COP 3330 – Object Oriented Programming in Java
Multi-Class and Multi-File Programs using Static Methods

MathFunctions class – Same File

Organizationally, it may make sense to put several methods (in our case static methods since we haven't yet learned how to write instance methods) in a separate class than the class that has our main method.

For this course, we'll put the public method of the class first in the file, and after that completes, we'll follow with the code for the other classes we need for the program, separately, **so no class will be inside any other class.**

Since only one class in a file can be public and the name of the file must match the name of the public class, our support classes will not be public. Instead, they will have a default visibility modifier, which amounts to omitting the word public. Let's take a look at the layout of the file PrimeCheck3.java, which contains 2 classes: PrimeCheck3 and MathFunctions. To prove that the MathFunctions class can have more methods than just isPrime, I've added several math related methods to this class:

```
// Returns n factorial, 0 <= n <= 12.
public static int factorial(int n);

// Returns the number of ways to choose k items out of n
// 0 <= n <= 12, 0 <= k <= n.
public static int combination(int n, int k);

// Returns the number of divisors of n.
public static int numDivisors(int n);

// Returns the sum of divisors of n.
public static long sumDivisors(int n);

// Returns true if and only if n is a perfect number.
// A perfect number has its sum of proper divisors equal to
// itself.
public static boolean isPerfect(int n);

// Returns base raised to the power exp. exp must be non-negative.
public static int intPow(int base, int exp);
```

Starting on the next page is the whole code for PrimeCheck3.java. Notice that the PrimeCheck3 class comes first (begins and ends) with only a main method. This is followed by the entirety of the MathFunctions class, which is not public. I've omitted most of the comments for the sake of brevity. The code posted online has all comments in it.

```
import java.util.*;

public class PrimeCheck3 {

      public static void main(String[] args) {

            Scanner stdin = new Scanner(System.in);
            System.out.println("What number do you want to check?");
            int n = stdin.nextInt();

            if (MathFunctions.isPrime(n))
                  System.out.println(n+" is a prime number.");
            else
                  System.out.println(n+" is NOT a prime number.");

      }
}

class MathFunctions {

      public static boolean isPrime(int n) {

            if (n<2) return false;

            for (int i=2; i*i<=n; i++)
                  if (n%i == 0)
                        return false;

            return true;
      }

      public static int factorial(int n) {
            int res = 1;
            for (int i=1; i<=n; i++)
                  res *= i;
            return res;
      }

      public static int combination(int n, int k) {
            return factorial(n)/factorial(k)/factorial(n-k);
      }

      public static int numDivisors(int n) {

            int res = 0;
            for (int i=1; i*i<=n; i++) {

                  if (n%i == 0) {
                        res++;
                        if (n/i > i) res++;
                  }
            }
            return res;
      }
```

```
	public static long sumDivisors(int n) {

		long res = 0;
		for (int i=1; i*i<=n; i++) {

			if (n%i == 0) {
				res += i;
				if (n/i > i) res += (n/i);
			}
		}
		return res;
	}

	public static boolean isPerfect(int n) {
		return sumDivisors(n) == 2*((long)n);
	}

	public static int intPow(int base, int exp) {
		int res = 1;
		for (int i=1; i<=exp; i++)
			res = res*base;
		return res;
	}
}
```

It's possible to have several support classes, one after another, in the same file as long as they each have default visibility. (The absence of the word public when defining the class.) In this example, we see that we've included methods in the support class that aren't used by the public class in the file. This is okay, but not necessary. Some people, when designing programs in this manner prefer to remove all of the unused methods with others leave them in. For our purposes, it won't matter too much if you have an extra unused method or two left in a class for an assignment you submit.

<u>MathFunctions class – Different File</u>
It eventually may get unwieldy to put all necessary classes in a single file. Though we probably won't do any programs that are so big that we can't handle all the code in a single file in this class (nothing we do will exceed 500 lines probably), real life projects could have hundreds of thousands of lines of code and many, many classes. Even projects much smaller than this would benefit greatly from being stored in different files.

For this lecture, we'll look at one simple method for writing a program that uses multiple files, with each file storing one class. Later in the semester, we'll look at how to create our own package with multiple classes. (For things like this, IDEs are very useful as they autogenerate much of the necessary structure.)

If we want to CALL these methods from a different file, we must call them using the classname dot methodname syntax. To make sure that the method can be seen, all files that make calls to MathFunctions methods must reside in the same directory as MathFunctions.

In our example, we'll create four files. For the example to work, **<u>all of the files MUST BE placed in the same directory.</u>** Here are the four files for this example which in our sample programs are stored in the directory, "MathExample-NoPackage":

1. MathFunctions.java
2. PrimeCheck4.java
3. TacoBell.java
4. PerfectNums.java

Notice that in the MathFunctions.java file, we just have the identical code to what was previously presented, but we can make the class public, since it's in its own file.

In the other three files, we just have a single class with a single method: main. PrimeCheck4.java is exactly the same as the contents the class PrimeCheck3, except for the name of the class. Let's take a look at TacoBell.java, where we solve a combinations with repetition problem using the MathFunctions library:

```
import java.util.*;

public class TacoBell {
      public static void main(String[] args) {
            Scanner stdin = new Scanner(System.in);
            System.out.println("How many total items do you want to buy?");
            int n = stdin.nextInt();
            System.out.println("How many different items?");
            int r = stdin.nextInt();

            System.out.println("You can make "+MathFunctions.combination(n+r-
1,r-1)+" unique orders.");
      }
}
```

Here we can clearly see that the actual parameters to combination don't have to be just variables.

Finally, in the PerfectNums.java example, we utilize the fact that all known perfect numbers (numbers whose sum of proper divisors equals itself) are of the form $2^{p-1}(2^p - 1)$. Each of these is a perfect number given that p is prime and $2^p - 1$ is prime as well. In my code, instead of checking for the primality of p, I just create each number of this form and check if it's perfect or not. This greatly speeds up my search compared to a brute force check of every integer.

To compile, just compile the class you want to. Java will automatically go compile all classes it needs within the same directory to get the compilation of the current class done. (In our case, we can just compile PrimeCheck4.java, TacoBell.java and PerfectNums.java, depending on which of these applications we care to run.)

<u>Less Mathy Example – Static Methods with Strings</u>

Let's look at one more example with a single class but multiple static methods. Here we'll create a few methods with Strings. Consider the task of determining if a Strings begins and ends with the same character. Our method could take in the String in question as well as the desired character, then access the first and last characters of the string and see if they both match the desired letter. Here is the code for that method:

```
public static boolean StartsEndsSame(String str, char letter) {
      char check = str.charAt(0);
      return check == str.charAt(str.length()-1) && check == letter;
}
```

Unfortunately, there isn't a more succinct way to access the last character of a string than what is above. If we have to do this multiple times, it makes sense to store the result of str.length() in a shorter variable name.

Now that we've built this method, we can build another method that checks to see if both the first and last name of someone starts and ends with the letter a. Since names can have lower case or uppercase letters, our strategy here will be to store lowercase versions of both strings, then use these in two method calls to StartsEndsSame:

```
public static boolean APlusPlayer(String first, String last) {
      first = first.toLowerCase();
      last = last.toLowerCase();
      return StartsEndsSame(first, 'a') && StartsEndsSame(last, 'a');
}
```

Hopefully this example shows why functions are useful. We don't need to rewrite the logic for StartsEndsSame. We can call int multiple times. If we didn't have this method, we'd replicate the logic for it in two places and perhaps have errors in both places, making it harder to debug than just debugging a single method.

Let's define a monogram as the first two letters of one's first name and the first letter of their last name, all uppercase. We can write our own static method that calls existing String methods to perform this task. Our method will take in the first and last name of a person and return a String that is their monogram. First, we'll turn both strings into uppercase versions of themselves. Then, we'll use the substring method to extract out the pieces of both strings that we want. Finally, we'll use the concat method to put them together. Note: we can just use the + operator to concatenate strings, but I wanted to explicitly show the use of concat to illustrate the proper method of calling instance methods on objects. Here's the code:

```
public static String getMonogram(String first, String last) {
      String firstPart = first.toUpperCase().substring(0, 2);
      String secondPart = last.toUpperCase().substring(0, 1);
      return firstPart.concat(secondPart);
}
```

Finally, one last example, similar to the previous one. Given Strings first and last storing a person's first and last name, return a single String storing their rollbook entry, which is last name comma first name, with a space after the comma. Here's that method (one liner), and this time we take advantage of brevity of the + operator for string concatenation:

```
public static String getRollName(String first, String last) {
      return last+", "+first;
}
```