NUM12-J. Ensure conversions of numeric types to narrower types do not result in lost or misinterpreted data

Conversions of numeric types to narrower types can result in lost or misinterpreted data if the value of the wider type is outside the range of values of the narrower type. Consequently, all narrowing conversions must be guaranteed safe by range-checking the value before conversion.

Java provides 22 possible *narrowing primitive conversions*. According to *The Java Languagee Specification* (JLS), §5.1.3, "Narrowing Primitive Conversions" [JLS 2015]:

- `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`

Narrowing primitive conversions are allowed in cases where the value of the wider type is within the range of the narrower type.

Integer Narrowing

Integer type ranges are defined by the JLS, §4.2.1, "Integral Types and Values" [JLS 2015], and are also described in NUM00-J. Detect or prevent integer overflow.

The following table presents the rules for narrowing primitive conversions of integer types. In the table, for an integer type `T`, `n` represents the number of bits used to represent the resulting type `T` (precision).

From	To	Description	Possible Resulting Errors
Signed integer	Integral type T	Keeps only n lower-order bits	Lost or misinterpreted data
`char`	Integral type T	Keeps only n lower-order bits	Magnitude error; negative number even though `char` is 16-bit unsigned

When integers are cast to narrower data types, the magnitude of the numeric value and the corresponding sign can be affected. Consequently, data can be lost or misinterpreted.

Floating-Point to Integer Conversion

Floating-point conversion to an integral type `T` is a two-step procedure:

1. When converting a floating-point value to an `int` or `long` and the value is a `NaN`, a zero value is produced. Otherwise, if the value is not infinity, it is rounded toward zero to an integer value `V`:

- If `T` is `long` and `V` can be represented as a `long`, the `long` value `V` is produced.
- If `V` can be represented as an `int`, then the `int` value `V` is produced.

Otherwise,

- The value is negative infinity or a value too negative to be represented, and `Integer.MIN_VALUE` or `Long.MIN_VALUE` is produced.
- The value is positive infinity or a value too positive to be represented, and `Integer.MAX_VALUE` or `Long.MAX_VALUE` is produced.

2. If `T` is `byte`, `char`, or `short`, the result of the conversion is the result of a narrowing conversion to type `T` of the result of the first step

See the JLS, §5.1.3, "Narrowing Primitive Conversions" [JLS 2005], for more information.

Other Conversions

Narrower primitive types can be cast to wider types without affecting the magnitude of numeric values (see the JLS, §5.1.2, Widening Primitive Conversion" [JLS 2005]), for more information). Conversion from `int` or `long` to `float` or from `long` to `double` can lead to loss of precision (loss of least significant bits). No runtime exception occurs despite this loss.

Note that conversions from `float` to `double` or from `double` to `float` 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 (Integer Narrowing)

In this noncompliant code example, a value of type `int` is converted to a value of type `byte` without range checking:

```
class CastAway {
  public static void main(String[] args) {
    int i = 128;
    workWith(i);
  }

  public static void workWith(int i) {
    byte b = (byte) i;  // b has value -128
    // Work with b
  }
}
```

The resulting value may be unexpected because the initial value (128) is outside of the range of the resulting type.

Compliant Solution (Integer Narrowing)

This compliant solution validates that the value stored in the wider integer type is within the range of the narrower type before converting to the narrower type. It throws an `ArithmeticException` if the value is out of range.

```
class CastAway {
  public static void workWith(int i) {
    // Check whether i is within byte range
    if ((i < Byte.MIN_VALUE) || (i > Byte.MAX_VALUE)) {
      throw new ArithmeticException("Value is out of range");
    }

    byte b = (byte) i;
    // Work with b
  }
}
```

Alternatively, the `workWith()` method can explicitly narrow when the programmer's intent is to truncate the value:

```
class CastAway {
   public static void workWith(int i) {
     byte b = (byte)(i % 0x100); // 2^8;
     // Work with b
   }
}
```

Range-checking is unnecessary because the truncation that is normally implicit in a narrowing conversion is made explicit. The compiler will optimize the operation away, and for that reason, no performance penalty is incurred. Similar operations may be used for converting to other integral types.

Noncompliant Code Example (Floating-Point to Integer Conversion)

The narrowing primitive conversions in this noncompliant code example suffer from loss in the magnitude of the numeric value as well as a loss of precision:

```
float i = Float.MIN_VALUE;
float j = Float.MAX_VALUE;
short b = (short) i;
short c = (short) j;
```

The minimum and maximum `float` values are converted to minimum and maximum `int` values (`0x80000000` and `0x7fffffff` respectively). The resulting `short` values are the lower 16 bits of these values (`0x0000` and `0xffff`). The resulting final values (0 and 1) might be unexpected.

Compliant Solution (Floating-Point to Integer Conversion)

This compliant solution range-checks both the `i` and `j` variables before converting to the resulting integer type. Because both values are out of the valid range for a `short`, this code will always throw an `ArithmeticException`.

```
float i = Float.MIN_VALUE;
float j = Float.MAX_VALUE;
if ((i < Short.MIN_VALUE) || (i > Short.MAX_VALUE) ||
    (j < Short.MIN_VALUE) || (j > Short.MAX_VALUE)) {
  throw new ArithmeticException ("Value is out of range");
}

short b = (short) i;
short c = (short) j;
// Other operations
```

Noncompliant Code Example (`double` to `float` Conversion)

The narrowing primitive conversions in this noncompliant code example suffer from a loss in the magnitude of the numeric value as well as a loss of precision. Because `Double.MAX_VALUE` is larger than `Float.MAX_VALUE`, `c` receives the value `infinity`, and because `Double.MIN_VALUE` is smaller than `Float.MIN_VALUE`, `b` receives the value `0`.

```
double i = Double.MIN_VALUE;
double j = Double.MAX_VALUE;
float b = (float) i;
float c = (float) j;
```

Compliant Solution (`double` to `float` Conversion)

This compliant solution performs range checks on both `i` and `j` before proceeding with the conversions. Because both values are out of the valid range for a `float`, this code will always throw an `ArithmeticException`.

```
double i = Double.MIN_VALUE;
double j = Double.MAX_VALUE;
if ((i < Float.MIN_VALUE) || (i > Float.MAX_VALUE) ||
    (j < Float.MIN_VALUE) || (j > Float.MAX_VALUE)) {
  throw new ArithmeticException ("Value is out of range");
}

float b = (float) i;
float c = (float) j;
// Other operations
```

Risk Assessment

Casting a numeric value to a narrower type can result in information loss related to the sign and magnitude of the numeric value. As a result, data can be misrepresented or interpreted incorrectly.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
NUM12-J	Low	Unlikely	Medium	**P2**	**L3**

Automated Detection

Automated detection of narrowing conversions on integral types is straightforward. Determining whether such conversions correctly reflect the intent of the programmer is infeasible in the general case. Heuristic warnings could be useful.

Tool	Version	Checker	Description
Parasoft Jtest	10.3	**PB.NUM.CLP**	Implemented

Related Guidelines

SEI CERT C Coding Standard	INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data FLP34-C. Ensure that floating-point conversions are within range of the new type

ISO/IEC TR 24772:2010	Numeric Conversion Errors [FLC]
MITRE CWE	CWE-681, Incorrect Conversion between Numeric Types CWE-197, Numeric Truncation Error

Bibliography

[Harold 1999]	
[JLS 2005]	§4.2.1, "Integral Types and Values" §5.1.3, "Narrowing Primitive Conversions"
[Seacord 2015]	NUM12-J. Ensure conversions of numeric types to narrower types do not result in lost or misinterpreted data LiveLesson