ISO/IEC JTC 1/SC 22/WG23 N1475

Deleted: 036056

Date: 2025-03-12

Deleted: -02-080-109-02

ISO/IEC TR 24772–11

WG 23/N 1475

Notes on this document

This document is a draft of Avoiding programming language vulnerabilities in Java.

List of Java changes since Java 14

- Switch statements and expressions – possibly further enhancements (13)
- Sealed classes and interfaces
- Hidden classes
- Records
- Text Blocks

Java 15

- Vector API
- Sealed Classes

Java 16

- Restore always-strict FP semantics
- Enhanced pseudo-random number generators
- Pattern matching for switch statements (trial)
- Deprecate security manager for removal

Java 18

- Pattern matching for switch statements (second)
- Deprecate finalization for removal

Java 19 & 20

- Record patterns
- Virtual threads (preview)
- Vector API
- Structured concurrency

Java 21

- String templates

Sequenced collections

Record patterns

Pattern matching for switch

Virtual threads

Scoped values

Vector API

Structured concurrency

WG 23/N 1475

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages – Part 11 – Vulnerability descriptions for the programming language Java

Élément introductif — Élément principal — Partie n : Titre de la partie

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: International standard
Document subtype: if applicable
Document stage : (10) development stage
Document language: E

Deleted:

Baseline Edition ISO/IEC TR 24772–11

Participating in writeup 19 February 2025

Stephen Michell – convenor WG 23

Larry Wagoner

Sean McDonagh

Erhard Ploedereder

Excused

Tullio Vardanega

All issues discussed are captured in the document, either as comments or resolved issues. The previous version of this document is N1474.

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example), it may decide to publish a Technical Report. A Technical Report is entirely informative in nature and shall be subject to review every five years in the same manner as an International Standard.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 24772-11, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Introduction

This Technical Report provides guidance for the programming language Java, so that application developers considering Java or using Java will be better able to avoid the programming constructs that lead to vulnerabilities in software written in the Java language and their attendant consequences. This guidance can also be used by developers to select source code evaluation tools that can discover and eliminate some constructs that could lead to vulnerabilities in their software. This report can also be used in comparison with companion Technical Reports and with the language-independent report, TR 24772–1, to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

This technical report part is intended to be used with TR 24772–1, which discusses programming language vulnerabilities in a language independent fashion.

It should be noted that this Technical Report is inherently incomplete. It is not possible to provide a complete list of programming language vulnerabilities because new weaknesses are discovered continually. Any such report can only describe those that have been found, characterized, and determined to have sufficient probability and consequence.

Information Technology — Programming Languages — Avoiding vulnerabilities in programming languages — Vulnerability descriptions for the programming language Java

6. Scope

This Technical Report specifies software programming language vulnerabilities to be avoided in the development of systems where assured behaviour is required for security, safety, mission-critical and business-critical software. In general, this guidance is applicable to the software developed, reviewed, or maintained for any application.

Vulnerabilities described in this Technical Report document the way that the vulnerability described in the language-independent TR 24772–1 are manifested in Java

2. Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

The Java Language Specification, Java SE 10 Edition, 2018-02-20, https://docs.oracle.com/javase/specs/

3. Terms and definitions, symbols and conventions

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, in TR 24772–1, the Oracle Java Glossary (https://www.oracle.com/technetwork/java/glossary-135216.html) and the following apply. Other terms are defined where they appear in *italic* type.

The following terms are in alphabetical order, with general topics referencing the relevant specific terms.

3.1.1

access
read or modify the value of an object

Note: Modify includes the case where the new value being stored is the same as the previous value. Expressions that are not evaluated do not access objects.

3.1.2

behaviour
external appearance or action

Note: See: implementation-defined behaviour, undefined behaviour, unspecified behaviour

3.1.3

bit
unit of data storage in the execution environment large enough to hold an object that has one of two values

Note: It need not be possible to express the address of each individual bit of an object.

3.1.4

byte
addressable unit of data storage large enough to hold any member of the basic character set of the execution environment

Note: It is possible to uniquely express the address of each individual byte of an object. A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit.

3.1.5

character
abstract member of a set of elements used for the organization, control, or representation of data

3.1.6

correctly rounded result
representation in the result format that is nearest in value, subject to the current rounding mode, to what the result would be given unlimited range and precision

3.1.7

implementation
particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment

3.1.8

implementation-defined behaviour
behaviour where multiple options are permitted by the standard and where each implementation documents how the choice is made

3.1.9

implementation-defined value
value not specified in the standard where each implementation documents how the choice for the value is selected

3.1.10

implementation limit
restriction imposed upon programs by the implementation

3.1.11

memory location
object of scalar type, or a maximal sequence of adjacent bit-fields all having nonzero width

3.1.12

multibyte character
sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment, where the extended character set is a superset of the basic character set

3.1.13

thread

independent path of execution within a program

3.1.14

undefined behaviour
use of a non-portable or erroneous program construct or erroneous data

Deleted: of

Note: Undefined behaviour ranges from completely ignoring the situation with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

Deleted: completely

3.1.15

unspecified behaviour
use of an unspecified value, or other behaviour where the language standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

Note: For example, unspecified behaviour is the order in which the arguments of a function are evaluated.

4. Language concepts

Java was originally developed at Sun Microsystems (acquired by Oracle Corporation in 2010) in the early 1990s. Java was initially defined as a syntactic superset of the C programming language: adding object oriented features such as classes, encapsulation, dynamic dispatch, namespaces, and templates. It was designed to be platform independent through the use of platform independent byte code which is then interpreted by the Java Virtual Machine (JVM) on whichever platform it is executed on. Java espoused the Write Once, Run Anywhere (WORA) goal.

While there is a core of Java that is syntactically identical to C, it has always been the case that there are significant differences between the two. Since Java was developed, the two languages have diverged even further, both adding features not present in the other. Notwithstanding that, there is still a significant syntactic and semantic overlap between C and Java.

At its core, Java was designed to address some weaknesses that existed in other languages through the addition of security management features. Some key features of Java are:

- Java uses a Garbage Collector to manage memory without the use of explicit commands to erase memory or to aggregate freed space.
- Java provides ease of code reuse through inheritance.
- The javac compiler transforms Java code into byte code instead of into machine executable instructions. The byte code is then interpreted and run by a Java Virtual Machine (JVM) on a particular platform.
- Classes provide single inheritance of specifications and code.
- Interfaces provide multiple inheritance of specifications.

Subsequently, in many cases, the additional features of Java provide mechanisms for avoiding vulnerabilities based in memory management and other areas that are susceptible to language misuse, and these are reflected in the following sections.

Java does have some inherently unsafe features. For instance, as its name implies, sun.misc.Unsafe is considered unsafe for general use, though it does provide some low level programming features such as reinterpretation of data. Documentation is not widely available, and its use usually relies on miscellaneous web postings, leading to even more unsafe use.

5. Avoiding programming language vulnerabilities in Java

In addition to the generic programming rules from ISO/IEC 24772-1:2024 clause 5.4, additional rules from this section apply specifically to the Java programming language. The recommendations of this section are restatements of recommendations from clause 6 but represent ones stated frequently or that are considered particularly noteworthy by the authors. Clause 6 of this document contains the full set of recommendations, as well as explanations of the problems that led to the recommendations made.

Every avoidance mechanism provided in this section is supported by material in Clause 6 of this document, as well as other important recommendations.

Index		Reference
1	Access all private data components only through getter and setter methods. For class-based enums, ensure that enum values are not mutable by making members in an enum type private, by setting the members in the constructor and by not providing setter methods.	6.61 Concurrent data access [CGX]
2	Check the value of a larger type before converting it to a smaller type to see if the value in the larger type is within the range of the smaller type. Use comments to document cases where intentional loss of data due to narrowing is expected and acceptable.	6.6 Conversion errors [FLC]
3	Use defensive programming techniques to check whether an operation will overflow or underflow the receiving data type. These techniques can be omitted if it can be shown by static analysis (e.g., at compile time) that overflow or underflow is not possible.	6.15 Arithmetic wrap-around error [FIF]
4	Include checks for null prior to making use of objects. Less preferably, handle exceptions raised by attempts to dereference null values.	6.13 Null pointer dereference [XYH]
5	Mark all variables observable by another thread or hardware agent as volatile.	6.18 Dead store [WXQ]
6	Ensure that when the identifier that a method uses is identical to an identifier in the class that the correct identifier is used through the use or non-use of “this”.	6.20 Identifier name reuse [YOW]
7	Avoid the use of expressions with side effects for multiple parameters to functions, since the order in which the parameters are evaluated and hence the side effects occur is unspecified.	6.32 Passing parameters and return values [CSJ]
8	Use *try-with-resources*, which extends the behaviour of the try/catch construct to allow access to resources without having to close them afterwards, as the resource closures are done automatically.	6.36 Ignored error status and unhandled exceptions [OYB]
9	Enable verbose garbage collection to see a detailed trace of the garbage collector’s actions. Reduce the number of temporary objects to minimize the impact and need for garbage collection. Enable verbose garbage collection and profiling to locate and fix memory leaks to reduce the need for garbage collection.	6.39 Memory leaks and heap fragmentation [XYL]
10	Use Java profiler tools that monitor and diagnose memory leaks.	6.39 Memory leaks and heap fragmentation [XYL]

11	Keep the inheritance graph as shallow as possible to simplify the review of inheritance relationships and method overridings.	6.41 Inheritance [RIP]
12	Be aware that native code can lack many of the protections afforded by Java, such as bounds checks on structures not being performed on native methods, and explicitly perform the necessary checks. Use a foreign function interface such as JNI to provide a clear separation between Java and the other language. Minimize the use of those issues known to be error-prone when interfacing between languages, such as: 1. passing character strings 2. dimension, bounds, and layout issues of arrays 3. interfacing with other parameter mechanisms such as call by reference, value, or name 4. handling faults, exceptions, and errors, and 5. bit representation.	6.47 Inter-language calling [DJS]
13	Always have an appropriate response for checked exceptions since even things that should never happen do happen occasionally.	6.50 Unanticipated exceptions from library routines [HJW]
14	Use the Java ExecutorService framework for thread group management.	6.62 Concurrency – Premature termination [CGS]

6. Specific Guidance for Java Vulnerabilities

6.1 General

This clause contains specific advice for Java about the possible presence of vulnerabilities as described in ISO/IEC 24772-1:2024 and provides specific guidance on how to avoid them in Java code. This section mirrors ISO/IEC 24772-1:2024 clause 6 in that the vulnerability "Type System [IHN]" is found in 6.2 of ISO/IEC TR 24772–1, and Java specific guidance is found in clause 6 and its subclauses in this document.

6.2 Type System [IHN]

6.2.1 Applicability to language

Java is a statically typed language. Java is also a strongly typed language, as it requires all variables to be typed and places restrictions on the values that a variable can hold. There are two categories of types in Java: primitive types and reference types. Primitive types are `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, *`enum`*, and `double`. Reference types are the class, interface, and array types. Records are a restricted form of classes that are intended to hold immutable data, cannot participate in inheritance, and cannot be abstract.

When performing an arithmetic operation composed of all integers, all operands are first converted to an int. If all of the operands are floating point, all operands are first converted to the double type. When performing operations with mixed data types, the smaller type is converted to a larger type. For instance, adding a short to an int results in the short being upsized to an int before the operation is performed. Java requires explicit casting when going from a larger primitive type to a smaller one. Implicit casting is allowed when going from a smaller primitive type to a larger one, even though it is likely that precision is lost in the conversion. This and other type conversion vulnerabilities are discussed in more depth in sections 6.6 Conversion errors [FLC], 6.15 Arithmetic wrap-around error [FIF], and 6.44 Polymorphic variables [BKK].

For reference types, no explicit cast is required when assigning an object of a child type to a variable of its parent type; however, an explicit cast is required when assigning an object designated by a parent type reference to a variable of any of its child types. A ClassCastException will be thrown at runtime unless the parent type reference is referring to an object of the child type.

The vulnerability documented in ISO/IEC 24772-1:2024 relating to the ability to distinguish integer types representing different physical units (such as meters or feet) exists in Java. It can be mitigated by generating distinct classes for each dimensional type and creating operators and conversion methods that correctly perform the conversations.

6.2.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.6.5.
- Consider using classes instead of base types for values with physical properties, such as weight or size.
- Avoid deeply nested or complicated record types to minimize the possibility of unexpected behavior.

Commented [SM1]: Many normal paragraphs are showing up like section headers. Sean, please fix throughout document.

6.3 Bit representations [STR]

6.3.1 Applicability to language

The vulnerabilities described in ISO/IEC 24772-1:2024 6.3 apply to Java.

Java supports a variety of sizes for integers, such as `byte`, `short`, `int`, and `long`, but Java only supports signed integer types. This simplifies the understanding and use of integer types; however, Java supports unsigned arithmetic using static methods in class `Integer`. The result of the unsigned arithmetic is an unsigned integer. No mixed operations are provided.

Java also supports various bitwise operators that facilitate bit manipulations, such as left and right shifts and bitwise `&` and `|`. Some of these bit manipulations can cause unexpected results. For instance, Java differentiates between a signed right shift and an unsigned right shift. The signed right shift is performed using the operator ">>" whereas the unsigned right shift is performed using the operator ">>>". Although Java has simplified its language by only having signed integers, it has relegated the issue of whether the sign bit is shifted right to the choice of operator. It is easy to confuse the two operators ">>" and ">>>" and do a signed right shift instead of an unsigned right shift or vice versa. For instance,

```
int a, b, c, d;
a = 0b00101000;  // a = 0010 0100
b = a >> 3;         // signed right shift yields b = 0000 0100
c = 0b11110100;  // c = 1111 0100
d = c >> 3;   // signed right shift of a negative number yields d = 1111 1110

int e, f, g, h;
e = 0b00101000;  // e = 0010 1000
f = e >>> 3;     // unsigned right shift yields f = 0000 0101
g = 0b11110100;  // g = 1111 0100
h = g >>> 3; // unsigned right shift of a negative number yields h = 0001 1110
```

Another issue that can arise is that Java stores data in big-endian format, also known as network byte order. This can cause issues when interfacing with little endian languages such as C.

6.3.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.3.5.
- Ensure that the unsigned and signed right shift operators are not confused with each other.
- Avoid manipulating numbers using unsigned arithmetic operations in class `Integer`.
- Use java.nio.ByteBuffer to convert byte order between little endian to big endian.

6.4 Floating-point arithmetic [PLF]

6.4.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.4 applies to Java.

Java implements a subset of ISO/IEC/IEEE 60559:2011 Floating-point arithmetic.

Java permits the floating-point data types `float` and `double`. Due to the approximate nature of floating-point representations, using floating-point data types in situations where equality is to be tested or where rounding could accumulate over multiple iterations could lead to unexpected results and potential vulnerabilities. Instead of testing equality, comparison against a threshold will yield the intended effect, for example:

```
final double THRESHOLD = .00001;
double f1,f2;
// … assignments and operations on f1 and f2
if (Math.abs(f1 – f2) < THRESHOLD){
   . . .
}
```

As with most data types, Java is flexible in how `float` and `double` can be used. For instance, Java allows the use of floating-point types to be used as loop counters and in equality statements, even though, in some cases, these will not have the expected behaviour. For example:

```
float x;
for (x=0f; x!=1f; x+=0.0000001){
   . . .
}
```

creates a scenario in which the loop likely will not terminate after 10,000,000 iterations. The representations used for `x` and the accumulated effect of many iterations cause `x` to not be identical to 1.0, causing the loop to continue to iterate forever.

Similarly, it is undecidable if the Boolean test

```
float x=1.336f;
float y=2.672f;
if (x == (y/2)){
   . . .
}
```

evaluates to true. Given that `x` and `y` are constant values, it is expected that consistent results will be achieved on the same platform. However, it is questionable whether the logic performs as expected when a float that is twice that of another is tested for equality when divided by 2 as above.

Overflow in Java yields `Infinity` and underflow yields 0.0. In neither case is an exception raised.

Floating point operations are platform dependent. Different platforms can yield different results. To counter this problem, Java introduced the `strictfp` keyword. After version 17 of Java, the `strictfp` modifier ensures that all floating point operations yield the same result across different JVMs and platforms. For example:

```
public class FloatingSum {
   public strictfp float sum() {
      float num1 = 5e+7;
      float num2 = 3e+9;
      return (num1 + num2);
   }
   public static strictfp void main(String[] args) {
           FloatingSum t = new FloatingSum();
           System.out.println (t.sum());
       }
}
```

Sometimes very high precision is necessary in calculations. Multiple calculations that exacerbate imprecise calculations and platform differences can cause unexpected results. To achieve higher precision and more predictable performance, the Java class BigDecimal provides a variety of rounding choices to give better control over rounding behavior.

6.4.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.4.5.
- Use thresholds in comparisons instead of equality.
- Use the `strictfp` keyword to ensure consistent floating point results across different JVMs and platforms.
- If possible, use integers instead of floating point numbers.
- Use the `BigDecimal` class to provide better precision such as for monetary or financial calculations and to mitigate rounding issues, when performing high precision arithmetic or where more granular control is needed.

6.5 Enumerator issues [CCB]

6.5.1 Applicability to language

The vulnerability of arrays indexed by enumerations discussed in ISO/IEC 24772-1:2024 6.5 does not directly exist in Java since arrays in Java can only be indexed by `int` values. This mapping can easily be created, however, by indexing an array by the ordinals of an enum type, which can result in a subset of the issues discussed in ISO/IEC 24772-1:2024. In particular, arrays with 'holes' are difficult to create, but maintenance on an enumeration type that inserts values between other enum values could result in array indexing errors.

The vulnerabilities related to user-provided encodings do not exist in Java since the enumerator capability does not rely upon a user-provided encoding. Also, because enum constants are associated with a specific type, the vulnerability associated with the mapping of enums to integer types is absent in Java.

The enumerator capability provided by Java has its own set of vulnerabilities, which are discussed here.

Enums in Java can be done outside of a class or as part of a class. The basic `enum` type (outside of a class `enum`) comprises a set of named discrete constant values as in the example:

```
public enum Weekday {SUN, MON, TUE, WED, THU, FRI, SAT};

String [] WeekdayString = new String[Weekday.SAT.ordinal];
WeekdayString[Weekday.SUN.ordinal] = "Sunday";
```

Each of the keywords must be unique. `Enum` constants are implicitly static and final and cannot be changed once created. The basic `enum` type in Java does not contain any public fields or methods that change state, so the basic `enum` is immutable and cannot be changed.

enum declarations define classes, collectively referred to as *enum types,* which implicitly extend `java.lang.Enum`. Java enum types thus have fields and methods. A more extensive example from the Java Joda.org date and time classes provides an illustration of the associated methods for an `enum`:

```
public enum Month implements TemporalAccessor, TemporalAdjuster {
   JANUARY,   FEBRUARY,   MARCH,      APRIL,
   MAY,       JUNE,       JULY,       AUGUST,
   SEPTEMBER, OCTOBER,    NOVEMBER,   DECEMBER;

    private static final Month[] ENUMS = Month.values();

    public static Month of(int month) {
       if (month < 1 || month > 12) {
          throw new DateTimeException("Invalid value for MonthOfYear: " + month);
       }
       return ENUMS[month - 1];
    }

 // additional methods…
```

```
}
```

However, the flexibility that Java offers with enum types can lead to issues, as the following illustrates:

```
public enum Sea {

  BERING (2261060,3937),
  // ...
  MEDITERRANEAN (2509698,5267);

  private int area;
  public int maxDepth;  // Public

  Continent(int area, int maxDepth) {
    // ...
  }

  public void setArea(int area) {  // Allows modification of private field
    this.area = area;
  }
}
```

When `enum` fields are public, Java allows them to be mutable. This can lead to unexpected consequences, such as accidental or malicious changes to the object, while users expect enums to be immutable. Fields in an `enum` should be private, set in the constructor, and have no setter methods.

Java 14 added the notion of a `switch` expression. A `switch` expression, unlike a switch statement, guarantees coverage of all enumeration values by its choices when applied to a basic `enum` type under the circumstances shown in the examples in 6.27 “Switch statements and static analysis [CLL]”.

6.5.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms from ISO/IEC 24772-1:2024 6.5.5.
- For class-based enums, ensure that `enum` values are not mutable by making members in an `enum type` private, by setting the members in the constructor, and by not providing setter methods.
- Set all `enum` fields to be final.
- Use an `enum` type to select from a limited set of choices to make possible the use of tools to detect omissions of possible values such as in switch statements.

6.6 Conversion errors [FLC]

6.6.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.6 applies to Java, although the consequences are mitigated by checks in the language. In Java, automatic type conversion is permitted if both types are compatible and the target type is wider than the source type, so there can be no loss of data.

In Java, automatic type conversion is permitted if both types are compatible and the target type is larger than the source type, so there can be no loss of data. From the smallest to the largest capacity is the order: `byte`, `short`, `char`, `int`, `long`, `float`, and `double`. For example, a `byte` can be implicitly cast to any of the others since all of the others have a larger capacity, but a `float` can only be implicitly cast to a `double` since there could be a loss of data if a `float` is cast to something smaller, such as an `int`.

There are 19 possible instances of widening primitive conversions in Java. These are:

- `byte` to `short`, `int`, `long`, `float`, or `double`
- `short` to `int`, `long`, `float`, or `double`
- `char` to `int`, `long`, `float`, or `double`
- `int` to `long`, `float`, or `double`
- `long` to `float` or `double`
- `float` to `double`

Though a floating point number can store larger numbers than an integer, precision could still be lost when converting an `int` to a `long` or a `float`, or from a `long` to a `double`. Because of the way floating point numbers are stored, the least significant bits can be lost in the conversion. Converting from the smaller integral types, such as a short to a floating point type or a conversion from an int to a double, will not result in a loss of precision.

Going in the opposite direction from a larger type to a smaller type requires explicit casting. Though there must be explicit casting, the use of explicit casting does not prevent either the production of an incorrect truncated value or the loss of precision (from floating-point) in the conversion. A `long` containing a value not representable in `int` will yield an incorrect value when explicitly downcast to an `int`. Data can be lost when a `float` is explicitly downcast to an `int`.

The vulnerabilities from ISO/IEC 24772-1:2024 6.6 related to the loss of values due to narrowing apply to Java. In addition, the vulnerabilities related to implicit change of units or sets of values with maximums and minimums being exceeded but not generating exceptions also apply.

There are 22 possible instances of narrowing primitive conversions in Java where a potential loss of precision could occur. These are:

- `short` to `byte` or `char`
- `char` to `byte` or `short`
- `int` to `byte`, `short`, or `char`

- `long` to `byte`, `short`, `char`, or `int`
- `float` to `byte`, `short`, `char`, `int`, or `long`
- `double` to `byte`, `short`, `char`, `int`, `long`, or `float`

The use of an incorrect result of a downcast as an out-of-range index value will result in an exception. Thus, the vulnerabilities associated with out-of-range indexing cannot happen in Java. The vulnerability associated with unhandled exceptions is discussed in 6.36 Ignored error status and unhandled exceptions. Behaviours such as termination of the executable or denial-of-service remain.

6.6.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.6.5.
- Check the value of a larger type before converting it to a smaller type to see if the value in the larger type is within the range of the smaller type.
- Use comments to document cases where intentional loss of data due to narrowing is expected and acceptable.
- Be aware that conversion from certain integral types to floating types can result in a loss of the least significant bits.

6.7 String termination [CJM]

This vulnerability does not apply to Java because Java does not use a string termination character.

6.8 Buffer boundary violation (buffer overflow) [HCB]

The vulnerabilities from buffer boundary violation documented in ISO/IEC 24772-1:2024 6.8 resulting in undefined behaviours do not apply to Java, because Java has inherent protections in the language to prevent buffer boundary violations. The vulnerabilities associated with denial of service or termination of the program are possible, depending upon how related exceptions are handled. See 6.36 Ignored error status and unhandled exceptions [OYB].

6.9 Unchecked array indexing [XYZ]

This vulnerability described in ISO/IEC 24772-1:2024 6.9 does not apply to Java because Java performs explicit out-of-bounds checks and raises an exception if the bounds are violated. The vulnerabilities associated with denial of service or termination of the program are possible, depending upon how related exceptions are handled. See 6.36 Ignored error status and unhandled exceptions [OYB].

6.10 Unchecked array copying [XYW]

The vulnerability documented in ISO/IEC 24772-1:2024 6.10 does not apply to Java because Java performs explicit range checks and raises an exception if the ranges are not compatible. The vulnerabilities associated with denial of service or termination of the program are possible, depending upon how related exceptions are handled. See 6.36 Ignored error status and unhandled exceptions [OYB].

6.11 Pointer type conversions [HFC]

With the exception of conversions of references (Java's equivalent to pointers) along the inheritance hierarchies, which are described in 6.44, the vulnerability described in ISO/IEC 24772-1:2024 6.11 does not apply to Java since no other conversions between references are permitted.

6.12 Pointer arithmetic [RVG]

The vulnerability described in ISO/IEC TR 62443-1 6.12 does not apply to Java because Java does not permit arithmetic on references.

6.13 Null pointer dereference [XYH]

6.13.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.13 applies to Java. Prior to making use of a reference to an object, verification needs to be made to ensure that the reference is not null. This can be accomplished through an explicit runtime check or other means of ensuring a reference is not null. Though a null dereference is mitigated in Java by compile-time or run-time checks that ensure that no null-value can be dereferenced, it is better to not rely exclusively on catching the exceptions. The exception `NullPointerException` is implicitly raised upon such dereferencing and needs to be handled, or else the vulnerability of a failing system or components prevails.

An alternative mechanism that has been available since Java 8 called `Optional`, which can be used to encapsulate the potential null values safely to avoid generating a null pointer exception. Optional.IsPresent returns the value `present` if there is a valid value, or `absent` if the reference would be null to let one deal with null values without raising an exception.

6.13.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.13.5.
- Include checks for `null` prior to making use of objects. Less preferably, handle exceptions raised by attempts to dereference null values.
- Consider using the `Optional` class (`java.util.Optional`) to handle objects as `present` or `absent` instead of checking for null values.

6.14 Dangling reference to heap [XYK]

The vulnerability described in ISO/IEC 24772-1:2024 6.14 does not apply to Java because, in Java, an object's lifetime is controlled by the references to the object. Deallocation is only done by the garbage collector if no references to the object exist. If any reference still exists, the object will still exist.

6.15 Arithmetic wrap-around error [FIF]

6.15.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.15 exists in Java. Given the fixed size of integer data types, continuously adding a positive value to an integer eventually results in a value that cannot be represented in the space allocated. For Java, this is defined as an overflow. The integer operators do not indicate overflow, so the potential exists for unexpected, meaningless, or incorrect arithmetic results as a result of the overflow.

Similarly, repeatedly subtracting from an integer leads to underflow. The integer operators also do not indicate underflow in any way.

For example, consider the following code for an integer operation:

```
int foo( int i ) {
   i++;
   return i;
}
```

Calling `foo` with the value of 2147483647 results in `i` containing the value of -2147483648 after the `i++` statement. Continuing execution using such a value could result in unexpected results, such as overflowing a buffer and erroneous operation. The programmer could have been unaware that the value was getting too big to represent in the allocated space. As it is impossible for the compiler or an analysis tool to determine whether overflowing the variable is the expected behaviour, code should be annotated using comments if wrap-around is expected.

6.15.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC TR 24772-1:2024 6.15.5.
- Use defensive programming techniques to check whether an operation will overflow or underflow the receiving data type. For example
    - Check that an operation on an integer value will not cause wrapping, unless it can be shown that wrapping cannot occur. Any of the following operators have the potential to wrap:

```
a + b      a - b     a * b     a++       ++a       a--    --a
a += b     a -= b    a *= b    a << b    a<<=b     -a
```

- Check that an operation on a floating point value will not cause an overflow or underflow unless it can be shown that either cannot occur. Any of the following operators have the potential to overflow or underflow:

```
a + b      a – b     a * b     a/b    a%b       a++       ++a       a--
--a        a += b    a -= b    a *= b    a /= b    a %= b    a << b
a <<= b    -a
```

These techniques can be omitted if it can be shown by static analysis (e.g. at compile time) that overflow or underflow is not possible.

6.16 Using shift operations for multiplication and division [PIK]

6.16.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.16 exists in Java. Often, the use of a shift operator as a substitute for the use of the multiplication and division operators is to increase performance. The Java Virtual Machine (JVM) usually performs such optimizations automatically and can optimize for the current platform. Therefore, there usually is no difference in performance in the program execution when using a shift operator instead of a multiplication or division operator.

Java provides three shift operators: << (left shift), >> (signed right shift), and >>> (unsigned right shift). The signed right shift and the unsigned right shift will produce identical results for positive integers. However, for negative numbers, the two results will be different.

The left operand must be of type `int` or `long`. If the type of the left operand is of type `byte`, `short`, or `char`, then the left operand is promoted to type `int`. Since the promotion performs a sign extension, an unsigned right shift could cause the result of the shift to be an unexpected large positive integer.

Incorrect use of the shift operators could lead to incorrect arithmetic, buffer overruns, and incorrect loops.

6.16.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.16.5. Also, see, *6.15 Arithmetic Wrap-around Error [FIF]*.
- Include both positive and negative values in any testing of calculations involving right shifts to ensure correct operation.

6.17 Choice of clear names [NAI]

6.17.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.17 exists in Java. The possible confusion of names with typographically similar characters is not specific to Java, but Java is as prone to it as any other language. Depending upon the local character set, avoid having names that only differ by characters that can be confused, such as ‘O’ and ‘0’ or ‘l’ and ‘l’.

For Java, the maximum significant name length does not have a limit. Very long names can be problematic from the standpoint of readability and maintainability, even if Java does not impose a limit.

This issue is related to *6.20 Identifier name reuse [YOW]*, as they are both mechanisms by which the programmer could inadvertently use an object other than the one intended. This can lead to user confusion regarding variables and incorrect programming results.

6.17.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.17.5.
- Use names that are clear and non-confusing.
- Use consistency in choosing names.
- Use names that are appropriate to the scope of the code being written, such as short meaningful names in small constructs that involve only local scope, and more meaningful names when non-local classes or methods are being accessed.
- Choose names that are rich in meaning.

6.18 Dead store [WXQ]

6.18.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.18 exists in Java. Because Java is an imperative language, programs in Java can contain dead stores (memory locations that are written but never subsequently read or overwritten without an intervening read). This can result from an error in the initial design or implementation of a program, or from an incomplete or erroneous modification of an existing program. However, it can also be intended behaviour, for example when initializing a sparse array. It can be more efficient to clear the entire array to zero, and then assign the non-zero values, so the presence of dead stores should be regarded as a warning of a possible error, rather than an actual error.

Commented [LW2]: Xxx should this be "and then assign the non-zero values"?

Commented [SJM3R2]: I prefer your modified version

Commented [SM4R2]: Agreed. Changed.

The Java keyword `volatile` indicates to the compiler that the variable should not be cached since its value can be changed by entities outside of the scope of the program or by concurrent threads. A store into a volatile variable is not considered a dead store because accessing such a variable can cause additional side effects, such as input/output (memory-mapped I/O) or observability by a debugger or another thread of execution.

6.18.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.18.5.
- Use compilers and analysis tools to identify potential dead stores in the program.

- Mark all variables observable by another thread or hardware agent as `volatile`, also see *6.61 Concurrent data access [CGX].*

6.19 Unused variable [YZS]

6.19.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.19 exists in Java. Variables can be declared, but never used when writing code or the need for a variable can be eliminated in the code, but the declaration remains. Most Java compilers will report this as a warning and the warning can be easily resolved by removing the unused variable.

Having an unused variable in code indicates that warnings were either turned off during compilation or were ignored by the developer.

6.19.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.19.5.
- Resolve all compiler warnings for unused variables.

6.20 Identifier name reuse [YOW]

6.20.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.20 applies to Java. In Java, as in most languages, nested blocks create nested scopes. Moreover, methods in classes create nested scopes. The usual hiding rule applies to two identically named variables at different levels in these scopes.

Java does allow local variables in a subclass to have the same name as a superclass, as in:

```
class ExampleClass1 {
   public static void main(String[] args) {
      int i;
      class Local {
         int i;
         {
            for (int i = 0; i < 10; i++){
               System.out.println(i);
            }
         }
      }
      new Local();
   }
}
```

Although each of these situations likely resulted from decisions in designing Java that balanced alternatives, such as the need to avoid renaming local variables when such variables were in use in a superclass, these situations can cause issues when performing even routine maintenance such as accidental rebinds after maintenance changes. Variables that are distinct could become intermingled if careful consideration of the scope of the variables is not considered.

Java allows scoping so that a variable that is not declared within a method can be resolved to the class. To differentiate between the class member and a locally declared entity, Java provides the keyword `this` as shown in the following example:

```
public class usernameExample {

   private String username;

   public void setName(String username) {
      this.username = username;
   }
}
```

The keyword "`this`" allows the "`this.username`" to indicate that "`username`" refers to the class variable "`username`" instead of the method variable "`username`". In the following example:

```
public class usernameExample {

   private String username;
   private String oldName;

   public void setName(String username) {
      oldName = username;
      this.username = username;
   }
}
```

"`oldName`" is assigned to the method variable "username" when the programmer intended to assign oldName to the existing username before replacement (`this.username`).

Reuse of any publicly visible identifiers, public utility classes, interfaces, or packages in the Java Standard Library can cause confusion. For instance, naming an identifier, `Timer`, the same name as the public class `java.util.Timer` can cause confusion. Future maintainers of the code could be unaware that the identifier `Timer` refers to a custom class instead of the public class.

6.20.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.20.5.
- Ensure that when the identifier that a method uses is identical to an identifier in the class that the correct identifier is used through the use or non-use of "`this`".
- Choose unique names for any publicly visible identifiers, public utility classes, interfaces, and packages.

6.21 Namespace issues [BJL]

The vulnerability described in ISO/IEC 24772-1:2024 6.21 does not apply to Java since the importation of equally named entities is diagnosed as ambiguous by the compiler, making qualification of the names upon access mandatory.

Packages are one way that namespace issues can be handled when using the same name for two different classes. Should, for example, two classes have the same name, but in different packages, as shown here:

```
com.app1.model (package)
 …
Device (class)
 ...

com.app2.data (package)
…
Device (class)
 ...
```

If these two packages are both imported, then this requires either a name change of the Device class or the use of the full package and class name when referencing them.

An identical rule applies when two or more interfaces with equally named static constants are inherited. The use of the constant must be qualified by the interface name.

6.22 Missing initialization of variables [LAV]

6.22.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.22 related to initialization in a method does not apply to Java. Java requires that every variable in a program be initialized before it is used. With the exception of local variables, Java will assign a default value to variables that are not explicitly initialized. Local variables are not assigned a default value, though the compiler will ensure that each is initialized before use and report an error that a variable might not have been initialized if the compiler cannot determine that a variable has been initialized before use.

The vulnerability described in ISO/IEC 24772-1:2024 6.22 related to circular dependencies does exist in Java. Java does have the problem of circular dependency. If a class A, which has class B's Object, and class B is also

composed of Object of class A, there is an issue of circular dependency. Upon execution, the circular dependency will cause memory to be exhausted and a StackOverflowError to occur.

6.22.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Avoid circular dependencies if possible.
- To remove a circular dependency between objects A and B, create a proxy for one of them and derive that object from the proxy to remove the circular dependency.

6.23 Operator precedence and associativity [JCW]

6.23.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.23 exists in Java. The order of operator precedence for Java is well defined and is listed below in order from highest to lowest precedence.

Operator Precedence

Operators	Precedence
postfix	*expr*++ *expr*--
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||

ternary	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

As shown in the table above, operator precedence and associativity in Java are clearly defined, and mixing logical and arithmetic operations is allowed without parentheses. However, the language has more than 40 operators with the levels of precedence shown, and experience has shown that even experienced programmers do not always get the interpretation of complex expressions correct.

6.23.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.23.5.
- Use parentheses when combining operations in an expression to unambiguously specify the programmer's intent.

6.24 Side-effects and order of evaluation of operands [SAM]

6.24.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.24 exists in Java since Java allows methods and expressions to have side effects. The vulnerability is significantly mitigated by Java's prescribed left-to-right evaluation order so that the occurrence of side effects is deterministic.

If two or more side effects modify the same expression as in:

```
int[] array={10,20,30,40,50,60};
int i=2;
/* … */
i = array[i++];  // outcome is i == 30
```

the behaviour is undefined. Though the rules of Java concerning side effects are fairly straightforward, they can be confusing, such as in:

```
int i = 2;
int j = (i=3) * i;
System.out.println(j);
```

The assignment of `i=3` will occur first, and then the expression `j=i*i;` will be evaluated, leading to the printing out of 9.

Side effects, including assignments, in an argument to && can create an issue, for example in the following `if` statement:

```
if ( (aVar == 10) && (++i  < 25)){
    // do something
}
```

Should `aVar` not be equal to 10, then the `if` statement cannot be true, so the second half of the condition `(++i < 25)` will not be evaluated and thus `i` will not be incremented. Testing can give the false impression that the code is working, when it could just be that the values provided cause evaluations to be performed in a particular order that causes side effects to occur as expected.

Assert statements in Java are used as a diagnostic tool to test assumptions about a program. Assert statements should not contain side effects since although assert statements are enabled by default, the assert statements can be disabled as part of the build process. This could change the program results since the assert statements would not be executed if the assert statements are disabled.

6.24.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.24.5.
- Prohibit embedding ++, --, etc. in expressions.
- Simplify expressions to reduce or eliminate side effects, to avoid potential confusion and to improve maintainability.
- Prohibit side effects in assert statements.

Deleted: other

6.25 Likely incorrect expression [KOA]

6.25.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.25 exists in Java. Java has several instances of operators which are similar in structure, but vastly different in meaning, for example confusing the comparison operator "==" with assignment "=". Using an expression that is syntactically correct, but which could just be a null statement can lead to unexpected results. Consider:

```
int x, y;
/* … */
if (x = y){
  /* … */
}
```

A fair amount of analysis is likely required to determine whether the programmer intended to do an assignment as part of the `if` statement (valid in Java) or whether the programmer made the common mistake of using an "=" instead of a "==". In order to prevent this confusion, it is suggested that any assignments in contexts that are

easily misunderstood be moved outside of the Boolean expression. This would change the example code to the semantically equivalent:

```
int x,y;
/* … */
x = y;
if (x != 0) {
   /* … */
  }
```

This would clearly state what the programmer meant and that the assignment of `y` to `x` was intended.

Confusion of "`==`" and the `equals()` method can also cause problems. Consider:

```
int a=5;
int b=5;
if (a==b) {
       System.out.println("a==b is TRUE");
}
```

In this case, "`a==b is TRUE`" will be printed since the values contained in `a` and `b` are the same. However, in the following example:

```
String obj1 = new String("xyz");
String obj2 = new String("xyz");
if (obj1 == obj2)
   {
   System.out.println("obj1==obj2 is TRUE");
   }
else
   {
   System.out.println("obj1==obj2 is FALSE");
   }
```

"`obj1==obj2 is FALSE`" will be printed since the memory locations where `obj1` and `obj2` are stored are different. "`obj1==obj2 is TRUE`" would only be printed if the memory locations of `obj1` and `obj2` were the same as in the case:

```
String obj1 = new String("xyz");
String obj2 = obj1;
```

It is also possible for programmers to insert the "`;`" statement terminator prematurely. However, inadvertently doing this can drastically alter the meaning of code, even though the code is valid, as in the following example:

```
int a,b;
/* … */
if (a == b);  // the semi-colon will make this a null statement
{
 /* … */
```

```
    }
```

Because of the misplaced semi-colon, the code block following the `if` will always be executed. In this case, it is extremely likely that the programmer did not intend to put the semi-colon there and thus will end up with unexpected results.

Java also uses the "`>>>`" for the unsigned shift operator. This can be easily confused with the "`>>`" (signed right shift) which will produce identical results for positive values, but very different values for negative values.

Each of the following would be clearer and have less potential for problems if the embedded assignments were conducted outside of the expressions:

```
       int a,b,c,d;
       /* … */
       if ((a == b) || (c = (d-1))){. . .}  // the assignment to c will not
                                            // occur if a is equal to b
```

or:

```
      int a,b,c;
      /* … */
      foo (a=b, c);
```

Each is a valid Java statement, but each can have unintended results. They are better formulated as :

```
       int a,b,c,d;
       /* … */
       c = d-1;
       if ((a == b) || c) {. . .}
```

or

```
       int a,b,c;
       /* … */
       a = b;
       foo (a, c);
```

6.25.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.25.5.
- Explain statements with interspersed comments to clarify programming functionality and help future maintainers understand the intent and nuances of the code.
- Prohibit assignments embedded within expressions.
- Give null statements a source line of their own to clarify the intention that a statement was meant to be a null statement.

6.26 Dead and deactivated code [XYQ]

6.26.1 Applicability to language

Java allows the usual sources of dead code described in ISO/IEC 24772-1:2024 6.26 that are common to most conventional programming languages. To avoid dead code, there must be an execution path from the beginning of the constructor, method, instance initializer, or static initializer that contains the statement to the statement itself. If not, the result will in many cases be a compiler error or warning.

Java will not produce a compiler error or warning in what seems to be obvious cases of dead or deactivated code, such as in the following example:

```
{
int num = 10;
while (num > 15) {
     val = 5;
   }
}
```

Even though the statement "`val = 5;`" can never be reached, this code will not result in a compiler warning or error. `While` statements, `do` statements and `for` statements are afforded special treatment. Except in the case where the `while`, `do`, or `for` expressions have the constant value of `true`, the values of the expressions are not taken into account in determining reachability.

Java permits the use of line-oriented comments // or block oriented comments /* . . . */ which can be used to remove code from compilation by the compiler. Block oriented comments make it difficult for reviewers to distinguish active code from deactivated code.

6.26.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.26.5.
- Use "`//`" comment syntax instead of "`/*…*/`" comment syntax to avoid the inadvertent commenting out of sections of code.
- Use an IDE that adds additional capabilities to detect dead or unreachable code.

6.27 Switch statements and lack of static analysis [CLL]

6.27.1 Applicability to language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.27 apply to Java. Java contains both a switch statement and a switch expression.
!!! Reference JEP 361).

Java currently provides multiple styles of "switch" alternatives:

- The "old-style" switch statement that permits only a single value for each case choice and permits fall-through between cases using the ":" notation.
- The "new-style" switch statements (Java 21 and later) that permit multiple cases for each branch, adds implicit breaks at the end of the branch when the arrow notation "->" is used to begin the case.
- The switch expression (Java 14 and later) that returns a single value as a result, prohibits modification of all variables and uses new style "->" or old-style ":" notations for switching.
- An enhanced switch statement, where either (i) the type of the selector expression is not `char, byte, short, int, Character, Byte, Short, Integer, String`, or an `enum` type, or (ii) there is a case pattern or null literal associated with the switch block.

Pattern-matching and additional guards can be used to further constrain a case in "new-style" switch syntax, as in:

```
case String s when s.length() == 2 -> …
```

Old-style Java switch statements are error-prone as documented in ISO/IEC 24772-1:2024 and are discouraged for new code. If there is not a default case and the selecting value does not match any of the cases, then control shifts to the next statement after the switch statement block, which can cause logic errors. If such old-style code is present, an update that uses the "->" syntax as part of a switch expression or switch statement will improve static analysis and prevent unintended fall-throughs.

Switch expressions and switch statements that use the "->" syntax do not permit a fall-through from one case to another and hence do not permit a "break" in the construct.

Switch expressions and enhanced switch statements check the exhaustiveness of choices during compilation; for `enum` types and sealed classes, coverage is checked statically; for all other types, such as `int`, the presence of a `default` switch label is required by the language. For other switch statements, no checks for exhaustiveness are performed, making them vulnerable to unintentional fall-throughs.

When pattern matching is used in Java `switch` statements or expressions, it is important to be aware of case dominance issues where a more-general pattern unintentionally matches cases that should be handled by a more-specific pattern. This scenario can result in unexpected behavior if the order of cases is not carefully implemented and maintained. Java enforces a sequential scenario when potential overlap exists in two or more cases; the first matching SwitchRule is taken.

The presence of default SwitchRules carries the risk that the accidental omission of cases fails to be discovered, which can be corrected by explicitly enumerating all cases that are not error or "don't care" cases.

Commented [SJM5]: Do we want to add a definition for this to Section 3? For example … used `when` keyword …

Deleted: eration type

Deleted: and will fail compilation if coverage is not complete or if there is no default case. The example above would fail the coverage check if one of the enumeration literals (e.g. FRIDAY) is missing. If

Deleted: are used, then

Deleted: coverage is not checked and

Deleted: case

Deleted: necessary

Deleted: to catch unexpected cases

Deleted: potentially

Deleted: are

Another potential vulnerability is the lack of a `null` SwitchRule in an enhanced switch statement or switch expression over a value of reference type. When such a construct is invoked with a `null` value, a `NullPointerException` will occur.

6.27.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:202024 6.27.5 for Java switch statements and expressions.
- Prefer enhanced switch statements and switch expressions to guarantee exhaustiveness.
- Prefer the new style switch statements to the old style for all new code and for updates to old code.
- Prefer `enum` types with switch expressions to enable static completeness checks for the cases.
- For switch statements, include a `default` case to provide exhaustiveness of coverage and to support error handling.
- Prefer a coding style that requires explicit SwitchLabels instead of `default`.
- When using pattern matching in a `switch` statement or expression, order the case alternatives sequentially from most specific to least specific (enforced by the compiler in class-membership only).
- Include a `null` case to handle `null` values gracefully when switching over reference types.

6.28 Non-demarcation of control flow [EOJ]

6.28.1 Applicability to language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.28 apply to Java. Java lacks a keyword for an explicit terminator. Therefore, it is often not readily apparent which statements are part of a loop construct or an `if` statement.

Consider the following section of code:

```
void foo(int a, int[] b) {
    int i=0, count=0;
    //
    a = 0;
    for (i=0; i<10; i++)
        a += b[i];        //Did the programmer intend to include
                          // the next statement in the branch?
                          // If so, the programmer failed.
        Count++;
    System.out.printf("a=%d count=%d\n", a, count);
}
```

The programmer could have intended both "`a += b[i];`" and "`count++;`" to be the body of the loop, but as there are no enclosing brackets, the statement "`count++;`" is only performed once. Similarly, for if statements, the inclusion of statements on branches is susceptible to this error, for example:

```
int a,b,i;
```


Commented [SJM6]: Explicit 'return' is not permitted. The return is implied when using the '->' annotation.

Deleted: ¶
A `switch` expression chooses the correct case label, evaluates the selected expression, and returns its resulting value. The switch expression can be used as a direct replacement for the switch statement. Switch expressions do not permit a fall-through from one case to another and hence do not permit a "break" in the construct. ¶
¶
Switch expressions permit multiple case expressions to select an alternative, for example given:¶
¶

```
  enum Days {MONDAY, TUESDAY, WEDNESDAY,
THURSDAY, FRIDAY, SATURDAY, SUNDAY};¶
   Boolean isWeekday;¶
```

¶
the switch expression could have the form:¶
¶

```
public static Boolean isWeekDay (Days day) ¶
    {¶
    return switch(day) ¶
      case MONDAY, TUESDAY, WEDNESDAY, //
multiple cases can be combined¶
          THURSDAY, FRIDAY              ->
true;¶
                  // Control transfers to the
end of the switch block.¶
      case SATURDAY, SUNDAY             ->
false;¶
}¶
```

... [1]

Deleted: <#>Consider using switch expressions instead of switch statements and converting any switch statement to the corresponding switch expression.¶

Deleted: <#>basic

Deleted: <#>the

Deleted: <#>For switch statements, adopt a coding style that requires every nonempty case statement to be terminated with a `break` statement. Alternatively, if a direct fall through from one nonempty case to another is required that would violate the coding style, then this should be clearly documented by a comment, preferably one recognized by the analysis tool used.¶
Adopt a coding style that permits the selected language processor and analysis tools to verify that all cases are covered. Where this is not possible, use a default clause that diagnoses the error.¶
Adopt

Deleted: <#>the

Deleted: <#> of complex switch statements. This also applies to switch expressions where coverage is not checked by the language....

Deleted: statements

Commented [SM7]: Sean, please check.

Commented [SJM8R7]:

```
Object obj = "Hello";
switch (obj) {
    case String s ->
         System.out.println("It's
         a string");
    case Object o ->
         System.out.println("It's
```

... [2]

Commented [SJM9R7]: Can this comment be closed or do we want to include some of it in the text?

Deleted: .

Deleted: using type patterns in a `switch` statement

```
//
if (i == 10){
   a = 5;     // This is correct
   b = 10;
}
else
   a = 10;
   b = 5; // Incorrect since b = 5 will execute after either branch
```

If the assignments to `b` were added later and were expected to be part of each `if` and `else` clause (they are indented as such), the above code is incorrect: the assignment to `b` that was intended to be in the `else` clause is unconditionally executed.

If statements in Java are susceptible to another control flow problem since there is not a requirement for there to be an `else` statement for every `if` statement. An `else` statement in Java always belongs to the most recent `if` statement without an `else`. However, the situation could occur where it is not readily apparent to which `if` statement an `else` belongs due to the way the code is indented or aligned. For example:

```
int n1, n2, n3, rating;
rating = 0;
if (n1 >= n2)
   if (n1 >= n3)
      rating = n1;
else             // visually appears to be connected to first 'if'
                 // but actually belongs to the innermost 'if'
   rating = n3;
```

Based on the indentation, it would appear that the `else` belongs to the first `if`. However, since the `else` belongs to the most recent `if` without an `else` statement, the `else` would instead belong to the second `if` statement. The intended effect can be achieved through the use of braces as follows:

```
int n1, n2, n3, rating;
rating = 0;
if (n1 >= n2) {
   if (n1 >= n3) {
      rating = n1;
   }
}
else {          // this else belongs to the outermost 'if'
   rating = n3;
}
```

6.28.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms provided in ISO/IEC 24772-1:2024 6.28.5.
- Enclose the bodies of `if`, `else`, `while`, `for`, and similar constructs in braces to disambiguate the control flow.

6.29 Loop control variable abuse [TEX]

6.29.1 Applicability to language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.29 apply to Java. Java allows the modification of loop control variables within the loop, which can cause unexpected behaviour and can make the program more difficult to understand.

Since the modification of a loop control variable within a loop is infrequently encountered and unexpected, reviewers of Java code might not expect it and hence miss noticing the modification or not recognize its significance. Modifying the loop control variable can cause unexpected results. Loops can become infinite if the loop control variable is assigned a value such that the loop control test is never satisfied. Loops can unintentionally execute less iterations than expected, such as:

```
int a,i;
for (i=1; i<10; i++){
   …
   if (a > 7) {
      i = 10;
   }
   …
}
```

which would cause the for loop to exit once `a` is greater than `7`, regardless of the number of iterations that have occurred.

Java does not require the loop control variable to be an integer type. If, for example, it is a floating point type, the test for completion should not use equality or inequality, as floating point rounding can lead to mathematically inexact results, and hence an unterminated loop. The following can loop ten times or can loop indefinitely:

```
for (float x = 0.0f; x != 10.0f; x += 1.0f){
   . . .
}
```

The following is an improvement:

```
for (float x = 0.0f; x < 10.0f; x += 1.0f){
   . . .
}
```

Rounding can cause this loop to be performed ten or eleven times. To ensure this loop is performed ten times, `x` could be initialized to `0.5f`.

Enhanced `for` loops in Java provide for a simplified way of iterating through all elements of an array in order, as in the following:

```
for (int myIndex : myArray) {
   System.out.println (myIndex);
}
```

Unlike the conventional `for` statement, modifications to the loop variable do not affect the loop's iteration order over the array. This can cause unexpected results. Thus, it is better to declare the loop control variable as final to prevent this possible confusion, as the following illustrates:

```
for (final int myIndex : myArray) {
   System.out.println (myIndex);
}
```

By declaring `myIndex` as `final`, the Java compiler will reject any assignments within the loop.

6.29.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms of ISO/IEC 24772-1:2024 6.29.5.
- Prohibit the modification of a loop control variable within a loop.
- Declare all enhanced `for` statement loop variables final to cause the Java compiler to flag and reject any assignments made to the loop variable.
- Prohibit the use of floating point types as a loop control variable.
- Use enhanced `for` loops to eliminate the need for a loop control variable.

6.30 Off-by-one error [XZH]

6.30.1 Applicability to language

The vulnerability as documented in ISO/IEC 24772-1:2024 6.30 applies to Java.

Arrays are a common place for off-by-one errors to manifest. In Java, arrays are indexed starting at zero, causing the common mistake of looping from 0 to the size of the array as in:

```
public class arrayExample {
   public static void main (String[] args) {
```

```
        int intArray = new int[10];
        int i;
        for (i=0, i<=10, i++){
           a[i] = 5;
           . . .
        }
        return (0);
     }
  }
```

Java does provide protection in this case as any attempt to access an array with an index less than zero or greater than or equal to the length of the array will result in an `ArrayIndexOutOfBoundsException` to be thrown.

Java provides mechanisms to reduce the places where explicit bounds tests are required, such as:

1. Whole object copying, such as arrays, class objects, and containers;
2. `for` loops that run the entire structure without an explicit index count;
3. Java Maps provide a more secure way than arrays to manipulate objects because iterators implicitly obey bounds.

Programs in Java are susceptible to the usual off-by-one errors, such as looping less than the desired amount. Such errors will usually only be detected by doing thorough testing of the program.

6.30.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.30.5.
- Use careful programming, testing of boundary conditions, and static analysis tools to detect off-by-one errors in Java.
- Use Java facilities for whole-object copying.
- Use Maps and iterators in lieu of explicitly counted loops for accessing structures.

6.31 Unstructured programming [EWD]

6.31.1 Applicability to language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.31 apply to Java. Since Java is an object-oriented language, the structure inherent in the language helps to lead to well-structured programs. The Java language does not contain the `goto` statement. However, even though Java sets forth this structure and in spite of it, programmers can create unstructured code. Java does have the `continue`, `break`, `throw`, and `return` statements that can create complicated control flows when used in an undisciplined manner. Unstructured code can be more difficult for Java static analyzers to analyze. It is sometimes used deliberately to obfuscate the functionality of software. Code that has been modified multiple times by an assortment of programmers to add or remove functionality or to fix problems can be prone to become unstructured.

Many style guides recommend the use of no more than one `return` statement in a method. This style originated in assembly code where each return went directly back to the function caller, which is not true in modern

languages. In compiled Java code, the return statement always transfers to compiler-generated wrapper code that checks for exceptions, finalizes temporary variables and other state, and checks for a legal value to be returned.

Multiple returns are only a problem if various branches within a function perform disparate calculations and some return from within a branch while others take alternative action. Code, where a simple calculation such as a case expression results in a return from each branch with a unique value, is a valid pattern.

6.31.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.31.5.
- Write clear and concise structured code to make code as understandable as possible.
- Restrict or prohibit the use of `continue` and `break` in loops to encourage more structured programming.

6.32 Passing parameters and return values [CSJ]

6.32.1 Applicability to language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.32 apply to Java. All Java data types are permitted as the type of a method parameter. Method arguments should be validated to ensure that their value falls within the bounds of the method's anticipated values. Java passes any parameter that is of one of the eight primitive types by value. The parameter is evaluated and its value is assigned to the formal parameter of the method or constructor that is being called. Parameters provide information to the method from outside the scope of the method.

```
Public static int minFunction (int n1, int n2) {
    int min;
    if (n1 > n2){
        min = n2;
        }
    else {
        min = n1;
    }
    return min;
}
```

When the value of an object is passed as a parameter, effectively the reference to the object is passed. This allows the object to be changed in the method.

```
 Public class testObject {
      private int value;

      public static void main(String[] args) {
           testObject p = new testObject();
           p.value = 10;
           System.out.println("Before calling: " + p.value);
           increment(p);
           System.out.println("After calling: " + p.value);
      }

      public static void increment(testObject a){
           a.value++;
      }
}
```

However, when multiple parameters are passed, a vulnerability called "aliasing" can occur. For example

```
      public static void main(testObject a, testObject b) {
           a.value = 7;
           b.value = 21;
           System.out.println(a.value + b.value);  // Normally prints 28
                                                   // Sometimes prints 42
      }
```

Surprisingly, the value of `42` is printed in cases when main is called with variables denoting the same object, i.e. `main(x,y)` when `x == y`. Similar problems arise when the current instance is passed as a parameter to one of its methods.

Java also allows expressions such as the post increment expression "`i++`" to be passed as parameters. This can cause confusion and it is safer to perform the increment in a separate, prior statement to the call. The order of evaluation of parameters proceeds from left to right and care should be taken when side effects modify the same variables such as "`testMethod (i++, ++i)`".

6.32.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.32.5.
- Avoid expressions with side effects as parameters to methods.
- Write code to account for potential aliasing among parameters, including the current instance `this`.
- Avoid the use of expressions with side effects for multiple parameters to functions, since the order in which the parameters are evaluated and hence the side effects occur is unspecified.

6.33 Dangling references to stack frames [DCM]

This vulnerability as documented in ISO/IEC 24772-1:2024 6.33 does not apply to Java, because in Java any reference that does not point to a valid object will be garbage collected. References are also passed by value, meaning that Java creates a copy of the reference and passes the copy to the method.

Deleted: 6.33.1 Applicability to language

6.34 Subprogram signature mismatch [OTR]

6.34.1 Applicability to language

Except for vulnerabilities associated with a variable number of arguments, i.e. `varargs`, the vulnerability as documented in ISO/IEC 24772-1:2024 6.34 does not apply to Java since the compiler diagnoses mismatches.

If there are multiple member methods that are potentially applicable to a method invocation, overload resolution in the compiler determines the actual method to be called or, if multiple candidates remain, a compiler error results.

There are two concerns identified with this vulnerability. The first is if a subprogram is called with a different number of parameters than it expects. The second is if parameters of different types are passed than are expected.

Java supports variadic functions/methods, termed varargs, as shown in the following example:

```
public class classSample {
    void demoMethod(String… args) {
      for (String arg: args) {
         System.out.println(arg);
      }
    }

   public static void main(String args[] ){
      new classSample().demoMethod("water", "fire", "earth");
      new classSample().demoMethod("wood", "metal");
   }
}
```

A `varargs` argument must be the last argument in a multiple argument list and multiple `varargs`, even if of different primitive types, are not allowed. Though `varargs` can be useful, their usage can cause performance issues and possibly memory consumption issues leading to unexpected results. `Varargs` could also lead to heap pollution, which occurs when a variable of a parameterized type refers to an object that is not of that parameterized type.

6.34.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can avoid the use of the variable argument feature except in rare instances and instead use arrays to pass parameters.

6.35 Recursion [GDL]

6.35.1 Applicability to language

Java permits recursion, hence is subject to the vulnerabilities documented in ISO/IEC 24772-1:2024 6.35.

Deleted: problems

Deleted: described

6.35.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the guidance contained in ISO/IEC 24772-1:2024 6.35.5.
- If recursion is used, then catch the `java.lang.OutOfMemoryError` exception to handle insufficient storage due to recursive execution.

6.36 Ignored error status and unhandled exceptions [OYB]

6.36.1 Applicability to language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.36 exists in Java. Java mitigates the vulnerability by enforcing the handling of checked exceptions, but not for unchecked exceptions.

Java offers a set of predefined exceptions for error conditions that can be detected by checks that are compiled into a program. In addition, the programmer can define exceptions that are appropriate for their application. These exceptions are handled using an exception handler. Exceptions can be handled in the environment where the exception occurs or can be propagated out to an enclosing scope.

Java has two types of exceptions: checked and unchecked. A checked exception requires a response, and the existence of a response is checked at compile time. A method must either handle the exception or specify the exception using the `throws` keyword. This reduces the number of exceptions that are not properly handled. Unchecked exceptions are subclasses of RunTimeException and do not require handling since recovery is likely difficult or impossible, or the addition of an exception would not add significantly to the program's correctness and could be viewed as simply cluttering up the program needlessly.

Lack of handling of checked exceptions, such as `FileNotFoundException`, is detected at compile time. There must be a `try` and `catch` block to handle the exception, as in the following example:

```
public static void main(String[] args)
{
    try
    {
        FileReader file = new FileReader("datafile.txt");
```

```
        }
        catch (FileNotFoundException e)
        {
            // print the stack trace for this
            // throwable object on the standard error output stream
            e.printStackTrace();
        }
    }
```

Thus, the vulnerability of unhandled exceptions documented in ISO/IEC 24772-1:2024 6.36 does not apply to checked exceptions. The vulnerability does exist for unchecked exceptions.

Checked exceptions should not simply be suppressed by catching the exceptions with an empty or trivial catch block. The catch block must either recover from the exceptional condition, rethrow the exception by propagating it to an enclosing scope or throw an exception that is appropriate to the context of the catch block.

Unchecked exceptions, such as `ArithmeticException`, can be ignored in the program and the program will still compile. However, should an exception occur, how the exception should be handled might not be specified. Unchecked errors are mainly due to programming errors that should be fixed to prevent the unchecked exception from occurring again.

Variables defined in a try block are only local, so if they are needed in the catch block, define and initialize the variables outside of the try block.

6.36.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.36.5.
- Use *try-with-resources*, which extends the behaviour of the try/catch construct to allow access to resources without having to close them afterwards, as the resource closures are done automatically.
- Use unchecked exceptions in case an unanticipated exception occurs.
- Use *try-with-resources* for automatic resource management.

6.37 Type-breaking reinterpretation of data [AMV]

6.37.1 Applicability to language

Except for methods in `sun.misc.Unsafe`, Java is not subject to the vulnerabilities documented in ISO/IEC 24772-1:2024 6.37.

`sun.misc.Unsafe` provides some low level programming features, such as reinterpretation of data, but, as its name implies, is considered unsafe for general use. Documentation is not widely available, and its use usually relies on miscellaneous web postings, leading to even more unsafe use.

6.37.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Use `sun.misc.Unsafe` only when absolutely necessary to reinterpret data and carefully document its use.
- Consider segregating intended reinterpretation operations into distinct subprograms, as the presence of reinterpretation greatly complicates program understanding and static analysis.

6.38 Deep vs. shallow copying [YAN]

6.38.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 6.38 applies to Java.

The usual way of performing a copy of an object in Java is through the use of the `clone()` method. Using the default implementation of the `clone` method will result in a shallow copy with all of the resulting issues associated with a shallow copy. Unexpected results can occur if the elements of values are changed via some other reference. Using a deep copy that makes the original and cloned object totally disjoint comes at the cost of efficiency and performance. To create a deep copy of an object, the clone method has to be overridden. Since a deep copy is the exact duplicate of the original object, extensive use of deep copies can cause considerable dynamic memory use.

Another way of copying objects is to serialize them through the `Serializable` interface. An object can be serialized and then be deserialized to a new object. Since the constructor is not used for objects copied with clone or serialization, this can lead to improperly initialized data and prevents the use of the final member fields.

The constructor is not used for objects copied with `clone` or serialization. This can lead to improperly initialized data and prevents making member fields final.

6.38.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.38.5.
- Ensure that deep-copied objects are initialized properly.
- Be careful of excessive memory use when using deep copying.

6.39 Memory leaks and heap fragmentation [XYL]

6.39.1 Applicability to language

The vulnerabilities as documented in ISO IEC 24772-1 6.39 apply to Java but are mitigated by Java's built-in garbage collectors.

Java has automatic memory management along with several built-in Garbage Collectors (GC), including Serial, Parallel, G1, Concurrent Mark Sweek (CMS), Shenandoah, and the newest Z Garbage Collector (ZGC). Java selects the best garbage collector based on the platform, Java version, and JVM implementation, but the developer can override this selection and pick another GC. Nevertheless, memory leaks can occur in Java applications. Although objects are no longer being used by an application, the Garbage Collector cannot remove them from working memory if the objects are still being referenced. Left unchecked, this can result in the application increasingly consuming resources until a fatal `OutOfMemoryError` occurs.

Many scenarios can lead to a memory leak:

- Referencing a memory intensive object with a static field ties its lifecycle to the lifecycle of the JVM itself.
- Unclosed resources, such as database connections, input streams, and session objects.
- An instance of a non-static inner class (anonymous class) always requires an instance of the enclosing class and has, by default, an implicit reference to its containing instance. If this instance of the inner class object is used in an application, then even after the instance of the containing class goes out of scope, the instance of the containing class will not be garbage collected as long as the instance of the inner class exists.
- Overriding a class' `finalize()` method and then the objects of that class are not instantly garbage collected since the garbage collector queues them for finalization, which occurs at a later point in time.
- Reading a large `String` object and then calling intern() on that object will result in it being stored in the string pool, which is located in PermGen (permanent memory), where it will stay as long as the application runs.
- Using the `ThreadLocal` construct to isolate state to a particular thread and thus achieve thread safety so that each thread will hold an implicit reference to its copy of a ThreadLocal variable and will maintain its own copy instead of sharing the resource across multiple threads, as long as the thread is alive. This can introduce memory leaks if not used carefully.
- Calling applications written in programming languages that are prone to memory leaks.

Deleted: s.

6.39.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.39.5.
- Use a heap-analyzer tool to assist in detecting memory leaks.

- Enable verbose garbage collection to document and understand detailed traces of the garbage collector's actions.
- Use Java profiler tools that monitor and diagnose memory leaks.
- Set references to null once they are no longer needed so that the garbage collector can collect the designated object.
- Use reference objects from the `java.lang.ref` package instead of directly referencing objects to allow them to be easily garbage collected.

6.40 Templates and generics [SYM]

6.40.1 Applicability to language

The vulnerability as described in 24772-1:2024 6.40 exists in Java.

Generics allow programmers to specify, with a single method declaration, a set of related methods or, with a single class, a set of related types. At the heart of Java generics is type safety, which allows invalid types to be caught at compile time. The emphasis on type safety causes many problems to be averted.

Generics in Java are implemented with type erasure. That is, the generic type information is only available at compile time and not in the bytecode or at runtime. Thus, generics do not affect the signature of a method, resulting in the same signature for methods that have the same name and the same arguments. This can result in signature collision. In addition, this does not allow one to determine parameterized types using reflection.

Java allows the use of upper bounded, lower bounded and unbounded wildcards (“?”) in a generic. The use of a wildcard in generic programming can be useful but can also introduce uncertainty as to the intention during the maintenance cycle. Generic wildcards also add a level of complexity that might not be fully understood or comprehended by Java programmers who know the basics of generics, but not more sophisticated techniques like wildcard.

6.40.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.40.5.
- Use generic wildcards carefully and only when needed.
- Follow the acronym PECS for “Producer Extends, Consumer Super” – use extends when getting values out of a data structure, use super when putting values into a data structure, and use an explicit type when doing both. See 6.42 Violations of the Liskov substitution principle or the contract model.
- Use different names for methods to get different signatures.

Deleted: signature

6.41 Inheritance [RIP]

6.41.1 Applicability to language

The vulnerabilities as described in 24772-1:2024 6.41 exist in Java. Java supports inheritance but does not support multiple inheritance or cyclic inheritance for classes. This allows Java to avoid problems associated with multiple inheritance. Interfaces support multiple inheritance, but the vulnerabilities are centered on the inheritance of the implementation, which is missing from interfaces.

Java allows subclasses to override inherited methods, potentially causing difficulty in determining where in the hierarchy an invoked method is actually defined. An overriding method must specify the same name, parameter list, and return type as the method being overridden. The use of the keyword final in a method header will prevent the method from being overridden. For example, `final String getDate` will prevent `getDate` from being overridden in a subclass as the compiler will report an error if the method is overridden in a subclass.
The issues arising from inheritance are absent when composition is used, especially when using library classes. Apart from this mitigation to accidental or malicious overriding, all other vulnerabilities described in ISO/IEC 24772-1:2024 6.41 for single inheritance apply.

Version 17 of Java finalized sealed classes that restrict the extension of that class by subclasses to subclasses permitted to do so either explicitly or by being defined in the same module. This restriction brought some order to the Java derivation hierarchies but introduced the vulnerability caused by late additions of subclasses in the same module not intended to be so permitted.

Potential issues can arise when developers misuse the sealed feature for classes, leading to situations where the compiler cannot guarantee exhaustive checks for subclasses, potentially causing unexpected behavior in code that relies on inheritance hierarchies, especially when combined with pattern matching in switch statements. (XXXCHECK!!!)
If a sealed class does not explicitly list all permitted subclasses in its `permits` clause, and a new subclass is created outside the specified list, the compiler might not catch this as an error, potentially leading to unexpected behavior in code that assumes only the listed subclasses exist. (XXXMAYBE -- Check).

For vulnerabilities associated with classes and switch statements/expressions, see 6.27 … .

6.41.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.41.5.
- Use composition as an alternative to inheritance.
- Use interfaces when multiple inheritance is required.
- Keep the inheritance graph as shallow as possible to simplify the review of inheritance relationships and

Deleted: I

Commented [SM10]: Check these with real code.

Commented [SM11]: XXX - Sean - check with real code, please.

Commented [SM12]: Cover in 6.27 and move there.

Commented [SM13]: XXX fill in.

method overridings.

- Explicitly list all allowed subclasses in the `permits` clause of a sealed class to ensure the compiler can check for exhaustive subclass coverage.
- When using pattern matching with a switch statement on a sealed class, take advantage of the possibility to check that all possible subclasses are covered by a case.
- Evaluate the desirability of a sealed class and design the permitted subclasses carefully to balance flexibility and control.

Commented [SM14]: Move to 6.27.

6.42 Violations of the Liskov substitution principle or the contract model [BLP] Error! Bookmark not defined.

6.42.1 Applicability to language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.42 apply to Java. Since Java supports inheritance, it is important that developers abide by the Liskov substitution principle. In particular, no restrictions on parameters to an overridden method can be permitted, if that restriction does not exist in the base class.

Precondition and postcondition checks are not supported in Java, but assertions can be used to implement them at runtime.

6.42.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.42.5.
- Use assertions to implement precondition and postcondition checks.
- Use static analysis tools to verify assertions.

6.43 Redispatching [PPH] Error! Bookmark not defined.

6.43.1 Applicability to language

The vulnerability as documented in ISO/IEC 24772-1:2024 6.43 exists in Java. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time rather than compile time. When an overridden method is called through a superclass reference, Java determines which version (superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made dynamically at run time. For methods that are overridden in subclasses in the object being initialized, the overriding methods are used and thus the redispatching problem of infinite recursion could manifest.

6.43.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.43.5.
- Prevent redispatching where it is not necessary and document the behaviour if redispatching is required.

6.44 Polymorphic variables [BKK]

6.44.1 Applicability to language

The vulnerabilities related to upcasts in ISO/IEC 24772-1:2024 6.44 exist in Java.

The vulnerabilities related to unsafe casts do not exist in Java since unsafe casts are not permitted in Java.

Downcasts from a superclass to a subclass in the same type hierarchy are legal and will not be flagged by the compiler. In the following example:

- `Subclass` extends `Superclass` and declares `method()`.
- `BadDowncast` declares a `main()` method that instantiates `Superclass`. `BadDowncast` then downcasts this object to `Subclass`, which raises the exception ClassCastException because the instance currently designated by `subclass` is not an instance of `Subclass`.
- If, however, the value of `Superclass` were an instance of `Subclass`, the downcast will succeed and `subclass.method()` will be called.

```
class Superclass
{
}

class Subclass extends Superclass
{
    void method()
    {
    }
}

public class BadDowncast
{
    public static void main(String[] args)
    {
        Superclass superclass = new Superclass();
        Subclass subclass = (Subclass) superclass; // raises an exception
        subclass.method();
    }
}
```

6.44.2Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024.

6.45 Extra intrinsics [LRM]

The vulnerability as documented in ISO/IEC 24772-1:2024 6.45 does not exist in Java, since Java does not provide any intrinsic that can conflict with a user-defined name. All language-provided capabilities outside the standard operators reside in named library classes, and the usual name resolution rules apply.

6.46 Argument passing to library functions [TRJ]

6.46.1 Applicability to language

The vulnerability as documented in ISO/IEC 24772-1:2024 6.46 applies to Java.

Parameter validation should always be performed in public methods since the caller is out of scope of its implementation. In public methods or other instances where such validation is not performed or it is unsure whether it is performed, the calling routine should perform parameter validation.

There are open source libraries that provide for preconditions to be placed on parameters. For instance, the open source library Guava provides utilities such as `checkArgument`, as illustrated in this example:

```
public static double sqrt (double value)
{
   Preconditions.checkArgument(value >= 0.“, "negative value:”%s", value);
   // …perform calculation of the square root

 }
```

6.46.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.46.5.
- Avoid assumptions about the values of parameters.
- Implement precondition checks to validate parameters and establish a strategy for each interface to check parameters in either the calling or receiving routines.

6.47 Inter-language calling [DJS]

6.47.1 Applicability to language

The vulnerabilities in ISO/IEC 24772-1:2024 6.47 exist in Java when working with components developed in other languages. Interfacing with other languages can be difficult. Though Java attempts to make interfacing with other languages easier, it can still be rather complicated. Foreign Function Interfaces (FFI) are one way to provide a clean API for communicating between the languages. The Java Native Interface (JNI) is a typical FFI designed to make a foreign function interface easier and safer. JNI can be used to interface with C/C++, assembly, and other languages. The pitfalls of using JNI or other FFI are generally that of impacted performance and, because of the

many issues related to interfacing between languages, correctness potentially causing issues where the code sometimes works, but not reliably because of the complexities of the interface. FFIs can introduce issues that are difficult to debug because of the complexities and lack of transparency within the interface.

6.47.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.47.5.
- Use a foreign function interface such as JNI to provide a clear separation between Java and the other language, but be aware that foreign function interfaces can be error prone and lack transparency, making debugging harder.
- Be aware that native code can lack many of the protections afforded by Java, such as bounds checks on structures not being performed on native methods, and explicitly perform the necessary checks.
- Minimize the use of those issues known to be error-prone when interfacing between languages, such as:
    1. passing character strings
    2. dimension, bounds, and layout issues of arrays
    3. interfacing with other parameter mechanisms such as call by reference, value, or name
    4. handling faults, exceptions, and errors, and
    5. bit representation.

6.48 Dynamically-linked code and self-modifying code [NYY]

6.48.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1:2024 6.48 exists in Java as explained below.

The Java Virtual Machine (JVM) does not allow access to random locations in memory, so modifying an already loaded byte code for self-modifying code is not possible from a Java program. However, new classes and methods that have not been loaded can be written or modified as a Java program is executing and then loaded.

Class loaders are responsible for loading Java classes during runtime dynamically to the JVM. When the runtime environment needs to load a new class for an application, the class is located and loaded by one of three types of class loaders in the following order: bootstrap class loader, extension class loader, and system class loader. The bootstrap class loader is responsible for loading all core Java classes. The extension class loader is a child of the bootstrap class loader and loads classes from the extension directories. The system class loader is responsible for loading code from the path specified by the CLASSPATH environment variable or, alternatively, by the `-classpath` option. The `-classpath` option will take precedence over the CLASSPATH environment variable. Altering either of these could lead to executing code that is different from what was tested.

The Java platform allows for JAR files to be digitally signed, thus providing a mechanism for verification of the origin of the file.

Java classes are not loaded into memory all at once, but when required by an application. Thus, if a class is changed while a program is running and before it has been loaded into memory, the new version will be used. Java also allows for class reloading. Thus, a program employing class reloading makes it possible for an attacker to modify a class while a program runs.

Since Java version 21, warnings are issued when agents are dynamically loaded into a running JVM and future releases will prohibit dynamic loading by default. The dynamic loading of the agents can be disabled after startup with the `-XX:-EnableDynamicAgentLoading` option.

6.48.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.48.5.
- Prohibit the dynamic modification of classes.
- Verify through the use of signatures that dynamically linked or shared code being used is the same as that which was tested.
- Retest when dynamically linked or shared code has changed before using the application.
- Review all warnings related to dynamic loading that are presented.

6.49 Library signature [NSQ]

6.49.1 Applicability to language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.49 exist in Java as explained below.

Integrating Java and another language into a single executable relies on knowledge of how to interface the method/function calls, argument lists, and data structures so that symbols match in the object code during linking.

Arrays and other data structures are often interpreted by another language differently than the way that Java interprets or stores them in memory. This can cause issues with transferring data between Java and the receiving language. For instance, it is common to use one-dimensional arrays to pass array data to and from programs in another language since the way that Java stores multidimensional arrays is significantly different than that of C, C++, and other languages.

Issues can arise when Java interfaces with a language that does not support garbage collection. Java can perform garbage collection and delete objects before the other non-garbage collection language being called is finished with them. Issues can also arise with the integration of non-Java exception handling or other error handling mechanisms, e.g. exit codes.

To alleviate some of these issues, wrappers can be used. Though wrappers can make the interfacing easier, wrappers can be error-prone and impact performance through the overhead of the wrapper.

6.49.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.49.5.
- Use a tool, if possible, to automatically create interface wrappers.
- Be wary of making assumptions about argument lists, data structures and error handling mechanisms, as other languages are likely to have differences in these areas.

6.50 Unanticipated exceptions from library routines [HJW]

6.50.1 Applicability to language

If the library routine is a Java routine, the vulnerabilities described in ISO/IEC 24772-1:2024 6.50 do not apply to Java with the minor exception of unhandled unchecked exceptions since all checked exceptions are part of the specification of the library routines and handling them is enforced by the compiler and runtime system.

For foreign libraries, see 6.49 Library signature.

Though a response to a checked exception is required, it is unfortunately too common for a programmer to assume that a checked exception could not possibly happen and instead of putting appropriate code in to handle the unexpected event, the programmer does just enough to get a clean compile by inserting an empty catch block as in the following example:

```
public void whatCouldPossiblyGoWrong() {
    try {
        // do something
    } catch (NumberFormatException e) {
        // this will never happen
    }
}
```

6.50.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Follow the mitigation mechanisms of ISO/IEC 24772-1:2024 6.50.5.
- Always have an appropriate response for checked exceptions since even things that should never happen do happen occasionally.

6.51 Pre-processor directives [NMP]

The vulnerability as described in ISO IEC 24772-1 6.51 does not apply to Java, as Java does not have a preprocessor.

6.52 Suppression of language-defined run-time checking [MXB]

The vulnerability as described in ISO IEC 24772-1 6.52 does not apply to Java since runtime checks cannot be suppressed.

6.53 Provision of inherently unsafe operations [SKL]

6.53.1 Applicability to language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.53 apply to Java.

The Java compiler generates the "uses unsafe or unchecked operations" warning for code considered to be unsafe. However, as it is a warning, it can be ignored.

Although Java is inherently a safe language, it does allow some operations that are inherently unsafe. For example, one undocumented class, `sun.misc.Unsafe contains code that is recognized to be inherently unsafe` but is often required for low-level programming. For instance, it allows the creation of an instance of a class without invoking its constructor code, initialization code, and various other JVM security checks. The `allocateMemory()` method in `sun.misc.Unsafe` also allows the creation of huge objects, larger than `Integer.MAX_VALUE`, that are invisible to the garbage collector and the JVM.

Another unsafe operation is the deserialization of data from external sources. Java version 17 finalized a filter package that permits the examination of data prior to deserialization.

6.53.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.53.5.
- Analyze the Java warnings "uses unsafe or unchecked operations" to determine whether action is needed or whether it is appropriate to leave the code as is.
- Only use the class `sun.misc.Unsafe` in specialized instances where the capabilities it provides are essential. It should not be used for everyday use to evade Java protections.
- Document all uses of unsafe code with in-place comments and provide evidence that all such uses function correctly and safely.
- Name unsafe extensions with names that retain the `unsafe` nomenclature.
- Apply Java's input stream filter capability for deserialization of external data.

6.54 Obscure language features [BRS]

6.54.1 Applicability of language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.54 apply to Java. There are ways that a feature of the language can be easily misused, and as such, restrictions on the feature are commonly expressed in coding

standards in software development organizations. For instance, the inclusion of statements other than loop control statements should not be included in a `for()` statement. For instance:

```
for (i = 0; total=0; i < 50; i++){

    total += value[i];

    }
```

Though the above code is legal, the inclusion of the non-loop control statement, `total=0`, reduces the maintainability and readability of the code.

Other features are unique to Java, and programmers schooled in other languages might not use these features since they are not as familiar with them as they would be with a feature that is common to both their native language(s) and Java. Finally, some features, such as the logical right shift (“>>>”) operator, are only applicable under rare circumstances, and there are alternative ways of achieving the same result and thus programmers could forget that the feature exists in the language.

Problems can also arise from the use of a combination of features that are rarely used together or fraught with issues if not used correctly. This can cause unexpected results and potential vulnerabilities.

6.54.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.54.5.
- Specify coding standards that restrict or ban the use of features or combinations of features that have been observed to lead to vulnerabilities in the operational environment for which the software is intended.

6.55 Unspecified behaviour [BQF]

6.55.1 Applicability of language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.55 apply to Java.

The Java specification is fairly complete and leaves very little unspecified. Two areas that lack full specification are:

- The garbage-collection algorithm used and any internal optimization that is performed. Since when garbage collection happens can be unpredictable, timing issues can be introduced. Garbage collection behaviour can be influenced by changing the heap size since the default garbage collector is scheduled to execute when free space on the heap goes below implementation-defined limits.
- Optimization of Java virtual machine instructions can cause portions of instructions to be skipped or reordered. Among others, this can influence timing behaviours, stack usage or heap usage.

6.55.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.55.5.
- Prohibit reliance on unspecified behaviour because the behaviour can change at each instance. Any code that makes assumptions about the behaviour of something that is unspecified should be replaced.
- Reduce the number of temporary objects to minimize the impact and need for garbage collection.
- Increase the Java heap size to reduce the frequency and amount of time spent doing garbage collection.
- Enable verbose garbage collection and profiling to locate and fix memory leaks to reduce the need for garbage collection.

6.56 Undefined behaviour [EWF]

6.56.1 Applicability of language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.56 apply to Java. Java is a well-defined language but has some areas of undefined behaviour. Areas of undefined behaviour are:

- The exact timing and scheduling of multiple threads. This is the primary area where undefined behaviour is experienced in Java.
- Calling a non-final method of the same class in the constructor. The undefined behaviour occurs if this method is overridden in a subclass. Notice that construction occurs from the superclass to the subclass. In some virtual machines, the local attributes will be constructed, the superclass constructor will finish its execution then, when the constructor of subclass is reached the attributes will be constructed again, overriding previously defined values.
- Interpreting a byte array as characters using the default encoding instead of the encoding used to produce the byte array and lacking a valid character representation for some of the bytes in the default encoding.
- How soon a finalizer will be invoked, which thread will invoke the finalizer for any given object, and the ordering of finalize method calls are all unspecified.
- Details of how and when garbage collection will occur, even when the garbage collection is explicitly invoked.
- If circularly declared classes are detected at runtime, then a ClassCircularityError is thrown. Otherwise, the behaviour is undefined and could lead to a StackOverflowError being thrown.

6.56.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.56.5.

6.57 Implementation–defined behaviour [FAB]

6.57.1 Applicability to language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.57 apply to Java, relating generally to the connection between the JVM and the underlying operation system. Java has very little implementation-defined behaviour as Java is a Write Once Run Anywhere (WORA) language. The Java operating model is that the Java source code is compiled and converted into byte code. The byte code is designed to be platform independent.

The main areas of implementation-defined behaviour relate to the connection between the JVM and the underlying operation systems, such as Windows and Unix. File name conventions, use of file path separators, thread behaviours, and network access mechanisms can have different observable behaviours.

For the instance of file path separators, an example of an area that is implementation defined are the two static variables in the java.io.File class, which will be used to make file path separation Java code platform independent. File.separator is the String value that an operating system uses to separate file paths. For instance, on Unix based systems, the a "/" is used, whereas on a Windows based system, a "\" is used. In order to make code platform independent, when creating a file path, use:

```
String filePath = "temp" + File.separator + "abcd.txt"
```

instead of the platform dependent

```
String filePath = "temp/abcd.txt".
```

6.57.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.57.5.

6.58 Deprecated language features [MEM]

6.58.1 Applicability to language

The vulnerabilities documented in ISO/IEC 24772-1:2024 6.58 apply to Java. As with other languages, it is recommended that deprecated classes, methods, and fields not be used. Java provides a way to express deprecation because as a class evolves, its API inevitably changes. Methods are renamed for consistency, improved methods are added, and fields change. To facilitate the transition to the new APIs, Java supports two mechanisms for the deprecation of a class, method, or field: an annotation and the `Javadoc` tag, which is the old method. Java annotations were introduced in Java 5 and are the preferred method. For either mechanism, existing calls to the old API continue to work, but the annotation causes the compiler to issue a warning when it finds references to deprecated program elements. Comments are inserted in the code prior

to the `@Deprecated` annotation to warn users against using the deprecated item and provide information on what should be used instead. However, in some instances where there is not a suitable replacement, users should simply not use the method.

```
public class ADeprecatedExmp {

    /**
     * @Deprecated
     * reason(s) why it was deprecated
     */
    @Deprecated
    public void showDeprecatedMessage(){
        System.out.println("This method is marked as deprecated");
    }

    public static void main(String a[]){

        ADeprecatedExmp mde = new ADeprecatedExmp();
        mde.showDeprecatedMessage();
    }
}
```

6.58.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.58.5.
- Use the Java annotation and a Javadoc tag to indicate deprecation of classes, methods, or member fields
- Rewrite code that uses deprecated language features to remove such use whenever possible.

6.59 Concurrency – Activation [CGA]

6.59.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1:2024 6.59 applies to Java.

Java will throw an exception if a thread cannot be created. For example, the `Java.lang.OutOfMemoryError` exception occurs when the system lacks the resources to create a new thread. A `try/catch` block can be used to ensure that if an `OutOfMemoryError` is encountered, then threads can be gracefully shut down and resources cleanly released. It is generally not recommended that any other recovery be attempted.

A thread that has visibility to another thread object `t` can test `t.isAlive()` to determine if the thread has been created and has not terminated yet.

Java provides a `ThreadGroup` class that contains a mechanism for multiple threads to be treated as one object rather than as individual objects (note that adding a thread to a group is a one-at-a-time activity). Thus, a single method call can be used to interrupt, suspend, or resume all of the threads within a group. However, many of

Commented [SM15]: The reference manual released in fall 2019 does not say that an exception is thrown if a thread creation fails, but it does say that any termination of a thread because of an exception raises an exception in the head of the task group for that thread, which is likely the same issue. This then becomes an issue of creating threads inside of a try-catch block but then we must resolve whether or not the creating thread must remain in the block until the created threads complete.

yyy Larry, we cannot find any mention that thread groups are deprecated

Commented [I16R15]: allowThreadSuspension, resume, stop, and suspend have all been deprecated from the ThreadGroup class. See https://stackoverflow.com/questions/18897621/why-is-not-safe-to-use-java-lang-threadgroup, https://rules.sonarsource.com/java/RSPEC-3014, https://wiki.sei.cmu.edu/confluence/display/java/THI01-J.+Do+not+invoke+ThreadGroup+methods , etc.

Commented [SM17R15]: Resolved.

Commented [WLD18]: yyy Do we need this or should it be deleted?

Commented [SM19R18]: It is appropriate.

these methods have been deprecated, flawed, or are insecure and thus it is recommended that these deprecated methods be avoided.

Alternatively, the Java `ExecutorService` is a framework provided by the JDK that simplifies the execution of tasks in asynchronous mode. The abstraction through the use of the framework relieves the developer from doing direct thread management by separating thread management and creation from the rest of the application. It allows the developer to create tasks and allows the framework to decide how, when, and where to execute the task on a thread. Effectively, executors execute potentially concurrent code but use the resources of underlying concurrency agents (such as threads) to perform the calculations. The underlying concurrency agents are not discarded but are reused for other executor computations. This means the user is not concerned with thread creation or termination, although issues related to shared data and synchronization still apply.

Extensions of the executor framework are the classes `FutureTask, Futures,` and `CompletableFutures,` which provide a framework for composing, combining, and executing asynchronous computation steps and handling errors. These use the concept of "tasks" that have less overhead than threads, but they can use the threading model to implement them as described above in the executor framework.

Virtual threads are lightweight threads managed by the JVM. Virtual threads require significantly fewer resources, enabling a large number of concurrent tasks to run efficiently and with a high throughput within a single process. Virtual threads excel when dealing with tasks that spend most of their time waiting for input/output operations since they can be easily suspended and resumed when needed. While great for I/O bound tasks, virtual threads are not designed for long-running CPU intensive operations. Because virtual threads are very lightweight, a stack trace might not accurately represent the full execution path of a program, making debugging more complex. When dealing with highly asynchronous operations, the interleaved nature of virtual threads can make it harder to debug the flow of execution and identify potential issues.

6.59.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.59.5.
- If running out of memory to create threads, Increase the amount of memory available for Java threads following system-specific conventions, e.g. on a Linux-based implementation by using the `java "-Xmx"` option.
- Lower the number of dynamically created threads, if possible, to avoid resource exhaustion.
- Avoid using the ThreadGroup class due to its inherent issues with memory leaks, deadlocks, race conditions, and synchronization.
- Use a framework such as the Java Executor Framework (`java.util.concurrent.Executor`), FutureTask (`java.util.concurrent.FutureTask`), Future

Commented [WLD20]: See: https://openjdk.java.net/jeps/8252885

Commented [SJM21]: SJM
The following was deleted:
"Java executor framework (java.util.concurrent.Executor), released with the JDK 5 is used to run the Runnable objects without creating new threads every time and mostly re-using the already created threads. Managing threads- through a framework such as this can avert potential problems with thread management."

java.util.concurrent.Executor is mentioned in 6.59.2 so I am wondering if there should be some, more-abbreviated, reference to it here in 6.59.1?

Commented [LW22R21]: That text was not in the version that Stephen mailed out after the meeting, so I don't know where it came from. In just a visual compare between this version and the after the meeting version Stephen sent out, it doesn't look like I did much in this section (probably should have since the comments were from 2021). However, I agree that there should be some mention of java.util.concurrent.Executor in section 1 since it is in section 2.

(`java.util.concurrent.Future`) and CompletableFuture (`java.util.concurrent.CompletableFuture`) to provide for more efficient management of concurrency.
- Use when performing asynchronous processing of data.
- Use care when implementing virtual threads since they work differently than traditional threads.

6.60 Concurrency – Directed termination [CGT]

6.60.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2024 6.60 applies to Java.

Terminating a thread in Java used to be done by calling the `java.lang.Thread.stop()` method. `java.lang.Thread.stop()` has been deprecated as it is inherently unsafe, leading to an inconsistent state of operation, such as monitored objects being corrupted.

Another way of directing the termination of a thread is through the use of the `java.lang.Thread.interrupt()` method. Both the initiating thread, which generates the interrupt, and the receiving thread, which should handle the interrupt, must cooperate in this process. For the interrupt mechanism to work correctly, the receiving thread must support its own interruption. In order to catch and process interrupts, each thread needs to occasionally check to see if the interrupt has been generated, for if it does not, then the interrupt will be effectively ignored. However, interrupting a thread in a sleeping or waiting state causes that state to be terminated with an `InterruptedException` exception. This exception needs to be handled by the interrupted thread, or else the thread will terminate.

The recommended way to stop a thread is by using a status variable whose changes must be synchronized. The thread periodically checks the variable and uses the value to determine whether it should gracefully terminate. This method avoids the use of interrupts or exceptions.

Either method of terminating a thread in Java depends on the programmer to decide exactly how to respond to the sent interrupt or to a synchronized status variable being set to indicate the need for termination.

Since the creation of a thread is expensive, Executor frameworks maintain a thread pool that contains a collection of pre-initialized threads that can be assigned tasks as needed. When a task is complete, the thread is not terminated, but simply returned to the thread pool so it can be assigned as needed to another task. This avoids the need to explicitly terminate a thread.

6.60.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.58.5.
- Use a synchronized status variable to indicate that a thread should exit in preference to `Thread.interrupt()`.
- If using `Thread.interrupt()`, ensure that all cases are handled and that the responses of an interrupted thread are safe.

Commented [SM23]: yyy – Erhard says this is wrong. Steve – reread Java document. Consider the situations.

Commented [SM24R23]: Resolved. Wording is correct.

Commented [WLD25]: From the Java specification: 17.2.3 Interruptions
Interruption actions occur upon invocation of Thread.interrupt, as well as methods defined to invoke it in turn, such as ThreadGroup.interrupt.
Let t be the thread invoking u.interrupt, for some thread u, where t and u may be the same. This action causes u's interruption status to be set to true. Additionally, if there exists some object m whose wait set contains u, then u is removed from m's wait set. This enables u to resume in a wait action, in which case this wait will, after re-locking m's monitor, throw InterruptedException.
Invocations of Thread.isInterrupted can determine a thread's interruption status. The static method Thread.interrupted may be invoked by a thread to observe and clear its own interruption status.

Commented [WLD26]: See https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html

Commented [WLD27]: I suspect the second sentence is the problem.

Commented [WLD28]: Yyy Text added to address the interrupted call and synchronized space.

6.61 Concurrent data access [CGX]

6.61.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2024 6.61 applies to Java.

Some data elements of Java can be shared between threads, while other data elements cannot. Data elements that can be shared between threads are termed shared memory or heap memory. All instance fields, static fields, and array elements are stored in heap memory and thus can be shared. Other data elements, such as local variables, formal method parameters, and exception handler parameters, are never shared between threads. The obvious issue is that data elements shared between threads must be synchronized to be accessed safely.

Concurrent access to an object needs to be synchronized to prevent data races and unforeseen results. To avoid unsynchronized access among threads, Java provides the `synchronized` keyword. Java provides `synchronized` methods to ensure non-interleaved access to an object of a class. The `synchronized` keyword indicates that a mutual-exclusion lock is implicitly acquired for the executing thread. For example:

```
public synchronized void tallyTotal (int newValue){
      this.total += newValue;
}
```

Once the method is executed, the lock is released. While the executing thread owns the lock, no other thread can acquire the lock, thus preventing an interleaving of two invocations of any `synchronized` method on the same object. In addition, single statements can be synchronized on an object, such as `synchronized(x);` `x.notify();` Calls on `x.notify(), x.notifyAll()` and `x.wait()` outside of synchronization on object `x` yield an exception.

Furthermore, Java provides private components to disallow direct access to components by users of the class. When these capabilities are combined, the functionality of simple monitors can be achieved, provided that all modifying accesses to private data components are performed via synchronized methods (as opposed to access by direct access, e.g., x.data. For conditional waiting to be achieved, Java provides the `wait()` and `notify()/notifyAll()` primitives.

Data elements that are shared between threads or executors without the use of `synchronized` can have their new values cached and can experience delays in the writing of their value to the shared memory. Other threads reading the current shared memory will get the old value until the cache value is written. Java provides the primitive `volatile` to ensure that all changes to a variable are atomic and the result is visible to all other threads that can also be accessing the variable. Alternatively, cache-coherence protocols on multiprocessor architectures can serve the same purpose. For example, 64-bit operations can be problematic since the operation could be performed as two separate 32-bit operations to a non-volatile long or double in many computers. Because other threads can read the value after the first write of 32 bits and before the second write, the value could be

incorrect. By declaring the `long` or `double` variable as `volatile`, the writes and reads of the `long` or `double` variables are always atomic. Note, however, that many types or classes cannot be declared `volatile`.

Since concurrent execution of threads is more common now with multicore processors, the order of execution can be very important. Examination of the source code will be misleading since compilers or firmware/hardware often reorder statements to optimize performance within each thread, but this reordering could affect the resulting execution order, leading to different results than expected. In addition, the sequencing of events between thread executions is unpredictable unless synchronization takes place between the threads in question.

Commented [WLD29]: Text modified. Is the modified text o.k.?

Commented [SM30]: Write a paragraph that recommends when using executors to avoid the explicit sharing of data.

6.61.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.61.5.
- Form 'happens-before' relationships through the use of the `java.util.concurrent` package.
- Use the `volatile` keyword to force a data element to always go to main memory for its reads and writes
- Mark as `private` all data components that are accessed by multiple threads.
- Apply the `synchronized` keyword to methods that access the same data components of an object to prevent multiple invocations of methods on the same object from interleaving.
- Access all data components, including private components, only through `synchronized` getter and setter methods.

6.62 Concurrency – Premature termination [CGS]

6.62.1 Applicability to language

Java is susceptible to premature termination of threads, as documented in ISO/IEC 24772-1:2024 6.62.

Commented [SM31]: The Java standard says that an exception is raised in the head of a thread group if a thread terminates due to an exception. This needs to be documented here and a recommendation that thread group heads handle such exceptions and deal with threads that terminate because of an exception.

Commented [WLD32]: This is documented in the last paragraph.

Java provides the `java.lang.Thread.isAlive()` method to test if a thread is alive. The method will return `true` if the thread is alive and `false` otherwise. This allows the thread to be monitored to see if it is still functioning. Note that a call to `ThreadIsAlive` is asynchronous with the execution of the thread being queried, so it is subject to a race condition with the termination of the queried thread.

Commented [SM33]: Investigate how adding a thread to a thread group -- *Investigate how adding a thread to a thread group mitigates premature termination of that thread. We believe that an exception is raised to the owner of the thread group but which thread catches it.*

Java has a thread group feature. A thread group forms a tree of threads and other thread groups in which every thread group except the initial thread group has a parent. A Java thread group is implemented by the `java.lang.ThreadGroup` class. However, many of the methods of the `ThreadGroup` class, such as resume(), stop(), and suspend(), have been deprecated and should not be used. Other methods in the class, such as `activeCount()` and `enumerate()`, are not thread safe.

Commented [WLD34]: Yyy This is covered in 6.59. Suggest deleting this.

Threads that exit unexpectedly are vulnerable to the issues raised in ISO/IEC 24772-1:2024 6.62.3. Premature termination as a result of an unexpected exception can be handled either by a per-thread static method (set by `Thread.setUncaughtExceptionHandler()`) or by a static `ThreadGroup` method (optionally set by `ThreadGroup.setDefaultUncaughtExceptionHandler()`). In either case, no notifications to other threads occur unless explicitly programmed. As a simpler remedy, the thread that is terminating can have the relevant exception handler installed and can use normal thread notifications.

The CompletableFuture class contains methods for composing, combining, and executing asynchronous computation. Among the methods in the `CompletableFuture class is` the method `isCompletedExceptionally(), which` can be used to determine if the CompletableFuture completed in any exceptional fashion.

6.62.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.62.5.
- Use the `java.lang.Thread.isAlive()` method to check as needed to see if a thread is still active.
- When appropriate, use the Java `ExecutorService` framework for concurrency management using tasks.
- Use the `java.util.concurrent.CompletableFuture.IsCompletedExceptionally()` to determine whether a future completed normally or exceptionally.
- Ensure that each thread handles all exceptions that can arise during its activation and execution and provides appropriate notification upon termination to interested other threads.
- Use the `Thread.setDefaultUncaughtExceptionHandler()` method in thread groups to handle unexpected exceptions.

6.63 Lock protocol errors [CGM]

6.63.1 Applicability to language

Java is susceptible to lock protocol errors, as documented in ISO/IEC 24772-1:2024 6.63. Java allows a synchronization mechanism for communicating between threads, which is implemented using monitors. Each object in Java is associated with a monitor, which a thread locks by accessing a `synchronized` method and unlocks upon leaving the outermost synchronized method. Every object has an intrinsic lock associated with it. A thread that needs exclusive and consistent access to an object's fields acquires the object's intrinsic lock by accessing a `synchronized` method, accessing the object's fields, and then releasing the intrinsic lock when it is finished with them.

The `Java.lang.Thread` class has six potential states for a thread: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, and TERMINATED. Three of these are states that indicate that the thread is waiting are BLOCKED, WAITING and TIMED_WAITING.

- BLOCKED indicates that the thread is waiting for a monitor lock.

- WAITING indicates that the thread is waiting on another thread to perform a particular action. Future objects can be used to indicate when a thread has an object ready for the main thread to use. This allows the main thread to keep track of the progress and result from another thread.
- TIMED_WAITING indicates that the thread is waiting for another thread to perform an action for up to a specified waiting time.

Each of these states provide an indication of ways that a thread can be waiting on another thread's actions so as to attempt to alleviate lock protocol errors. Though Java has intrinsic language features for managing lock protocol errors, per the Java specification, "The Java programming language neither prevents nor requires detection of deadlock conditions." It is recommended in the Java specification that conventional techniques for deadlock avoidance be used since Java does not inherently have preventions.

The BlockingQueue interface, `java.util.concurrent.BlockingQueue`, is a thread safe queue that permits multiple threads to insert or extract elements without concurrency issues. If the queue is empty, a thread will be blocked from taking an element until one is added to the queue. Similarly, if the queue is full, a thread will be blocked from adding additional elements.

For example, in a producer/consumer scenario, both kinds of threads need to synchronize over a buffer; in addition, producers need to wait when the buffer is full and consumers need to wait when the buffer is empty. It is the responsibility of each thread to inform the other kind when an element is taken off the buffer, which then is no longer full, or an element is added to the buffer, which then is no longer empty. However, Java waits on the synchronized object, not a signal of a specific condition. `Notify()` notifies the object, which releases the top element on the wait queue. In the unlikely but possible event that a producer notifies, but the top element on the queue happens to be another producer, the wrong kind of thread is awakened. If the buffer is full at this time, the awakened producer waits and so do all threads, including consumers, forever, unless another consumer arrives and gets the queue going again. Response times of the threads become unpredictable and possibly reach infinity. Therefore, to be on the safe side, `notifyAll()` is to be used to awaken all queued entries. As only one consumer can win, all others will have to wait again; this creates performance issues.

Java also provides a mechanism to schedule and release threads explicitly via the `wait`() and `signal`() functions. A thread can `wait(E)` on a timed event or on an arbitrary event. All threads waiting on a non-timed event are waiting until a `notify(E)` or `notifyAll(E)` is called. The first releases only the first thread to wait, while `notifyAll`(E) releases all waiting threads. Interrupt will also release a thread from a wait queue, but with an exception state set. The vulnerabilities that can result from the use of this mechanism are: A nasty vulnerability is the existence of only a single waiting queue for each synchronized object since:

1. Two or more threads can execute a notify() almost simultaneously and the waiting thread will have no knowledge as to which notify event it was connected.
2. A thread can be interrupted and notified almost simultaneously, and there is no specification as to which condition the released thread will respond, either a normal continuation or the posting of an exception.

It is fundamentally important that, within synchronized methods, wait calls are only placed to the object that is the synchronization object. Waiting on other objects is highly likely to result in an immediate deadlock since the lock on the synchronized object is not freed by the `wait()`.

6.63.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.63.5.
- Use the intrinsic monitor features coupled with conventional techniques to avoid lock protocol errors.
- Use `java.util.concurrent.BlockingQueue` when sharing queues among threads.
- Use `java.lang.Object.wait` to cause the current thread to wait until another thread invokes the `notify()` or `notifyAll()` method or a specified amount of time has elapsed.
- When using `wait()` and `notify()`, make the wait/release set as granular as possible so that precise control can be exercised over the concurrency paradigm and the locking paradigms. Prefer using `wait` and `notify` and `synchronized` data to model mailboxes between pairs of threads in preference to broad-based monitors.

6.64 Reliance on external format strings [SHL]

6.64.1 Applicability to language

Java provides string classes to interpret the data read or format the output. These strings include all of the features described in ISO/IEC 24772-1:2024 6.64.1. The `java.util.Scanner` class allows for the parsing of strings using regular expressions. The `java.lang.String` allows for the creation and manipulation of strings. In Java, strings are immutable. Once a string object is created its data or state cannot be changed, instead a new string object is created. Though Java has classes that can help avoid external format strings, strings originating outside of the trust boundary always need verification to ensure trust and before use. The standard Java library implementation will throw an exception if a string does not match the corresponding format specification.

Checking strings without normalizing them first can cause validation logic, and in particular, blacklisting comparisons, to be inaccurate. Similarly, if path names and other such strings with more than one possible representation are not canonicalized before comparing, inaccurate results can occur.

6.64.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.64.5.
- Normalize strings before validating them.
- Canonicalize path names and other strings that have more than one possible representation.
- Use Java classes for importing, exporting, and manipulating strings.

6.65 Modifying constants [UJO]

6.65.1 Applicability to language

Java provides a capability called *reflection* that allows constants that are declared `final` to be changed. Much like the use of `sun.misc.Unsafe`, a programmer must intentionally perform a series of steps to alter the value of an object marked `final`. In the interest of security, it is not uncommon that the use of the method needed to do this is forbidden by a security manager in many enterprise server environments.

6.65.2 Avoidance mechanisms for language users

To avoid the vulnerabilities or mitigate their ill effects, Java software developers can:

- Apply the avoidance mechanisms contained in ISO/IEC 24772-1:2024 6.65.5.
- Avoid declaring an object `public final` if it needs to be changed over the lifetime of a program .
- Prohibit modification of `final` constants.

7. Language specific vulnerabilities for Java

[Intentionally blank]

Bibliography

[1] Gosling, James, et al., *The Java Language Specification, Java SE 10 Edition*, 2018-02-20.

[2] Long, Fred, et al., The CERT Oracle Secure Coding Standard for Java, Upper Saddle River, NJ, Addison Wesley, 2012.

Page 29: [1] Deleted Stephen Michell 2/19/25 2:57:00 PM

Page 29: [2] Commented [SJM8R7] McDonagh, Sean 2/19/25 12:40:00 PM

```
Object obj = "Hello";
switch (obj) {
    case String s ->
         System.out.println("It's
         a string");
    case Object o ->
         System.out.println("It's
         an object");
}
```

Output:

```
It's a string
========================================
Object obj = "Hello";
switch (obj) {
    case Object o ->
         System.out.println("It's
         an object");
    case String s ->
         System.out.println("It's
         a string"); // ERROR:
                                   This case label is
                                dominated by a preceding case label
}
```