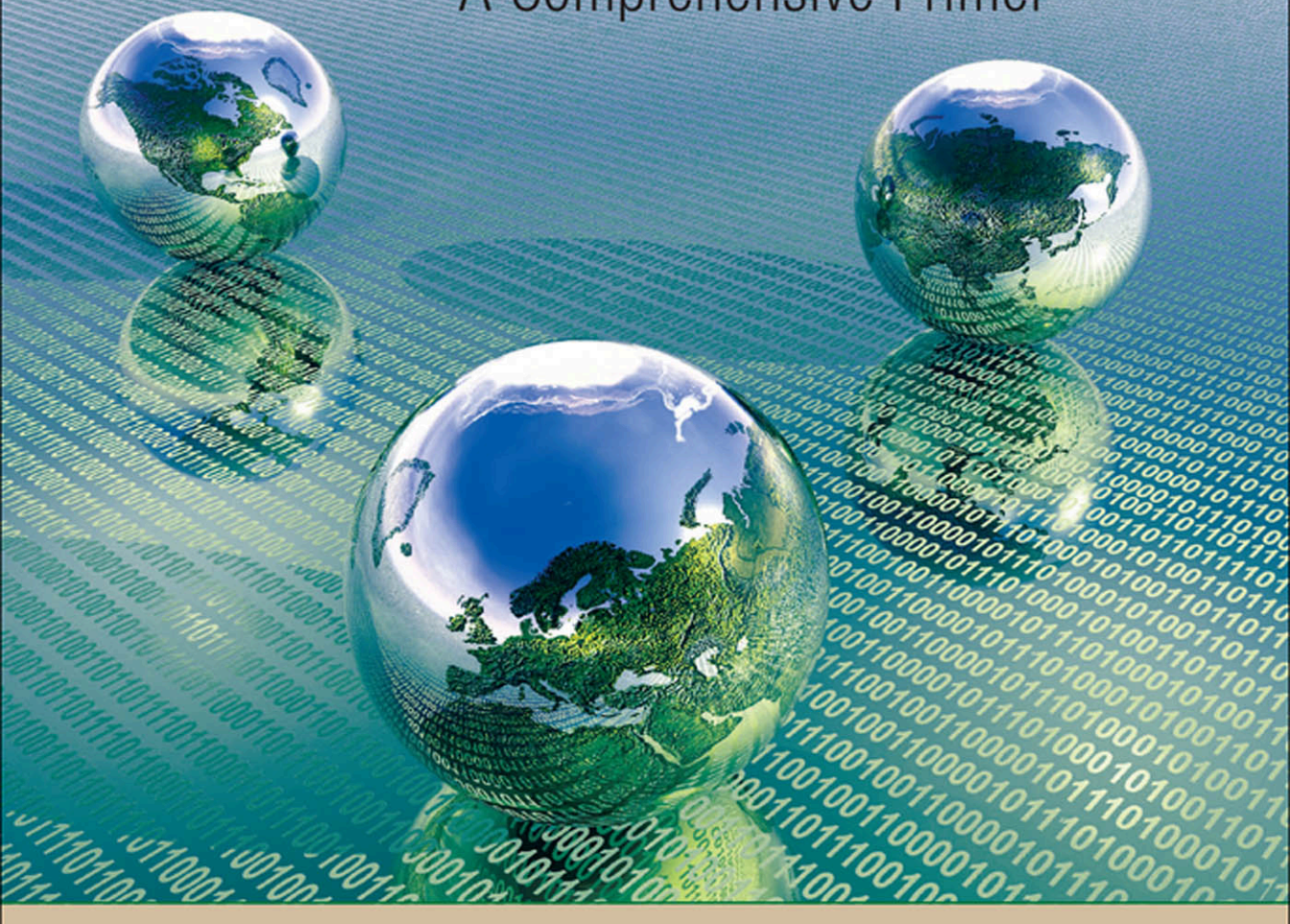
EXAM 1Z0-808

A Programmer's Guide to Java® SE 8 Oracle Certified Associate (OCA)

A Comprehensive Primer



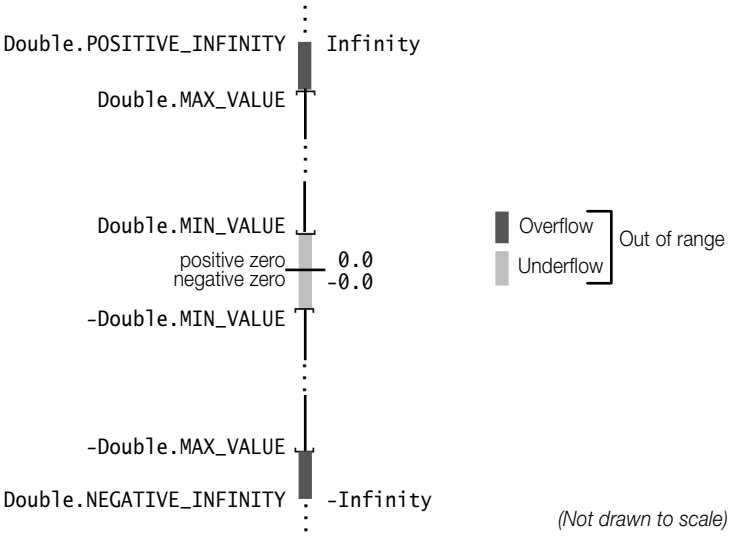
Khalid A. Mughal • Rolf W. Rasmussen

A Programmer's Guide to

Java® SE 8

Oracle Certified Associate (OCA)

Figure 5.3 *Overflow and Underflow in Floating-Point Arithmetic*



Certain operations have no mathematical result, and are represented by *NaN* (*Not a Number*). For example, calculating the square root of -1 results in NaN. Another example is (floating-point) dividing zero by zero:

```
System.out.println(0.0 / 0.0);        // Prints: NaN
```

NaN is represented by the constant named `NaN` in the wrapper classes `java.lang.Float` and `java.lang.Double`. Any operation involving NaN produces NaN. Any comparison (except inequality `!=`) involving NaN and any other value (including NaN) returns `false`. An inequality comparison of NaN with another value (including NaN) always returns `true`. However, the recommended way of checking a value for NaN is to use the static method `isNaN()` defined in both wrapper classes, `java.lang.Float` and `java.lang.Double`.

Strict Floating-Point Arithmetic: `strictfp`

Although floating-point arithmetic in Java is defined in accordance with the IEEE-754 32-bit (`float`) and 64-bit (`double`) standard formats, the language does allow JVM implementations to use other extended formats for intermediate results. This means that floating-point arithmetic can give different results on such JVMs, with possible loss of precision. Such a behavior is termed *non-strict*, in contrast to being *strict* and adhering to the standard formats.

To ensure that identical results are produced on all JVMs, the keyword `strictfp` can be used to enforce strict behavior for floating-point arithmetic. The modifier `strictfp` can be applied to classes, interfaces, and methods. A `strictfp` method ensures that all code in the method is executed strictly. If a class or interface is declared to be `strictfp`, then all code (in methods, initializers, and nested classes

and interfaces) is executed strictly. If the expression is determined to be in a `strictfp` construct, it is executed strictly. Strictness, however, is not inherited by the subclasses or subinterfaces. Constant expressions are always evaluated strictly at compile time.

Unary Arithmetic Operators: -, +

The unary operators have the highest precedence of all the arithmetic operators. The unary operator - negates the numeric value of its operand. The following example illustrates the right associativity of the unary operators:

```
int value = - -10;              // (-(-10)) is 10
```

Notice the blank needed to separate the unary operators; otherwise, these would be interpreted as the decrement operator -- (§5.9, p. 176), which would result in a compile-time error because a literal cannot be decremented. The unary operator + has no effect on the evaluation of the operand value.

Multiplicative Binary Operators: *, /, %

*Multiplication Operator: **

The multiplication operator * multiplies two numbers.

```
int    sameSigns     = -4    * -8;   // result:  32
double oppositeSigns =  4    * -8.0; // Widening of int 4 to double. result: -32.0
int    zero          =  0    * -0;   // result:   0
```

Division Operator: /

The division operator / is overloaded. If its operands are integral, the operation results in *integer division*.

```
int    i1 = 4  / 5;   // result: 0
int    i2 = 8  / 8;   // result: 1
double d1 = 12 / 8;   // result: 1.0; integer division, then widening conversion
```

Integer division always returns the quotient as an integer value; that is, the result is truncated toward zero. Note that the division performed is integer division if the operands have integral values, even if the result will be stored in a floating-point type. The integer value is subjected to a widening conversion in the assignment context.

An `ArithmeticException` is thrown when integer division with zero is attempted, meaning that integer division by zero is an illegal operation.

If any of the operands is a floating-point type, the operation performs *floating-point division*, where relevant operand values undergo binary numeric promotion:

```
double d2 = 4.0 / 8;      // result: 0.5
double d3 = 8 / 8.0;      // result: 1.0
```

```
float d4  = 12.0F / 8;    // result: 1.5F

double result1 = 12.0 / 4.0 * 3.0;       // ((12.0 / 4.0) * 3.0) which is 9.0
double result2 = 12.0 * 3.0 / 4.0;       // ((12.0 * 3.0) / 4.0) which is 9.0
```

Remainder Operator: %

In mathematics, when we divide a number (the *dividend*) by another number (the *divisor*), the result can be expressed in terms of a *quotient* and a *remainder*. For example, when 7 is divided by 5, the quotient is 1 and the remainder is 2. The remainder operator % returns the remainder of the division performed on the operands.

```
int quotient  = 7 / 5;   // Integer division operation: 1
int remainder = 7 % 5;   // Integer remainder operation: 2
```

For *integer remainder operation*, where only integer operands are involved, evaluation of the expression (x % y) always satisfies the following relation:

$x == (x \ / \ y) * y + (x \ \% \ y)$

In other words, the right-hand side yields a value that is always equal to the value of the dividend. The following examples show how we can calculate the remainder so that this relation is satisfied:

Calculating (7 % 5):

```
7 == (7 / 5) * 5 + (7 % 5)
  == (  1  ) * 5 + (7 % 5)
  ==           5 + (7 % 5)
2 ==               (7 % 5)          (7 % 5) is equal to 2
```

Calculating (7 % -5):

```
7 == (7 / -5) * -5 + (7 % -5)
  == (  -1  ) * -5 + (7 % -5)
  ==             5 + (7 % -5)
2 ==                 (7 % -5)       (7 % -5) is equal to 2
```

Calculating (-7 % 5):

```
-7 == (-7 / 5) * 5 + (-7 % 5)
   == (  -1  ) * 5 + (-7 % 5)
   ==           -5 + (-7 % 5)
-2 ==                (-7 % 5)       (-7 % 5) is equal to -2
```

Calculating (-7 % -5):

```
-7 == (-7 / -5) * -5 + (-7 % -5)
   == (   1   ) * -5 + (-7 % -5)
   ==             -5 + (-7 % -5)
-2 ==                  (-7 % -5)    (-7 % -5) is equal to -2
```

The remainder can be negative only if the dividend is negative, and the sign of the divisor is irrelevant. A shortcut to evaluating the remainder involving negative operands is the following: ignore the signs of the operands, calculate the remainder, and negate the remainder if the dividend is negative.

```
int  r0 =  7  %  7;     //  0
int  r1 =  7  %  5;     //  2
long r2 =  7L % -5L;    //  2L
int  r3 = -7  %  5;     // -2
long r4 = -7L % -5L;    // -2L
boolean relation = -7L == (-7L / -5L) * -5L + r4;  // true
```

An `ArithmeticException` is thrown if the divisor evaluates to zero.

Note that the remainder operator accepts not only integral operands, but also floating-point operands. The *floating-point remainder* `r` is defined by the relation

$r == a - (b * q)$

where `a` and `b` are the dividend and the divisor, respectively, and `q` is the *integer* quotient of `(a/b)`. The following examples illustrate a floating-point remainder operation:

```
double  dr0 =  7.0  %  7.0;    //  0.0
float   fr1 =  7.0F %  5.0F;   //  2.0F
double  dr1 =  7.0  % -5.0;    //  2.0
float   fr2 = -7.0F %  5.0F;   // -2.0F
double  dr2 = -7.0  % -5.0;    // -2.0
boolean fpRelation = dr2  == (-7.0) - (-5.0) * (long)(-7.0 / -5.0);  // true
float   fr3 = -7.0F %  0.0F;   // NaN
```

Additive Binary Operators: +, -

The addition operator + and the subtraction operator - behave as their names imply: They add and subtract values, respectively. The binary operator + also acts as *string concatenation* if any of its operands is a string (§5.8, p. 174).

Additive operators have lower precedence than all the other arithmetic operators. Table 5.6 includes examples that show how precedence and associativity are used in arithmetic expression evaluation.

Table 5.6 *Examples of Arithmetic Expression Evaluation*

Arithmetic expression	Evaluation	Result when printed
`3 + 2 - 1`	`((3 + 2) - 1)`	`4`
`2 + 6 * 7`	`(2 + (6 * 7))`	`44`
`-5 + 7 - -6`	`(((-5) + 7) - (-6))`	`8`
`2 + 4 / 5`	`(2 + (4 / 5))`	`2`
`13 % 5`	`(13 % 5)`	`3`
`11.5 % 2.5`	`(11.5 % 2.5)`	`1.5`
`10 / 0`		`ArithmeticException`
`2 + 4.0 / 5`	`(2.0 + (4.0 / 5.0))`	`2.8`
`4.0 / 0.0`	`(4.0 / 0.0)`	`Infinity`
`-4.0 / 0.0`	`((-4.0) / 0.0)`	`-Infinity`
`0.0 / 0.0`	`(0.0 / 0.0)`	`NaN`

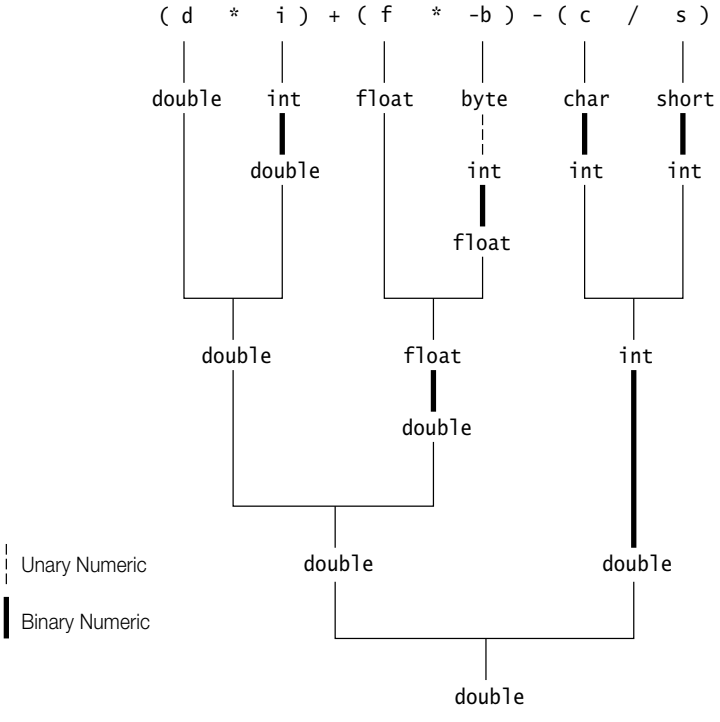
Numeric Promotions in Arithmetic Expressions

Unary numeric promotion is applied to the single operand of the unary arithmetic operators - and +. When a unary arithmetic operator is applied to an operand whose type is narrower than `int`, the operand is promoted to a value of type `int`, with the operation resulting in an `int` value. If the conditions for implicit narrowing conversion are not fulfilled (p. 160), assigning the `int` result to a variable of a narrower type will require a cast. This is demonstrated by the following example, where the `byte` operand `b` is promoted to an `int` in the expression `(-b)`:

```
byte b = 3;           // int literal in range. Narrowing conversion.
b = (byte) -b;        // Cast required on assignment.
```

Binary numeric promotion is applied to operands of binary arithmetic operators. Its application leads to type promotion for the operands, as explained in §5.2, p. 149. The result is of the promoted type, which is always type `int` or wider. For the expression at (1) in Example 5.2, numeric promotions proceed as shown in Figure 5.4. Note the integer division performed in evaluating the subexpression `(c / s)`.

Figure 5.4 *Numeric Promotion in Arithmetic Expressions*



Example 5.2 *Numeric Promotion in Arithmetic Expressions*

```
public class NumPromotion {
  public static void main(String[] args) {
    byte   b = 32;
    char   c = 'z';                // Unicode value 122 (\u007a)
    short  s = 256;
    int    i = 10000;
    float  f = 3.5F;
    double d = 0.5;
    double v = (d * i) + (f * -b) - (c / s);     // (1) 4888.0D
    System.out.println("Value of v: " + v);
  }
}
```

Output from the program:

```
Value of v: 4888.0
```

In addition to the binary numeric promotions in arithmetic expression evaluation, the resulting value can undergo an implicit widening conversion if assigned to a variable. In the first two declaration statements that follow, only assignment conversions take place. Numeric promotions take place in the evaluation of the right-hand expression in the other declaration statements.

```
Byte   b = 10;       // Constant in range: narrowing and boxing on assignment.
Short  s = 20;       // Constant in range: narrowing and boxing on assignment.
char   c = 'z';      // 122 (\u007a)
int    i = s * b;    // Values in s and b promoted to int: unboxing, widening.
long   n = 20L + s;  // Value in s promoted to long: unboxing, widening.
float  r = s + c;    // Value in s is unboxed. This short value and the char
                     // value in c are promoted to int, followed by implicit
                     // widening conversion of int to float on assignment.
double d = r + i;    // Value in i promoted to float, followed by implicit
                     // widening conversion of float to double on assignment.
```

Binary numeric promotion for operands of binary operators implies that each operand of a binary operator is promoted to type `int` or a broader numeric type, if necessary. As with unary operators, care must be exercised in assigning the value resulting from applying a binary operator to operands of these types.

```
short h = 40;          // OK: int converted to short. Implicit narrowing.
h = h + 2;             // Error: cannot assign an int to short.
```

The value of the expression `h + 2` is of type `int`. Although the result of the expression is in the range of `short`, this cannot be determined at compile time. The assignment requires a cast.

```
h = (short) (h + 2);   // OK
```

Notice that applying the cast operator `(short)` to the individual operands does not work:

```
h = (short) h + (short) 2;     // The resulting value should be cast.
```