CS 2300 Week 05 Lab: Prime Numbers.

As usual, start a new Intellij IDEA with a name something like **Lab05Prime**.

PART 1:

1.1. Create a package and class with `lab05.PrimeMethods`. This class will hold two methods -- `isPrime` and `makePrimeTable` -- for dealing with prime numbers – and a main method used to test the other methods.

1.2. Make a "JavaDoc comment block" like this:

```
/**

*/
```

just above the class. Put the following info in it:

Your name,
The class,
The date,
Anything else that seems relevant.

1.3. Add a `main` method to the class, exactly like in previous labs. Write an opening line: **Prime Testing**. Write a closing line: **Bye**. Try this out to make sure it works (i.e., "Start small").

PART 2:

2.1. The key logic in this project is a method to determine if an integer is a **prime number**. A prime number is a number that is greater than 1, and is **only dividable by itself, and by 1**. For example, 2, 3, 5, 7, 11, 13, 17, 19 are the first few prime numbers. 4, 6, 8, 9, 10, 12, 15 are not prime because they are divisible by 2 or by 3. 1 is usually not considered a prime number.

2.2 The `isPrime` method is `public static` so that it can be used by other classes. Start by adding this line near the top of the class:

```
public static boolean isPrime(int n){
```

2.3. When you hit enter, a closing brace `}` should appear on the next line. There will be a red mark because so far this is not a legal `boolean` method, because there is no return line. This will be fixed presently.

2.4. Go to the line above this method, and start a **JavaDoc Comment** by writing `/**` and hitting enter. This will cause the IDE to create the following JavaDoc comment:

```
/**
 *
 * @param n
 * @return
 */
```

2.5. Add to the comment so that it looks like this:

```
/**
 * isPrime method tests primality of integers.
 * @param n: The integer to test.
 * @return true if n is prime, false if n is not prime
 */
```

This is another example of a **JavaDoc comment**. It is a good habit to always write such a comment for each method you write. If you do this, the JavaDoc program can automatically write a JavaDoc for the class. A JavaDoc is what you see when you look up any built in Java stuff. For example, if you Google on "Java 17 String", the first hit is the **API** for the String class. The API (Application Programming Interface) is a JavaDoc automatically created from the Java code for the String class. This is pretty cool!

2.6.Put the line

```
return true;                          // Temporary -- not finished yet!
```

inside the method. This should remove any mark indicating a syntax error. Doing this fixes the syntax error, but introduces a logic error: this version of the method indicates that all integers are prime! The reason for doing this is so you have a program that will compile. This helps because any other errors you might make will be found by the IDE. This line will continue as the last line of the method. The correct logic will be to return `false` for all `n` that are not prime. It this does not happen, then the last line will return `true`. When the method is correct, remove the **inline comment** (`Temp …`) because it is no longer temporary.

2.7. At the top of the `isPrime` method, put in the lines:

```
if (n < 2){                           // negative numbers, 0, and 1 are not prime
    return false;
}
```

This **inline comment** is an example of a pretty good comment. It explains why this test (the `if` statement) is included. Now we do not have to worry about what happens for `n < 2`, because in this case, `false` is returned, because the number is not a prime. Observe that the two inline comments above start in about the middle of the page, and line up with each other. This is a common style and makes your code look professional.

2.8. A number **n** is prime only if **no number m with 1 < m < n divides n**, so we can loop through this range of numbers and test using the `%` operator to check for divisibility. If we find some **m** that evenly divides **n**, we will immediately return `false`, because the **n** is not prime. Only for values of `n` that pass all the tests will `true` be returned.

2.9. In the middle of the method, after the first test, but before the final `return` statement, put in the following code:

```
for (int m = 2; m < n; ++m){          // check for a divisor
    if (n % m == 0){
        return false;
    }
}
```

This loop uses the mod operator, `%`. Using the mod operator like this `n % m` evaluates to the **remainder** when you divide `n` by `m`. If you get `0`, this means there is no remainder, so `m` divides evenly into `n`, so `n` cannot be a prime number! If any divisors are found, `false` is returned immediately, and you do not get to the end of the loop.

2.10. If the above loop runs all the way to the end without returning `false`, then it will proceed to the next statement, namely `return true`, which is correct because we did not find any divisors. So remove the `//`

`Temporary` from the `return true` line at the end of the method, because now the code is correct, so the line is not temporary any more.

2.11. This code should work, but get in the habit of ALWAYS TESTING ASAP! To do a very easy test, add the following lines to the `main` method:

```
System.out.println("   6 is prime? ( no)" + isPrime(6));
System.out.println("   7 is prime? (yes)" + isPrime(7));
System.out.println("7917 is prime? ( no)" + isPrime(7917));
System.out.println("7919 is prime? (yes)" + isPrime(7919));
```

Note how the tests are written so that the answers line up. It might take a couple seconds to run this program for the bigger numbers.

2.12. It turns out that we can slightly revise the program so that it runs much faster! If **n** is **not** prime, then **n = a*b** for some integers **a** and **b**. It cannot happen that both **a** and **b** are bigger than the square root of **n**, because otherwise **a*b > n**. So, if **n = a*b**, then either **a** or **b** is $\leq \sqrt{n}$. For this reason, we do not have to check all the way up to n, but only up to the last integer $\leq \sqrt{n}$.

There are (at least) two ways to make a loop run like this:

1. `for (int m = 2; m * m <= n; ++m){}`

2.
```
int last = (int) Math.sqrt(n);
for (int m = 2; m <= last; ++m){}
```

Using the first way slows things down by having to do one extra multiplication each time through the loop. This might make the program take twice as long to run. This is usually not a big deal, but we might want to avoid it.

Using the second way requires a square root, which might also take a long time to calculate (sometimes you can't win!). It would probably be faster for a large number. Notice that `Math.sqrt(n)` returns a `double`, so we must **cast** it into an `int` by doing this:

```
(int) Math.sqrt(n).
```

Casting one type of variable into another tells the compiler that we know what we are doing, so don't worry about the possibility of losing some accuracy. In this this situation, we don't care about losing some accuracy, but the compiler has no way to know that. Casting basically tells the compiler to not worry, I know what I am doing.

Replace the previous loop with one of these, for example

```
for (int m = 2; m * m <= n; ++m){
    if (n % m == 0){
        return false;
    }
}
```

2.13. Test this by running the program again. You should **always test as soon as possible**. Add some more tests to the four already there. Be sure to test 0, 1, and a negative number, so you can be sure the method deals correctly with

atypical input. When you are testing, try to think of everything that is unusual, or might trick the code for some reason. If you try it out on some numbers that you do not know if they are prime, write it like this:

```
System.out.println("54321 is prime (?): " + isPrime(54321));
```

2.14. If you enter really big numbers, you might have to wait awhile. This is slow work for big numbers. This is a key fact used in encryption, which is mostly based on factoring large numbers, such as ones with 100 digits. Factoring a number this big might take 1000 years, so don't wait around!

2.15. Besides the fact that it might take too long to factor a big number, this method is limited to how big of a number can be held in the `int` data type (about 2 billion). If we wanted to, we could do this program using `long`s instead of `int`s. Then we could test numbers up to 9223372036854775807. This would take a very long time, because it would have to test numbers up to about 3,037,000,500, which might take forever.

BTW, I found the maximum long given above like this:

```
System.out.println("max long: " + Long.MAX_VALUE);
```

This is an example of how Java has all the info you need for anything!

2.16. The method can be modified to report the first factor of a nonprime number. That is **not** part of this lab, but you might enjoy playing around with it, and perhaps using it to completely factor a number.

PART 3.

3.1 Now create another method in the same class with the signature:

```
public static void makePrimeTable(int primesToFind, int primesPerLine){}
```

The job of this method is to create a table of the first bunch of prime numbers and write them out as a table. The first argument, `primesToFind`, determines how many primes to find. The second argument, `primesPerLine`, indicates how many prime numbers to print on each line of the table. Each prime number is to be written in a field of width 12. Observe that the two arguments (`primesToFind` and `primesPerLine`) have well-chosen names, so there is little danger of mixing them up.

3.2 The table is to have a **header**, which means some sort of title at the top. The header is to look something like this:

```
Prime Number Table                                    CS 2300 Lab 05
Ralph Kelsey                                             2022 09 21
```

Use your own name and the correct date.

3.3. We will do a little extra work so that the right hand side of the header lines up correctly with the right hand side of the last column of prime numbers. Note that the right edge must be in the column that is 12 * `primesPerLine` from the start. For example, if `primesPerLine` is 5, the entire width is 12 * 5 = 60. In this case, we want the table to look like this:

```
Prime Number Table                              CS 2300 Lab 05
Ralph Kelsey                                        2022 09 21

           2           3           5           7          11
          13          17          19          23          29
```

and so on. An easy way to do this is just to guess how many blank spaces to put in the middle, try it out, and come back and fix it, which usually takes a few tries. BUT, if you change `primesPerLine`, then you have to do it all over again, which is a bummer.

It will be more fun to put in some smart code to ensure that the header is in the correct place no matter how many numbers get printed per line. Do it like this:

Put the following code at the top of the `makePrimeTable` method:

```
int widthOfTable = 12 * primesPerLine;
String stringLeft1 = "Prime Number Table";
String stringRight1 = "CS 2300 Lab Week 05";
int extraSpaces1 = widthOfTable - stringLeft1.length() - stringRight1.length();

// This should print one line of a table header the same width as the table
System.out.println("");
System.out.print(stringLeft1);
for (int i = 0; i < extraSpaces1; ++i){
    System.out.print(" "); // one extra space
}
System.out.print(stringRight1);
System.out.println(“”);
```

Read this through and see if you can figure out what is being done. Try it out and see if the output looks good. Here is what is going on:

3.3.1. Above, we used the Java `String` class. We **instantiate** two strings named `stringLeft1` and `stringRight1` (for row 1 left, and row 1 right). Then we figure out `widthOfTable`, how wide the entire row should be. Then we figure out `extraSpaces1`, how many blank spaces are needed in the middle to make everything in row 1 line up correctly.

3.3.2. Notice the `System.out.print`, as opposed to `System.out.println`. This means write something out, but DO NOT start a new line afterwards.

3.3.3. Then the row 1, left corner thing `stringLeft1` is printed. Then a loop is used to print as many blank spaces as needed. Then the row 1, right corner thing is printed, then finally a `System.out.println(“”);`, to start a new line.

3.3.4. Now check out that this is working. This might take a couple tries. When it is looking good, add another chunk of code similar to the above, but this time to write the second line of the header.

Do not put in another copy of `int widthOfTable = 12 * primesPerLine;`, or the compiler will give you a nasty red line for using the same name twice.

(Hint: Do about the same thing, but with 2 rather than 1). Test this again. The two lines should end up in the same place. Cool.

3.4. In the main method, add the line

```
makePrimeTable(100, 5);
```

to call the `makePrimeTable` method.

3.5. Run the program. So far, it will only print out the top two rows. Check it out, and fix if necessary.

3.6. In the `makePrimeTable` method, add code to print two more blank lines after the header.

3.7. You are about to write a loop to build the table. We will use two variables to keep track of what we are doing. First, `candidateNumber` is the number we are checking to see if it is prime or not. If it is, we will write it on the table. The **counter** `numberFound` keeps track of how many prime numbers we have found so far, so we will know if it is time for a new line to start. Add these two lines to your code:

```
int numberFound = 0;                  // number of primes found so far
int candidateNumber = 0;              // see if this number is prime
```

3.8. Now write a `while` loop. Control it by how many primes have been found so far; i.e., end the loop when you have found `primesToFind` prime numbers. An easy way to do this is

```
while (numberFound < primesToFind){}
```

3.9. The first thing to do inside the loop is increment `candidateNumber` with the `++` operator. Thus the first number actually tested will be 1.

3.10. The rest of the code will be inside a **block** controlled by an `if` statement. Start it like this:

```
if (isPrime(candidateNumber)) {}
```

3.11. Note that here we are using the `isPrime` method already written. We know it works, because we have tested it. This method can be used like this because it is `static`.

3.12. If the number is not prime, we do not go inside the block, and just move on and the next number.

3.13. If the number is prime, then we go inside the block and do the following tasks.

3.14. Check and see if it is time for a new line. You can tell this by `numberFound % primesPerLine == 0`. If so, use `System.out.println("")` to start a new line.

3.15. Increment the `numberFound` counter. (this means write: `++numberFound;`).

3.16. Write out the prime number found. We want to print these all in a **field of width 12**, like this:

```
System.out.printf("%12d", candidateNumber);
```

3.17. Here we use a **placeholder rule**, which starts with a `%` sign. Sometimes the word **mask** is used for placeholder rules.

Recall that **`%d`** means: print one `int` (in **d**ecimal notation) right here. Here we write `%12d`, which means print the `int` using exactly 12 spaces (a **field of width 12**), **right justified**. Right justified means pushed over to the right side of

the 12 spaces. Also, if the number does not fit, it will overflow past the 12 spaces. The `printf` statement prints whatever is inside the string (whatever is surrounded by the double quotes: `" "`). After the string, you put in as many variables as there are `%` placeholder codes. Then the first variable gets printed using the first mask, the second variable with the second, etc. In this case, there is only one mask (`%12d`) and one variable (`candidateNumber`).

3.18. After the table of prime numbers is completed, print out a couple of blank lines at the end.

PART 4. That's it!! Turn it in.

What to hand in. Submit your Java class to the Blackboard Assignment in the usual way.

PART 5. Just for fun.

If you are having fun, and don't want the fun to stop, add some features so that you can list all the **twin primes**. Twin primes are pairs of primes that are 2 apart, such as (3,5), (5,7), (11,13), (17,19), etc. One of the longest standing unsolved problems in math is called the **twin prime conjecture**. This conjecture (a statement of suspected fact, that is so far unproven) is that there are infinitely many twin prime pairs. This conjecture has been around for about 200 years, and just in the last year, a lot of progress has been made. It is very likely that there are infinitely many twin primes! But, so far, no one knows for sure.