COP 3330 – Object Oriented Programming in Java
Static Methods

Static Methods in Java

In all programming languages, we have subroutines called "functions." Functions are critical in programming because they do the following:

1. Allow for code reuse
2. Reduce time debugging since the same error isn't replicated in many places in the code.
3. Make code much more readable.
4. Reduce the number of variables, processes the programmer has to keep straight in their mind at the same time, which, in turn, reduces the number of errors being made.

Since this class is set up such that all students took a formal programming course before it, all students should be aware of functions in either the C or Python language.

The equivalent of a function in C or a regular (non-class) function in Python in Java is a static method. Recall that "static" means "belongs to the class." This means that we require a class to exist to create a static method, but we don't require any objects to exist in order to create a static method.

Here is the framework for most of the static methods in Java (and all static methods must reside within a class) that we'll write:

```
public static RETTYPE METHODNAME(FORMAL PARAMETER LIST)
```

For now we'll make all of our methods public. Later, we'll revisit what this means when we learn about visibility modifiers. In order for a method to be static, you must type the keyword static. This is followed by the return type of the method. If we don't want the method to return anything, we make this void. Otherwise, it can be either a primitive type or a reference type. So far the only reference type we've learned that might be worth returning is a String reference. This is followed by the name of the method and parentheses. Within the parentheses, we put our formal parameter list.

In Java, the formal parameter list must be a comma separated list of terms where is term is a pair:

```
TYPE FORMALPARAMETERNAME
```

Just like C and Python, if a method isn't void, then it must return an expression of the correct type via a return statement which has the following syntax:

```
return EXPRESSION;
```

Let's adapt an example from the previous lecture, PrimeCheck.java. In that example, we read in a single number from the user and determined whether it was prime or not. A natural improvement to this code would be to write a method that takes in an integer and returns true if the integer it takes in is prime, and false otherwise. First, here's the method:

Non-Void Method Example: isPrime method
Here is the method in isolation:

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

     // First prime number is 2.
     if (n<2) return false;

     // Try potential divisors until square root.
     // If any works one works, it's not prime.
     for (int i=2; i*i<=n; i++)
          if (n%i == 0)
               return false;

     // If we get here it's prime.
     return true;
}
```

Notice that because we have return statements, we have no need to ever declare a boolean variable. Also, we don't have to worry quite as much about flow control, because a return statement gets us out of the code, so after we get to a particular point in the method, we can be guaranteed that particular things are true. Finally, notice that in this method, there is no else tied to the if. One of the most common error beginning students make is the desire to write some code in a matching if, but if n isn't divisible by i, there is simply no action to be taken. In fact, in the majority of static methods, it's likely the case that they will be without an else. Most decisions are one way decisions...if something is true (or false), then an action can be taken, but if the opposite is true, no action can be taken.

Formally, to call a method, we would do:

```
CLASSNAME.METHODNAME(ACTUAL PARAMETER LIST)
```

Since a method belongs to a class, we must first list the class name, (we did this with Math methods), then a dot, then the method name, followed by parentheses containing the actual parameters passed to the method.

If you omit the classname, then Java assumes the method must be in the class that you are in. Thus, if you write a static method in a class, AND call it from a different place in that class, you can omit the classname and dot. If you are calling a static method from a different class than the one the method belongs to, then you must use the classname and dot before the method name.

For this lecture we will ONLY deal with programs involving a single class in a single file, so we won't need to use the classname dot syntax. In the next lecture we'll look at both how to create programs with multiple classes in one file and how to create a program with multiple classes in multiple files.

Let's look at the whole program, in the class PrimeCheck2, that asks the user for a positive integer and determines whether or not it's prime:

PrimeCheck2.java
```
import java.util.*;

public class PrimeCheck2 {

      public static void main(String[] args) {

            Scanner stdin = new Scanner(System.in);

            // Read the number.
            System.out.println("What number do you want to check?");
            int n = stdin.nextInt();

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

      } // end main

      public static boolean isPrime(int n) {

            // First prime number is 2.
            if (n<2) return false;

            // Try potential divisors until square root.
            // If any works one works, it's not prime.
            for (int i=2; i*i<=n; i++)
                  if (n%i == 0)
                        return false;

            // If we get here it's prime.
            return true;
      }
}
```

Since we're in the class, we can just make our method call as isPrime(n). Notice that in this example we happened to call the variable in main n and the formal parameter in the method n as well. Two thing:

1. The names of variables in two different methods are independent. (They can be the same or different, it doesn't matter.) Download the code and change the name of the variable in main to prove it to yourself.

2. It's not necessary that we pass isPrime a single variable. We can give it any sort of expression that evaluates to an integer.

We'll test out the second item a bit later.

Void method example: printChars method and supporting methods
In practicing nested loops, it's common to repeat printing a particular character some number of times. There's no "answer" to return in this subtask, so the method to solve this problem will be void. Here's the method:

```
     public static void printChars(char ch, int numtimes) {
          for (int i=0; i<numtimes; i++)
               System.out.print(ch);
     }
```

This looks pretty much like the previous static method we saw, but since there's no value to return, a return statement is not needed. The method just does its job and completes.

We can use this method to write another method that prints a left-justified right triangle of n rows with a particular character. Here is a design with the '^' character and 7 rows:

```
^
^^
^^^
^^^^
^^^^^
^^^^^^
^^^^^^^
```

Here's the code for the method:

```
     public static void leftRightTri(char ch, int n) {
          for (int i=1; i<=n; i++) {
               printChars(ch, i);
               printChars('\n', 1);
          }
     }
```

Once we have printChars as a subtask, then we can think about our new tasks in such a way that we can print any character we want, any number of times. For this task, we want to print ch, 1 time, followed by a new line, then 2 times, followed by a new line, and so forth. So, our pattern is that on row i, we print i copies of ch, followed by printing one newline character. So, we want our loop to run from 1 to n, inclusive, and then use our loop variable to specify the number of times ch is printed. BUT, we always print only 1 copy of the newline character!

Now, let's consider the task of printing a similar design, but right-justified, so there are some spaces preceding the character ch on some rows. Here is a design for 3 rows with character 'q':

```
  q
 qq
qqq
```

Let the rows be numbered 1 to n. On row number i, we must do the three following things:

1. Print n – i spaces.
2. Print i copies of character ch.
3 Print 1 copy of the newline character.

Here's the method that accomplishes this:

```
public static void rightRightTri(char ch, int n) {
     for (int i=1; i<=n; i++) {
          printChars(' ', n-i);
          printChars(ch, i);
          printChars('\n', 1);
     }
}
```

See how once we jot down the algorithm in words, if we have methods that solve appropriate subtasks, then the code basically writes itself.

Now let's add on the task of printing a pyramid. This will be similar to a right-justified right triangle, but have more characters ch on most of the rows. Instead of increasing the number of times ch appears on a subsequent row by 1, we'll increase it by 2, to create a vertical line of symmetry. Here is a pyramid design for n = 4 and the character '+':

```
   +
  +++
 +++++
+++++++
```

As you might imagine, the code is VERY similar to the previous method. Just change printing i copies of the character ch to 2*i – 1 copies of that character:

```
public static void pyramid(char ch, int n) {
     for (int i=1; i<=n; i++) {
          printChars(' ', n-i);
          printChars(ch, 2*i-1);
          printChars('\n', 1);
     }
}
```

Now, imagine that we want this picture with the rows reversed, so a backwards pyramid of sorts. Also, for reasons that will soon be clear, imagine that you want to add some number of extra spaces at the beginning of every line (in relation to the typical number of spaces on each line already added). We can create a method similar to the one above, BUT, we need to add a formal parameter to do our task: the number of extra spaces to add to each row. In addition, we'll have to reverse the order of our loop by starting at n and counting down to 1.

Here's the code:

```
public static void backPyramid(char ch, int n, int addLeftSpaces){

    for (int i=n; i>0; i--) {
        printChars(' ', n-i+addLeftSpaces);
        printChars(ch, 2*i-1);
        printChars('\n', 1);
    }
}
```

You might be asking, why on earth did we add extra flexibility to this function to allow us to pad it with extra spaces on the left? It's so that we can easily solve the next task:

Printing a diamond.

Let's define an n carat diamond of character ch as having 2n – 1 rows, where the number of copies of ch increases from 1, 3, 5, ..., 2n – 1, and then decreases 2n – 3, 2n – 5, ..., 3, 1 and is symmetric about both the middle row and the vertical line containing ch on the first row. Here is a 4 carat diamond with the character '*':

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

Of course, on the console, the colors won't print! Notice that the characters highlighted in yellow are nothing but a pyramid of size 4 and the characters highlighted in blue are a backPyramid of size 3, but one that's been "moved to the right" by 1 space. Now you can see why I added that parameter to the backPyramid method – it was necessary if I wanted to split my diamond task up into two smaller subtasks!!! With no further ado, here's the diamond code:

```
     public static void diamond(char ch, int n) {
          pyramid(ch, n);
          backPyramid(ch, n-1, 1);
     }
```

In the course examples, additional methods are added that print out numbers in any arithmetic sequence the user would like and these methods can further be used to print out some fun patterns!