Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds

Mailing lists
Wiki · IRC

Bylaws · Census
Legal

Workshop

JEP Process

Source code
Mercurial
GitHub

Tools
Git
jtreg harness

Groups
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web

Projects
(overview, archive)
Amber
Babylon
CRaC
Caciocavallo
Closures
Code Tools
Coin
Common VM Interface
Compiler Grammar
Detroit
Developers' Guide
Device I/O
Duke
Galahad
Graal
IcedTea
JDK 7
JDK 8
JDK 8 Updates
JDK 9
JDK (..., 21, 22, 23)
JDK Updates
JavaDoc.Next
Jigsaw
Kona
Kulla
Lambda
Lanai
Leyden
Lilliput
Locale Enhancement
Loom
Memory Model Update
Metropolis
Mission Control
Multi-Language VM
Nashorn
New I/O
OpenJFX
Panama
Penrose
Port: AArch32
Port: AArch64
Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: Mobile
Port: PowerPC/AIX
Port: RISC-V
Port: s390x
Portola
SCTP
Shenandoah
Skara
Sumatra
Tiered Attribution
Tsan
Type Annotations
Valhalla
Verona
VisualVM
Wakefield
Zero
ZGC

JEP 421: Deprecate Finalization for Removal

Authors	Brent Christian, Stuart Marks
Owner	Brent Christian
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	18
Component	core-libs / java.lang
Discussion	core dash libs dash dev at openjdk dot java dot net
Effort	S
Duration	S
Reviewed by	Alex Buckley, Brian Goetz, Kim Barrett
Endorsed by	Brian Goetz, Mikael Vidstedt
Created	2021/09/30 20:24
Updated	2022/09/02 17:48
Issue	8274609

Summary

Deprecate finalization for removal in a future release. Finalization remains enabled by default for now, but can be disabled to facilitate early testing. In a future release it will be disabled by default, and in a later release it will be removed. Maintainers of libraries and applications that rely upon finalization should consider migrating to other resource management techniques such as the `try`-with-resources statement and cleaners.

Goals

- Help developers understand the dangers of finalization.
- Prepare developers for the removal of finalization in a future version of Java.
- Provide simple tooling to help detect reliance upon finalization.

Motivation

Resource leaks

Java programs enjoy automatic memory management, wherein the JVM's garbage collector (GC) reclaims the memory used by an object when the object is no longer needed. However, some objects represent a resource provided by the operating system, such as an open file descriptor or a block of native memory. For such objects, it is insufficient merely to reclaim the object's memory; the program must also release the underlying resource back to the operating system, typically by calling the object's `close` method. If the program fails to do this before the GC reclaims the object then the information needed to release the resource is lost. The resource, still considered by the operating system to be in use, has *leaked*.

Resource leaks can be surprisingly common. Consider the following code that copies data from one file to another. In early versions of Java, developers typically used the `try-finally` construct to ensure that resources were released even if an exception occurred while copying:

```
FileInputStream  input  = null;
FileOutputStream output = null;
try {
    input  = new FileInputStream(file1);
    output = new FileOutputStream(file2);
    ... copy bytes from input to output ...
    output.close();  output = null;
    input.close();   input  = null;
} finally {
    if (output != null) output.close();
    if (input  != null) input.close();
}
```

This code is erroneous: If copying throws an exception, and if the `output.close()` statement in the `finally` block throws an exception, then the input stream will be leaked. Handling exceptions from all possible execution paths is laborious and difficult to get right. (The fix here involves a nested `try-finally` construct, and is left as an exercise for the reader.) Even if an unhandled exception occurs only occasionally, leaked resources can build up over time.

Finalization — and its flaws

Finalization, introduced in Java 1.0, was intended to help avoid resource leaks. A class can declare a *finalizer* — the method `protected void finalize()` — whose body releases any underlying resources. The GC will schedule the finalizer of an unreachable object to be called before it reclaims the object's memory; in turn, the `finalize` method can take actions such as calling the object's `close` method.

At first glance, this seems like an effective safety net for preventing resource leaks: If an object containing a still-open resource becomes unreachable (the `input` object above) then the GC will schedule the finalizer, which will close the resource. In effect, finalization appropriates the power of garbage collection to manage non-memory resources (Barry Hayes, *Finalization in the Collector Interface*, International Workshop on Memory Management, 1992).

Unfortunately, finalization has several critical, fundamental flaws:

- *Unpredictable latency* — An arbitrarily long time may pass between the moment an object becomes unreachable and the moment its finalizer is called. In fact, the GC provides no guarantee that any finalizer will ever be called.
- *Unconstrained behavior* — Finalizer code can take any action. In particular, it can save a reference to the object being finalized, thereby *resurrecting* the object and making it reachable once again.
- *Always enabled* — Finalization has no explicit registration mechanism. A class with a finalizer enables finalization for every instance of the class, whether needed or not. Finalization of an object cannot be cancelled, even if it is no longer necessary for that object.
- *Unspecified threading* — Finalizers run on unspecified threads, in an arbitrary order. Neither threading nor ordering can be controlled.

These flaws were widely recognized over twenty years ago. Advice to be careful with Java finalization appeared as early as 1998 (Bill Venners, *Object Finalization and Cleanup: How to Design Classes for Proper Object Cleanup*, JavaWorld, May 1998) and featured prominently in Joshua Bloch's 2001 book *Effective Java* (Item 6: "Avoid finalizers"). The *SEI CERT Oracle Coding Standard for Java* has recommended against the use of finalizers since 2008.

Real-world consequences

The flaws of finalization combine to cause significant real-world problems in security, performance, reliability, and maintainability.

- *Security vulnerabilities* — If a class has a finalizer then a new instance of that class is eligible for finalization as soon as its constructor begins to execute. If the constructor throws an exception then the new instance is not destroyed, even though it might not be completely initialized. The new instance remains eligible for finalization and its finalizer can perform arbitrary actions on the object, including resurrecting it for later use. Malicious code can use this technique to produce ill-formed objects that cause unexpected errors, or to confuse otherwise correct code into misbehaving. (The same vulnerabilities apply to objects created by deserialization.)

  Simply omitting a finalizer from a class does not prevent this problem. A subclass can declare a finalizer and thereby gain access to invalidly constructed or deserialized objects. Mitigating this problem requires additional steps that would not be necessary if finalization were not part of the platform. This problem and techniques for mitigating it are described in the *Oracle Secure Coding Guidelines for Java SE* (see 4-5, "Limit the extensibility of classes and methods" and 7-3, "Defend against partially initialized instances of non-final classes").

- *Performance* — The mere presence of finalizers imposes a performance penalty: The GC must do extra work when objects are created, and also before and after finalizing them. For example, Hans Boehm described a 7-11x slowdown when adding finalization to a class. Finalization also leads to increased pause times for throughput-oriented collectors, and increased data structure overhead for low-latency collectors.

  Some classes provide an explicit method to release resources, such as `close`, as well as a finalizer, just to be safe. If the user forgets to call `close` then the finalizer can release the resource. However, since finalizers do not support cancellation, the performance penalty is always paid, even for unnecessary finalizers for already-released resources.

- *Unreliable execution* — Applications that use finalizers are at higher risk of intermittent and hard-to-diagnose failures. Finalizers are scheduled to run by the GC, but the GC typically operates only when necessary to satisfy memory allocation requests. If free memory is plentiful then the GC might run infrequently, inducing an arbitrary delay on finalization. When many resource-bearing objects accumulate on the heap awaiting finalization, the result can be a resource shortage that unpredictably breaks the application. In addition, finalization runs on an unspecified number of threads, so application threads might allocate resources more quickly than they can be released by finalizer threads, again leading to a resource shortage.

- *Difficult programming model* — Finalizers are surprisingly hard to implement correctly. As a rule, the finalizer of a class must call the finalizer of its superclass, because failing to do so may cause resource leaks. The developer is responsible for remembering to call `super.finalize()` and handle any exceptions. The Java compiler does not automatically insert this call into a finalizer, in contrast to how it automatically inserts a `super(...)` call into a constructor.

  Ensuring the correctness of finalizers in your own code is insufficient to prevent problems. Other components could subclass your code and override your finalizers. An incorrect finalizer implementation on their part could effectively break previously correct code.

  Finalizers execute on one or more system threads that are not known to the application. Thus, the presence of a finalizer in an otherwise single-threaded application inherently makes it multi-threaded. This introduces the potential for deadlocks and other threading problems.

  Ultimately, finalizers increase coupling in the architecture of an application. When several components in an application have finalizers, the finalization of one component's objects might delay or otherwise interfere with the finalization of another component's objects, especially since finalizer threads are shared across components. Such interference might result in a particular kind of finalizable object accumulating on the heap, resulting in shortages of one component's resources, as described above, and eventually in deleterious effects on other components.

Alternative techniques

Given the problems associated with finalization, developers should use alternative techniques to avoid resource leaks, namely `try`-with-resources and cleaners.

- *Try-with-resources* — Java 7 introduced the `try`-with-resources statement as an improvement on the `try-finally` construct shown above. This statement allows resources to be used in such a way that their `close` methods are guaranteed to be called, regardless of whether exceptions occur. The earlier example can be rewritten as follows:

  ```
  try (FileInputStream input = new FileInputStream(file1);
       FileOutputStream output = new FileOutputStream(file2)) {
      ... copy bytes from input to output ...
  }
  ```

  `try`-with-resources handles all exceptional cases properly, avoiding the need for the safety net of finalization. Any resource opened and closed within a single lexical scope should be converted to work with `try`-with-resources. If the instances of a class with a finalizer can be used exclusively within `try`-with-resources statements then the finalizer is likely unnecessary, and can be removed.

- *Cleaners* — Some resources are too long-lived to play well with `try`-with-resources, so Java 9 introduced the *cleaner* API to help release them. The cleaner API allows a program to register a *cleaning action* for an object that is run some time after the object becomes unreachable. Cleaning actions avoid many of the drawbacks of finalizers:

  - *No object resurrection* — Cleaning actions cannot access the object, so object resurrection is impossible.
  - *Enabled on-demand* — A constructor can register a cleaning action for a new object after the object is fully initialized. This means that a cleaning action never processes an uninitialized or partially initialized object. In addition, a program can cancel an object's cleaning action so that the GC no longer needs to schedule the action.
  - *No interference* — The developer can control which threads run cleaning actions, and so can prevent interference between cleaning actions. In addition, an erroneous or malicious subclass cannot interfere with cleaning actions set up by its superclass.

  However, like finalizers, cleaning actions are scheduled by the GC, so they may suffer from unbounded delays. Thus the cleaner API should not be used in situations where the timely release of a resource is required. In addition, cleaners should not be used to replace a finalizer that serves only as a safety net to protect against uncaught exceptions or missing calls of `close()` methods; in such cases, investigate using `try`-with-resources before converting the finalizer to a cleaner.

  One scenario in which cleaners, despite their latency, are valuable to advanced developers, is in implementing APIs that do not permit explicit `close()` methods. Consider a version of the `BigInteger` class that uses native memory in its underlying implementation. Adding a `close()` method to the `BigInteger` class would fundamentally change its programming model and would preclude certain optimizations. Since user code cannot `close` a `BigInteger`, the implementor must rely on the GC to schedule a cleaning action to release the native memory. The implementor can balance the interests of developers, who benefit from a simpler API, against runtime overhead. (The incubating Foreign Function & Memory API (JEP 419), which provides a better way to access native memory, supports the use of cleaners to deallocate native memory and avoid resource leaks.)

Summary

Finalization has serious flaws that have been widely recognized for decades. Its presence in the Java Platform burdens the entire ecosystem because it exposes all library and application code to security, reliability, and performance risks. It also imposes ongoing maintenance and development costs on the JDK, particularly on the GC implementations. To move the Java Platform forward, we will deprecate finalization for removal.

Description

We propose to:

- Add a command-line option to disable finalization, so that no finalizers are ever scheduled by the GC to run, and
- Deprecate all finalizers, and finalization-related methods, in the standard Java API.

Note that finalization is distinct from both the `final` modifier and the `finally` block of the `try-finally` construct. No changes are proposed to either `final` or `try-finally`.

Command-line option to disable finalization

Finalization remains enabled by default in JDK 18. A new command-line option `--finalization=disabled` disables finalization. A JVM launched with `--finalization=disabled` will not run any finalizers — not even those declared within the JDK itself.

You can use the option to help determine if your application relies on finalization, and to test how it will behave once finalization is removed. For example, you could first run an application load test *without* the option, so that finalization is enabled, and record metrics such as:

- Memory profiles for Java heap and/or native memory,
- Statistics from `BufferPoolMXBean` and `UnixOperatingSystemMXBean::getOpenFileDescriptorCount`, and
- The `jdk.FinalizerStatistics` event from JDK Flight Recorder (JFR). This event provides data about finalizer use at run time, as discussed in the JDK 18 release note. JFR can be used as follows:
  `java -XX:StartFlightRecording:filename=recording.jfr ...`
  `jfr print --events FinalizerStatistics recording.jfr`

You could then rerun the load test *with* the option, so that finalization is disabled. A significant degradation in the reported metrics, or an error or a crash, would indicate a need to investigate where the application relies on finalization. Substantially similar results between runs would provide some assurance that the application will not be impacted by the eventual removal of finalization.

Disabling finalization may have unpredictable consequences, so you should use the option only for testing and not in a production environment.

If finalization is disabled, JFR will not emit any `jdk.FinalizerStatistics` events. Also, `jcmd GC.finalizer_info` will report that finalization is disabled (instead of reporting the number of objects pending finalization).

For completeness, `--finalization=enabled` is supported.

Deprecating finalizers in the standard Java API

We will terminally deprecate these methods in the `java.base` and `java.desktop` modules, by annotating them with `@Deprecated(forRemoval=true)`:

- `java.lang.Object.finalize()`
- `java.lang.Enum.finalize()`
- `java.awt.Graphics.finalize()`
- `java.awt.PrintJob.finalize()`
- `java.util.concurrent.ThreadPoolExecutor.finalize()`
- `javax.imageio.spi.ServiceRegistry.finalize()`
- `javax.imageio.stream.FileCacheImageInputStream.finalize()`
- `javax.imageio.stream.FileImageInputStream.finalize()`
- `javax.imageio.stream.FileImageOutputStream.finalize()`
- `javax.imageio.stream.ImageInputStreamImpl.finalize()`
- `javax.imageio.stream.MemoryCacheImageInputStream.finalize()`

(Three other finalizers in `java.awt.**` were already terminally deprecated, and were removed from Java 18 independently of this JEP.)

In addition, we will:

- Terminally deprecate `java.lang.Runtime.runFinalization()` and `java.lang.System.runFinalization()`. These methods serve no purpose without finalization.
- Deprecate the `getObjectPendingFinalizationCount()` method in the `java.lang.management.MemoryMXBean` interface of the `java.management` module, by annotating it with `@Deprecated(forRemoval=false)`.

  This method is not part of the finalization mechanism, but rather queries the operation of the mechanism. We will deprecate this method because developers should avoid using it: Once finalization is removed, no objects will ever be pending and the method will always return zero. We will not terminally deprecate the method because we do not plan to remove it. `MemoryMXBean` is an interface, so removing a method could adversely impact a wide variety of independent implementations.

Future Work

We expect a lengthy transition period before removing finalization. This will provide time for developers to assess whether their systems rely upon finalization and to migrate their code as necessary. We also envision several other steps:

- The JDK itself makes significant use of finalizers. Some have already been removed or converted to use `Cleaner`. JDK-8253568 tracks the removal of the remaining finalizers.
- Well-known libraries such as Netty, Log4j, Guava, and Apache Commons use finalizers. We will work with their maintainers to ensure that these libraries can safely migrate away from finalization.
- We will publish documentation to help developers with several aspects of migrating away from finalization, such as:
  - How to find finalizers in library and application code (for example, using `jdeprscan`),
  - How to determine if a system relies on finalization, and
  - How to convert code with finalizers to use `try`-with-resources or cleaners.
- Future versions of the JDK may include some or all of the following changes:
  - Issue a warning at runtime when finalizers are in use,
  - Disable finalization by default, and require the `--finalization=enabled` option to re-enable it,
  - Later, remove the finalization mechanism, while leaving non-functioning APIs in place, and,
  - Finally, after most code has been migrated, remove the above terminally deprecated methods, including `Object::finalize`.

We do not plan to revisit the historically distinct roles played by `WeakReference` and `PhantomReference`. Conversely, we expect to update the Java Language Specification when we remove finalization, since finalizers interact with the Java Memory Model.