack. When the diner eats pancakes, they also take them from the top of the stack.

Back Pressure

Imagine now that our producer is much faster than our consumer. The work objects then accumulate in the queue. If the queue happens to be unbounded, we risk exhausting memory resources and potentially an unrecoverable exception: the application may crash. While not only is this experience jarring and unpleasant for the user, but in an unhandled error like this, you're almost assuredly going to lose whatever stateful information was present. Unless you've taken great care to be aware of—and react to—this circumstance, you may experience a sudden termination without an opportunity to perform any cleanup you might do normally. In Android, when a Bitmap instance is no longer being used, the recycle method can be used to mark each underlying memory allocation as unreachable and eligible for garbage collection. In an untidy system exit, you might not have an opportunity to do that and may risk leaking that data.

In this case, a wise choice is to use a bounded queue. But what should happen when the queue is full and a producer attempts to `put` an object?

We'll circle back to it with coroutines, but since we're only talking about threads for now, the answer is: it should block the producer thread until the consumer takes at least one object out of the queue. Although this blocking should be part of the design and anticipate whatever circumstance or logic branch might deliver the user to this point in the program. While blocking a thread seems harmful, a blocked producer allows the consumer to catch up and free up enough space into the queue so that the producer is released.

This mechanism is known as *back pressure*—the ability of a data consumer that can't keep up with incoming data to slow down the data producer. It's a very powerful way to design robust systems. Example 5-1 shows a implementation of back pressure.

Example 5-1. Back pressure example

```
fun main() {
    val workQueue = LinkedBlockingQueue<Int>(5)  // queue of size 5

    val producer = thread {
        while (true) {
            /* Inserts one element at the tail of the queue,
             * waiting if necessary for space to become available. */
            workQueue.put(1)
            println("Producer added a new element to the queue")
        }
    }
    val consumer = thread {
        while (true) {
            // We have a slow consumer - it sleeps at each iteration
            Thread.sleep(1000)
            workQueue.take()
            println("Consumer took an element out of the queue")
        }
    }
}
```

Since Java 7, a family of queues for this purpose is `BlockingQueue`—it's an interface, and implementations range from a single-ended queue with `LinkedBlockingQueue` to a double-ended queue with `LinkedBlockingDeque` (other implementations exist). The output of Example 5-1 is:

```
Producer added a new element to the queue
Producer added a new element to the queue
Producer added a new element to the queue
Producer added a new element to the queue
Producer added a new element to the queue
Consumer took an element out of the queue
Producer added a new element to the queue
Consumer took an element out of the queue
Producer added a new element to the queue
...
```

You can see that the producer quickly filled the queue with five elements. Then, on the sixth attempt to add a new element, it's blocked because the queue is full. One second later, the consumer takes an element out of the queue, releasing the producer which can now add a new element. At this point, the queue is full. The producer tries to add new elements but is blocked again. Again, one second later, the consumer takes one element—and so on.

It's important to note that the insertion of an element into a `BlockingQueue` isn't necessarily blocking. If you use the `put` method, then it blocks when the queue is full. Since `put` *might* block, we say that this is a blocking call. However, there's another method available to add a new element: `offer`, which attempts to immediately add the new element and returns a Boolean—whether or not the operation succeeded.

Since the `offer` method does not block the underlying thread and only returns false when the queue is full, we say that `offer` is nonblocking.

Had we used `offer` instead of `put` in Example 5-1, the producer would never be blocked, and the output would be filled with `Producer added a new element to the queue`. There would be no back pressure at all —don't do this!

The `offer` method can be useful in situations where losing work is affordable, or if blocking the producer thread isn't suitable. The same reasoning applies when taking an object out of the queue, with `take` and `poll`, which are respectively blocking and nonblocking.

Conversely, if the consumer is faster than the producer, then the queue eventually becomes empty. In the case of a `BlockingQueue`, using the `take` method on a consumer site will block until the producer adds new elements in the queue. So in this case, the consumer is slowed down to match the rate of the producer.

Summary

- A class or code is said to be thread-safe when its invariants are never violated. So, thread safety always refers to a policy that should be clearly defined in the class javadoc.
- A class can use internally thread-safe data structures while not being thread-safe.
- Avoid or reduce thread contention as much as possible. Thread contention is often the consequence of a poor locking strategy. An efficient way to reduce this risk is to do nonblocking calls whenever possible.
- Work queues is a pattern you will often encounter in Android and other platforms like backend services. It simplifies how a producer (like UI thread) offloads tasks to consumers (your background threads). Consumers process the tasks whenever they can. When the task completes, a consumer can use another work queue to send back to the original producer the result of its work.
- A bounded `BlockingQueue` blocks a `put` operation when it's full. So a too-fast producer eventually gets blocked, which gives consumers the opportunity to catch up. This is an implementation of back pressure, which has one major downside: the thread of the pro-

ducer might get blocked. Is it possible to have back pressure without blocking the producer thread? Yes—we'll see that in Chapter 9.

1 Actually, interleaving of threads can happen between lines of bytecode, not just between lines of normal Java.

2 IO operations aren't necessarily blocking. Nonblocking IO exists, though it's much more complicated to reason about. Android Link is helpful enough to warn you when you perform an HTTP request on the main thread, but other IO tasks—like reading a file or querying a database—do not do this. This may even be a deliberate and accepted practice if done under extremely thoughtful and careful supervision; while possible, this should be a rare exception to the standard.

3 Even for worker threads, executing a long-running task like working with 8-megapixel images, those blocking calls possibly block task packets the UI is waiting on.

4 Although not all work queues use this data structure arrangement. Some of them are more sophisticated, like Android's `MessageQueue`.

5 Even with one producer and one consumer, the queue must be thread-safe.