ERR00-J. Do not suppress or ignore checked exceptions

Programmers often suppress checked exceptions by catching exceptions with an empty or trivial `catch` block. Each `catch` block must ensure that the program continues only with valid invariants. Consequently, the `catch` block must either recover from the exceptional condition, rethrow the exception to allow the next nearest enclosing `catch` clause of a `try` statement to recover, or throw an exception that is appropriate to the context of the `catch` block.

Exceptions disrupt the expected control flow of the application. For example, no part of any expression or statement that occurs in the `try` block after the point from which the exception is thrown is evaluated. Consequently, exceptions must be handled appropriately. Many reasons for suppressing exceptions are invalid. For example, when the client cannot be expected to recover from the underlying problem, it is good practice to allow the exception to propagate outwards rather than to catch and suppress the exception.

Noncompliant Code Example

This noncompliant code example simply prints the exception's stack trace:

```
try {
  //...
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Printing the exception's stack trace can be useful for debugging purposes, but the resulting program execution is equivalent to suppressing the exception. Printing the stack trace can also leak information about the structure and state of the process to an attacker (see ERR01-J. Do not allow exceptions to expose sensitive information for more information). Note that even though this noncompliant code example reacts to the exception by printing out a stack trace, it then proceeds as though the exception were not thrown. That is, the behavior of the application is unaffected by the exception being thrown except that any expressions or statements that occur in the `try` block after the point from which the exception is thrown are not evaluated.

Compliant Solution (Interactive)

This compliant solution handles a `FileNotFoundException` by requesting that the user specify another file name:

```
volatile boolean validFlag = false;
do {
  try {
    // If requested file does not exist, throws FileNotFoundException
    // If requested file exists, sets validFlag to true
    validFlag = true;
  } catch (FileNotFoundException e) {
    // Ask the user for a different file name
  }
} while (validFlag != true);
// Use the file
```

To comply with ERR01-J. Do not allow exceptions to expose sensitive information, the user should only be allowed to access files in a user-specific directory. This prevents any other `IOException` that escapes the loop from leaking sensitive file system information.

Compliant Solution (Exception Reporter)

Proper reporting of exceptional conditions is context-dependent. For example, GUI applications should report the exception in a graphical manner, such as in an error dialog box. Most library classes should be able to objectively determine how an exception should be reported to preserve modularity; they cannot rely on `System.err`, on any particular logger, or on the availability of the windowing environment. As a result, library classes that wish to report exceptions should specify the API they use to report exceptions. This compliant solution specifies both an interface for reporting exceptions, which exports the `report()` method, and a default exception reporter class that the library can use. The exception reporter can be overridden by subclasses.

```
public interface Reporter {
  public void report(Throwable t);
}

class ExceptionReporterPermission extends Permission {
  // ...
}

public class ExceptionReporter {

  // Exception reporter that prints the exception
  // to the console (used as default)
  private static final Reporter PrintException = new Reporter() {
    public void report(Throwable t) {
      System.err.println(t.toString());
    }
  };

  // Stores the default reporter
  // The default reporter can be changed by the user
  private static Reporter Default = PrintException;

  // Helps change the default reporter back to
  // PrintException in the future
  public static Reporter getPrintException() {
    return PrintException;
  }

  public static Reporter getExceptionReporter() {
    return Default;
  }

  // May throw a SecurityException (which is unchecked)
public static void setExceptionReporter(Reporter reporter) {
    // Custom permission
    ExceptionReporterPermission perm = new
        ExceptionReporterPermission("exc.reporter");
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
      // Check whether the caller has appropriate permissions
      sm.checkPermission(perm);
    }
    // Change the default exception reporter
    Default = reporter;
  }
}
```

The `setExceptionReporter()` method prevents hostile code from maliciously installing a more verbose reporter that leaks sensitive information or that directs exception reports to an inappropriate location, such as the attacker's computer, by limiting attempts to change the exception reporter to callers that have the custom permission `ExceptionReporterPermission` with target `exc.reporter`.

The library may subsequently use the exception reporter in `catch` clauses:

```
try {
  // ...
} catch (IOException warning) {
  ExceptionReporter.getExceptionReporter().report(warning);
  // Recover from the exception...
}
```

Any client code that possesses the required permissions can override the `ExceptionReporter` with a handler that logs the error or provides a dialog box, or both. For example, a GUI client using Swing may require exceptions to be reported using a dialog box:

```
ExceptionReporter.setExceptionReporter(new ExceptionReporter() {
  public void report(Throwable exception) {
    JOptionPane.showMessageDialog(frame,
                                  exception.toString,
                                  exception.getClass().getName(),
                                  JOptionPane.ERROR_MESSAGE);
  }});
```

Compliant Solution (Subclass Exception Reporter and Filter-Sensitive Exceptions)

Sometimes exceptions must be hidden from the user for security reasons (see ERR01-J. Do not allow exceptions to expose sensitive information) . In such cases, one acceptable approach is to subclass the `ExceptionReporter` class and add a `filter()` method in addition to overriding the default `report()` method.

```
class MyExceptionReporter extends ExceptionReporter {
  private static final Logger logger =
      Logger.getLogger("com.organization.Log");

  public static void report(Throwable t) {
    t = filter(t);
    if (t != null) {
      logger.log(Level.FINEST, "Loggable exception occurred", t);
    }
  }

  public static Exception filter(Throwable t) {
    if (t instanceof SensitiveException1) {
      // Too sensitive, return nothing (so that no logging happens)
      return null;
    } else if (t instanceof SensitiveException2) {
      // Return a default insensitive exception instead
      return new FilteredSensitiveException(t);
    }
    // ...
```

```
      // Return for reporting to the user
      return t;
    }
  }


  // ...Definitions for SensitiveException1, SensitiveException2
  // and FilteredSensitiveException...
```

The `report()` method accepts a `Throwable` instance and consequently handles all errors, checked exceptions, and unchecked exceptions. The filtering mechanism is based on a *whitelisting* approach wherein only nonsensitive exceptions are propagated to the user. Exceptions that are forbidden to appear in a log file can be filtered in the same fashion (see FIO13-J. Do not log sensitive information outside a trust boundary). This approach provides the benefits of exception chaining by reporting exceptions tailored to the abstraction while also logging the low-level cause for future failure analysis [Bloch 2008].

Noncompliant Code Example

If a thread is interrupted while sleeping or waiting, it causes a `java.lang.InterruptedException` to be thrown. However, the `run()` method of interface `Runnable` cannot throw a checked exception and must handle `InterruptedException`. This noncompliant code example catches and suppresses `InterruptedException`:

```
class Foo implements Runnable {
  public void run() {
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      // Ignore
    }
  }
}
```

This code prevents callers of the `run()` method from determining that an interrupted exception occurred. Consequently, caller methods such as `Thread.start()` cannot act on the exception [Goetz 2006]. Likewise, if this code were called in its own thread, it would prevent the calling thread from knowing that the thread was interrupted.

Compliant Solution

This compliant solution catches the `InterruptedException` and restores the interrupted status by calling the `interrupt()` method on the current thread:

```
class Foo implements Runnable {
  public void run() {
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt(); // Reset interrupted status
    }
  }
}
```

Consequently, calling methods (or code from a calling thread) can determine that an interrupt was issued [Goetz 2006].

Exceptions

ERR00-J-EX0: Exceptions that occur during the freeing of a resource may be suppressed in those cases where failure to free the resource cannot affect future program behavior. Examples of freeing resources include closing files, network sockets, shutting down threads, and so forth. Such resources are often freed in `catch` or `finally` blocks and never reused during subsequent execution. Consequently, the exception cannot influence future program behavior through any avenue other than resource exhaustion. When resource exhaustion is adequately handled, it is sufficient to sanitize and log the exception for future improvement; additional error handling is unnecessary in this case.

ERR00-J-EX1: When recovery from an exceptional condition is impossible at a particular abstraction level, code at that level must not handle that exceptional condition. In such cases, an appropriate exception must be thrown so that higher level code can catch the exceptional condition and attempt recovery. The most common implementation for this case is to omit a `catch` block and allow the exception to propagate normally:

```
// When recovery is possible at higher levels
private void doSomething() throws FileNotFoundException {
  // Requested file does not exist; throws FileNotFoundException
  // Higher level code can handle it by displaying a dialog box and asking
  // the user for the file name
}
```

Some APIs may limit the permissible exceptions thrown by particular methods. In such cases, it may be necessary to catch an exception and either wrap it in a permitted exception or translate it to one of the permitted exceptions:

```
public void myMethod() throws MyProgramException {
  // ...
  try {
    // Requested file does not exist
    // User is unable to supply the file name
  } catch (FileNotFoundException e) {
    throw new MyProgramException(e);
  }
  // ...
}
```

Alternatively, when higher level code is also unable to recover from a particular exception, the checked exception may be wrapped in an unchecked exception and rethrown.

ERR00-J-EX2: An `InterruptedException` may be caught and suppressed when extending class `Thread` [Goetz 2006]. An interruption request may also be suppressed by code that implements a thread's interruption policy [Goetz 2006, p. 143].

Risk Assessment

Ignoring or suppressing exceptions can result in inconsistent program state.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR00-J	Low	Probable	Medium	**P4**	**L3**

Automated Detection

Detection of suppressed exceptions is straightforward. Sound determination of which specific cases represent violations of this rule and which represent permitted exceptions to the rule is infeasible. Heuristic approaches may be effective.

Tool	Version	Checker	Description
CodeSonar	4.5p1	**FB.BAD_PRACTICE.DE_MIGHT_IGNORE**	Method might ignore exception
Coverity	7.5	**MISSING_THROW**	Implemented

Parasoft Jtest	10.3	**SECURITY.UEHL.LGE, UC.UCATCH**	Implemented
SonarQube Java Plugin	4.11	**S1166**	

Related Vulnerabilities

AMQ-1272 describes a vulnerability in the ActiveMQ service. When ActiveMQ receives an invalid username and password from a Stomp client, a security exception is generated but is subsequently ignored, leaving the client connected with full and unrestricted access to ActiveMQ.

Related Guidelines

MITRE CWE	CWE-390, Detection of Error Condition without Action

Bibliography

[Bloch 2008]	Item 62, "Document All Exceptions Thrown by Each Method" Item 65, "Don't Ignore Exceptions"
[Goetz 2006]	Section 5.4, "Blocking and Interruptible Methods"
[JLS 2015]	Chapter 11, "Exceptions"

← ↑ →