

The Math Class

It is hard to avoid the `Math` class in any Java program that requires scientific or other numeric computations. As with the wrapper classes, the `Math` class is part of the `java.lang` package; so, methods may be used without an explicit `import` statement. Table 1 summarizes the most common methods in the `Math` class. For a full listing, see the Java API documentation. Unlike the wrapper classes, the `Math` class contains no constructor method: `Math` objects cannot be instantiated. Therefore, all methods in Table 1 are class methods (aka static methods).

Table 1. Methods in the `Math` Class

Method	Purpose
<code>abs(double a)</code>	returns a <code>double</code> equal to the absolute value of a <code>double</code> value <code>a</code>
<code>asin(double a)</code>	returns a <code>double</code> equal to the arc sine of an angle <code>a</code> , in the range of $-\pi/2$ through $\pi/2$
<code>exp(double a)</code>	returns a <code>double</code> equal to the exponential number <code>e</code> (i.e., 2.718...) raised to the power of a <code>double</code> value <code>a</code>
<code>log(double a)</code>	returns a <code>double</code> equal to the natural logarithm (base <code>e</code>) of a <code>double</code> value <code>a</code>
<code>pow(double a, double b)</code>	returns a <code>double</code> equal to of value of the first argument <code>a</code> raised to the power of the second argument <code>b</code>
<code>random()</code>	returns a <code>double</code> equal to a random number greater than or equal to 0.0 and less than 1.0
<code>rint(double a)</code>	returns a <code>double</code> equal to the closest <code>long</code> to the argument <code>a</code>
<code>round(double a)</code>	returns a <code>long</code> equal to the closest <code>int</code> to the argument <code>a</code>
<code>sin(double a)</code>	returns a <code>double</code> equal to the trigonometric sine of an angle <code>a</code>
<code>sqrt(double a)</code>	returns a <code>double</code> equal to the square root of a <code>double</code> value <code>a</code>
<code>tan(double a)</code>	returns a <code>double</code> equal to the trigonometric tangent of an angle <code>a</code>
<code>toDegrees(double a)</code>	returns a <code>double</code> equal to an angle measured in degrees equal to the equivalent angle measured in radians <code>a</code>
<code>toRadians(double a)</code>	returns a <code>double</code> equal to an angle measured in radians equal to the equivalent angle measured in degrees <code>a</code>

As well as methods, class definitions often include publicly-accessible variables or constants as "fields". The `Math` class includes two such fields: `E` and `PI`, representing common mathematical constants (see Table 2).

Table 2. Data Fields in the `Math` Class

Constant	Type	Purpose
<code>E</code>	<code>double</code>	holds the value of <code>e</code> , the base of natural logarithms,

		accurate to about 15 digits of precision
PI	double	holds the value of pi, the ratio of the circumference of a circle to its radius, accurate to about 15 digits or precision

To use these constants in an expression, they must be prefixed with "Math." (e.g., Math.PI) to inform the compiler of the whereabouts of their declarations. Note the use of uppercase characters for the identifiers. This is consistent with the naming conventions for constants, as given earlier.

Let's demonstrate a few of the methods in the Math class through demo programs.

Example: Equal-Tempered Musical Scale

The harmonic system used in most Western music is based on the "equal-tempered scale", as typified by the pattern of white and black keys on a piano. Each note on a piano keyboard is related in frequency (f) to its neighbor by the following formula:

$$f_{n+1} = f_n \times 2^{1/12}$$

where note $n + 1$ is "one semitone" above note n . The factor $2^{1/12}$ ensures that two notes separated by twelve steps in the equal-tempered scale differ in frequency by a factor of $2^{12/12} = 2$. This interval is known as an "octave". To allow musicians to travel and perform with different orchestras in different countries, a standard evolved and was adopted in the 1950s to ensure instruments were in tune with each other. The standard specifies that "A above middle C" should have a frequency of 440 Hz (see Figure 1). With this reference point, and with the $2^{1/12}$ frequency factor between adjacent notes, the frequency of any note on any instrument was standardized.

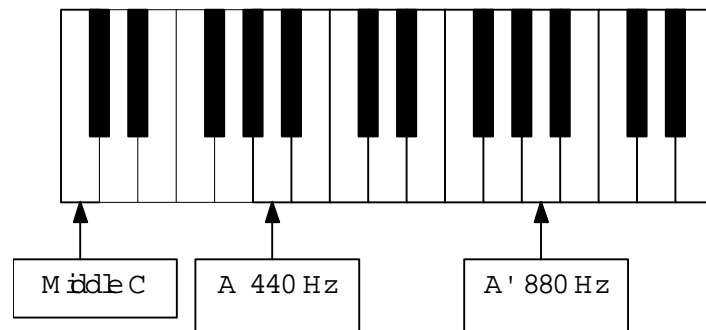


Figure 1. Notes and frequencies in the equal-tempered scale

As seen in Figure 1, the frequency of A' — one octave above A — is $440 \times 2^{12/12} = 880$ Hz. But what about the notes in between? Their frequencies are computed in the demo program *Frequencies* (see Figure 2).

```

1 public class Frequencies
2 {
3     public static void main(String[] args)
4     {
5         for (int i = 0; i < NOTES.length; ++i)
6         {
7             System.out.print(NOTES[i] + " ");
8             System.out.println(A440 * Math.pow(2, i / 12.0));
9         }
10    }
11    }
12    public static final double A440 = 440.0;
13    public static final String[] NOTES =
14        { "A ", "A#", "B ", "C ", "C#", "D ",
15          "D#", "E ", "F ", "F#", "G ", "G#", "A'" };
16 }

```

Figure 2. Frequencies.java

This program generates the following output:

```

A    440.0
A#   466.1637615180899
B    493.8833012561241
C    523.2511306011972
C#   554.3652619537442
D    587.3295358348151
D#   622.2539674441618
E    659.2551138257398
F    698.4564628660078
F#   739.9888454232688
G    783.9908719634985
G#   830.6093951598903
A'   880.0

```

The frequencies in the table above are computed iteratively for 13 notes beginning at 440 Hz (see line 8).¹ The calculation uses the `pow()` method in the `Math` class. Two arguments are required. The first is the base and the second is the power. The base is 2 in each case, but the power is increased each time through the loop to compute the frequency of the next note. Note the use of "12.0" in line 8. The ".0" is important, as it ensures the `int` variable `i` is promoted to a `double`.

In musical terms, the most consonant harmony is formed by playing two notes where the higher note is 1.5 times the frequency of the lower note. This interval is known as a musical "fifth" and it is equivalent to seven semitones in the equal-tempered scale. A fifth above A is E. Note in the table above that the frequency of E is 659.255 Hz. This is close to $1.5 \times 440 = 660$ Hz. The slight discrepancy is not due to a rounding error. In fact, the equal-tempered scale is a compromise over a system where notes are related by natural harmonic intervals. The difference between 659.255 Hz and 660 Hz is small, but important. One reason pianos are so difficult to tune is that each note must conform to the equal-tempered scale, so slight discrepancies from natural harmonic intervals must be tuned-in.

¹ The names of the notes are stored as an array of strings. Arrays are discussed in Chapter 6.

More information on the equal-tempered scale and other systems of tuning, see Backus' *The Acoustical Foundations of Music* [2].

Example: Calculating the Height of a Building

Our next example of methods in the `Math` class shows how to a surveyor might calculate the height of a building or other object given two measurements. One measurement is the surveyor's distance from the building, the other is the line-of-sight angle between the ground (which we assume is flat) and the roof of the building (see Figure 3).

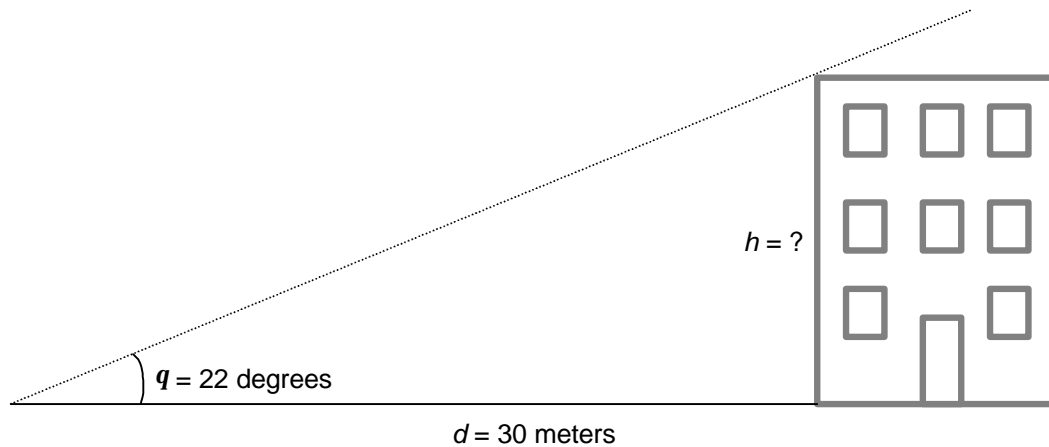


Figure 3. Calculating the height of a building

Figure 4 reviews the geometry. Clearly, x is the height of the building, y is the surveyor's distance from the building, and q is the line-of-sight angle between the ground and the roof. Our task is to compute $x = y \tan(q)$.

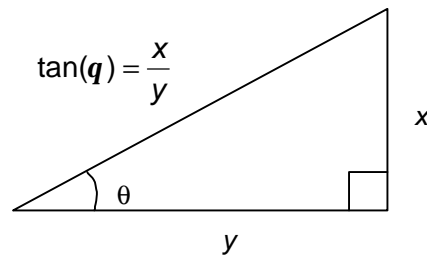


Figure 4. Review of tangent function

The demo program `Height` prompts the user to enter a distance in meters and a line-of-sight angle in degrees. From these, the height of the building is calculated.

```

1  import java.io.*;
2
3  public class Height
4  {
5      public static void main(String[] args) throws IOException
6      {
7          BufferedReader stdin =
8              new BufferedReader(new InputStreamReader(System.in), 1);
9
10         // Input distance to building
11         System.out.print("Enter distance to building: ");
12         double distance = Double.parseDouble(stdin.readLine());
13
14         // Input line-of-sight angle to top of building
15         System.out.print("Enter line-of-sight angle (degrees) to roof: ");
16         double angle = Double.parseDouble(stdin.readLine());
17
18         // Calculate height of building
19         double aRadians = Math.toRadians(angle);
20         double height = distance * Math.tan(aRadians);
21
22         // print height
23         System.out.println("Building is " + height + " meters high");
24     }
25 }

```

Figure 5. Height.java

A sample dialogue with this program follows: (User input is underlined.)

```

PROMPT> java Height
Enter distance (meters) to building: 30
Enter line-of-sight angle (degrees) to roof: 22
Building is 12.120786775054704 meters high

```

Two methods in the Math class are used. The `tan()` method computes the tangent of an angle passed to it as a double argument (line 20). However, the `tan()` method requires an angle in radians as an argument. The `toRadians()` method converts the inputted angle from degrees to radians (see line 19).

Example: Charging a Capacitor

In electronics, a well-known phenomenon is the charging characteristic of a capacitor. A representative circuit is shown in Figure 6. Capacitor C is in series with resistor R and a voltage V_i is applied to the RC pair through a switch.

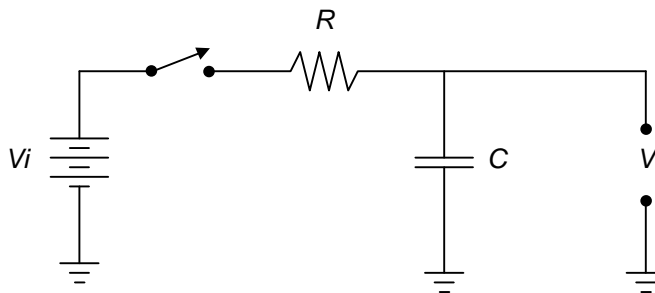


Figure 6. Charging a capacitor

Assuming an initial discharged state, the voltage across the capacitor, V , rises in time after the switch is closed according to the following formula:

$$V = V_i(1 - e^{-t/RC})$$

Let's examine a program to calculate V at several points in time after the switch is closed. To do this, we'll input sample values for C , R , and V_i .

The trickiest part in the calculation is raising the natural logarithm constant e to the power $-t/RC$. Java's `Math` class includes a data field, `Math.E`, that defines e to about 15 digits of precision, so we could use the `pow()` method, as we did in the `Frequencies` demo program (see Figure 2). However, the `exp()` method is a better choice. It computes e^n , where n is a double argument passed to the method. The demo program `CapacitorCharging` shows the `exp()` method in action.

```
1  import java.io.*;
2
3  public class CapacitorCharging
4  {
5      public static void main(String[] args) throws IOException
6      {
7          BufferedReader stdin =
8              new BufferedReader(new InputStreamReader(System.in), 1);
9
10         // Input value of capacitor (micro Farads)
11         System.out.print("Capacitance (uF): ");
12         double c = Double.parseDouble(stdin.readLine()) * 1000000;
13
14         // Input value of resistor (in kilo Ohms)
15         System.out.print("Resistance (k): ");
16         double r = Double.parseDouble(stdin.readLine()) / 1000;
17
18         // Input value of input voltage (v)
19         System.out.print("Applied voltage (v): ");
20         double vi = Double.parseDouble(stdin.readLine());
21
22         System.out.println("\nt (us)\tvoltage (v)");
23         System.out.println("-----");
24         for (int t = 1; t <= 1000000; t *= 10)
25         {
26             double v = vi * (1 - Math.exp(-t / (r * c)));
27             System.out.println(t + "\t" + v);
28         }
29     }
30 }
```

Figure 7. `CapacitorCharging.java`

A sample dialogue with this program follows:

```

PROMPT>java CapacitorCharging
Capacitance (uF): 1.5
Resistance (k): 33
Applied voltage (v): 12

t (us)      voltage (v)
-----
1           2.4242179371114503E-4
10          0.0024239975677593506
100         0.024217953426646677
1000        0.23999191951895327
10000       2.1950589463112236
100000      10.408455975563244
1000000     11.999999979790509

```

In the sample dialogue, the user specified a 1.5 μF capacitor in series with a 33 $\text{k}\Omega$ resistor with an applied voltage of 12 V. The output shows the capacitor voltage after 1 μs , 10 μs , 100 μs , and so on, up to 1,000,000 μs , or 1 second. The values inputted are in the most common units for capacitors (μF , or 10^{-6} farads) and resistors ($\text{k}\Omega$ or 10^3 ohms). Within the program, these are converted to farads (line 12) and ohms (line 16).

For more examples of electronics circuits, see *The Art of Electronics* by Horowitz and Hill [3].