|  |  |
| --- | --- |
| Infinity | The result when a number greater than 0.0 is divided (using /) by 0.0. |
| -Infinity | The result when a number less than 0.0 is divided (using /) by 0.0. |
| NaN | The result when 0.0 is divided (using /) by 0.0, and when any number is divided (using %) by 0.0. (NaN is short for *Not a Number*.) |

**(addition)**

The addition operator, +, adds the values of the two operands together. That is, when a and b are doubles, a + b evaluates to the double that is the sum of the values of a and b. (Some loss of precision is possible.) For example,

|  |  |  |
| --- | --- | --- |
| a | b | a + b |
| 1.2 | 8.9 | 10.1 |
| 4.3 | -9.3 | -5.000000000000001 |

**- (subtraction)**

With two operands, the subtraction operator, -, subtracts the value of the second from the value of the first. That is, when a and b are doubles, a - b evaluates to the double that is the result of subtracting the value of b from the value of a. (Some loss of precision is possible.) For example,

|  |  |  |
| --- | --- | --- |
| a | b | a - b |
| 1.2 | 8.9 | -7.7 |
| 4.3 | -9.3 | 13.600000000000001 |

With just one operand, the subtraction operator subtracts the value of that operand from 0.0.

**\* (multiplication)**

The multiplication operator, \*, multiplies the values of the two operands together. That is, when a and b are doubles, a \* b evaluates to the double that is the product of the values of a and b. (Some loss of precision is possible.) For example,

|  |  |  |
| --- | --- | --- |
| a | b | a \* b |
| 8.9 | 1.2 | 10.68 |
| -9.3 | 4.32 | -40.17600000000001 |

**/ (division)**

The division operator, /, divides the value of the first operand by the value of the second operand and, provided the second operand is not zero, returns the double that, when multiplied by the second operand's value, is equal (within the precision possible when calculating with doubles) to the first operand's value. More specifically, when a and b are doubles,

* if b has a non-zero value, then a / b evaluates to the double that, when multiplied by the value of b, yields the value of a (or very close to it).
* if b has the value 0.0 or -0.0 and
  + a has the value 0.0 or -0.0, then a / b evaluates to the special floating point value NaN;
  + a has a non-zero value then, provided only one of a and b has a minus sign, a / b evaluates to the special floating point value -Infinity; otherwise, it evaluates to the special floating point value Infinity.

For example,

|  |  |  |
| --- | --- | --- |
| a | b | a / b |
| 8.9 | 1.2 | 7.416666666666667 |
| -9.3 | 4.32 | -2.1527777777777777 |
| 8.9 | 0.0 | Infinity |
| 8.9 | -0.0 | -Infinity |
| 0.0 | 0.0 | NaN |
| -9.3 | 0.0 | -Infinity |
| -9.3 | -0.0 | Infinity |

**% (modulus)**

The modulus operator, %, divides the value of the first operand by the value of the second operand and, provided the second operand is not zero, returns the remainder. More specifically, when a and b are doubles,

* if b has a non-zero value, then a % b evaluates to the double that is determined as follows:

a - (b \* (double)(int)(a / b))

* if b has the value 0.0 or -0.0, then a % b evaluates to the special floating point value NaN.

The effect of the 2-fold cast (double)(int)(a / b) is to replace the decimal tail of a / b by the tail .0. (Of course, some loss of precision is possible.) For example,

|  |  |  |
| --- | --- | --- |
| a | b | a % b |
| 8.9 | 1.2 | 0.5000000000000007 |
| -9.3 | 3.42 | -2.460000000000001 |
| 8.9 | 0.0 | NaN |
| 0.0 | -0.0 | NaN |
| -9.3 | 0.0 | NaN |

**Calculating with special floating point values**

When working with ints, if you divide by 0 (using either / or %), the computation is halted (because an *ArithmeticException* is thrown) and a */ by zero* error is generated.

In contrast, when working with doubles, division by 0.0 or -0.0 yields one of the special values Infinity, -Infinity, and NaN and the computation continues without provoking an error. The special value -0.0 can be entered as a literal into a white edit box in any of our code fragments. This is not true, however, for the other three special values (Infinity, -Infinity, NaN). These will only feature in your work as values of expressions or sub-expressions. Nevertheless, the following table illustrates many of the interesting things that can happen if a calculation continues after one or more of these values has entered the picture:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| a | b | a + b | a - b | a \* b | a / b | a % b |
| Infinity | 1.0 | Infinity | Infinity | Infinity | Infinity | NaN |
| Infinity | 0.0 | Infinity | Infinity | NaN | Infinity | NaN |
| Infinity | -0.0 | Infinity | Infinity | NaN | -Infinity | NaN |
| Infinity | Infinity | Infinity | NaN | Infinity | NaN | NaN |
| Infinity | -Infinity | NaN | Infinity | -Infinity | NaN | NaN |
| Infinity | NaN | NaN | NaN | NaN | NaN | NaN |
| 1.0 | 0.0 | 1.0 | 1.0 | 0.0 | Infinity | NaN |
| 1.0 | -0.0 | 1.0 | 1.0 | -0.0 | -Infinity | NaN |
| 1.0 | Infinity | Infinity | -Infinity | Infinity | 0.0 | 1.0 |
| 1.0 | -Infinity | -Infinity | Infinity | -Infinity | -0.0 | 1.0 |
| 1.0 | NaN | NaN | NaN | NaN | NaN | NaN |
| -0.0 | 1.0 | 1.0 | -1.0 | -0.0 | -0.0 | -0.0 |
| -0.0 | 0.0 | 0.0 | -0.0 | -0.0 | NaN | NaN |
| -0.0 | -0.0 | -0.0 | 0.0 | 0.0 | NaN | NaN |
| -0.0 | Infinity | Infinity | -Infinity | NaN | -0.0 | -0.0 |
| -0.0 | -Infinity | -Infinity | Infinity | NaN | 0.0 | -0.0 |
| -0.0 | NaN | NaN | NaN | NaN | NaN | NaN |
| NaN | 1.0 | NaN | NaN | NaN | NaN | NaN |
| NaN | 0.0 | NaN | NaN | NaN | NaN | NaN |
| NaN | -0.0 | NaN | NaN | NaN | NaN | NaN |
| NaN | Infinity | NaN | NaN | NaN | NaN | NaN |
| NaN | -Infinity | NaN | NaN | NaN | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

In fact, there is a built-in Java method for calculating the floor of a floating point number. For any double d, the value of Math.floor( d ) is the double that is both less than or equal to d and equal to a mathematical integer. So (int)(Math.floor( a )) is a solution to part (a) that is correct whether a is positive or negative. The Math.floor method is not in the Advanced Placement Java subset.

Enter an expression into the long white box in the code fragment below so that, whenever a is assigned a floating point value that is greater than zero, the expression evaluates to the integer that results from rounding that value.

1. A suitable expression is as follows:

(int)(a + 0.5)

1. No. The only negative floating point numbers that the expression in part (a) rounds correctly are those that are halfway between two adjacent integers and those that are strictly between -0.5 and 0.0.

As in Exercise 14, we will be able to produce a single program that rounds both positive and negative floating point numbers equally well once we have introduced conditional branching.

In fact, there is a built-in Java method for rounding a floating point number to the nearest integer. For any double d, the value of Math.rint( d ) is the double that is closest to d and is equal to a mathematical integer. So (int)(Math.rint( a )) is a solution to part (a) that is correct whether a is positive or negative. The Math.rint method is not in the Advanced Placement Java subset.

In fact, there is a built-in Java method for calculating the floor of a floating point number. For any double d, the value of Math.floor( d ) is the double that is both less than or equal to d and equal to a mathematical integer. So (int)(Math.floor( a )) is a solution to part (a) that is correct whether a is positive or negative. The Math.floor method is not in the Advanced Placement Java subset.