APPENDIX A: OPERATOR PRECEDENCE IN JAVA

Java has well-defined rules for evaluating expressions, including *operator precedence*, *operator associativity*, and *order of operand evalution*. We describe each of these three rules.

Operator precedence. Operator precedence specifies the manner in which operands are grouped with operators. For example, `1 + 2 * 3` is treated as `1 + (2 * 3)`, whereas `1 * 2 + 3` is treated as `(1 * 2) + 3` because the multiplication operator has a higher precedence than the addition operator. You can use parentheses to override the default operator precedence rules.

Operator associativity. When an expression has two operators with the same precedence, the operators and operands are grouped according to their *associativity*. For example `72 / 2 / 3` is treated as `(72 / 2) / 3` since the division operator is left-to-right associate. You can use parentheses to override the default operator associativity rules.

Most Java operators are left-to-right associative. One notable exception is the assigment operator, which is right-to-left associative. As a result, the expression `x = y = z = 17` is treated as `x = (y = (z = 17))`, leaving all three variables with the value 17. Recall that an assignment statement evaluates to the value of its right-hand side. Associativity it not relevant for some operators. For example, `x <= y <= z` and `x++--` are invalid expressions in Java.

Precedence and associativity of Java operators. The table below shows all Java operators from highest to lowest precedence, along with their associativity. Most programmers do not memorize them all, and, even those that do, use parentheses for clarity.

Level	Operator	Description	Associativity

16	`()`
	`[]`
	`.`
		parentheses
		array access
		member access
			left-to-right

15	`++` `--`	unary post-increment unary post-decrement	left-to-right

14	`+`
	`-`
	`!`
	`~`
	`++`
	`--`
		unary plus
		unary minus
		unary logical NOT
		unary bitwise NOT
		unary pre-increment
		unary pre-decrement
			right-to-left

13	`()` `new`	cast object creation	right-to-left
12	`* / %`	multiplicative	left-to-right
11	`+ -` `+`	additive string concatenation	left-to-right
10	`<< >>` `>>>`	shift	left-to-right

9	`< <=`
	`> >=`
	`instanceof`
		relational	left-to-right

8	`==` `!=`	equality	left-to-right
7	`&`	bitwise AND	left-to-right
6	`^`	bitwise XOR	left-to-right
5	`|`	bitwise OR	left-to-right
4	`&&`	logical AND	left-to-right
3	`||`	logical OR	left-to-right
2	`?:`	ternary	right-to-left

1	`=  +=  -=`
	`*=  /=  %=`
	`&=  ^=  |=`
	`<<=  >>=  >>>=`
		assignment	right-to-left

0	`->`	lambda expression arrow	right-to-left

You'll find different (and usually equivalent) operator precedence tables on the web and in textbooks. They typically disagree in inconsequential ways because some operators cannot share operands, so their relative precedence order does not matter (e.g., `new` and `!`). There is no explicit operator precedence table in the Java Language Specification. Instead, the operator precedence and associativity rules are inferred via the grammar that defines the Java language.

Order of operand evaluation in Java. Associativity and precedence determine in which order Java groups operands and operators, but it does *not* determine in which order the operands are *evaluated*. In Java, the operands of an operator are always evaluated left-to-right. Similarly, argument lists are always evaluated left-to-right. So, for example in the expression `A() + B() * C(D(), E())`, the subexpressions are evaluated in the order `A()`, `B()`, `D()`, `E()`, and `C()`. Although, `C()` appears to the left of both `D()` and `E()`, we need the results of both `D()` and `E()` to evaluate `C()`. It is considered poor style to write code that relies upon this behavior (and different programming languages may use different rules).

Short-circuit evaluation. With three exceptions (`&&`, `||`, and `?:`), Java evaluates every operand of an operator before the operation is performed. For the logical AND (`&&`) and logical OR (`||`) operators, Java evaluate the second operand only if it is necessary to resolve the result. This is known as *short-circuit evaluation*. It allows statements like `if ((s != null) && (s.length() < 10))` to work reliably (i.e., invoke the `length()` method only if `s` is not `null`). Programmers rarely use the non short-circuiting versions (`&` and `|`) with `boolean` expressions.

Operator precedence gone awry. Sometimes the precedence order defined in a language do not conform with mathematical norms. For example, in Microsoft Excel, `-a^b` is treated as `(-a)^b` instead of `-(a^b)`. So `-3^2` evaluates to `9` instead of `-9`, which is the value that most mathematicians would expect. Microsoft acknowledges this quirk as a "design choice." One wonders whether the programmer was relying on the C precedence order in which unary operators have higher precedence than binary operators. This rule agrees with mathematical conventions for all C operators, but fails with the addition of the exponentiation operator. Once the order was established in Microsoft Excel 2.0, it could not easily be changed without breaking backward compatibility.

Operator associativity gone awry. Sometimes the associativity of an operator is implemented left-to-right in one programming language but right-to-left in another. An alarming example is exponentiation. In Wolfram Alpha and Google Sheets, the exponentiation operator is right-to-left associative, so `2 ^ 2 ^ 3` is treated as `2 ^ (2 ^ 3)`, which is 256. However, in Matlab and Excel, the exponentiation operator is left-to-right associative, so `2 ^ 2 ^ 3` is treated as as `(2 ^ 2) ^ 3`, which is 64. Exponentiation is not a binary operator in Java.

Exercises.

1. What is the result of the following code fragment? Explain.

```
System.out.println("1 + 2 = " + 1 + 2);
System.out.println("1 + 2 = " + (1 + 2));
```

   Answer: `1 + 2 = 12` and `1 + 2 = 3`, respectively. If either (or both) of the operands of the `+` operator is a string, the other is automatically cast to a string. String concatenation and addition have the same precedence and are left-to-right associative. The parentheses in the second statement ensures that the second `+` operator performs addition instead of string concatenation.

2. What does the following code fragment print?

```
System.out.println(1 + 2 + "abc");
System.out.println("abc" + 1 + 2);
```

   Answer: `3abc` and `abc12`, respectively. The `+` operator is left-to-right associative, whether it is string concatenation or addition.

3. Add parentheses to the following expression to make it more clear to other programmers.

```
year % 4 == 0 && year % 100 != 0 || year % 400 == 0
```

   Answer: LeapYear.java shows a variety of equivalent expressions, including the following reasonable alternative:

```
((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)
```

4. What is the value of the expression `1 <= 2 <= 3`? Explain.

   Answer: It leads to a compile-time error. The compiler parses the expression `1 <= 2 <= 3` as `(1 <= 2) <= 3`. This leads to a compile-time error because you can't use the `<=` operator to compare a `boolean` to an `int`. So, while the relational operators have left-to-right associativity, this property is not useful.

5. What is the result of the following code fragment? Explain.

```
boolean a = false;
boolean b = false;
boolean c = true;
System.out.println(a == b == c);
```

   Answer: It prints `true`. The equality operator is left-to-right associative, so `a == b` evaluates to `true` and this result is compared to `c`, whihc yields `true`.

6. What is the value of the expression `+--17`? Explain.

   Hint: The unary operators are right-to-left associative.

7. What is the value of the expression `--17`? Explain.

   Answer: It leads to a compile-time error because Java parses `--` as the pre-decrement operator (and not two unary minus operators).

8. What is the value of `z` after executing the following code fragment? Explain.

```
int x = 5;
int y = 10;
int z = ++x * y--;
```

9. What is the value of `y` after executing the following code fragment? Explain.

```
int x = 10;
int y = x+++x;
```

10. What is the value of `x` after executing the following code fragment? Explain.

```
int x = 10;
++++x;
```

    Answer: It leads to a compile-time error. Java parses it as `++(++x)`. The `++x` term increments the variable `x` and evaulates to the integer `11`. This leads to a compile-time error because you the pre-increment operator must be applied to a variable (not an integer). So, while the pre-increment operator has right-to-left associativity, this property is not useful.

Last modified on March 10, 2022.