MET12-J. Do not use finalizers

The garbage collector invokes object finalizer methods after it determines that the object is unreachable but before it reclaims the object's storage. Execution of the finalizer provides an opportunity to release resources such as open streams, files, and network connections that might not otherwise be released automatically through the normal action of the garbage collector.

A sufficient number of problems are associated with finalizers to restrict their use to exceptional conditions:

- There is no fixed time at which finalizers must be executed because time of execution depends on the Java Virtual Machine (JVM). The only guarantee is that any finalizer method that executes will do so sometime after the associated object has become unreachable (detected during the first cycle of garbage collection) and sometime before the garbage collector reclaims the associated object's storage (during the garbage collector's second cycle). Execution of an object's finalizer may be delayed for an *arbitrarily* long time after the object becomes unreachable. Consequently, invoking time-critical functionality such as closing file handles in an object's `finalize()` method is problematic.

- The JVM may terminate without invoking the finalizer on some or all unreachable objects. Consequently, attempts to update critical persistent state from finalizer methods can fail without warning. Similarly, Java lacks any guarantee that finalizers will execute on process termination. Methods such as `System.gc()`, `System.runFinalization()`, `System.runFinalizersOnExit()`, and `Runtime.runFinalizersOnExit()` either lack such guarantees or have been deprecated because of lack of safety and potential for deadlock.

- According to *The Java Language Specification* (JLS), §12.6, "Finalization of Class Instances" [JLS 2015]:

  > *The Java programming language imposes no ordering on `finalize()` method calls. Finalizers [of different objects] may be called in any order, or even concurrently.*

  One consequence is that slow-running finalizers can delay execution of other finalizers in the queue. Further, the lack of guaranteed ordering can lead to substantial difficulty in maintaining desired program invariants.

- Uncaught exceptions thrown during finalization are ignored. When an exception thrown in a finalizer propagates beyond the `finalize()` method, the process itself immediately stops and consequently fails to accomplish its sole purpose. This termination of the finalization process may or may not prevent all subsequent finalization from executing. The JLS fails to define this behavior, leaving it to the individual implementations.

- Coding errors that result in memory leaks can cause objects to incorrectly remain reachable; consequently, their finalizers are never invoked.

- A programmer can unintentionally resurrect an object's reference in the `finalize()` method. When this occurs, the garbage collector must determine yet again whether the object is free to be deallocated. Further, because the `finalize()` method has executed once, the garbage collector cannot invoke it a second time.

- Garbage collection usually depends on memory availability and usage rather than on the scarcity of some other particular resource. Consequently, when memory is readily available, a scarce resource may be exhausted in spite of the presence of a finalizer that could release the scarce resource if it were executed. See FIO04-J. Release resources when they are no longer needed and TPS00-J. Use thread pools to enable graceful degradation of service during traffic bursts for more details on handling scarce resources correctly.

- It is a common myth that finalizers aid garbage collection. On the contrary, they increase garbage-collection time and introduce space overheads. Finalizers interfere with the operation of modern generational garbage collectors by extending the lifetimes of many objects. Incorrectly programmed finalizers could also attempt to finalize reachable objects, which is always counterproductive and can violate program invariants.

- Use of finalizers can introduce synchronization issues even when the remainder of the program is single-threaded. The `finalize()` methods are invoked by the garbage collector from one or more threads of its choice; these threads are typically distinct from the `main()` thread, although this property is not guaranteed. When a finalizer is necessary, any required cleanup data structures must be protected from concurrent access. See the JavaOne presentation by Hans J. Boehm [Boehm 2005] for additional information.

- Use of locks or other synchronization-based mechanisms within a finalizer can cause deadlock or starvation. This possibility arises because neither the invocation order nor the specific executing thread or threads for finalizers can be guaranteed or controlled.

Object finalizers have also been deprecated since Java 9. See MET02-J. Do not use deprecated or obsolete classes or methods for more information.

Because of these problems, finalizers must not be used in new classes.

Noncompliant Code Example (Superclass's finalizer)

Superclasses that use finalizers impose additional constraints on their extending classes. Consider an example from JDK 1.5 and earlier. The following noncompliant code example allocates a 16 MB buffer used to back a Swing `JFrame` object. Although the `JFrame` APIs lack `finalize()` methods, `JFram` extends `AWT.Frame`, which does have a `finalize()` method. When a `MyFrame` object becomes unreachable, the garbage collector cannot reclaim the storage for the byte buffer because code in the inherited `finalize()` method might refer to it. Consequently, the byte buffer must persist *at least* until the inherited `finalize()` method for class `MyFrame` completes its execution and cannot be reclaimed until the following garbage-collection cycle.

```
class MyFrame extends JFrame {
  private byte[] buffer = new byte[16 * 1024 * 1024];
  // Persists for at least two GC cycles
}
```

Compliant Solution (Superclass's finalizer)

When a superclass defines a `finalize()` method, make sure to decouple the objects that can be immediately garbage collected from those that must depend on the finalizer. This compliant solution ensures that the `buffer` can be reclaimed as soon as the object becomes unreachable.

```
class MyFrame {
  private JFrame frame;
  private byte[] buffer = new byte[16 * 1024 * 1024]; // Now decoupled
}
```

Noncompliant Code Example (`System.runFinalizersOnExit()`)

This noncompliant code example uses the `System.runFinalizersOnExit()` method to simulate a garbage-collection run. Note that this method is deprecated because of thread-safety issues.

According to the Java API [API 2014] class `System`, `runFinalizersOnExit()` method documentation,

> *Enable or disable finalization on exit; doing so specifies that the finalizers of all objects that have finalizers that have not yet been automatically invoked are to be run before the Java runtime exits. By default, finalization on exit is disabled.*

The class `SubClass` overrides the `protected finalize()` method and performs cleanup activities. Subsequently, it calls `super.finalize()` to make sure its superclass is also finalized. The unsuspecting `BaseClass` calls the `doLogic()` method, which happens to be overridden in the `SubClass`. This resurrects a reference to `SubClass` that not only prevents it from being garbage-collected but also prevents it from calling its finalizer to close new resources that may have been allocated by the called method. As detailed in MET05-J. Ensure that constructors do not call overridable methods, if the subclass's finalizer has terminated key resources, invoking its methods from the superclass might result in the observation of an object in an inconsistent state. In some cases, this can result in `NullPointerException`.

```
class BaseClass {
  protected void finalize() throws Throwable {
    System.out.println("Superclass finalize!");
    doLogic();
  }

  public void doLogic() throws Throwable {
    System.out.println("This is super-class!");
  }
}

class SubClass extends BaseClass {
  private Date d; // Mutable instance field

  protected SubClass() {
    d = new Date();
  }

  protected void finalize() throws Throwable {
    System.out.println("Subclass finalize!");
    try {
      //  Cleanup resources
      d = null;
    } finally {
      super.finalize();  // Call BaseClass's finalizer
    }
  }

  public void doLogic() throws Throwable {
    // Any resource allocations made here will persist

    // Inconsistent object state
    System.out.println(
        "This is sub-class! The date object is: " + d);
    // 'd' is already null
  }
}

public class BadUse {
  public static void main(String[] args) {
    try {
      BaseClass bc = new SubClass();
      // Artificially simulate finalization (do not do this)
      System.runFinalizersOnExit(true);
    } catch (Throwable t) {
      // Handle error
    }
  }
}
```

This code outputs:

```
Subclass finalize!
Superclass finalize!
This is sub-class! The date object is: null
```

Compliant Solution

Joshua Bloch [Bloch 2008] suggests implementing a `stop()` method explicitly such that it leaves the class in an unusable state beyond its lifetime. A private field within the class can signal whether the class is unusable. All the class methods must check this field prior to operating on the class. This is akin to the "initialized flag"–compliant solution discussed in OBJ11-J. Be wary of letting constructors throw exceptions. As always, a good place to call the termination logic is in the `finally` block.

Exceptions

MET12-J-EX0: Finalizers may be used when working with native code because the garbage collector cannot reclaim memory used by code written in another language and because the lifetime of the object is often unknown. Again, the native process must not perform any critical jobs that require immediate resource deallocation.

Any subclass that overrides `finalize()` must explicitly invoke the method for its superclass as well. There is no automatic *chaining* with `finalize`. The correct way to handle it is as follows:

```
protected void finalize() throws Throwable {
  try {
    //...
  } finally {
    super.finalize();
  }
}
```

A more expensive solution is to declare an anonymous class so that the `finalize()` method is guaranteed to run for the superclass. This solution is applicable to public nonfinal classes. "The finalizer guardian forces `super.finalize` to be called if a subclass overrides `finalize()` and does not explicitly call `super.finalize`" [JLS 2015].

```
public class Foo {
  // The finalizeGuardian object finalizes the outer Foo object
  private final Object finalizerGuardian = new Object() {
    protected void finalize() throws Throwable {
      // Finalize outer Foo object
    }
  };
  //...
}
```

The ordering problem can be dangerous when dealing with native code. For example, if object `A` references object `B` (either directly or reflectively) and the latter gets finalized first, `A`'s finalizer may end up dereferencing dangling native pointers. To impose an explicit ordering on finalizers, make sure that `B` remains reachable until `A`'s finalizer has concluded. This can be achieved by adding a reference to `B` in some global state variable and removing it when `A`'s finalizer executes. An alternative is to use the `java.lang.ref` references.

MET12-J-EX1: A class may use an empty final finalizer to prevent a finalizer attack, as specified in OBJ11-J. Be wary of letting constructors throw exceptions.

Risk Assessment

Improper use of finalizers can result in resurrection of garbage-collection-ready objects and result in denial-of-service vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MET12-J	Medium	Probable	Medium	P8	L2

Automated Detection

Tool	Version	Checker	Description
CodeSonar	4.2	**FB.BAD_PRACTICE.FI_EMPTY** **FB.BAD_PRACTICE.FI_EXPLICIT_INVOCATION** **FB.BAD_PRACTICE.FI_FINALIZER_NULLS_FIELDS** **FB.BAD_PRACTICE.FI_FINALIZER_ONLY_NULLS_FIELDS** **FB.BAD_PRACTICE.FI_MISSING_SUPER_CALL** **FB.BAD_PRACTICE.FI_NULLIFY_SUPER** **FB.MALICIOUS_CODE.** **FI_PUBLIC_SHOULD_BE_PROTECTED** **FB.BAD_PRACTICE.FI_USELESS**	Empty finalizer should be deleted Explicit invocation of finalizer Finalizer nulls fields Finalizer nulls fields Finalizer does not call superclass finalizer Finalizer nullifies superclass finalizer Finalizer should be protected, not public Finalizer does nothing but call superclass finalizer
Coverity	7.5	**CALL_SUPER** **DC.THREADING** **FB.FI_EMPTY** **FB.FI_EXPLICIT_INVOCATION** **FB.FI_FINALIZER_NULLS_FIELDS** **FB.FI_FINALIZER_ONLY_NULLS_FIELDS** **FB.FI_MISSING_SUPER_CALL** **FB.FI_NULLIFY_SUPER** **FB.FI_USELESS** **FB.FI_PUBLIC_SHOULD_BE_ PROTECTED**	Implemented

Parasoft Jtest	10.3	**EJB.MNDF, GC.FCF, GC.FM, GC.IFF, GC.NCF, PB.API.OF,UC.EF, UC.FCSF**	
SonarQube	6.7	**S1113** **S1111** **S1174** **S2151** **S1114**	**The Object.finalize() method should not be overriden** **The Object.finalize() method should not be called** **"Object.finalize()" should remain protected (versus public) when overriding** **"runFinalizersOnExit" should not be called** **"super.finalize()" should be called at the end of "Object.finalize()" implementations**

Related Vulnerabilities

AXIS2-4163 describes a vulnerability in the `finalize()` method in the Axis web services framework. The finalizer incorrectly calls `super.finalize()` before doing its own cleanup, leading to errors in `GlassFish` when the garbage collector runs.

Related Guidelines

MITRE CWE	CWE-586, Explicit call to `Finalize()` CWE-583, `finalize()` Method Declared Public CWE-568, `finalize()` Method without `super.finalize()`

Bibliography

[API 2014]	Class `System` `finalize()`
[Bloch 2008]	Item 7, "Avoid Finalizers"
[Boehm 2005]	
[Coomes 2007]	"'Sneaky' Memory Retention"
[Darwin 2004]	Section 9.5, "The Finalize Method"
[Flanagan 2005]	Section 3.3, "Destroying and Finalizing Objects"
[JLS 2015]	§12.6, "Finalization of Class Instances"