NUM50-J. Convert integers to floating point for floating-point operations

Incautious use of integer arithmetic to calculate a value for assignment to a floating-point variable can lead to loss of information. For example, integer arithmetic always produces integral results, discarding information about any possible fractional remainder. Furthermore, there can be loss of precision when converting integers to floating-point values. See NUM13-J. Avoid loss of precision when converting primitive integers to floating-point for additional information. Correct programming of expressions that combine integer and floating-point values requires careful consideration.

Operations that could suffer from integer overflow or loss of a fractional remainder should be performed on floating-point values rather than integral values.

Noncompliant Code Example

In this noncompliant code example, the division and multiplication operations are performed on integral values; the results of these operations are then converted to floating point:

```java
short a = 533;
int b = 6789;
long c = 4664382371590123456L;
float d = a / 7;    // d is 76.0 (truncated)
double e = b / 30;  // e is 226.0 (truncated)
double f = c * 2;   // f is -9.1179793305293046E18
```

The results of the integral operations are truncated to the nearest integer and can also overflow. As a result, the floating-point variables \( d \), \( e \), and \( f \) are initialized incorrectly because the truncation and overflow take place before the conversion to floating point.

Note that the calculation for \( c \) violates NUM00-J. Detect or prevent integer overflow.

Compliant Solution (Floating-Point Literal)

This compliant solution performs the multiplication and division operations on floating-point values, avoiding both the truncation and the overflow seen in the noncompliant code example. In every operation, at least one of the operands is of a floating-point type, forcing floating-point multiplication and division and avoiding truncation and overflow.

```java
short a = 533;
int b = 6789;
long c = 4664382371590123456L;
float d = a / 7.0f;       // d is 76.14286
double e = b / 30.;       // e is 226.3
double f = (double)c * 2; // f is 9.328764743180247E18
```

Another compliant solution is to eliminate the truncation and overflow errors by storing the integers in the floating-point variables before performing the arithmetic operations:

```java
short a = 533;
int b = 6789;
long c = 4664382371590123456L;
float d = a;
double e = b;
double f = c;

d /= 7;   // d is 76.14286
e /= 30;  // e is 226.3
f *= 2;   // f is 9.328764743180247E18
```

As in the previous compliant solution, this practice ensure that at least one of the operands of each operation is a floating-point number. Consequently, the operations are performed on floating-point values.
In both compliant solutions, the original value of \( c \) cannot be represented exactly as a `double`. The representation of type `double` has only 48 mantissa bits, but a precise representation of the value of \( c \) would require 56 mantissa bits. Consequently, the value of \( c \) is rounded to the nearest value that can be represented by type `double`, and the computed value of \( f \) (9.328764743180247E18) differs from the exact mathematical result (9328564743180246912). This loss of precision is one of the many reasons correct programming of expressions that mix integer and floating-point operations or values requires careful consideration. See NUM13-J, Avoid loss of precision when converting primitive integers to floating-point for more information about integer-to-floating-point conversion. Even with this loss of precision, however, the computed value of \( f \) is far more accurate than that produced in the noncompliant code example.

**Noncompliant Code Example**

This noncompliant code example attempts to compute the whole number greater than the ratio of two integers. The result of the computation is 1.0 rather than the intended 2.0.

```java
int a = 60070;
int b = 57750;

double value = Math.ceil(a/b);
```

As a consequence of Java's numeric promotion rules, the division operation performed is an `integer` division whose result is truncated to 1. This result is then promoted to `double` before being passed to the `Math.ceil` function.

**Compliant Solution**

This compliant solution casts one of the operands to `double` before the division is performed:

```java
int a = 60070;
int b = 57750;

double value = Math.ceil(a/(double) b);
```

As a result of the cast, the other operand is automatically promoted to `double`. The division operation becomes a double divide, and `value` is assigned the correct result of 2.0. As in the previous compliant solutions, this practice ensures that at least one of the operands of each operation is a floating-point number.

**Applicability**

Improper conversions between integers and floating-point values can yield unexpected results, especially from precision loss. In some cases, these unexpected results can involve overflow or other exceptional conditions.

It is acceptable to perform operations using a mix of integer and floating-point values when deliberately exploiting the properties of integer arithmetic before conversion to floating point. For example, use of integer arithmetic eliminates the need to use the `floor()` method. Any such code must be clearly documented to help future maintainers understand that this behavior is intentional.

**Automated Detection**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SonarQube</td>
<td>6.7</td>
<td>S2184</td>
<td></td>
</tr>
</tbody>
</table>

**Bibliography**

[JLS 2013] §5.1.2, "Widening Primitive Conversion"