NUM13-J. Avoid loss of precision when converting primitive integers to floating-point

The following 19 specific conversions on primitive types are called the **widening primitive conversions**:

- 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

Conversion from int or long to float or from long to double can lead to loss of precision (loss of least significant bits). In these cases, the resulting floating-point value is a rounded version of the integer value, using IEEE 754 round-to-nearest mode. Despite this loss of precision, *The Java Language Specification (JLS)* requires that the conversion and rounding occur silently, that is, without any runtime exception (see the JLS, §5.1.2, "Widening Primitive Conversion" [JLS 2015], for more information). Conversions from integral types smaller than int to a floating-point type and conversions from int to double can never result in a loss of precision. Consequently, programs must ensure that conversions from an int or long to a floating-point type or from long to double do not result in a loss of required precision.

Note that conversions from float to double can also lose information about the overall magnitude of the converted value (see NUM53-J. Use the strictfp modifier for floating-point calculation consistency across platforms for additional information).

Noncompliant Code Example

In this noncompliant code example, two identical large integer literals are passed as arguments to the `subFloatFromInt()` method. The second argument is coerced to float, cast back to int, and subtracted from a value of type int. The result is returned as a value of type int.

This method could have unexpected results because of the loss of precision. In FP-strict mode, values of type float have 23 mantissa bits, a sign bit, and an 8-bit exponent (see NUM53-J. Use the strictfp modifier for floating-point calculation consistency across platforms for more information about FP-strict mode). The exponent allows type float to represent a larger range than that of type int. However, the 23-bit mantissa means that float supports exact representation only of integers whose representation fits within 23 bits; float supports only approximate representation of integers outside that range.

```java
strictfp class WideSample {
    public static int subFloatFromInt(int op1, float op2) {
        return op1 - (int)op2;
    }

    public static void main(String[] args) {
        int result = subFloatFromInt(1234567890, 1234567890);
        // This prints -46, not 0, as may be expected
        System.out.println(result);
    }
}
```

Noncompliant Code Example

In this noncompliant code example, two identical large integer literals are passed as arguments to the `subFloatFromInt()` method. The second argument is coerced to float, cast back to int, and subtracted from a value of type int. The result is returned as a value of type int.

This method could have unexpected results because of the loss of precision. In FP-strict mode, values of type float have 23 mantissa bits, a sign bit, and an 8-bit exponent (see NUM53-J. Use the strictfp modifier for floating-point calculation consistency across platforms for more information about FP-strict mode). The exponent allows type float to represent a larger range than that of type int. However, the 23-bit mantissa means that float supports exact representation only of integers whose representation fits within 23 bits; float supports only approximate representation of integers outside that range.

```java
strictfp class WideSample {
    public static int subFloatFromInt(int op1, float op2) {
        return op1 - (int)op2;
    }

    public static void main(String[] args) {
        int result = subFloatFromInt(1234567890, 1234567890);
        // This prints -46, not 0, as may be expected
        System.out.println(result);
    }
}
```

Note that conversions from long to either float or double can lead to similar loss of precision.

Compliant Solution (**ArithmeticException**)

This compliant solution range checks the argument of the integer argument `op1` to ensure it can be represented as a value of type float without a loss of precision:
strictfp class WideSample {
    public static int subFloatFromInt(int op1, float op2)
        throws ArithmeticException {

        // The significand can store at most 23 bits
        if ((op2 > 0x007fffff) || (op2 < -0x800000)) {
            throw new ArithmeticException("Insufficient precision");
        }

        return op1 - (int)op2;
    }

    public static void main(String[] args) {
        int result = subFloatFromInt(1234567890, 1234567890);
        System.out.println(result);
    }
}

In this example, the subFloatFromInt() method throws ArithmeticException. This general approach, with appropriate range checks, can be used for conversions from long to either float or double.

Compliant Solution (Wider Type)

This compliant solution accepts an argument of type double instead of an argument of type float. In FP-strict mode, values of type double have 52 mantissa bits, a sign bit, and an 11-bit exponent. Integer values of type int and narrower can be converted to double without a loss of precision.

strictfp class WideSample {
    public static int subDoubleFromInt(int op1, double op2) {
        return op1 - (int)op2;
    }

    public static void main(String[] args) {
        int result = subDoubleFromInt(1234567890, 1234567890);
        // Works as expected
        System.out.println(result);
    }
}

Note that this compliant solution cannot be used when the primitive integers are of type long because Java lacks a primitive floating-point type whose mantissa can represent the full range of a long.

Exceptions

NUM13-J-EX0: Conversion from integral types to floating-point types without a range check is permitted when suitable numerical analysis demonstrates that the loss of the least significant bits of precision is acceptable.

Risk Assessment

Converting integer values to floating-point types whose mantissa has fewer bits than the original integer value can result in a rounding error.

<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>NUM13-J</td>
<td>Low</td>
<td>Unlikely</td>
<td>Medium</td>
<td>P2</td>
<td>L3</td>
</tr>
</tbody>
</table>

Automated Detection

Automatic detection of casts that can lose precision is straightforward. Sound determination of whether those casts correctly reflect the intent of the programmer is infeasible in the general case. Heuristic warnings 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.AIC</td>
<td>Implemented</td>
</tr>
</tbody>
</table>
Related Guidelines

| SEI CERT C Coding Standard | FLP36-C. Preserve precision when converting integral values to floating-point type |

Bibliography

| JLS 2015 | §5.1.2, "Widening Primitive Conversion" |
| Seacord 2015 | NUM13-J. Avoid loss of precision when converting primitive integers to floating-point LiveLesson |