**2.01: Order of Operations**

* 15 + 2 \* (5 + 9 / 2) + 7.5 / 3
  + Would be 36.5 in Algebra; 35.5 in Java because of consideration of integers and floats

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator** | **Operation** | **From** |
| 1 | +, - | Unary (single) plus; unary minus | Right to left |
| 2 | \*, /, % | Multiplication; division; modulus | Left to right |
| 3 | +, -, + | Addition; subtraction; concatenation (adds two strings into one) | Left to right |
| 4 | = | Assignment | Right to left |

* Example:
  + 6+15/3+23+7+3=
    - 6 + (15/3) + (23) + 10=
    - 6 + 5 + 8 + 10=
    - 29

**2.02: Printing Arithmetic Operations**

* No notes

**2.03: Primitive Data Types**

* Two types of variables in Java: primitive data types (defined by a programming language) and reference variables (data type whose instances are not stored directly in that type)
* int sum = 15 + 23 + 45;
  + The variable **sum** is stored as an **integer** as the **sum of 15, 23, and 45**
* Modulus:
  + Divide 17 by 3, you get 5 with a remainder of 2
  + Modulus looks for the remainder: 17/3 = 5; 17 % 3 = 2
* public static void main(String[ ] args)  
  {  
        // Declare integer variables  
        int iNum1 = 25;  
        int iNum2 = 9;  
        // Addition   
        System.out.println("Addition");  
        System.out.println(iNum1 + " + " + iNum2 + " = " + (iNum1 + iNum2));  
        System.out.println(43.21 + " + " + 3.14+ " = " + (43.21 + 3.14));  
        System.out.println();

**2.04: Primitive Data Types – Doubles**

* [**int**](javascript:void(0);) is used for integer values (e.g., -3, 52, 365, etc.)
* [**double**](javascript:void(0);) is used for decimal values (e.g., 3.141592654, -273.15, etc.)
* [**char**](javascript:void(0);) handles single keyboard characters (e.g., A, d, \*)
* [**boolean**](javascript:void(0);) is restricted to logical values (e.g., true or false)

**02.06: Primitive Data Type Conversions**

| **Part 1 Primitive Data Type** | | |
| --- | --- | --- |
| **Primitive Data Type** | **Size (bytes)** | **Range** |
| boolean | 1 | true or false |
| char | 2 | Unicode character set |
| byte | 1 | -27 to 27-1 (-128 to 127) |
| short | 2 | -215 to 215-1 (-32768 to 32767) |
| int | 4 | -231 to 231-1 (-2147483648 to 2147483647) |
| long | 8 | -263 to 263-1 |
| float | 4 | -3.4 x1038 to 3.4 x 1038 |
| double | 8 | -1.8 x 10308 to 1.8 x 10308 |

* **Assignment Conversion** occurs when a value with less precision is assigned to a variable with greater precision. For example:
  + double dNum = 100;
    - Becomes 100.0 in a widening conversion
* Not possible with integers (cannot convert great precision to little precision with int. EX:
  + int iNum = 3.141592654
    - Compiler will throw a possible loss of precision error
* **Arithmetic Promotion** also happens automatically in an expression of mixed types. For example:
  + double x = 3.14 \* 10 \* 10;
    - Output will be a double; the 3.14 in the statement promotes the integers 10 into a double; 10.0.
* **Casting** allows promotion and demotion of types, but it must be done explicitly. For example:
  + int x = (int)3.14 \* 10 \* 10;
    - The (int) statement converts 3.14 into 3 and then multiplies by 10 twice, returning an integer response.
      * Only treats it as 3 in this case; does not change anything in memory
    - This cast is a **narrowing conversion**
      * Should be avoided; causes a loss of information

**02.07: Pitfalls, Surprises, and Shortcuts**