FIO14-J. Perform proper cleanup at program termination

When certain kinds of errors are detected, such as irrecoverable logic errors, rather than risk data corruption by continuing to execute in an indeterminate state, the appropriate strategy may be for the system to quickly shut down, allowing the operator to start it afresh in a determinate state. Section 6.46, "Termination Strategy [REU]," [ISO/IEC TR 24772:2010] says:

> *When a fault is detected, there are many ways in which a system can react. The quickest and most noticeable way is to fail hard, also known as fail fast or fail stop. The reaction to a detected fault is to immediately halt the system. Alternatively, the reaction to a detected fault could be to fail soft. The system would keep working with the faults present, but the performance of the system would be degraded. Systems used in a high availability environment such as telephone switching centers, e-commerce, or other "always available" applications would likely use a fail soft approach. What is actually done in a fail soft approach can vary depending on whether the system is used for safety critical or security critical purposes. For fail-safe systems, such as flight controllers, traffic signals, or medical monitoring systems, there would be no effort to meet normal operational requirements, but rather to limit the damage or danger caused by the fault. A system that fails securely, such as cryptologic systems, would maintain maximum security when a fault is detected, possibly through a denial of service.*

And:

> *The reaction to a fault in a system can depend on the criticality of the part in which the fault originates. When a program consists of several tasks, each task may be critical, or not. If a task is critical, it may or may not be restartable by the rest of the program. Ideally, a task that detects a fault within itself should be able to halt leaving its resources available for use by the rest of the program, halt clearing away its resources, or halt the entire program. The latency of task termination and whether tasks can ignore termination signals should be clearly specified. Having inconsistent reactions to a fault can potentially be a vulnerability.*

Java provides two options for program termination: `Runtime.exit()` (which is equivalent to `System.exit()`) and `Runtime.halt()`.

`Runtime.exit()`

`Runtime.exit()` is the typical way of exiting a program. According to the Java API [API 2014], `Runtime.exit()`:

> *terminates the currently running Java virtual machine by initiating its shutdown sequence. This method never returns normally. The argument serves as a status code; by convention, a nonzero status code indicates abnormal termination.*
>
> *The virtual machine's shutdown sequence consists of two phases. In the first phase all registered shutdown hooks, if any, are started in some unspecified order and allowed to run concurrently until they finish. In the second phase all uninvoked finalizers are run if finalization-on-exit has been enabled. Once this is performed the virtual machine halts.*
>
> *If this method is invoked after the virtual machine has begun its shutdown sequence, then if shutdown hooks are being run, this method will block indefinitely. If shutdown hooks have already been run and on-exit finalization has been enabled, then this method halts the virtual machine with the given status code if the status is nonzero; otherwise, it blocks indefinitely.*
>
> *The `System.exit()` method is the conventional and convenient means of invoking this method.*

The `Runtime.addShutdownHook()` method can be used to customize `Runtime.exit()` to perform additional actions at program termination. This method uses a `Thread`, which must be initialized but unstarted. The thread starts when the Java Virtual Machine (JVM) begins to shut down. Because the JVM usually has a fixed time to shut down, these threads should not be long-running and should not attempt user interaction.

`Runtime.halt()`

`Runtime.halt()` is similar to `Runtime.exit()` but does *not* run shutdown hooks or finalizers. According to the ava API [API 2014], `Runtime.halt()`

> *forcibly terminates the currently running Java virtual machine. This method never returns normally.*
> *This method should be used with extreme caution. Unlike the exit method, this method does not cause shutdown hooks to be started and does not run uninvoked finalizers if finalization-on-exit has been enabled. If the shutdown sequence has already been initiated, then this method does not wait for any running shutdown hooks or finalizers to finish their work.*

Java programs do not flush unwritten buffered data or close open files when they exit, so programs must perform these operations manually. Programs must also perform any other cleanup that involves external resources, such as releasing shared locks.

Noncompliant Code Example

This example creates a new file, outputs some text to it, and abruptly exits using `Runtime.exit()`. Consequently, the file may be closed without the text actually being written.

```
public class CreateFile {
  public static void main(String[] args)
                        throws FileNotFoundException {
    final PrintStream out =
        new PrintStream(new BufferedOutputStream(
                       new FileOutputStream("foo.txt")));
    out.println("hello");
    Runtime.getRuntime().exit(1);
  }
}
```

Compliant Solution (`close()`)

This solution explicitly closes the file before exiting:

```
public class CreateFile {
  public static void main(String[] args)
                        throws FileNotFoundException {
    final PrintStream out =
    new PrintStream(new BufferedOutputStream(
        new FileOutputStream("foo.txt")));
    try {
      out.println("hello");
    } finally {
    try {
      out.close();
   } catch (IOException x) {
     // Handle error
   }
    }
    Runtime.getRuntime().exit(1);
  }
}
```

Compliant Solution (Shutdown Hook)

This compliant solution adds a shutdown hook to close the file. This hook is invoked by `Runtime.exit()` and is called before the JVM is halted.

```
public class CreateFile {
  public static void main(String[] args)
                        throws FileNotFoundException {
    final PrintStream out =
        new PrintStream(new BufferedOutputStream(
                       new FileOutputStream("foo.txt")));
    Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
        public void run() {
          out.close();
        }
    }));
    out.println("hello");
    Runtime.getRuntime().exit(1);
  }
}
```

Noncompliant Code Example (`Runtime.halt()`)

This noncompliant code example calls `Runtime.halt()` instead of `Runtime.exit()`. The `Runtime.halt()` method stops the JVM without invoking any shutdown hooks; consequently, the file is not properly written to or closed.

```
public class CreateFile {
  public static void main(String[] args)
                        throws FileNotFoundException {
    final PrintStream out =
          new PrintStream(new BufferedOutputStream(
                        new FileOutputStream("foo.txt")));
    Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
        public void run() {
          out.close();
        }
    }));
    out.println("hello");
    Runtime.getRuntime().halt(1);
  }
}
```

Noncompliant Code Example (Signal)

When a user forcefully exits a program, for example by pressing the Ctrl+C keys or by using the `kill` command, the JVM terminates abruptly. Although this event cannot be captured, the program should nevertheless perform any mandatory cleanup operations before exiting. This noncompliant code example fails to do so.

```
public class InterceptExit {
  public static void main(String[] args)
                        throws FileNotFoundException {
    InputStream in = null;
    try {
      in = new FileInputStream("file");
      System.out.println("Regular code block");
      // Abrupt exit such as ctrl + c key pressed
      System.out.println("This never executes");
    } finally {
      if (in != null) {
        try {
          in.close();  // This never executes either
        } catch (IOException x) {
          // Handle error
        }
      }
    }
  }
}
```

Compliant Solution (`addShutdownHook()`)

Use the `addShutdownHook()` method of `java.lang.Runtime` to assist with performing cleanup operations in the event of abrupt termination. The JVM starts the shutdown hook thread when abrupt termination is initiated; the shutdown hook runs concurrently with other JVM threads.

According to the Java API [API 2014], class `Runtime`, method `addShutdownHook()`,

> *A* shutdown hook *is simply an initialized but unstarted thread. When the virtual machine begins its shutdown sequence it will start all registered shutdown hooks in some unspecified order and let them run concurrently. When all the hooks have finished it will then run all uninvoked finalizers if finalization-on-exit has been enabled. Finally, the virtual machine will halt. Once the shutdown sequence has begun it can be stopped only by invoking the halt method, which forcibly terminates the virtual machine....*
> *Once the shutdown sequence has begun it is impossible to register a new shutdown hook or de-register a previously registered hook.*

Some precautions must be taken because the JVM might be in a sensitive state during shutdown. Shutdown hook threads should

- Be lightweight and simple.
- Be thread-safe.
- Hold locks when accessing data and release those locks when done.
- Avoid relying on system services, because the services themselves may be shut down (for example, the logger may be shut down from another hook).

To avoid race conditions or deadlock between shutdown actions, it may be better to run a series of shutdown tasks from one thread by using a single shutdown hook [Goetz 2006].

This compliant solution shows the standard method to install a hook:

```
public class Hook {

  public static void main(String[] args) {
    try {
      final InputStream in = new FileInputStream("file");
      Runtime.getRuntime().addShutdownHook(new Thread() {
          public void run() {
            // Log shutdown and close all resources
            in.close();
          }
      });

     // ...
    } catch (IOException x) {
      // Handle error
    } catch (FileNotFoundException x) {
      // Handle error
    }
  }
}
```

The JVM can abort for external reasons, such as an external `SIGKILL` signal (POSIX) or the `TerminateProcess()` call (Windows), or because of memory corruption caused by native methods. Shutdown hooks may fail to execute as expected in such cases because the JVM cannot guarantee that they will be executed as intended.

Risk Assessment

Failure to perform necessary cleanup at program termination may leave the system in an inconsistent state.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO14-J	Medium	Likely	Medium	**P12**	**L1**

Automated Detection

Tool	Version	Checker	Description
Parasoft Jtest	10.3	**OPT.CIO, OPT.CCR, OPT.CRWD**	Implemented

Related Guidelines

SEI CERT C Coding Standard	ERR04-C. Choose an appropriate termination strategy
SEI CERT C++ Coding Standard	VOID ERR04-CPP. Choose an appropriate termination strategy
ISO/IEC TR 24772:2010	Termination Strategy [REU]
MITRE CWE	CWE-705, Incorrect Control Flow Scoping

Android Implementation Details

Although most of the code examples are not applicable to the Android platform, the principle is applicable to Android. Aprocess on Android can be terminated in a number of ways: `android.app.Activity.finish()` and the related `finish()` methods, `android.app.Activity.moveTaskToBack(boolean flag)`, `android.os.Process.killProcess(int pid)`, and `System.exit()`.

Bibliography

[API 2014]	Class `Runtime.exit()`
[ISO/IEC TR 24772:2010]	Section 6.46, "Termination Strategy [REU]"