Computing values in Java works much like you'd expect from most other languages:

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class ExpressionOperand {
    public static void main(final String[] args) {
        final int x = (3 * 6) + 24;
        System.out.println(x);
    }
}
```

There are two things to be aware of:

1. Like in C or C++, but unlike Javascript or Python3, division of two integers yields an integer.
2. Operators given two different types (like `2 + 1.7`) promote, or automatically convert, the more limited type. `2` will be converted to the floating point `2.0`, and the resulting value will be a floating point that must be stored in a variable of type `float` or `double`.

Integer and floating point division

There are two types of division, and each is sometimes useful. **Integer division** takes two *integers* and evaluates to an integer. **Floating-point** division takes two *floating-point* numbers (numbers with a decimal point) and evaluates to a floating-point number.

Let's say I have 18 cents and 5 nieces. How much money should I give to each? With floating point division, 18/5 = 3.6, but it's hard to distribute .6 cents in cash, and my nieces prefer hard currency. Integer division would give the value 3. If both operands of `/` are integers, Java uses integer division. If either operand is a floating point value, then Java uses floating point division.

You can ensure floating point division by adding a decimal point and a zero:

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class DivisionTypes {

    public static void main(final String[] args) {
        System.out.println(18 / 5);
        System.out.println(18.0 / 5.0);
    }
}
```

If you want the remainder that would be left over after integer division, you can use the **modulus operator** `%`:

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class Modulus {

    public static void main(final String[] args) {
        // Three cents left over after distributing
        // three pennies to each of my five nieces:
        System.out.println(18 % 5);
    }
}
```

Typecasting operators

Sometimes, you need to force Java to convert one type of data to another. For example, if you have two `int` variables and would like to divide them to get a fractional number, you might convert those values into `double` values. You can do this by typecasting the values, or **casting**, for short. A cast operator is written using parentheses and the name of the target type, and precedes the value to be cast. For example, `(double) 5` yields the value `5.0`.

For now, we'll mostly cast between numerical types. Unlike in Javascript or Python, you cannot cast an `int` or `double` to or from a string, but must rather use special methods of the `String`, `Integer`, or `Double` class.

Implicit vs. explicit casting

Sometimes, Java does casting for you automatically. For example, if you write `18.0 / 5`, then Java will decide to cast `5` into a floating point `5.0`, an implicit cast. Another example is `double x = 5`. The `5` on the right is an integer data type, but Java knows to cast it into a double.

On the other hand, `int x = 5.6` will not work. Java will notice that you are trying to put a floating point into an `int`, and warn you that this may entail a loss of precision. You can reassure Java by using a casting operator: `int x = (int) 5.6`. Notice that when casting to an integer, the part after the decimal is truncated, not rounded, so this expression will compute the value 5, not 6.

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class Cast {

    public static void main(final String[] args) {
        int numerator = 18;
        int denominator = 5;
        System.out.println((double) numerator / denominator);
    }
}
```

Combined assignment and arithmetic operators

Like many other languages, Java permits arithmetic and assignment to be combined, with the operators `+=`, `-=`, `/=`, `*=`. Thus, `x += 4` would increase the value of `x` by `4`.

Increment operators `--` and `++`

Java, Javascript, C, and C++ all have increment operators that work the same way; Python does not. The `++` operator adds one to a value, `--` subtracts. These operators are very useful in Java `for` and `while` loops, which we will see soon.

These operators are most cleanly used like the combined assignment operators in Python: you can replace the line of code `x += 1` with `x++`.

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class Increment {

    public static void main(final String[] args) {
        int x;

        x = 5;
        x += 1;
        System.out.println("x = " + x);

        x++;
        System.out.println("x = " + x);
    }
}
```

Variable assignments like `x = 4` change the value of a variable. In C and Java, these assignments also produce a value that can be used. `x = 4` produces the value `4`. So `y = (x = 4)` would set y to have the value `4`. (The parentheses are not needed, but clarify what is happening.)

Similarly, `x++` produces a value: the value of `x` before adding `1` to `x`. This is called the **post-increment** operator, since the value is produced, and then the variable is incremented.

On the other hand, `++x` increments the variable first, and then produces the value: the **pre-increment operator**.

Overly-clever use of `++` and `--` operators are a common source of hard-to-find errors. I almost always write `x++` on its own line of code, and never use `++x`.

Just because you can write hard-to-read code like `x += ( y-- - (x = 4) )` doesn't mean you should.

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class Preincrement {

    public static void main(final String[] args) {
        int x = 5;
        int y = 5;

        // The ++ operator changes the value
        //  of the variable, but you can also
        //  use the result in an expression:

        System.out.println(x++);  // post-increment
        System.out.println(++y);  // pre-increment

        // assignment statements are also expressions.
        //  However, the code below is bad: it's too likely
        //  to be a typo; probably you wanted ==, the
        //  the equality comparison.
        System.out.println(x = y);

        // Truly horrible programming style:
        System.out.println(x += (y-- - (x = 4) ));
    }
}
```