NUM00-J. Detect or prevent integer overflow

Programs must not allow mathematical operations to exceed the integer ranges provided by their primitive integer data types. According to *The Java Language Specification (JLS)*, §4.2.2, "Integer Operations" [JLS 2015]:

The built-in integer operators do not indicate overflow or underflow in any way. Integer operators can throw a `NullPointerException` if unboxing conversion of a null reference is required. Other than that, the only integer operators that can throw an exception are the integer divide operator `/` and the integer remainder operator `%`, which throw an `ArithmeticException` if the right-hand operand is zero, and the increment and decrement operators `++` and `--` which can throw an `OutOfMemoryError` if boxing conversion is required and there is insufficient memory to perform the conversion.

The integral types in Java, representation, and inclusive ranges are shown in the following table taken from the JLS, §4.2.1, "Integral Types and Values" [JLS 2015]:

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
<th>Inclusive Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8-bit signed two's-complement</td>
<td>128 to 127</td>
</tr>
<tr>
<td>short</td>
<td>16-bit signed two's-complement</td>
<td>32,768 to 32,767</td>
</tr>
<tr>
<td>int</td>
<td>32-bit signed two's-complement</td>
<td>2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>long</td>
<td>64-bit signed two's-complement</td>
<td>9,223,372,036,854,775,808 to 9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>char</td>
<td>16-bit unsigned integers representing UTF-16 code units</td>
<td>\u0000 to \uffff (0 to 65,535)</td>
</tr>
</tbody>
</table>

The following table shows the integer overflow behavior of the integral operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Overflow</th>
<th>Operator</th>
<th>Overflow</th>
<th>Operator</th>
<th>Overflow</th>
<th>Operator</th>
<th>Overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-</code></td>
<td>Yes</td>
<td><code>--</code></td>
<td>No</td>
<td><code>&lt;</code></td>
<td>No</td>
<td><code>&lt;</code></td>
<td>No</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Yes</td>
<td><code>*=</code></td>
<td>No</td>
<td><code>&gt;</code></td>
<td>No</td>
<td><code>&gt;</code></td>
<td>No</td>
</tr>
<tr>
<td><code>/</code></td>
<td>No</td>
<td>`</td>
<td>=`</td>
<td>No</td>
<td><code>\</code></td>
<td>No</td>
<td><code>\</code></td>
</tr>
<tr>
<td><code>%</code></td>
<td>No</td>
<td><code>&lt;&lt;</code></td>
<td>No</td>
<td><code>^</code></td>
<td>No</td>
<td><code>^</code></td>
<td>No</td>
</tr>
<tr>
<td><code>++</code></td>
<td>Yes</td>
<td><code>&gt;&gt;</code></td>
<td>No</td>
<td><code>-</code></td>
<td>No</td>
<td><code>!</code></td>
<td>No</td>
</tr>
<tr>
<td><code>--</code></td>
<td>Yes</td>
<td><code>{=</code></td>
<td>No</td>
<td><code>=</code></td>
<td>No</td>
<td><code>Unary +</code></td>
<td>No</td>
</tr>
<tr>
<td><code>=</code></td>
<td>No</td>
<td><code>=</code></td>
<td>No</td>
<td><code>Unary -</code></td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Because the ranges of Java types are not symmetric (the negation of each minimum value is one more than each maximum value), even operations such as unary negation can overflow if applied to a minimum value. Because the `java.lang.math.abs()` method returns the absolute value of any number, it can also overflow if given the minimum `int` or `long` as an argument.

When a mathematical operation cannot be represented using the supplied integer types, Java’s built-in integer operators silently wrap the result without indicating overflow. The silent wrap can result in incorrect computations and unanticipated outcomes. Failure to account for integer overflow has resulted in failures of real systems, for example, when implementing the `compareTo()` method. The meaning of the return value of the `compareTo()` method is defined only in terms of its sign and whether it is zero; the magnitude of the return value is irrelevant. Consequently, an apparent but incorrect optimization would be to subtract the operands and return the result. For operands of opposite signs, this approach can result in integer overflow, consequently violating the `compareTo()` contract [Bloch 2008].

Comparison of Compliant Techniques

Following are the three main techniques for detecting unintended integer overflow:

- **Precondition testing.** Check the inputs to each arithmetic operator to ensure that overflow cannot occur. Throw an `ArithmeticException` when the operation would overflow if it were performed; otherwise, perform the operation.

- **Upcasting.** Cast the inputs to the next larger primitive integer type and perform the arithmetic in the larger size. Check each intermediate result for overflow of the original smaller type and throw an `ArithmeticException` if the range check fails. Note that the range check must be performed after each arithmetic operation; larger expressions without per-operation bounds checking can overflow the larger type. Downcast the final result to the original smaller type before assigning to a variable of the original smaller type. This approach cannot be used for type `long` because `long` is already the largest primitive integer type.

- **BigInteger.** Convert the inputs into objects of type `BigInteger` and perform all arithmetic using `BigInteger` methods. Type `BigInteger` is the standard arbitrary-precision integer type provided by the Java standard libraries. The arithmetic operations implemented as methods of this type cannot overflow; instead, they produce the numerically correct result. Consequently, compliant code performs only a single range check just before converting the final result to the original smaller type and throws an `ArithmeticException` if the final result is outside the range of the original smaller type.
The precondition testing technique requires different precondition tests for each arithmetic operation. This approach can be somewhat more difficult to implement and to audit than either of the other two approaches.

The upcast technique is the preferred approach when applicable. The checks it requires are simpler than those of the previous technique; it is substantially more efficient than using BigInteger. Unfortunately, it cannot be applied to operations involving type long, as there is no bigger type to upcast to.

The BigInteger technique is conceptually the simplest of the three techniques because arithmetic operations on BigInteger cannot overflow. However, it requires the use of method calls for each operation in place of primitive arithmetic operators, which can obscure the intended meaning of the code. Operations on objects of type BigInteger can also be significantly less efficient than operations on the original primitive integer type.

Precondition Testing

The following code example shows the necessary precondition checks required for each arithmetic operation on arguments of type int. The checks for the other integral types are analogous. These methods throw an exception when an integer overflow would otherwise occur; any other conforming error handling is also acceptable. Since ArithmeticException inherits from RuntimeException, we do not need to declare it in a throws clause.

```java
static final int safeAdd(int left, int right) {
    if (right > 0 ? left > Integer.MAX_VALUE - right
                  : left < Integer.MIN_VALUE - right) {
        throw new ArithmeticException("Integer overflow");
    } return left + right;
}

static final int safeSubtract(int left, int right) {
    if (right > 0 ? left < Integer.MIN_VALUE + right
                   : left > Integer.MAX_VALUE + right) {
        throw new ArithmeticException("Integer overflow");
    } return left - right;
}

static final int safeMultiply(int left, int right) {
    if (right > 0 ? left > Integer.MAX_VALUE/right
                   || left < Integer.MIN_VALUE/right
                   || right < -1 ? left > Integer.MIN_VALUE/right
                   || left < Integer.MAX_VALUE/right
                   : right == -1
                   && left == Integer.MIN_VALUE) {
        throw new ArithmeticException("Integer overflow");
    } return left * right;
}

static final int safeDivide(int left, int right) {
    if ((left == Integer.MIN_VALUE) && (right == -1)) {
        throw new ArithmeticException("Integer overflow");
    } return left / right;
}

static final int safeNegate(int a) {
    if (a == Integer.MIN_VALUE) {
        throw new ArithmeticException("Integer overflow");
    } return -a;
}

static final int safeAbs(int a) {
    if (a == Integer.MIN_VALUE) {
        throw new ArithmeticException("Integer overflow");
    } return Math.abs(a);
}
```

These method calls are likely to be inlined by most just-in-time (JIT) systems.

These checks can be simplified when the original type is char. Because the range of type char includes only positive values, all comparisons with negative values may be omitted.
Noncompliant Code Example

Either operation in this noncompliant code example could result in an overflow. When overflow occurs, the result will be incorrect.

```java
public static int multAccum(int oldAcc, int newVal, int scale) {
    // May result in overflow
    return oldAcc + (newVal * scale);
}
```

Compliant Solution (Precondition Testing)

This compliant solution uses the `safeAdd()` and `safeMultiply()` methods defined in the "Precondition Testing" section to perform secure integral operations or throw `ArithmeticException` on overflow:

```java
public static int multAccum(int oldAcc, int newVal, int scale) {
    return safeAdd(oldAcc, safeMultiply(newVal, scale));
}
```

Compliant Solution (Java 8, `Math.*Exact()`)  

This compliant solution uses the `addExact()` and `multiplyExact()` methods defined in the `Math` class. These methods were added to Java as part of the Java 8 release, and they also either return a mathematically correct value or throw `ArithmeticException`. The `Math` class also provides `SubtractExact()` and `negateExact()` but does not provide any methods for safe division or absolute value.

```java
public static int multAccum(int oldAcc, int newVal, int scale) {
    return Math.addExact(oldAcc, Math.multiplyExact(newVal, scale));
}
```

Compliant Solution (Upcasting)

This compliant solution shows the implementation of a method for checking whether a value of type `long` falls within the representable range of an `int` using the upcasting technique. The implementations of range checks for the smaller primitive integer types are similar.

```java
public static long intRangeCheck(long value) {
    if ((value < Integer.MIN_VALUE) || (value > Integer.MAX_VALUE)) {
        throw new ArithmeticException("Integer overflow");
    }
    return value;
}

public static int multAccum(int oldAcc, int newVal, int scale) {
    final long res = intRangeCheck((long) oldAcc) + intRangeCheck((long) newVal * (long) scale);
    return (int) res; // Safe downcast
}
```

Note that this approach cannot be applied to values of type `long` because `long` is the largest primitive integral type. Use the `BigInteger` technique instead when the original variables are of type `long`.

Compliant Solution (`BigInteger`)

This compliant solution uses the `BigInteger` technique to detect overflow:

```java
public static long intRangeCheck(long value) {
    if ((value < Integer.MIN_VALUE) || (value > Integer.MAX_VALUE)) {
        throw new ArithmeticException("Integer overflow");
    }
    return value;
}

public static long multAccum(long oldAcc, long newVal, long scale) {
    final long res = intRangeCheck((long) oldAcc) + intRangeCheck((long) newVal * (long) scale);
    return res; // Safe downcast
}
```
private static final BigInteger bigMaxInt =
    BigInteger.valueOf(Integer.MAX_VALUE);
private static final BigInteger bigMinInt =
    BigInteger.valueOf(Integer.MIN_VALUE);

public static BigInteger intRangeCheck(BigInteger val) {
    if (val.compareTo(bigMaxInt) == 1 ||
        val.compareTo(bigMinInt) == -1) {
        throw new ArithmeticException("Integer overflow");
    }
    return val;
}

public static int multAccum(int oldAcc, int newVal, int scale) {
    BigInteger product =
        BigInteger.valueOf(newVal).multiply(BigInteger.valueOf(scale));
    BigInteger res =
        intRangeCheck(BigInteger.valueOf(oldAcc).add(product));
    return res.intValue(); // Safe conversion
}

Noncompliant Code Example (AtomicInteger)

Operations on objects of type AtomicInteger suffer from the same overflow issues as other integer types. The solutions are generally similar to the solutions already presented; however, concurrency issues add additional complications. First, potential issues with time-of-check, time-of-use (TOCTOU) must be avoided (see VNA02-J. Ensure that compound operations on shared variables are atomic for more information). Second, use of an AtomicInteger creates happens-before relationships between the various threads that access it. Consequently, changes to the number of accesses or order of accesses can alter the execution of the overall program. In such cases, you must either choose to accept the altered execution or carefully craft your implementation to preserve the exact number of accesses and order of accesses to the AtomicInteger.

This noncompliant code example uses an AtomicInteger, which is part of the concurrency utilities. The concurrency utilities lack integer overflow checks.

class InventoryManager {
    private final AtomicInteger itemsInInventory = new AtomicInteger(100);

    //...
    public final void nextItem() {
        itemsInInventory.getAndIncrement();
    }
}

Consequently, itemsInInventory can wrap around to Integer.MIN_VALUE when the nextItem() method is invoked when itemsInInventory == Integer.MAX_VALUE.

Compliant Solution (AtomicInteger)

This compliant solution uses the get() and compareAndSet() methods provided by AtomicInteger to guarantee successful manipulation of the shared value of itemsInInventory. This solution has the following characteristics:

- The number and order of accesses to itemsInInventory remain unchanged from the noncompliant code example.
- All operations on the value of itemsInInventory are performed on a temporary local copy of its value.
- The overflow check in this example is performed in inline code rather than encapsulated in a method call. This is an acceptable alternative implementation. The choice of method call versus inline code should be made according to your organization’s standards and needs.
The two arguments to the `compareAndSet()` method are the expected value of the variable when the method is invoked and the intended new value. The variable’s value is updated only when the current value and the expected value are equal [API 2006] (refer to VNA02-J. Ensure that compound operations on shared variables are atomic for more details).

**Exceptions**

**NUM00-J-EX0**: Depending on circumstances, integer overflow could be benign. For example, many algorithms for computing hash codes use modular arithmetic, intentionally allowing overflow to occur. Such benign uses must be carefully documented.

**NUM00-J-EX1**: Prevention of integer overflow is unnecessary for numeric fields that undergo bitwise operations and not arithmetic operations (see NUM01-J. Do not perform bitwise and arithmetic operations on the same data for more information).

**Risk Assessment**

Failure to perform appropriate range checking can lead to integer overflows, which can cause unexpected program control flow or unanticipated program behavior.

<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>NUM00-J</td>
<td>Medium</td>
<td>Unlikely</td>
<td>Medium</td>
<td>P4</td>
<td>L3</td>
</tr>
</tbody>
</table>

**Automated Detection**

Automated detection of integer operations that can potentially overflow is straightforward. Automatic determination of which potential overflows are true errors and which are intended by the programmer is infeasible. Heuristic warnings might be helpful.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverity</td>
<td>7.5</td>
<td>BAD_SHIFT OVERFLOW_BEFORE_WIDEN</td>
<td>Implemented</td>
</tr>
<tr>
<td>Parasoft Jtest</td>
<td>10.3</td>
<td>PB.NUM.(ICO,BSA,CACO)</td>
<td></td>
</tr>
</tbody>
</table>

**Related Guidelines**

<table>
<thead>
<tr>
<th>SEI CERT C Coding Standard</th>
<th>INT32-C. Ensure that operations on signed integers do not result in overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO/IEC TR 24772:2010</td>
<td>Wrap-around Error [XYY]</td>
</tr>
</tbody>
</table>
| MITRE CWE                  | CWE-682, Incorrect Calculation  
CWE-190, Integer Overflow or Wraparound  
CWE-191, Integer Underflow (Wrap or Wraparound) |

**Android Implementation Details**

Mezzofanti for Android contained an `integer overflow` that prevented the use of a big SD card. Mezzofanti contained an expression:

```java
(int) StatFs.getAvailableBlocks() * (int) StatFs.getBlockSize()
```
to calculate the available memory in an SD card, which could result in a negative value when the available memory is larger than `Integer.MAX_VALUE`. Note that these methods are deprecated in API level 18 and replaced by `getAvailableBlocksLong()` and `getBlockSizeLong()`.

### Bibliography

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[API 2006]</td>
<td>Class <code>AtomicInteger</code></td>
</tr>
<tr>
<td>[Bloch 2005]</td>
<td>Puzzle 27, “Shifty i’s”</td>
</tr>
<tr>
<td>[Bloch 2008]</td>
<td>Item 12, “Minimize the Accessibility of Classes and Members”</td>
</tr>
<tr>
<td>[Java Tutorials]</td>
<td><code>Primitive Data Types</code></td>
</tr>
<tr>
<td>[JLS 2015]</td>
<td>§4.2.1, &quot;Integral Types and Values&quot;</td>
</tr>
<tr>
<td></td>
<td>§4.2.2, &quot;Integer Operations&quot;</td>
</tr>
<tr>
<td></td>
<td>§15.22, &quot;Bitwise and Logical Operators&quot;</td>
</tr>
<tr>
<td>[Seacord 2005]</td>
<td>Chapter 5, &quot;Integers&quot;</td>
</tr>
</tbody>
</table>