Donate

Stay safe, friends. Learn to code from home. Use our free 2,000 hour curriculum.

31 JANUARY 2019 / #PROGRAMMING

# Code review checklist: how to tackle issues with Java concurrency

by Roman Leventov

At the Apache Druid community, we are currently preparing a parts of the checklist as posts on Medium to gather more ideas for checklist items. Hopefully, somebody will find it useful in practice.

By the way, it seems me that creating project-specific checklists for code reviews should be a powerful idea, yet I don't see any existing examples among large open source projects.

This post contains checklist items about problems that arise with the multithreaded Java code.

Thanks to Marko Topolnik, Matko Medenjak, Chris Vest, Simon Willnauer, Ben Manes, Gleb Smirnov, Andrey Satarin, Benedict Jin, and Petr Janeček for reviews and contributions to this post. The checklist is not considered complete, comments and suggestions are welcome!

Update: this checklist is now available on Github.

## 1. Design

1.1. If the patch introduces a new subsystem with concurrent code, is **the necessity for concurrency rationalized in the patch description**? Is there a discussion of alternative design approaches that could simplify the concurrency model of the code (see the next item)?

1.2. Is it possible to apply one or several design patterns (some of them are listed below) to significantly **simplify the concurrency model of the code, while not considerably compromising other quality aspects**, such as overall simplicity, efficiency, testability, extensibility, etc?

**Immutability/Snapshotting.** When some state should be updated, a created, published and used, while some concurrent threads may still use older copies or snapshots. See [EJ Item 17], [JCIP 3.4], items 4.5 and 9.2 in this checklist, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, persistent data structures.

**Divide and conquer.** Work is split into several parts that are processed independently, each part in a single thread. Then the results of processing are combined. Parallel Streams (see section 14) or `ForkJoinPool` (see items 10.4 and 10.5) can be used to apply this pattern.

**Producer-consumer.** Pieces of work are transmitted between worker threads via queues. See [JCIP 5.3], item 6.1 in this checklist, CSP, SEDA.

**Instance confinement.** Objects of some root type encapsulate some complex hierarchical child state. Root objects are solitarily responsible for the safety of accesses and modifications to the child state from multiple threads. In other words, composed objects are synchronized rather than synchronized objects are composed. See [JCIP 4.2, 10.1.3, 10.1.4].

**Thread/Task/Serial thread confinement.** Some state is made local to a thread using top-down pass-through parameters or `ThreadLocal`. See [JCIP 3.3]. Task confinement is a variation of the idea of thread confinement that is used in conjunction with the divide-and-conquer pattern. It usually comes in the form of lambda-captured “context” parameters or fields in the per-thread task objects. Serial thread confinement is an extension of the idea of thread confinement for the producer-consumer pattern, see [JCIP 5.3.2].

Donate

2.1. For every class, method, and field that has signs of being thread-safe, such as the `synchronized` keyword, `volatile` modifiers on fields, use of any classes from `java.util.concurrent.*`, or third-party concurrency primitives, or concurrent collections: do their Javadoc comments include

- **The justification for thread safety**: is it explained why a particular class, method or field has to be thread-safe?
- **Concurrent control flow documentation**: is it enumerated from what methods and in contexts of what threads (executors, thread pools) each specific method of a thread-safe class is called?

Wherever some logic is parallelized or the execution is delegated to another thread, are there comments explaining why it's worse or inappropriate to execute the logic sequentially or in the same thread? See also item 14.1 in this checklist about parallel `Stream` use.

2.2. If the patch introduces a new subsystem that uses threads or thread pools, are there **high-level descriptions of the threading model, the concurrent control flow (or the data flow) of the subsystem** somewhere, e. g. in the Javadoc comment for the package in `package-info.java` or for the main class of the subsystem? Are these descriptions kept up-to-date when new threads or thread pools are added or some old ones deleted from the system?

Description of the threading model includes the enumeration of threads and thread pools created and managed in the subsystem, and external pools used in the subsystem (such as `ForkJoinPool.commonPo` priorities, and the lifecycle of the managed threads and thread pools. A high-level description of concurrent control flow should be an overview and tie together concurrent control flow documentation for individual classes, see the previous item. If the producer-consumer pattern is used, the concurrent control flow is trivial and the data flow should be documented instead.

Describing threading models and control/data flow greatly improves the maintainability of the system, because in the absence of descriptions or diagrams developers spend a lot of time and effort to create and refresh these models in their minds. Putting the models down also helps to discover bottlenecks and the ways to simplify the design (see item 1.2).

2.3. For classes and methods that are parts of the public API or the extensions API of the project: is it specified in their Javadoc comments whether they are (or in case of interfaces and abstract classes designed for subclassing in extensions, should they be implemented as) **immutable, thread-safe or not thread-safe**? For classes and methods that are (or should be implemented as) thread-safe, is it documented precisely with what other methods (or themselves) they may be called concurrently from multiple threads? See also [EJ Item 82] and [JCIP 4.5].

If the `@com.google.errorprone.annotations.Immutable` annotation is used to mark immutable classes, Error Prone static analysis tool is capable to detect when a class is not actually immutable (see the relevant bug pattern).

2.4. For subsystems, classes, methods, and fields that use some concurrency design patterns, either high-level (such as those

Donate

checked locking, see section 8 in this checklist): are the used **concurrency patterns pronounced in the design or implementation comments** for the respective subsystems, classes, methods, and fields? This helps readers to make sense out of the code quicker.

2.5. Are `ConcurrentHashMap` and `ConcurrentSkipListMap` objects stored in fields and variables of `ConcurrentHashMap` or `ConcurrentSkipListMap` or **`ConcurrentMap` type**, but not just `Map` ?

This is important, because in code like the following:

```
ConcurrentMap<String, Entity> entities = getEntities();if (!entitie
```

It should be pretty obvious that there might be a race condition because an entity may be put into the map by a concurrent thread between the calls to `containsKey()` and `put()` (see item 4.1 about this type of race conditions). While if the type of the `entities` variable was just `Map<String, Enti`ty> it would be less obvious and readers might think this is only slightly suboptimal code and pass by.

It's possible to turn this advice into an inspection in IntelliJ IDEA.

2.6. An extension of the previous item: are ConcurrentHashMaps on which `compute()` , `computeIfAbsent()` , `computeIfPresent()` , or `merge()` methods are called stored in fields and variables of `ConcurrentHashMap` type rather than `ConcurrentMap` ? This is because `ConcurrentHashMap` (unlike the generic `ConcurrentMap` interface) guarantees that the lambdas passed into `compute()` -like methods are performed atomically per key, and the thread safety of the class may depend on that guarantee.

This advice may seem to be overly pedantic, but if used in conjunction with a static analysis rule that prohibits calling `compute()` -like methods on `ConcurrentMap` -typed objects that are not ConcurrentHashMaps (it's possible to create such inspection in IntelliJ IDEA too) it could prevent some bugs: e. g. **calling** `compute()` **on a** `ConcurrentSkipListMap` **might be a race condition** and it's easy to overlook that for somebody who is used to rely on the strong semantics of `compute()` in `ConcurrentHashMap` .

2.7. Is **`@GuardedBy` annotation used**? If accesses to some fields should be protected by some lock, are those fields annotated with `@GuardedBy` ? Are private methods that are called from within critical sections in other methods annotated with `@GuardedBy` ? If the project doesn't depend on any library containing this annotation (it's provided by `jcip-annotations` , `error_prone_annotations` , `jsr305` and other libraries) and for some reason it's undesirable to add such dependency, it should be mentioned in Javadoc comments for the respective fields and methods that accesses and calls to them should be protected by some specified locks.

See [JCIP 2.4] for more information about `@GuardedBy` .

Using `GuardedBy` is especially beneficial in together with Error Prone, which is able to statically check for unguarded accesses to fields and methods with `@GuardedBy` annotations.

2.8. If in a thread-safe class some **fields are accessed both from within critical sections and outside of critical sections**, is it explained in comments why this is safe? For example, unprotected read-only access to a reference to an immutable object might be benignly racy (see item 4.5).

2.9. Regarding every field with a `volatile` modifier: **does it really need to be** `volatile`? Does the Javadoc comment for the field explain why the semantics of `volatile` field reads and writes (as defined in the Java Memory Model) are required for the field?

2.10. Is it explained in the **Javadoc comment for each mutable field in a thread-safe class that is neither** `volatile` **nor annotated with** `@GuardedBy`, why that is safe? Perhaps, the field is only accessed and mutated from a single method or a set of methods that are specified to be called only from a single thread sequentially (as described in item 2.1). This recommendation also applies to `final` fields that store objects of non-thread-safe classes when those objects could be mutated from the methods of the enclosing thread-safe class. See items 4.2–4.4 in this checklist about what could go wrong with such code.

## 3. Excessive thread safety

3.1. An example of excessive thread safety is a class where every modifiable field is `volatile` or an `AtomicReference` or other atomic, and every collection field stores a concurrent collection (e. g. `ConcurrentHashMap`), although all accesses to those fields are synchronized.

**There shouldn't be any "extra" thread safety in code, there should be just enough of it.** Duplication of thread safety confuses readers because they might think the extra thread safety precautions are (or used to be) needed for something but will fail to find the purpose.

The exception from this principle is the `volatile` modifier on the lazily initialized field in the safe local double-checked locking pattern which is the recommended way to implement double-checked locking, initialized object has all `final` fields*. Without that `volatile` modifier the thread safety of the double-checked locking could easily be broken by a change (addition of a non-`final` field) in the class of lazily initialized objects, though that class should not be aware of subtle concurrency implications. If the class of lazily initialized objects is *specified* to be immutable (see item 2.3) the `volatile` is still unnecessary and the UnsafeLocalDCL pattern could be used safely, but the fact that some class has all `final` fields doesn't necessarily mean that it's immutable.

See also section 8 in this post about double-checked locking.

3.2. Aren't there **`AtomicReference`**, `AtomicBoolean`, `AtomicInteger` **or** `AtomicLong` **fields on which only** `get()` **and** `set()` **methods are called?** Simple fields with `volatile` modifiers can be used instead, but `volatile` might not be needed too; see item 2.9.

## 4. Race conditions

4.1. Aren't **ConcurrentHashMaps updated with multiple separate** `containsKey()`, `get()`, `put()` **and** `remove()` **calls** instead of a single call to `compute()`/`computeIfAbsent()`/`computeIfPresent()`/`replace()`?

4.2. Aren't there **point read accesses such as** `Map.get()`, `containsKey()` **or** `List.get()` **outside of critical sections to a non-thread-safe collection such as** `HashMap` **or** `ArrayList`, while new entries can be added to the collection concurrently, even though there is a happens-before edge between the moment when some entry is put into the collection and the moment when the same entry is point-queried outside of a critical section?

grows and changes its internal structure from time to time (`HashMap` rehashes the hash table, `ArrayList` reallocates the internal array). At such moments races might happen and unprotected point read accesses might fail with `NullPointerException`, `ArrayIndexOutOfBoundsException`, or return `null` or some random entry.

Note that this concern applies to `ArrayList` even when elements are only added to the end of the list. However, a small change in `ArrayList`'s implementation in OpenJDK could have disallowed data races in such cases at very little cost. If you are subscribed to the concurrency-interest mailing list, you could help to bring attention to this problem by reviving this thread.

4.3. A variation of the previous item: isn't a non-thread-safe collection such as `HashMap` or `ArrayList` **iterated outside of a critical section**, while it can be modified concurrently? This could happen by accident when an `Iterable`, `Iterator` or `Stream` over a collection is returned from a method of a thread-safe class, even though the iterator or stream is created within a critical section.

Like the previous item, this one applies to growing ArrayLists too.

4.4. More generally, aren't **non-trivial objects that can be mutated concurrently returned from getters** on a thread-safe class?

4.5. If there are multiple variables in a thread-safe class that are **updated at once but have individual getters**, isn't there a race condition in the code that calls those getters? If there is, the variables should be made `final` fields in a dedicated POJO, that serves as a snapshot of the updated state. The POJO is stored in a field of the thread-safe class, directly or as an `AtomicReference`. Multiple getters the POJO. This allows avoiding a race condition in the client code by reading a consistent snapshot of the state at once.

This pattern is also very useful for crafting safe and reasonably simple non-blocking code: see item 9.2 in this checklist and [JCIP 15.3.1].

4.6. If some logic within some critical section depends on some data that principally is part of the internal mutable state of the class, but was read outside of the critical section or in a different critical section, isn't there a race condition because the **local copy of the data may become out of sync with the internal state by the time when the critical section is entered**? This is a typical variant of check-then-act race condition, see [JCIP 2.2.1].

4.7. Aren't there **race conditions between the code (i. e. program runtime actions) and some actions in the outside world** or actions performed by some other programs running on the machine? For example, if some configurations or credentials are hot reloaded from some file or external registry, reading separate configuration parameters or separate credentials (such as username and password) in separate transactions with the file or the registry may be racing with a system operator updating those configurations or credentials.

Another example is checking that a file exists (or not exists) and then reading, deleting, or creating it, respectively, while another program or a user may delete or create the file between the check and the act. It's not always possible to cope with such race conditions, but it's useful to keep such possibilities in mind. Prefer static methods from `java.nio.file.Files` class instead of methods from the old `java.io.File` for file system operations. Methods from `Files` are more sensitive to file system race conditions and tend to throw exceptions hard even to detect race conditions.

## 5. Replacing locks with concurrency utilities

5.1. Is it possible to use concurrent collections and/or utilities from `java.util.concurrent.*` and **avoid using locks with** `Object.wait()` / `notify()` / `notifyAll()` ? Code redesigned around concurrent collections and utilities is often both clearer and less error-prone than code implementing the equivalent logic with intrinsic locks, `Object.wait()` and `notify()` ( `Lock` objects with `await()` and `signal()` are not different in this regard). See [EJ Item 81] for more information.

5.2. Is it possible to **simplify code that uses intrinsic locks or** `Lock` **objects with conditional waits by using Guava's** `Monitor` **instead**?

## 6. Avoiding deadlocks

6.1. If a thread-safe class is implemented so that there are nested critical sections protected by different locks, **is it possible to redesign the code to get rid of nested critical sections**? Sometimes a class could be split into several distinct classes, or some work that is done within a single thread could be split between several threads or tasks which communicate via concurrent queues. See [JCIP 5.3] for more information about the producer-consumer pattern.

6.2. If restructuring a thread-safe class to avoid nested critical sections is not reasonable, was it deliberately checked that the locks are acquired in the same order throughout the code of the class? **Is the locking order documented in the Javadoc comments for the fields where the lock objects are stored?**

6.3. If there are nested critical sections protected by several

**associated with some business logic entities), are the locks ordered before the acquisition**? See [JCIP 10.1.2] for more information.

6.4. Aren't there **calls to some callbacks (listeners, etc.) that can be configured through public API or extension interface calls within critical sections** of a class? With such calls, the system might be inherently prone to deadlocks because the external logic executed within a critical section might be unaware of the locking considerations and call back into the logic of the project, where some more locks may be acquired, potentially forming a locking cycle that might lead to deadlock. Let alone the external logic could just perform some time-consuming operation and by that harm the efficiency of the system (see the next item). See [JCIP 10.1.3] and [EJ Item 79] for more information.

## 7. Improving scalability

7.1. **Are critical sections as small as possible?** For every critical section: can't some statements in the beginning and the end of the section be moved out of it? Not only minimizing critical sections improves scalability, but also makes it easier to review them and spot race conditions and deadlocks.

This advice equally applies to lambdas passed into `ConcurrentHashMap`'s `compute()`-like methods.

See also [JCIP 11.4.1] and [EJ Item 79].

7.2. Is it possible to **increase locking granularity**? If a thread-safe class encapsulates accesses to map, is it possible to **turn critical sections into lambdas passed into** `ConcurrentHashMap.compute()` or `computeIfAbsent()` or `computeIfPresent()` methods to enjoy effective per-

`ed` or an equivalent? See [JCIP 11.4.3] for more information about lock striping.

7.3. Is it possible to **use non-blocking collections instead of blocking ones?** Here are some possible replacements within JDK:

- `Collections.synchronizedMap(HashMap)`, `Hashtable` → `ConcurrentHashMap`
- `Collections.synchronizedSet(HashSet)` → `ConcurrentHashMap.newKeySet()`
- `Collections.synchronizedMap(TreeMap)` → `ConcurrentSkipListMap`. By the way, `ConcurrentSkipListMap` is not the state of the art concurrent sorted dictionary implementation. `SnapTree` is more efficient than `ConcurrentSkipListMap` and there have been some research papers presenting algorithms that are claimed to be more efficient than SnapTree.
- `Collections.synchronizedSet(TreeSet)` → `ConcurrentSkipListSet`
- `Collections.synchronizedList(ArrayList)`, `Vector` → `CopyOnWriteArrayList`
- `LinkedBlockingQueue` → `ConcurrentLinkedQueue`
- `LinkedBlockingDeque` → `ConcurrentLinkedDeque`

Was it considered to **use one of the array-based queues from the JCTools library instead of** `ArrayBlockingQueue`? Those queues from JCTools are classified as blocking, but they avoid lock acquisition in many cases and are generally much faster than `ArrayBlockingQueue`.

`ass, . ..>?` Note, however, that unlike `ConcurrentHashMap` with its `computeIfAbsent()` method `ClassValue` doesn't guarantee that per-class value is computed only once, i. e. `ClassValue.computeValue()` might be executed by multiple concurrent threads. So if the computation inside `computeValue()` is not thread-safe, it should be synchronized separately. On the other hand, `ClassValue` does guarantee that the same value is always returned from `ClassValue.get()` (unless `remove()` is called).

7.5. Was it considered to **replace a simple lock with a** `ReadWriteLock`? Beware, however, that it's more expensive to acquire and release a `ReentrantReadWriteLock` than a simple intrinsic lock, so the increase in scalability comes at the cost of reduced throughput. If the operations to be performed under a lock are short, or if a lock is already striped (see item 7.2) and therefore very lightly contended, **replacing a simple lock with a** `ReadWriteLock` **might have a net negative effect** on the application performance. See this comment for more details.

7.6. Is it possible to use a **`StampedLock`** **instead of a** `ReentrantReadWriteLock` when reentrancy is not needed?

7.7. Is it possible to use **`LongAdder`** **for "hot fields"** (see [JCIP 11.4.4]) instead of `AtomicLong` or `AtomicInteger` on which only methods like `incrementAndGet()`, `decrementAndGet()`, `addAndGet()` and (rarely) `get()` is called, but not `set()` and `compareAndSet()`?

## 8. Lazy initialization and double-checked locking

8.1. For every lazily initialized field: **is the initialization code thread-safe and might it be called from multiple threads concurrently?** If the

used or the initialization should be eager.
8.2. If a field is initialized lazily under a simple lock, is it possible to use double-checked locking instead to improve performance?

8.3. Does double-checked locking follow the SafeLocalDCL pattern, as noted in item 3.1 in this checklist?

If the initialized objects are immutable a more efficient UnsafeLocalDCL pattern might also be used. However, if the lazily-initialized field is not `volatile` and there are accesses to the field that bypass the initialization path, the value of the **field must be carefully cached in a local variable**. For example, the following code is buggy:

```
private MyClass lazilyInitializedField;



void foo() {  if (lazilyInitializedField != null) { // (1)    // Ca
```

It might result in a `NullPointerException`, because although a non-null value is observed when the field is read the first time at line 1, the second read at line 2 could observe null.

The above code could be fixed as follows:

```
void foo() {  MyClass lazilyInitialized = this.lazilyInitializedFie
```

See "Wishful Thinking: Happens-Before Is The Actual Ordering" for more information.

8.4. In each particular case, doesn't the **net impact of double-checked locking and lazy field initialization on performance and complexity overweight the benefits of lazy initialization?** Isn't it ultimately better to initialize the field eagerly?

8.5. If a field is initialized lazily under a simple lock or using double-checked locking, does it really need locking? If nothing bad may happen if two threads do the initialization at the same time and use different copies of the initialized state, a benign race could be allowed. The initialized field should still be `volatile` (unless the initialized objects are immutable) to ensure there is a happens-before edge between threads doing the initialization and reading the field.

See also [EJ Item 83] and "Safe Publication this and Safe Initialization in Java".

## 9. Non-blocking and partially blocking code

9.1. If there is some non-blocking or semi-symmetrically blocking code that mutates the state of a thread-safe class, was it deliberately checked that if a **thread on a non-blocking mutation path is preempted after each statement, the object is still in a valid state**? Are there enough comments, perhaps before almost every statement where the state is changed, to make it relatively easy for readers of the code to repeat and verify the check?

9.2. Is it possible to simplify some non-blocking code by **confining all mutable state in an immutable POJO and update it via compare-**

Instead of a POJO, a single `long` value could be used if all parts of the state are integers that can together fit 64 bits. See also [JCIP 15.3.1].

## 10. Threads and Executors

10.1. **Are Threads given names** when created? Are ExecutorServices created with thread factories that name threads?

It appears that different projects have different policies regarding other aspects of `Thread` creation: whether to make them daemon with `setDaemon()`, whether to set thread priorities and whether a `ThreadGroup` should be specified. Many of such rules can be effectively enforced with forbidden-apis.

10.2. Aren't there threads created and started, but not stored in fields, a-la **`new Thread(...).start()`**, in some methods that may be called repeatedly? Is it possible to delegate the work to a cached or a shared `ExecutorService` instead?

10.3. **Aren't some network I/O operations performed in an `Executors.newCachedThreadPool()`-created `ExecutorService`?** If a machine that runs the application has network problems or the network bandwidth is exhausted due to increased load, CachedThreadPools that perform network I/O might begin to create new threads uncontrollably.

10.4. **Aren't there blocking or I/O operations performed in tasks scheduled to a `ForkJoinPool`** (except those performed via a `managedBlock()` call)? Parallel `Stream` operations are executed in the common `ForkJoinPool` implicitly, as well as the lambdas passed into `CompletableFuture`'s methods whose names end with "Async".

as that may happen during logging) and operations that may rarely block (such as `ConcurrentHashMap.put()` calls) usually shouldn't disqualify all their callers from execution in a `ForkJoinPool` or in a parallel `Stream`. See Parallel Stream Guidance for the more detailed discussion of those tradeoffs.

See also section 14 in this checklist about parallel Streams.

10.5. Opposite of the previous item: **can non-blocking computations be parallelized or executed asynchronously by submitting tasks to** `ForkJoinPool.commonPool()` **or via parallel Streams instead of using a custom thread pool** (e. g. created by one of the static factory methods from `ExecutorServices`)? Unless the custom thread pool is configured with a `ThreadFactory` that specifies a non-default priority for threads or a custom exception handler (see item 10.1) there is little reason to create more threads in the system instead of reusing threads of the common `ForkJoinPool`.

## 11. Thread interruption and `Future` cancellation

11.1. If some code propagates `InterruptedException` wrapped into another exception (e. g. `RuntimeException`), is **the interruption status of the current thread restored before the wrapping exception is thrown?**

Propagating `InterruptedException` wrapped into another exception is a controversial practice (especially in libraries) and it may be prohibited in some projects completely, or in specific subsystems.

11.2. If some method **returns normally after catching an** `InterruptedException`, is this coherent with the (documented) semantics of the usually makes sense only in two types of methods:

- `Runnable.run()` or `Callable.call()` themselves, or methods that are intended to be submitted as tasks to some Executors as method references. `Thread.currentThread().interrupt()` should still be called before returning from the method, assuming that the interruption policy of the threads in the `Executor` is unknown.
- Methods with “try” or “best effort” semantics. Documentation for such methods should be clear that they stop attempting to do something when the thread is interrupted, restore the interruption status of the thread and return.

If a method doesn’t fall into either of these categories, it should propagate `InterruptedException` directly or wrapped into another exception (see the previous item), or it should not return normally after catching an `InterruptedException`, but rather continue execution in some sort of retry loop, saving the interruption status and restoring it before returning (see an example from JCIP). Fortunately, in most situations, it’s not needed to write such boilerplate code: **one of the methods from `Uninterruptibles` utility class from Guava can be used.**

11.3. If an **`InterruptedException` or a** `TimeoutException` **is caught on a** `Future.get()` **call** and the task behind the future doesn’t have side effects, i. e. `get()` is called only to obtain and use the result in the context of the current thread rather than achieve some side effect, is the future canceled?

task cancellation.

## 12. Time

12.1. Are values returned from `System.nanoTime()` **compared in an overflow-aware manner**, as described in the documentation for this method?

12.2. Does the code that compares values returned from `System.currentTimeMillis()` **have precautions against "time going backward"**? This might happen due to time correction on a server. Values that are returned from `currentTimeMillis()` that are less than some other values that have already been seen should be ignored. Otherwise, there should be comments explaining why this issue is not relevant for the code.

Alternatively, `System.nanoTime()` could be used instead of `currentTimeMillis()`. Values returned from `nanoTime()` never decrease (but may overflow — see the previous item). Warning: `nanoTime()` didn't always uphold to this guarantee in OpenJDK until 8u192 (see JDK-8184271). Make sure to use the freshest distribution.

In distributed systems, the leap second adjustment causes similar issues.

12.3. Do **variables that store time limits and periods have suffixes identifying their units**, for example, "timeoutMillis" (also -Seconds, -Micros, -Nanos) rather than just "timeout"? In method and constructor parameters, an alternative is providing a `TimeUnit` parameter next to a "timeout" parameter. This is the preferred option for public APIs.

Donate

**negative arguments as zeros?** This is to obey the principle of least astonishment because all timed blocking methods in classes from `java.util.concurrent.*` follow this convention.

## 13. Thread safety of Cleaners and native code

13.1. If a class manages native resources and employs `java.lang.ref.Cleaner` (or `sun.misc.Cleaner`; or overrides `Object.finalize()`) to ensure that resources are freed when objects of the class are garbage collected, and the class implements `Closeable` with the same cleanup logic executed from `close()` directly rather than through `Cleanable.clean()` (or `sun.misc.Cleaner.clean()`) to be able to distinguish between explicit `close()` and cleanup through a cleaner (for example, `clean()` can log a warning about the object not being closed explicitly before freeing the resources), is it ensured that even if the **cleanup logic is called concurrently from multiple threads, the actual cleanup is performed only once**? The cleanup logic in such classes should obviously be idempotent because it's usually expected to be called twice: the first time from the `close()` method and the second time from the cleaner or `finalize()`. The catch is that the cleanup *must be concurrently idempotent, even if `close()` is never called concurrently on objects of the class*. That's because the garbage collector may consider the object to become unreachable before the end of a `close()` call and initiate cleanup through the cleaner or `finalize()` while `close()` is still being executed.

Alternatively, `close()` could simply delegate to `Cleanable.clean()` (`sun.misc.Cleaner.clean()`) which is thread-safe. But then it's impossible to distinguish between explicit and automatic cleanup.

13.2. In a class with some native state that has a cleaner or overrides `finalize()`, are **bodies of all methods that interact with the native state wrapped with** `try { ... } finally { Reference.reachabilityFence(this); }`, including constructors and the `close()` method, but excluding `finalize()`? This is needed because an object could become unreachable and the native memory might be freed from the cleaner while the method that interacts with the native state is being executed, that might lead to use-after-free or JVM memory corruption.

`reachabilityFence()` in `close()` also eliminates the race between `close()` and the cleanup executed through the cleaner or `finalize()` (see the previous item), but it may be a good idea to retain the thread safety precautions in the cleanup procedure, especially if the class in question belongs to the public API of the project because otherwise if `close()` is accidentally or maliciously called concurrently from multiple threads, the JVM might crash due to double memory free or, worse, memory might be silently corrupted, while the promise of the Java platform is that whatever buggy some code is, as long as it passes bytecode verification, thrown exceptions should be the worst possible outcome, but the virtual machine shouldn't crash. Item 13.4 also stresses on this principle.

`Reference.reachabilityFence()` has been added in JDK 9. If the project targets JDK 8 and Hotspot JVM, any method with an empty body is an effective emulation of `reachabilityFence()`.

See the documentation for `Reference.reachabilityFence()` and this discussion in the concurrency-interest mailing list for more information.

**not to free native resources**, but merely to return heap objects to some pools, or merely to report that some heap objects are not returned to some pools? This is an antipattern because of the tremendous complexity of using cleaners and `finalize()` correctly (see the previous two items) and the negative impact on performance (especially of `finalize()`), that might be even larger than the impact of not returning objects back to some pool and thus slightly increasing the garbage allocation rate in the application. If the latter issue arises to be any important, it should better be diagnosed with async-profiler in the allocation profiling mode ( `-e alloc` ) than by registering cleaners or overriding `finalize()`.

This advice also applies when pooled objects are direct ByteBuffers or other Java wrappers of native memory chunks. `async-profiler -e malloc` could be used in such cases to detect direct memory leaks.

13.4. If some **classes have some state in native memory and are used actively in concurrent code, or belong to the public API of the project, was it considered making them thread-safe**? As described in item 13.2, if objects of such classes are inadvertently accessed from multiple threads without proper synchronization, memory corruption and JVM crashes might result. This is why classes in the JDK such as `java.util.zip.Deflater` use synchronization internally despite `Deflater` objects are not intended to be used concurrently from multiple threads.

Note that making classes with some state in native memory thread-safe also implies that the **native state should be safely published in constructors.** This means that either the native state should be stored exclusively in `final` fields, or `VarHandle.storeStoreFence()` should be called in constructors after full initialization of the native state. If the project targets JDK 9 and `VarHandle` is not available, the same `nized (this) { ... }`.

## 14. Parallel Streams

14.1. For every use of parallel Streams via `Collection.parallelStream()` or `Stream.parallel()`: **is it explained why parallel `Stream` is used in a comment preceding the stream operation?** Are there back-of-the-envelope calculations or references to benchmarks showing that the total CPU time cost of the parallelized computation exceeds 100 microseconds?

Is there a note in the comment that parallelized operations are generally I/O-free and non-blocking, as per item 10.4? The latter might be obvious momentary, but as codebase evolves the logic that is called from the parallel stream operation might become blocking accidentally. Without comment, it's harder to notice the discrepancy and the fact that the computation is no longer a good fit for parallel Streams. It can be fixed by calling the non-blocking version of the logic again or by using a simple sequential `Stream` instead of a parallel `Stream`.

Bonus: is forbidden-apis configured for the project and are `java.util.StringBuffer`, `java.util.Random` and `Math.random()` prohibited? `StringBuffer` and `Random` are thread-safe and all their methods are `synchronized`, which is never useful in practice and only inhibits the performance. In OpenJDK, `Math.random()` delegates to a global static `Random` instance. `StringBuilder` should be used instead of `StringBuffer`, `ThreadLocalRandom` or `SplittableRandom` should be used instead of `Random`.

## Reading List

field semantics.

- [EJ] "Effective Java" by Joshua Bloch, Chapter 11. Concurrency.
- [JCIP] "Java Concurrency in Practice" by Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea.
- Posts by Aleksey Shipilёv:
  Safe Publication and Safe Initialization in Java
  Java Memory Model Pragmatics
  Close Encounters of The Java Memory Model Kind
- When to use parallel streams written by Doug Lea, with the help of Brian Goetz, Paul Sandoz, Aleksey Shipilev, Heinz Kabutz, Joe Bowbeer, …

Donate

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

**You can make a tax-deductible donation here.**

**Our Nonprofit**

About

Alumni Network

Open Source

Shop

Support

Sponsors

Academic Honesty

Code of Conduct

Privacy Policy

Terms of Service

Copyright Policy

**Trending Guides**

2019 Web Developer Roadmap

Python Tutorial

CSS Flexbox Guide

JavaScript Tutorial

Python Example

HTML Tutorial

Linux Command Line Guide

JavaScript Example

Git Tutorial

React Tutorial

Java Tutorial

Linux Tutorial

CSS Tutorial

jQuery Example

SQL Tutorial

CSS Example

React Example

Angular Tutorial

Bootstrap Example

How to Set Up SSH Keys

WordPress Tutorial

PHP Example

Donate