Nested Loops

To investigate nested loops, we'll look at printing out some different star patterns. Let's consider that we want to print out a square as follows:

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

We know we can print out one line of this square as follows:

```
System.out.println("*****");
```

Now, we can utilize the for loop to do this 5 times:

```
for (int i=0; i<5; i=i+1)
    System.out.println("*****");
```

But, this would only work if we wanted a square of size 5. What if we wanted the user to choose the side length of the square? Then, when writing our program, we couldn't just do a single println with a fixed number of stars in it. Instead, we would have to utilize the user's input and print each star out one by one, so we printed out the right number of them on a single row. After that, we could advance to the next line.

So, we would need a loop that went through each line we were printing. But, then, when we were printing a single line, we'd need ***another loop*** inside of the previous loop we wrote. This is what we call a nested loop structure.

Here is a full program that accomplishes our task:

```
public class Square {
	public static void main( String args[] )
	{
		// Get the user input.
		Scanner stdin = new Scanner(System.in);
		System.out.println("How big do you want your square?");
		int n = stdin.nextInt();

		// Go through each row.
		for (int i=0; i<n; i++) {

			// Print exactly n stars on a single row.
			for (int j=0; j<n; j++)
				System.out.print("*");

			// Advance to the next row.
			System.out.println();
		}

	}
}
```

The key here is that when i = 0, j cycles through the values 0, 1, 2, …, n-1 and the print inside the j loop triggers exactly n times. Then, the newline prints. Then, i gets set to 1 and the whole process repeats.

Now, consider the situation that you want to print out a slightly more difficult pattern, such as a triangle:

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

The trick for effectively using a for loop is to make use of the index variable. The index variable is the one initialized in the initialization statement. In our previous example this was the variable i. If you want to repeat your code exactly, several number of times, then you will not have i appear in the body of your for loop. However, most of the time you will actually want to make use of the index variable inside of your loop. In fact, to print out this triangle, we must use the index variable inside of our loop.

Our general construct will loop like the following:

```
for (int i=0; i < 5; i=i+1) {
        <Print out one line of the triangle>
}
```

Now the problem we run into is that every line of the triangle is NOT the same. Thus, even though we must place the same code into the for loop body, it must execute differently every time. But, how is that possible???

The key is that if you place i in the for loop body, we know that i will take on different values for each iteration of the for loop. Thus, even though the code will look the exact same, it could potentially execute differently on different iterations. (We saw this earlier when we printed out the appropriate line number in the first example.)

We can now refine our construct to look like the following:

```
for (int i=0; i < 5; i=i+1) {
   <Print out the line in the triangle with (i+1) stars in it.>
}
```

Now, all we have to do is figure out how to print a line with exactly (i+1) stars.

But wait, this is asking how to do something exactly a specific number of times! We know what we can use to do that – a for loop! So our code will look like:

```
for (int i=0; i < 5; i=i+1) {
   for (int j=0; j <= i; j=j+1) {
      System.out.print(“*”);
   }
}
```

Now, this isn’t completely correct. The problem is that we do not have any println’s. This means the cursor will never get to a second line and all the stars will be printed on the same line.

We know that we must advance to the next line once we are done printing the current line. We can do that as follows:

```
for (int i=0; i < 5; i=i+1) {
   for (int j=0; j <= i; j=j+1) {
      System.out.print(“*”);
   }
   System.out.println();
}
```

In this example, notice that the inner loop is dependent upon the loop index of the outer loop. This allows for different behavior when each different row prints.

Printing Out all Prime Numbers in a Range

In our next program with nested loops, we'll print out all of the prime numbers within a range of integers. Recall that a prime number is one that is 2 or greater which only has two divisors: 1 and itself. We can check for primality through trial division, making sure that each integer in between 2 and n-1, inclusive do NOT divide evenly into n. As previously discussed, we can use the mod operator to check divisibility.

In this program, we'll prompt the user to enter in a low and high bound, and then we'll print out all of the primes between low and high, inclusive.

Our strategy is as follows:

1. Get the user input.
2. Set up a loop that goes from low to high.
3. Inside that loop, set up the mechanism to check the current number for primality.

```
public class PrimeRange {
	public static void main( String args[] )
	{
		// Get the low and high bounds for the prime search.
		Scanner stdin = new Scanner(System.in);
		System.out.println("Enter the low and high bounds.");
		int low = stdin.nextInt();
		int high = stdin.nextInt();

		System.out.print("The primes from "+low+" to "+high+" are ");

		// Try each number in range for primality.
		for (int i=low; i<=high; i++) {

			// Now, try dividing i by each relevant potential divisor.
			boolean isPrime = true;
			for (int div=2; div<i; div++)
				if (i%div == 0)
					isPrime = false;

			// Only print if none of the trial divisions worked.
			if (isPrime)
				System.out.print(i+" ");
		}
		System.out.println();

	}
}
```

Inside the outer loop(i), our goal is to determine if i is prime or not. Thus, we must try to divide i by 2, 3, …, i-1. We use the loop variable div to accomplish this goal. If any of these values is a divisor of i, we simply note that i isn't prime by changing a boolean variable from true to false. Note: inside of the if, one could break out, since once we find a divisor, there is no point in checking future values.

Drawing a CheckerBoard in a GUI

Now that you've been introduced to drawing objects onto a canvas using Java Swing, we can utilize nested loops to draw a checkerboard. A checkerboard is an design of 8 x 8 squares which are colored red and black, alternately. For this example, we are just going to modify the draw function to be different than the draw function from the assignment PaintStuff. First, we'll simply do a nested loop structure that emulates going through 8 rows and columns. NUMSQ is a constant set to 8.

```
for (int i=0; i<NUMSQ; i++) {
    for (int j=0; j<NUMSQ; j++) {
```

One key question is how to decide what color to make a square. If the top left square is red, then the square directly to its right and below are black, and then the same logic can be applied to those two squares. What we see is that if i+j is even, then the square is red and if i+j is odd, the square is black. Notice that whenever we move right or down, we change the sum i+j by 1, because we are adding 1 to either i or j by moving right or down. Thus, if we are on an even sum square and it's red, we are guaranteed to move to a black square by changing our parity of the sum from even to odd. The argument works the other way around when moving from a black square. So, we can set our color as follows, right inside of the loop:

```
for (int i=0; i<NUMSQ; i++) {
    for (int j=0; j<NUMSQ; j++) {

         if ((i+j)%2 == 0)
              g.setColor(Color.RED);
         else
              g.setColor(Color.BLACK);
```

Finally, we must draw a filled rectangle. To do this, let's introduce some more constants:

```
final public static int SQSIZE = 50;
final public static int OFFX = 45;
final public static int OFFY = 35;
```

SQSIZE is the number of pixels on the side of each individual square, OFFX is the offset in X from the left side of the display and OFFY is the offset in Y from the top of the display. When i and j are zero, our top left pixel value is simply (OFFX, OFFY). As i and j change, we just add the appropriate multiple of SQSIZE to our offset. Here is the full draw method:

```
public void draw(Graphics g) {

     g.setColor(this.backgroundColor);
     g.fillRect(0, 0, SCRSIZE, SCRSIZE);
     g.setColor(this.characterColor);

     for (int i=0; i<NUMSQ; i++) {
         for (int j=0; j<NUMSQ; j++) {

               if ((i+j)%2 == 0)
                     g.setColor(Color.RED);
               else
                     g.setColor(Color.BLACK);

               g.fillRect(SQSIZE*i+OFFX, SQSIZE*j+OFFY, SQSIZE, SQSIZE);
         }
     }
 }
```