ERR08-J. Do not catch NullPointerException or any of its ancestors

Programs must not catch `java.lang.NullPointerException`. A `NullPointerException` exception thrown at runtime indicates the existence of an underlying `null` pointer dereference that must be fixed in the application code (see EXP01-J. Do not use a null in a case where an object is required for more information). Handling the underlying null pointer dereference by catching the `NullPointerException` rather than fixing the underlying problem is inappropriate for several reasons. First, catching `NullPointerException` adds significantly more performance overhead than simply adding the necessary null checks [Bloch 2008]. Second, when multiple expressions in a `try` block are capable of throwing a `NullPointerException`, it is difficult or impossible to determine which expression is responsible for the exception because the `NullPointerException` `catch` block handles any `NullPointerException` thrown from any location in the `try` block. Third, programs rarely remain in an expected and usable state after a `NullPointerException` has been thrown. Attempts to continue execution after first catching and logging (or worse, suppressing) the exception rarely succeed.

Likewise, programs must not catch `RuntimeException`, `Exception`, or `Throwable`. Few, if any, methods are capable of handling all possible runtime exceptions. When a method catches `RuntimeException`, it may receive exceptions unanticipated by the designer, including `NullPointerException` and `ArrayIndexOutOfBoundsException`. Many `catch` clauses simply log or ignore the enclosed exceptional condition and attempt to resume normal execution; this practice often violates ERR00-J. Do not suppress or ignore checked exceptions. Runtime exceptions often indicate bugs in the program that should be fixed by the developer and often cause control flow vulnerabilities.

Noncompliant Code Example (`NullPointerException`)

This noncompliant code example defines an `isName()` method that takes a `String` argument and returns true if the given string is a valid name. A valid name is defined as two capitalized words separated by one or more spaces. Rather than checking to see whether the given string is null, the method catches `NullPointerException` and returns false.

```
boolean isName(String s) {
  try {
    String names[] = s.split(" ");

    if (names.length != 2) {
      return false;
    }
    return (isCapitalized(names[0]) && isCapitalized(names[1]));
  } catch (NullPointerException e) {
    return false;
  }
}
```

Compliant Solution

This compliant solution explicitly checks the `String` argument for `null` rather than catching `NullPointerException`:

```
boolean isName(String s) {
  if (s == null) {
    return false;
  }
  String names[] = s.split(" ");
  if (names.length != 2) {
    return false;
  }
  return (isCapitalized(names[0]) && isCapitalized(names[1]));
}
```

Compliant Solution

This compliant solution omits an explicit check for a null reference and permits a `NullPointerException` to be thrown:

```
boolean isName(String s) /* Throws NullPointerException */ {
  String names[] = s.split(" ");
  if (names.length != 2) {
    return false;
  }
  return (isCapitalized(names[0]) && isCapitalized(names[1]));
}
```

Omitting the null check means that the program fails more quickly than if the program had returned false and lets an invoking method discover the null value. A method that throws a `NullPointerException` without a null check must provide a precondition that the argument being passed to it is not null.

Noncompliant Code Example (Null Object Pattern)

This noncompliant code example is derived from the logging service Null Object design pattern described by Henney [Henney 2003]. The logging service is composed of two classes: one that prints the triggering activity's details to a disk file using the `FileLog` class and another that prints to the console using the `ConsoleLog` class. An interface, `Log`, defines a `write()` method that is implemented by the respective log classes. Method selection occurs polymorphically at runtime. The logging infrastructure is subsequently used by a `Service` class.

```
public interface Log {
  void write(String messageToLog);
}

public class FileLog implements Log {
  private final FileWriter out;

  FileLog(String logFileName) throws IOException {
    out = new FileWriter(logFileName, true);
  }

  public void write(String messageToLog) {
    // Write message to file
  }
}

public class ConsoleLog implements Log {
  public void write(String messageToLog) {
    System.out.println(messageToLog); // Write message to console
  }
}

class Service {
  private Log log;

  Service() {
    this.log = null; // No logger
  }

  Service(Log log) {
    this.log = log; // Set the specified logger
  }

  public void handle() {
    try {
      log.write("Request received and handled");
    } catch (NullPointerException npe) {
      // Ignore
    }
  }

  public static void main(String[] args) throws IOException {
    Service s = new Service(new FileLog("logfile.log"));
    s.handle();

    s = new Service(new ConsoleLog());
    s.handle();
  }
}
```

Each `Service` object must support the possibility that a `Log` object may be `null` because clients may choose not to perform logging. This noncompliant code example eliminates null checks by using a `try-catch` block that ignores `NullPointerException`.

This design choice suppresses genuine occurrences of `NullPointerException` in violation of ERR00-J. Do not suppress or ignore checked exceptions. It also violates the design principle that exceptions should be used only for exceptional conditions because ignoring a null `Log` object is part of the ordinary operation of a server.

Compliant Solution (Null Object Pattern)

The Null Object design pattern provides an alternative to the use of explicit null checks in code. It reduces the need for explicit null checks through the use of an explicit, safe *null object* rather than a null reference.

This compliant solution modifies the no-argument constructor of class `Service` to use the *do-nothing* behavior provided by an additional class, `Log.NULL`; it leaves the other classes unchanged.

```
public interface Log {

  public static final Log NULL = new Log() {
    public void write(String messageToLog) {
      // Do nothing
    }
  };

  void write(String messageToLog);
}

class Service {
  private final Log log;

  Service(){
    this.log = Log.NULL;
  }

  // ...
}
```

Declaring the `log` reference final ensures that its value is assigned during initialization.

An acceptable alternative implementation uses accessor methods to control all interaction with the reference to the current log. The accessor method to set a log ensures use of the null object in place of a null reference. The accessor method to get a log ensures that any retrieved instance is either an actual logger or a null object (but never a null reference). Instances of the null object are immutable and are inherently thread-safe.

Some system designs require returning a value from a method rather than implementing do-nothing behavior. One acceptable approach is use of an exceptional value object that throws an exception before the method returns [Cunningham 1995]. This approach can be a useful alternative to returning `null`.

In distributed environments, the null object must be passed by copy to ensure that remote systems avoid the overhead of a remote call argument evaluation on every access to the null object. Null object code for distributed environments must also implement the `Serializable` interface.

Code that uses this pattern must be clearly documented to ensure that security-critical messages are never discarded because the pattern has been misapplied.

Noncompliant Code Example (Division)

This noncompliant code example assumes that the original version of the `division()` method was declared to throw only `ArithmeticException`. However, the caller catches the more general `Exception` type to report arithmetic problems rather than catching the specific exception `ArithmeticException` type. This practice is risky because future changes to the method signature could add more exceptions to the list of potential exceptions the caller must handle. In this example, a revision of the `division()` method can throw `IOException` in addition to `ArithmeticException`. However, the compiler will not diagnose the lack of a corresponding handler because the invoking method already catches `IOException` as a result of catching `Exception`. Consequently, the recovery process might be inappropriate for the specific exception type that is thrown. Furthermore, the developer has failed to anticipate that catching `Exception` also catches unchecked exceptions.

```
public class DivideException {
  public static void division(int totalSum, int totalNumber)
    throws ArithmeticException, IOException  {
    int average  = totalSum / totalNumber;
    // Additional operations that may throw IOException...
    System.out.println("Average: " + average);
  }
  public static void main(String[] args) {
    try {
      division(200, 5);
      division(200, 0); // Divide by zero
    } catch (Exception e) {
      System.out.println("Divide by zero exception : "
                         + e.getMessage());
    }
  }
}
```

Noncompliant Code Example

This noncompliant code example attempts to solve the problem by specifically catching `ArithmeticException`. However, it continues to catch `Excepti on` and consequently catches both unanticipated checked exceptions and unanticipated runtime exceptions.

```
try {
  division(200, 5);
  division(200, 0); // Divide by zero
} catch (ArithmeticException ae) {
  throw new DivideByZeroException();
} catch (Exception e) {
  System.out.println("Exception occurred :" + e.getMessage());
}
```

Note that `DivideByZeroException` is a custom exception type that extends `Exception`.

Compliant Solution

This compliant solution catches only the specific anticipated exceptions (`ArithmeticException` and `IOException`). All other exceptions are permitted to propagate up the call stack.

```
import java.io.IOException;

public class DivideException {
  public static void main(String[] args) {
    try {
      division(200, 5);
      division(200, 0); // Divide by zero
    } catch (ArithmeticException ae) {
      // DivideByZeroException extends Exception so is checked
      throw new DivideByZeroException();
    } catch (IOException ex) {
      ExceptionReporter.report(ex);
    }
  }

  public static void division(int totalSum, int totalNumber)
                              throws ArithmeticException, IOException  {
    int average  = totalSum / totalNumber;
    // Additional operations that may throw IOException...
    System.out.println("Average: "+ average);
  }
}
```

The `ExceptionReporter` class is documented in ERR00-J. Do not suppress or ignore checked exceptions.

Compliant Solution (Java SE 7)

Java SE 7 allows a single `catch` block to catch multiple exceptions of different types, which prevents redundant code. This compliant solution catches the specific anticipated exceptions (`ArithmeticException` and `IOException`) and handles them with one catch clause. All other exceptions are permitted to propagate to the next `catch` clause of a `try` statement on the stack.

```
import java.io.IOException;

public class DivideException {
  public static void main(String[] args) {
    try {
      division(200, 5);
      division(200, 0); // Divide by zero
    } catch (ArithmeticException | IOException ex) {
      ExceptionReporter.report(ex);
    }
  }

  public static void division(int totalSum, int totalNumber)
                              throws ArithmeticException, IOException  {
    int average  = totalSum / totalNumber;
    // Additional operations that may throw IOException...
    System.out.println("Average: "+ average);
  }
}
```

Exceptions

ERR08-J-EX0: A `catch` block may catch all exceptions to process them before rethrowing them (filtering sensitive information from exceptions before the call stack leaves a trust boundary, for example). Refer to ERR01-J. Do not allow exceptions to expose sensitive information and weaknesses CWE 7 and CWE 388 for more information. In such cases, a `catch` block should catch `Throwable` rather than `Exception` or `RuntimeException`.

This code sample catches all exceptions and wraps them in a custom `DoSomethingException` before rethrowing them:

```
class DoSomethingException extends Exception {
  public DoSomethingException(Throwable cause) {
    super(cause);
  }

  // Other methods

};

private void doSomething() throws DoSomethingException {
  try {
    // Code that might throw an Exception
  } catch (Throwable t) {
    throw new DoSomethingException(t);
  }
}
```

Exception wrapping is a common technique to safely handle unknown exceptions. For another example, see ERR06-J. Do not throw undeclared checked exceptions.

ERR08-J-EX1: Task processing threads such as worker threads in a thread pool or the Swing event dispatch thread are permitted to catch `RuntimeException` when they call untrusted code through an abstraction such as the `Runnable` interface [Goetz 2006, p. 161].

ERR08-J-EX2: Systems that require substantial fault tolerance or graceful degradation are permitted to catch and log general exceptions such as `Throwable` at appropriate levels of abstraction. For example:

- A real-time control system that catches and logs all exceptions at the outermost layer, followed by warm-starting the system so that real-time control can continue. Such approaches are clearly justified when program termination would have safety-critical or mission-critical consequences.
- A system that catches all exceptions that propagate out of each major subsystem, logs the exceptions for later debugging, and subsequently shuts down the failing subsystem (perhaps replacing it with a much simpler, limited-functionality version) while continuing other services.

Risk Assessment

Catching `NullPointerException` may mask an underlying null dereference, degrade application performance, and result in code that is hard to understand and maintain. Likewise, catching `RuntimeException`, `Exception`, or `Throwable` may unintentionally trap other exception types and prevent them from being handled properly.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR08-J	Medium	Likely	Medium	**P12**	**L1**

Automated Detection

Tool	Version	Checker	Description
CodeSonar	5.1p0	**PMD.Strict-Exceptions.AvoidCatchingThrowable**	Avoid catching Throwable
Parasoft Jtest	10.3	**EXCEPT.NCNPE**	Implemented
SonarQube	6.7	**S1181**	Throwable and Error should not be caught
		S1696	"NullPointerException" should not be caught