Secure Coding Standards (Selected) in Java

Dr. Natarajan Meghanathan
Associate Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Standards

- Standard-1: Detect or Prevent Integer Overflow
- Standard-2: Do Not Use Floating Point Values for Precise Computation
- Standard 3: Do not Attempt Comparisons with NaN
- Standard 4: Check Floating Point Inputs for Exceptional Values
- Standard 5: Conversions of Numeric Types to Narrower Types should not result in Lost or Misinterpreted Data

Standard-1: Detect or Prevent Integer Overflow

- Programs should not permit arithmetic operations to exceed the ranges provided by the various primitive integer data types.
  - In the Java language, the only integer operators that can throw an exception are the / and % operators, which throw an *Arithmetic Exception* if the right-hand operand is a 0. In addition, the -- or ++ unary operators throw an *OutofMemoryError* if the decrement or increment operation requires insufficient memory.

Type	Representation	Inclusive Range
`byte`	8-bit signed two's-complement	-128 to 127
`short`	16-bit signed two's-complement	-32,768 to 32,767
`int`	32-bit signed two's-complement	-2,147,483,648 to 2,147,483,647
`long`	64-bit signed two's-complement	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Vulnerable Program: Integer Overflow

```
class intOverflow{

   public static void main(String[] args){

    int arg1 = Integer.parseInt(args[0]);
    int arg2 = Integer.parseInt(args[1]);

    int sum = arg1 + arg2;

    System.out.println("int sum: "+sum);

   }

}
```

```
C:\res\SCA\8-intOverflow>java intOverflow 2147483645 4
int sum: -2147483647
```

Solution # 1: Pre-condition Testing

- Idea: 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.

```java
class intOverflow{

   public static int safeAdd(int left, int right)
                      throws ArithmeticException{

      if (right > 0 ?
                 left > Integer.MAX_VALUE - right :
                 left < Integer.MIN_VALUE - right){
           throw new ArithmeticException("Integer overflow");
      }

      return left + right;
   }

   public static void main(String[] args){

    int arg1 = Integer.parseInt(args[0]);
    int arg2 = Integer.parseInt(args[1]);

    try{
      int sum = safeAdd(arg1, arg2);
      System.out.println("int sum: "+sum);
      }
     catch(ArithmeticException ae){
         System.out.println(ae);
     }

   }
}
```

Pre-condition Testing for Addition

```
C:\res\SCA\8-intOverflow>java intOverflow 2147483645 4
java.lang.ArithmeticException: Integer overflow

C:\res\SCA\8-intOverflow>java intOverflow 2 4
int sum: 6
```

Code Segments for Safe Arithmetic

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

For the sake of understanding,
Assume Integer.MAX_VALUE = 127
Integer.MIN_VALUE = -128

```
static final int safeMultiply(int left, int right)
                 throws ArithmeticException {
  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;
}
```

left = 65; right = 2 (→ left > Integer.MAX_VALUE/right)

left = - 65; right = 2 (→ left < Integer.MIN_VALUE/right)

left = 65; right = -2 (→ left > Integer.MIN_VALUE/right)

left = - 65; right = -2 (→ left < Integer.MAX_VALUE/right)

Source: https://www.securecoding.cert.org/confluence/display/java/NUM00-J.+Detect+or+prevent+integer+overflow

Code Segments for Safe Arithmetic

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

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

Source: https://www.securecoding.cert.org/confluence/display/java/NUM00-J.+Detect+or+prevent+integer+overflow

Solution # 2: Upcasting

- Idea:
  - Cast the inputs to the next larger integer type
  - Do the arithmetic operation on the larger type
  - Check the value of each intermediate result and final result to see if it would still fit within the range of the original integer type; if not raise an *ArithmeticException*
  - Downcast the final result to the original smaller type before assigning the result to a variable of smaller type and throw an

Vulnerable Program

```
class Upcasting{

   // Evaluating the expression (a + b - c)
   // where a, b and c are of type short

   public static void main(String[] args){

    short short_a = Short.parseShort(args[0]);
    short short_b = Short.parseShort(args[1]);
    short short_c = Short.parseShort(args[2]);

    short result = (short) (short_a + short_b - short_c);

    System.out.println("result in short: "+ result);

    }

}
```

```
C:\res\SCA\8-intOverflow>java Upcasting 3 4 5
result in short: 2

C:\res\SCA\8-intOverflow>java Upcasting 32760 10 5
result in short: 32765

C:\res\SCA\8-intOverflow>java Upcasting 32760 100 5
result in short: -32681
```

How is this possible???

Solution using Upcasting

```
class Upcasting{

   public static int checkShortRange(int value)
      throws ArithmeticException{

          if (value > Short.MAX_VALUE || value < Short.MIN_VALUE)
             throw new ArithmeticException("Integer overflow");

          return value;
   }

      public static int safeAdd(int left, int right)
                        throws ArithmeticException{

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

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

Continued…..

```java
public static void main(String[] args){

 try{
  short short_a = Short.parseShort(args[0]);
  short short_b = Short.parseShort(args[1]);
  short short_c = Short.parseShort(args[2]);

  short result = (short)
                checkShortRange(
                   safeSubtract(
                      checkShortRange( safeAdd(short_a, short_b) ),
                      short_c  )
                          );

  System.out.println("result in short: "+ result);
 }
 catch(ArithmeticException ae){
    System.out.println(ae);
 }

}

}
```

```
C:\res\SCA\8-intOverflow>java Upcasting 3 4 5
result in short: 2

C:\res\SCA\8-intOverflow>java Upcasting 32760 10 5
java.lang.ArithmeticException: Integer overflow

C:\res\SCA\8-intOverflow>java Upcasting 32760 100 5
java.lang.ArithmeticException: Integer overflow

C:\res\SCA\8-intOverflow>java Upcasting 32740 10 5
result in short: 32745
```

Standard 2: Do Not Use Floating Point Values for Precise Computation

- 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.

<u>Vulnerable Code: Program requiring Precise Computation</u>

```
class floatingPointValue{

   public static void main(String[] args){

     double dollar = 1.00;
     double dime = 0.10;

     int number = 7;

     System.out.println(" A dollar less "+number+" dimes is $"+
(dollar - number*dime) );

   }
}
```

```
C:\res\SCA\11-floatDouble>java floatVsDouble
 A dollar less 7 dimes is $0.29999999999999993
```

Solution: Use Integer types

```
class floatingPointValue{

   public static void main(String[] args){

     int dollar = 100;
     int dime = 10;

     int number = 7;

     System.out.println(" A dollar less "+number+
       " dimes is "+ (dollar - number*dime) +" cents" );

   }

}
```

```
C:\res\SCA\11-floatDouble>java floatingPointValue
 A dollar less 7 dimes is 30 cents
```

Do not use Floating Point Values as Loop Counters

```
class floatLoopCounters{

    public static void main(String[] args){

        int counter = 1;

        for (double i = 0.1; i <= 2.0; i += 0.1){
            System.out.println(i+"\t"+counter);
            counter++;
        }

    }

}
```

```
C:\res\SCA\11-floatDouble>java floatLoopCounters
0.1     1
0.2     2
0.30000000000000004     3
0.4     4
0.5     5
0.6     6
0.7     7
0.7999999999999999      8
0.8999999999999999      9
0.9999999999999999      10
1.0999999999999999      11
1.2     12
1.3     13
1.4000000000000001      14
1.5000000000000002      15
1.6000000000000003      16
1.7000000000000004      17
1.8000000000000005      18
1.9000000000000006      19

C:\res\SCA\11-floatDouble>
```

Solution: Use Integer Loop Counter

```
class floatLoopCounters{

    public static void main(String[] args){

        for (int counter = 1; counter <= 20; counter++){
            System.out.println((counter*0.1)+"\t"+counter);
        }

    }

}
```

```
C:\res\SCA\11-floatDouble>java floatLoopCounters
0.1     1
0.2     2
0.30000000000000004     3
0.4     4
0.5     5
0.6000000000000001      6
0.7000000000000001      7
0.8     8
0.9     9
1.0     10
1.1     11
1.2000000000000002      12
1.3     13
1.4000000000000001      14
1.5     15
1.6     16
1.7000000000000002      17
1.8     18
1.9000000000000001      19
2.0     20

C:\res\SCA\11-floatDouble>
```

Standard 3: Do not Attempt Comparisons with NaN

- Use of the numerical comparison operators (<, <=, >, >=, ==) with NaN (not a number) returns false, if either or both operands are NaN.
- Use of the inequality operator (!=) returns true, if either operand is NaN.

```
class NaNComparison{

  public static void main(String[] args){

   double x = 0.0;
   double result = Math.sin(1/x);

   if ( result == Double.NaN ){
      System.out.println("result is NaN");
   }
   else{
      System.out.println("result is not a NaN");
    }
  }

}
```

```
C:\res\SCA\12-NaNComparison>java NaNComparison
result is not a NaN
```

Solution: Use the Double.isNaN(double) Method

```
class NaNComparison{

   public static void main(String[] args){

    double x = 0.0;
    double result = Math.sin(1/x);

    if ( Double.isNaN(result) ){
       System.out.println("result is NaN");
    }
    else{
       System.out.println("result is not a NaN "+result);
     }

   }

}
```

```
C:\res\SCA\12-NaNComparison>java NaNComparison
result is NaN
```

Standard 4: Check Floating Point Inputs for Exceptional Values

- Floating-point numbers can take on three exceptional values: infinity, -infinity, and NaN (not-a-number). These values are produced as a result of exceptional or otherwise unresolvable floating-point operations, such as division by zero, or can be input by the user.

```
class floatingInputCheck{

   public static void main(String[] args){

      double arg1 = Double.parseDouble(args[0]);
      double arg2 = Double.parseDouble(args[1]);

      if (arg1 >= Double.MAX_VALUE - arg2){
          System.out.println("integer overflow error..");
      }
      else{
          System.out.println("sum is: "+(arg1+arg2) );
      }
   }

}
```

```
C:\res\SCA\11-floatDouble>java floatingInputCheck 34 NaN
sum is: NaN
```

Solution: Check Values before Use

```
class floatingInputCheck{

   public static void main(String[] args){

     double arg1 = Double.parseDouble(args[0]);
     double arg2 = Double.parseDouble(args[1]);

     if (Double.isNaN(arg1) || Double.isNaN(arg2) ){
          System.out.println("Input(s) is a NaN");
          return;
     }

     if (Double.isInfinite(arg1) || Double.isInfinite(arg2) ){
          System.out.println("Input(s) is infinite");
          return;
     }

     if (arg1 >= Double.MAX_VALUE - arg2){
          System.out.println("integer overflow error..");
          return;
     }

     System.out.println("Sum is "+(arg1+arg2) );
   }
}
```

```
C:\res\SCA\11-floatDouble>java floatingInputCheck 34 NaN
Input(s) is a NaN

C:\res\SCA\11-floatDouble>java floatingInputCheck 34 Infinity
Input(s) is infinite

C:\res\SCA\11-floatDouble>java floatingInputCheck 34 45
Sum is 79.0
```

Standard 5: Conversions of Numeric Types to Narrower Types should not result in Lost or Misinterpreted Data

```
class narrowConversions{

   public static byte castByte(int x){
      return (byte) x;
   }

   public static void main(String[] args){

     int x = 128;
     byte b_x = castByte(x);
     System.out.println(b_x);
   }
}
```

```
C:\res\SCA\13-narrowConversions>java narrowConversions
-128
```

Solution: Range Check the Values before Conversion

```
class narrowConversions{

  public static byte castByte(int x) throws ArithmeticException{
     if (x > Byte.MAX_VALUE || x < Byte.MIN_VALUE)
        throw new ArithmeticException("byte overflow");

     return (byte) x;
  }

  public static void main(String[] args){

    int x = Integer.parseInt(args[0]);

    try{
      byte b_x = castByte(x);
      System.out.println(b_x);
    }
    catch(ArithmeticException ae){
       System.out.println(ae);
    }

  }
}
```

```
C:\res\SCA\13-narrowConversions>java narrowConversions 128
java.lang.ArithmeticException: byte overflow

C:\res\SCA\13-narrowConversions>java narrowConversions 12
12
```