ISO/IEC JTC 1/SC 22/WG 23 N 0299

Further revised draft language-specific annex for Java

Date	15 December 2010
Contributed by	Ben Brosgol
Original file name	JavaVulnerabilitiesAnnex-markup-on.pdf
Notes	Replaces N0294

Annex Java
(Informative)
Vulnerability descriptions for language Java

Java.1 Identification of standards and other references

[JLS 2005]	J. Gosling, B. Joy, G. Steele, G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley; 2005.
[JP 2005]	J. Bloch and N. Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases.* Addison-Wesley; 2005.

Java.2 General terminology and concepts

~~<<TBD>>~~

boxing An implicit conversion applied to a value from a primitive type and yielding a reference to a (dynamically allocated) object of a reference type; the reference type is either `Object` or one of the wrapper classes for primitive types (`Integer`, `Long`, etc.). See [JLS2005, §5.1.7]. For example:

```
int j = 1000;
Integer i1 = new Integer(j); // Explicit constructor call
Integer i2 = j; // Boxing conversion from int to Integer
Object  o  = j; // Boxing conversion from int to Object
```

Garbage Collector A component of the Java run-time environment that arranges automatic storage reclamation (with space compaction to avoid fragmentation) for objects that are no longer accessible from the program.

Java.3 Type System [IHN]

Java.3. 0 Status, history, and bibliography

2010-11-13	Initial version (Brosgol)
2010-11-26	Updated format, added discussion of boxing issues (Brosgol)
2010-12-14	Clarified discussion and examples based on comments at Meeting #16 (Brosgol)
2010-12-15	Added explanation based on further comments at Meeting #16 (Brosgol)

Java.3.1 Applicability to language

Java's type mechanism, and in particular the rules governing the use of class types, protects against many kinds of type errors (manipulation of data objects with inappropriate operations). However, the primitive types in Java (`int`, `double`, etc) are weakly typed. For example, there is no way to indicate to the compiler that two `int` values –say one representing a student grade and another representing an index into an array – are incompatible and thus should not be assigned to each other, used as operands in an arithmetic expression, etc .

One possible workaround is to define a "wrapper" class to simulate a strongly-typed scalar type so that the compiler detects attempts to use a scalar as an object of the wrapper class or *vice versa*; e.g.:

```
public class Grade{
  public int grade;
}
…
Grade g = new Grade();
int i = 0;
i = g; // illegal
g = i; // illegal
```

This style may be appropriate in some circumstances but ~~adds overhead and notational clumsiness and~~ for several reasons is ~~in general is~~ not a realistic solution in general:~~.~~

- *Overhead*. Constructing an object of a wrapper class (and having the Garbage Collector keep track of it) is more expensive than simply pushing a local variable on the stack or having a primitive field in an object.

- *Complexity because of reference semantics*. The sharing inherent in reference semantics, when a reference is passed as a parameter or assigned to a variable, complicates the job of program analysis. Defining the wrapper class as immutable can address this issue but induces extra object construction and thus additional overhead.

- *Notational clumsiness*. If a grade is implemented as an int, then literals are available and it is trivial to do simple things like subtracting two grades. With grade implemented as a class, the programmer's job is more complicated (explicit "delegation" methods need to be defined) and the absence of operator overloading will make the program text harder to understand.

Arrays are also weakly typed:

```
double[] temperatures = new double[n];
double[] distances = new double[m];
… // initialize the arrays
temperatures = distances; // Legal but likely an error
```

As a result of this assignment, each array element `temperatures[i]` actually represents a distance and not a temperature. A method that operates on either the entire array (assuming temperature values) or on an individual element will thus misinterpret the data.

A complete and efficient solution to the general "units" problem – enforcing at compile time that scalar data are used consistently – remains a subject of research in the programming language community. However, Java's weak typing for primitive data means that even simple cases of type misuse will not be detected by the compiler.

Java's boxing rules ~~[JLS2005, §5.1.7] provide implicit conversions (via implicit dynamic allocation) from a value of a primitive type to a reference whose type is either `Object` or the type of one of the primitive wrapper classes (`Integer`, `Long`, etc). These rules~~ present a typing vulnerability; some of the subtleties of reference versus value identity for the result of a boxing conversion can yield unintuitive (and possibly implementation-dependent) ~~results~~effects. In particular, whether different occurrences of a boxing conversion for the same primitive value always evaluate to the same reference is not completely defined by Java semantics. The following example illustrates the problem:

```
class TestBoxing{
  public static void main(String[] args){
     Object o1 = 127;
     Object o2 = 127;

     // Do the two boxing conversions for the value 127
     // yield the same reference?
     System.out.println("Value is " + (Integer)o1);
     if (o1==o2){
         System.out.println("Objects equal");
     }
     else {
         System.out.println("Objects not equal");
     }

     o1=128;
     o2=128;

     // Do the two boxing conversions for the value 128
     // yield the same reference?
     System.out.println("Value is " + (Integer)o1);
     if (o1==o2){
         System.out.println("Objects equal");
     }
     else {
         System.out.println("Objects not equal");
     }
  }
}
```

This program produces the following output:

```
Value is 127
Objects equal
Value is 128
Objects not equal
```

Whether the objects that box 128 are equal is not defined by Java semantics.

Java.3.2 Guidance to language users

Users need to exercise care when dealing with values of the primitive types, to ensure that assignment, arithmetic operations, parameter passing, casts, etc., are consistent with the purpose for the values.

Comments in the source program explaining the intended use of variables of primitive types may be helpful.

In principle it is possible that annotations, combined with appropriate static analysis tools, could detect particular kinds of misuses of primitive types. <<Need to verify this claim.>>

Programmers need to be alert to the issues with implicitly boxed values, and in particular the use of "==" (or "!=") on references obtained from boxing conversions. Again, static analysis may help to detect such cases.

Java.4 Bit Representation [STR]

Java.4.0 Status, history, and bibliography

2010-11-26 Initial version (Brosgol)

Java.4.1 Applicability to language

Java includes operators for bitwise processing of values of integral types:

- Unary complement ("**~**")[JLS 2005, §15.15.5]
- Binary operators for **and** ("&"), exclusive **or** ("^"), and inclusive **or** ("|") [JLS 2005, §15.22]
- Binary operators for left shift ("**<<**"), logical right shift ("**>>**"), and arithmetic right shift ("**>>>**") [JLS 2005, §15.19].

Java also includes some bit-manipulation methods for the integral types' wrapper classes (`Byte`, `Char`, `Short`, `Integer`, `Long`), for example `rotateRight`, `rotateLeft`, `bitCount`, and a few others.

Since Java mandates the size for integral types, it avoids some of the problems that arise in other languages. Further, Java is generally not used for the low-level (and platform-dependent) processing where the need for bit manipulation typically arises. However, several issues still arise:

- Common practice dictates that bit manipulation should be reserved for unsigned types, but all the integral types in Java except for `char` are signed. Using a signed type to simulate an unsigned type requires care (e.g. with literals and the relational operators).
- The right operand to the shift operators – the shift count – may be of any integral type, but only the rightmost 5 bits (if the left operand is an `int`) or 6 bits (if the left operand is a `long`) are used. This may cause some surprising results (see [JP 2005, Puzzle 57] for an example).
- Java's promotion of byte and short arithmetic operands to 32 bits uses sign extension, even for the bitwise processing operators, resulting in effects that might not be obvious. For example:

```
class TestBitOps{
  public static void main(String[] args){
     int   left   = 0xFFFF0000;
     short right  = (short)0xFFFF;
     int   result = left & right;
     System.out.println(Integer.toHexString(left));
     System.out.println(Integer.toHexString(right));
     System.out.println(Integer.toHexString(result));
  }
}
```

This program produced the following output:

```
ffff0000
ffffffff
ffff0000
```

I.e., the 16-bit value of the right operand is sign extended when the value is promoted to 32 bits, so that it becomes `0xffffffff` and not `0x0000ffff`.

Java.4.2 Guidance to language users

Programmers need to understand the semantics of type promotion for operands of the bitwise processing operators, and the way in which the shift count is determined, as discussed in the preceding section. Further, when performing bitwise processing on values coming in from sources external to the Java implementation, programmers may need to take into account the platform-specific endianness (both bit-order and byte-order).

Java.5 Floating-point Arithmetic [PLF]

Java.5.0 Status, history and bibliography

2010-12-13 Initial version (Brosgol)

[IEEE 1985] IEEE Std 754-1985. *IEEE Standard for Binary Floating-Point Arithmetic*
ieeexplore.ieee.org/servlet/opac?punumber=2355

[IEEE 2008] IEEE Std 754-2008. *IEEE Standard for Floating-Point Arithmetic*
ieeexplore.ieee.org/servlet/opac?punumber=4610933

[Go 1991] D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic", *ACM Computing Surveys*, March 1991.
Reprinted, with a supplemental section "Differences Among IEEE 754 Implementations", in Sun Microsystems *Numerical Computation Guide*, 2002.
docs.sun.com/app/docs/doc/806-3568

[Ka 2000] W. Kahan. *Marketing versus Mathematics and other Ruminations on the Design of Floating-Point Arithmetic*. August 2000.
www.cs.berkeley.edu/~wkahan/MktgMath.pdf

[KD 2004] W. Kahan & J. D. Darcy. *How Java's Floating-Point Hurts Everyone Everywhere* (originally presented 1 March 1998 at the invitation of the *ACM 1998 Workshop on Java for High-Performance Network Computing*); 2004.
www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf

[Da 1998] J. Darcy. *Borneo 1.0.2: Adding IEEE 754 floating point support to Java*. May 1998.
For link see Borneo language home page: sonic.net/~jddarcy/Borneo/

Java 5.1 Applicability to language

Java's floating-point facility[JLS 2005, §4.2.3, 4.2.4] is based on, but not fully conformant with, the IEEE 754 binary floating-point standard [IEEE 1985] (currently under revision as [IEEE 2008]).

Java supplies two primitive floating-point types, `float` (32 bits) and `double` (64 bits) and allows two representational formats for each of these types:

- *FP-strict*. The format for these types is exactly as specified in IEEE 754 for significand (mantissa) size and exponent range.
- *Not FP*-strict. The implementation is permitted to represent float and double values using wider exponent ranges than is specified for these types in IEEE 754. The Java representations are respectively referred to as *float-extended-exponent* and *double-extended-exponent*. Note that these do not correspond to the IEEE 754 single extended or double extended formats since the precision of the significand in the Java formats is not extended.

Aside from compile-time constants (which are always FP-strict), the determination of whether or not the value of an expression is represented in FP-strict format is local to a given context [JLS 2005, §15.4] with explicit syntax required (the strictfp modifier) to obtain FP-strict.

Regardless of the format, each of the types float and double includes the IEEE 754 special values for positive and negative infinity and NaN (Not a Number), and also distinguishes +0.0 from -0.0 (although these values compare as equal). Java requires support of IEEE 754 denormalized floating point and gradual underflow via "subnormal" values (versus "flush to zero"on underflow). It uses "round to nearest" for inexact results; if the exact result is midway between two representable values, then it is rounded to even (the value with least significant bit zero).

Java's rules for expression evaluation [JLS 2005, §15.7.3] prevent the optimizer from performing certain program transformations that could change the meaning of floating-point expressions. For example, parentheses must be obeyed for grouping operands with operators; properties (such as the associative law) that hold for mathematical real numbers cannot be assumed to hold for floating point.

The main vulnerability issues with Java's floating point arise from a program not doing what the programmer expects. Several aspects of Java semantics contribute to this problem

- *Numeric conversion*. The following example was cited in [KD 2004]:

```
// Convert from Fahrenheit to Celsius
double c, f;
…
c = (f – 32) * 5/9;     // works correctly
c = (f – 32) * (5/9);   // compiles, gives incorrect result at run time
```

The first assignment statement works since the right side is parsed as `(((f-32)*5)/9)`; each operator has a `double` as its left operand, thus causing an implicit cast of each of the `int` literals to double. In the second assignment, however, the right side of the multiplication is treated as an integer expression `5/9` and thus integer division is performed (yielding 0) before the implicit cast to `double`.

- *Precision of intermediate results*. Java prohibits computing intermediate results in wider precision than the operand types require. The following example ([Ka 2000, p. 26]) illustrates the problem:

```
double z;
float v, w, x, y;
…
z = v*w + x*y;
```

The Java rules [JLS 2005, §4.2.4] require that the computation of z be equivalent to:

```
float p, q;
p = v*w; // rounded
q = x*y; // rounded
r = p+q; // rounded
z = r;   // extend to double
```

The multiple roundings reduce the accuracy of the final result `z`. In contrast, pre-ANSI C required the intermediate results to be computed as `double`, with an effect equivalent to the following pseudocode:

```
double p, q;
p ← v*w; // exact
q ← x*y; // exact
z ← p+q; // rounded
```

This gives a more accurate result than would be obtained through the Java rules.

- *Lack of operator overloading.* Although this is generally perceived as a readability issue, the absence of operator overloading also causes problems with complex number computation. With operator overloading and an `Imaginary` class, a complex number could be expressed as `x + y*i` or `x + i*y`. In the absence of operator overloading, a `Complex` class in Java would need to work on pairs `(x, y)`. Java's type coercion rules would result in mishandling `-0.0`, leading to computational anomalies [KD2004, pp. 11-15].

Other problems with Java floating point are due to the language's incomplete compliance with IEEE 754. Some strongly-worded critiques ([Ka 2000], [KD2004]) identify several issues that complicate the job of writing correct numeric programs in Java, including the following:

- *Lack of support for single-extended and double-extended precision formats.*

This absence exacerbates the multiple-roundings issue described above.

- *Lack of support for dynamic directed rounding modes*

IEEE 754 specifies four rounding modes (round to nearest, round towards zero, round towards +∞, round towards −∞) with dynamic program control over the choice. This capability is extremely useful; e.g., running the same program multiple times with different rounding modes helps test for numeric instability [KD 2004, pp. 57-58]. Java lacks this capability.

- *Lack of support for exception flags*

IEEE 754 defines several conditions that may arise during a floating-point operation: *Invalid Operation* (e.g., `0.0/0.0` or `SQRT(-1.0)`), *Overflow*, *Division by Zero*, *Underflow*, and *Inexact Result*. These conditions are known as *exceptions* in IEEE 754, but their effect is not the same as exceptions in Java since the operation continues and delivers a well-defined result. By default this result will be NaN (for invalid operation), one of the infinities (for overflow and division by zero), or a subnormal (for underflow). The program can deal with the exceptions by installing trap handlers (returning a result to complete the operation) or by querying and "lowering" a sticky exception-related flag that was raised by the operation.

Java provides no mechanism for dealing with exception flags or for installing trap handlers. [KD 2004, p. 24] identifies the problem with the absence of exception flag support:

> *Without flags, detecting rare creations of ∞ and NaN before they disappear requires programmed tests and branches that, besides duplicating tests already performed by the hardware, slow down the program and impel a programmer to make decisions prematurely in many cases. Worse, a plethora of tests and branches undermines a program's modularity, clarity and concurrency.*
>
> *With flags, fewer tests and branches are necessary because they can be postponed to propitious points in the program. They almost never have to appear in lowest-level methods nor innermost loops.*

Note that for a language like Java that supports multiple threads, the flags would need to be implemented on a per-thread basis.

A set of extensions to Java that would address these issues, the language Borneo [Da 1998], has been proposed but has not (yet) been implemented.

Java 5.2 Guidance to language users

Users of floating-point in Java need to be alert to these issues and ensure that their floating-point computations deliver the accuracy required. In some cases workarounds are available, for example by declaring variables to be of wider precision (double) than might be needed by the final result.

TR 24772, §6.4.5, suggests that programmers not use Boolean tests for equality on floating-point values, since such tests could cause various anomalies. This advice, however, is not necessarily correct. As noted in [Go 1991, p. 212]:

> *… some people think that the solution to such anomalies is never to compare floating-point numbers for equality, but instead to consider them equal if they are within some error bound* E*. This is hardly a cure-all because it raises as many questions as it answer[s]. What should the value of* E *be? If* x<0 *and* y>0 *are within* E*, should they really be considered to be equal, even though they have different signs? Furthermore, the relation defined by this rule*
>
> $a \sim b \Leftrightarrow |a - b| < E$
>
> *is not an equivalence relation because* a~b *and* b~c *does not imply that* a~c*.*

Thus it may be perfectly acceptable to use equality for floating-point comparison, provided that the programmer has done the necessary analysis.

Java.X Title [ZZZ] *<Template for further sections>*

Java.X.0

Java.X.0 Status, history and bibliography

Java X.1 Applicability to language

Java X.2 Guidance to language users

Java.54 Deprecated Language Features [MEM]

<<TBD>>

Java. 55 Implications for standardization

Java.55.1 Introduction

This section discusses how the various vulnerability mitigation approaches discussed above might be realized in future versions of the Java Language Specification.

Java.55.2 Type System [IHN]

A variety of language enhancements could help address this issue. For example, a simple typedef facility that allows the programmer to supply aliases for the primitive types would at least document the intended uses for variables of the type, even if it the aliasing type and the underlying primitive type are treated as equivalent. A more complete solution that allows user definition of new primitive types would be a major change, and the benefit is probably not worth the cost in semantic complexity and impact on implementations.

Java.55.3 Bit Representation [STR]

<<TBD>>

…

Java.55.4 Floating point Arithmetic [PLF]

The shortcomings in Java floating-point have been known for some time; a proposal for addressing these has appeared in the Borneo language proposal [Da 1998] and could be considered for a future version of Java.

Java.55.52 Deprecated Language Features [MEM]

<<TBD>>