The Java language provides two primitive floating-point types, `float` and `double`, which are associated with the single-precision 32-bit and double-precision 64-bit format values and operations specified by IEEE 754 [IEEE 754]. Each of the floating-point types has a fixed, limited number of mantissa bits. Consequently, it is impossible to precisely represent any irrational number (for example, pi). Further, because these types use a binary mantissa, they cannot precisely represent many finite decimal numbers, such as 0.1, because these numbers have repeating binary representations.

When precise computation is necessary, such as when performing currency calculations, floating-point types must not be used. Instead, use an alternative representation that can completely represent the necessary values.

When precise computation is unnecessary, floating-point representations may be used. In these cases, you must carefully and methodically estimate the maximum cumulative error of the computations to ensure that the resulting error is within acceptable tolerances. Consider using numerical analysis to properly understand the problem. See Goldberg’s work for an introduction to this topic [Goldberg 1991].

**Noncompliant Code Example**

This noncompliant code example performs some basic currency calculations:

```java
double dollar = 1.00;
double dime = 0.10;
int number = 7;
System.out.println("A dollar less "+ number+" dimes is "+(dollar - number * dime));
```

Because the value 0.10 lacks an exact representation in Java floating-point type (or any floating-point format that uses a binary mantissa), on most platforms, this program prints the following:

```
A dollar less 7 dimes is $0.29999999999999993
```

**Compliant Solution**

This compliant solution uses an integer type (such as `int`) and works with cents rather than dollars:

```java
int dollar = 100;
int dime = 10;
int number = 7;
System.out.println("A dollar less "+ number+" dimes is "+(dollar - number * dime));
```

This code correctly outputs the following:

```
A dollar less 7 dimes is $0.30
```

**Compliant Solution**

This compliant solution uses the `BigDecimal` type, which provides exact representation of decimal values. Note that on most platforms, computations performed using `BigDecimal` are less efficient than those performed using primitive types.
import java.math.BigDecimal;

BigDecimal dollar = new BigDecimal("1.0");
BigDecimal dime = new BigDecimal("0.1");
int number = 7;
System.out.println("A dollar less " + number + " dimes is $" +
    (dollar.subtract(new BigDecimal(number).multiply(dime))));

This code outputs the following:

A dollar less 7 dimes is $0.3

Risk Assessment

Using floating-point representations when precise computation is required can result in a loss of precision and incorrect values.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM04-J</td>
<td>Low</td>
<td>Probable</td>
<td>High</td>
<td>P2</td>
<td>L3</td>
</tr>
</tbody>
</table>

Automated Detection

Automated detection of floating-point arithmetic is straightforward. However, determining which code suffers from insufficient precision is not feasible in the general case. Heuristic checks, such as flagging floating-point literals that cannot be represented precisely, could be useful.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parasoft Jtest</td>
<td>10.3</td>
<td>PB.NUM.UBD</td>
<td>Implemented</td>
</tr>
</tbody>
</table>

Related Guidelines

- The CERT C Secure Coding Standard: FLP02-C. Avoid using floating-point numbers when precise computation is needed
- The CERT C++ Secure Coding Standard: VOID FLP02-CPP. Avoid using floating point numbers when precise computation is needed
- ISO/IEC TR 24772:2010: Floating-Point Arithmetic [PLF]

Android Implementation Details

The use of floating-point on Android is not recommended for performance reasons.

Bibliography

- [Bloch 2008] Item 48, "Avoid float and double If Exact Answers Are Required"
- [Bloch 2005] Puzzle 2, "Time for a Change"
- [Goldberg 1991]
- [IEEE 754]
- [JLS 2015] §4.2.3, Floating-Point Types, Formats, and Values
- [Seacord 2015] NUM04-J. Do not use floating-point numbers if precise computation is required LiveLesson