COP 3330 – Object Oriented Programming in Java
Use of Build In Class: Math Random

Math Class

Every programming language has built in functions (called methods in Java) that help the programmer with various tasks. One of the most common collection of such functions is mathematical functions. To use these in C, you include math.h and to use these in Python you import math. In Java, where each collection of methods resides in a class, Math is naturally included. It's part of the package called java.lang, so if you did have to import it you would write:

```
import java.lang.Math;
```

but naturally, everything from the package java.lang is imported.

The most important thing to learn how to do in Java is to learn how to read Java Docs. These are the pages that provide documentation on each Java class. Here is a link to the documentation for the Math class:

https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

Most classes have instance methods, methods that belong to the object. But, the Math class does not have any instance methods. Instead, it is a collection of static methods because there is no "Math object" per se.

Let's take a look at a typical method in the Math class, one that students usually learn early on:

Modifier and Type	Method	Description
`static double`	`sqrt(double a)`	Returns the correctly rounded positive square root of a double value.

This method returns a double and is static, meaning that it belongs to the Math class and not a Math object (which doesn't exist). It's name is sqrt. It takes in a single parameter that is a double, and it returns the square root of that parameter.

If you click on the method name, you get some further details:

Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is the same as the argument.

(Note: Both the table and the special cases are directly taken from the Java Doc link stated above on 1/19/2026.)

Notice that the special cases are precisely concerning actual parameters you should never give the function. (The function should only receive non-negative real numbers.) But, instead of causing a

run-time error if you don't adhere to these rules (what Java used to do), newer versions of Java handle these cases with the use of a constant named NaN. Nearly all operations with NaN result in another NaN. This at least makes sure that a program with this sort of issue doesn't crash.

To call a static method in a class (that is outside of the class that you are in), use the following syntax:

```
Classname.methodName(actual parameters)
```

In C, you don't put a classname to the left of a dot before calling a function. In Python, this is precisely how you call a function/method that is imported.

Of course, we follow the usual conventions for calling methods as is true in other languages:

1. Methods that are void (return nothing are typically called on a line by themselves.
2. Methods that return something (int, double, etc.) are typically called in a larger line where an expression of the return type would be valid. They can never be the left-hand side of an assignment statement, since that has to be a variable.

A second very common method in the Math class is the pow method:

Modifier and Type	Method	Description
`static double`	`pow(double a, double b)`	Returns the value of the first argument raise to the power of the second argument.

Below is a code snippet from a program that calculates both roots of a quadratic equation if they are real. Assume that a, b and c are declared as doubles and already store the appropriate values:

```
double discriminant = Math.pow(b,2) - 4*a*c;

if (discriminant < 0)
     System.out.println("Your quadratic has complex roots.");

else {
    double r1 = (-b + Math.sqrt(discriminant))/(2*a);
    double r2 = (-b - Math.sqrt(discriminant))/(2*a);

    System.out.println("The roots are " + r1 + " and " + r2 + ".");
}
```

We first calculate the discriminant to avoid calling the sqrt method with a negative number. Once we have this quantity stored in a variable and screen out the complex root case, we can use it to calculate both roots of the quadratic. Note that it would be valid to do the following, illustrating that actual parameters can be any expression of the appropriate type:

```
double r1 = (-b + Math.sqrt(b*b-4*a*c)/(2*a);
```

In general, the amount one uses the Math class depends precisely on how quantitative the applications one's code requires. Some applications never touch it while others use it extensively. Here are a few other methods that are commonly used from the Math class:

Modifier and Type	Method	Description
`static double`	`abs(double a)`	Returns the absolute value of a double value.
`static double`	`cbrt(double a)`	Returns the cube root of a double value.
`static double`	`ceil(double a)`	Returns the smallest double value that is greater than or equal to the argument and is equal to a mathematical integer.
`static double`	`cos(double a)`	Returns the trigonometric cosine of an angle. The angle must be in radians.
`static double`	`exp(double a)`	Returns Euler's number e raised to the power of a double value.
`static double`	`floor(double a)`	Returns the largest double value that is less than or equal to the argument and is equal to a mathematical integer.
`static double`	`log(double a)`	Returns the natural logarithm (base e) of a double value.
`static double`	`log10(double a)`	Returns the base 10 logarithm of a double value;
`static double`	`max(double a, double b)`	Returns the greater of two double values.
`static double`	`min(double a, double b)`	Returns the smaller of two double values.
`static double`	`random()`	Returns a double value with a positive sign greater or equal to 0.0 and less than 1.0
`static double`	`sin(double a)`	Returns the trigonometric sine of an angle. The angle must be in radians.
`static double`	`tan(double a)`	Returns the trigonometric tangent of an angle. The angle must be in radians.
`static double`	`toDegrees(double angRad)`	Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
`static double`	`toRadians(double angDeg)`	Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

Use these or other methods from the Math class as needed.

One note: The Math class provides one rather restrictive method to generate random numbers. I prefer not to use this method (as we'll see on the next page), but I've provided it here since many people do use it.

Random Class

The class in Java intended to be used to generate pseudorandom numbers is the Random Class. Unlike the Math class, there is a Random object. Thus, one must first create a Random object before being able to call any of the methods in the Random class, since all of the methods in the Random class are instance methods (belong to the object).

Here is a link to the Java Doc for the Random class:
https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

In order to create an object, there is a special method that must be called: a constructor. A constructor always has the same exact name as the class and must be called with the keyword new. Here is the Java Doc listing of the two constructors for the Random class:

Constructor	Description
`Random()`	Creates a new random number generator.
`Random(long seed)`	Creates a new random number generator using a single long seed.

We will typically use the first constructor and although one can create many objects of a single Class, for nearly all programs with random numbers, we will only create one Random object, but we'll call methods multiple times ON that same object. Here is a valid call to the first constructor:

```
Random rndObj = new Random();
```

After this method call, rndObj is a reference to a Random object and different instance methods can be called on this Random object.

The Random class doesn't have nearly as many methods as the Math class. Here is a listing of what we'll probably use most:

Modifier and Type	Method	Description
`boolean`	`nextBoolean()`	Returns the next pseudorandom, uniformly distributed boolean value from this random number generator's sequence.
`double`	`nextDouble()`	Returns the next pseudorandom, uniformly distributed double value between 0.0 and 1.0 from this random number generator's sequence.
`double`	`nextGaussian()`	Returns the next pseudorandom, Gaussian ("normally") distributed double value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence.
`int`	`nextInt(int bound)`	Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

In the notes we'll look at two separate programs that utilize random numbers. The first is the classic guessing game. The program will generate a random integer in between 1 and 100, inclusive and then the user will have to guess the integer. After each guess, the user will be told if their guess is too low or too high, until they get the number. Here is the portion of the code before the loop that generates the number:

```
Random r = new Random();
int num = r.nextInt(MAX) + 1;
```

From here, the code resembles the usual logic that is shown in this classic example.

In the other example, we simulate playing the game of Craps. In this game, the player rolls a pair of dice. If she rolls a sum of 7 or 11, she wins. If she rolls a sum of 2, 3 or 12, she loses. Otherwise, the game continues. Call the sum of her dice, "the point." She will continue rolling the pair of dice until she rolls a total of 7 or a total equal to the point. If she gets a sum of 7 first, she loses. If she gets a sum equal to the point first, she wins.

Although the sum of two dice rolls is always in between 2 and 12, it would be incorrect to simulate the roll of two dice this way:

```
int sum = r.nextInt(11) + 2;
```

Can you see why? (The reason is that this code generates each value in between 2 and 12 with roughly equal probability, but when you roll two standard six-sided dice, some totals are more likely than others. For example, the probability of rolling a sum of seven is 1/6 while the probability of rolling a sum of 2 is only 1/36.) Thus, the correct way to simulate the roll of two fair dice is as follows:

```
int die1 = r.nextInt(6) + 1;
int die2 = r.nextInt(6) + 1;
int sum = die1 + die2;
```

Of course, we could do this in a single line as follows:

```
int sum = r.nextInt(6) + r.nextInt(6) + 2;
```

If we don't need to print out the individual values on each die, this will suffice. Incidentally, what would be wrong with:

```
int sum = 2*r.nextInt(6) + 2;
```

Although this ***logically equivalent*** to the line before, it is not. Why?

The code for the Craps program is provided on the next page.

```
import java.util.*;

public class Craps {

    public static void main(String[] args) {

        Scanner stdin = new Scanner(System.in);
        Random r = new Random();

        int die1 = Math.abs(r.nextInt())%6 + 1;
        int die2 = Math.abs(r.nextInt())%6 + 1;
        int sum = die1+die2;

        System.out.println("You rolled a "+sum+" for your first roll.");
        System.out.println("Hit enter to continue.");
        String dummy = stdin.nextLine();

        if (sum == 7 || sum == 11) {
            System.out.println("You win!!!");
        }
        else if (sum == 2 || sum == 3 || sum == 12) {
            System.out.println("Sorry, you lose.");
        }
        else {

            System.out.println("The game will continue.");
            System.out.println("Your roll of "+sum+" is now your point.");
            System.out.println("If you roll that again before a 7, you win!");
            System.out.println("Hit enter to continue.");
            dummy = stdin.nextLine();

            int point = sum;

            do {

                die1 = Math.abs(r.nextInt())%6 + 1;
                die2 = Math.abs(r.nextInt())%6 + 1;
                sum = die1+die2;

                System.out.println("You rolled a "+sum+" for your this roll.");
                System.out.println("Hit enter to continue.");
                dummy = stdin.nextLine();

            } while (sum != 7 && sum != point);

            if (sum == point)
                System.out.println("You win, you hit your point!");
            else
                System.out.println("Sorry, you lose :(");

        } // end if
    } // end main
} // end class
```

Our initial roll is taken care of with the three lines originally shown. Then we screen out the winning and losing cases in two branches of an if statement. The else branch of that if statement takes care of the situation when the game continues. The point is stored in a single variable and then a do-while loop (since it must run at least once) ensues with new rolls of a pair of dice. This loop continues running until either 7 or the user's point is obtained.