Loops

Why a Loop?

This section will discuss three different kinds of *loops* in Java. Loops are structures that repeat the same set of actions over and over until a specified condition becomes false.

To see why loops are useful, suppose we want to print the sum of 100 numbers entered by the user. With what we've seen so far, we'd have to write 100 separate prompts and input statements – this would be a big mess. Really, for each number we want to do exactly the same thing – print a prompt and read in the number. With a loop, we can repeat these two steps a certain number of times without having to write each one out separately.

While Loops

The easiest kind of loop is a *while loop*. This loop executes a set of instructions repeatedly until a given condition becomes false.

Syntax

Here is the syntax for a while loop:

```
while (condition) {
	//statements
}
```

The condition is evaluated before anything inside the loop is executed. If the condition is false, we immediately skip to the code after the loop. If the condition is true, we execute the statements inside the loop, and then check the condition again. If the condition is false, we leave the loop. If it is still true, we execute the loop again. We repeat this process until the condition becomes false.

Examples

Here is how we can use a while loop to print the sum of 100 numbers entered by the user:

```
//keep track of the sum of the elements we've seen so far
int sum = 0;

//keep track of how many elements we've asked for
int count = 0;

//keep looping while we haven't asked for 100 elements
Scanner s = new Scanner(System.in);
while (count < 100) {
	//ask for the next number
	System.out.print("Enter a number: ");
```

```
          int num = Integer.parseInt(s.nextLine());

          //add the number to the sum we have so far
          sum = sum + num;

          //add one to our count (we've asked for one more number)
          count = count + 1;
     }

     //the loop is over – sum now holds the sum of 100 values

     //print the sum
     System.out.println("The sum is: " + sum);
```

We can also put if-statements inside of loops (or loops inside of if-statements). In this example, we want to print the sum of 100 positive numbers. If the user enters a negative number, we want to print an error and not add the number to our total.

```
     int sum = 0;
     int count = 0;

     Scanner s = new Scanner(System.in);
     while (count < 100) {
          System.out.print("Enter a number: ");
          int num = Integer.parseInt(s.nextLine());

          //only add the number if it is positive
          if (num > 0) {
               sum = sum + num;
               count = count + 1;
          }
          //otherwise, print an error
          else {
               System.out.println(num + " is not positive");
          }
     }

     //the loop is over – sum now holds the sum of 100 values

     //print the sum
     System.out.println("The sum is: " + sum);
```

Do-While Loops

A *do-while loop* is similar to a while loop, but its condition is evaluated at the end of the loop instead of at the beginning. This means that a do-while loop will always execute at least once, but a while loop might not (because the condition might be false in the beginning).

Syntax
Here’s the syntax of a do-while loop:

```
do {
     //statements
} while (condition);      //Notice the semi-colon!
```

Examples
Let’s say we want to add up a list of numbers typed by the user until they type a 0. Here’s how:

```
int num, sum;  //declare several variables of the same type

Scanner s = new Scanner(System.in);
sum = 0;
do {
     System.out.print(“Enter an integer: “);
     num = Integer.parseInt(s.nextLine());
     sum = sum + num;
} while (num != 0);

System.out.println(“The sum is “ + sum);
```

This loop immediately asks for a number before checking any kind of condition. Note that the last number typed by the user (a 0, which ends the loop) IS added to the sum. However, this is OK because adding zero to a number does not change the number.

For Loops

For loops are the most complicated loop, but they are probably used the most often. Many times you use a **loop counter** to keep track of how many times the loop has executed. If you use a while loop, you initialize the loop counter before the loop, have the while loop with a certain condition, and update the loop counter somewhere in the loop code. A for loop combines those three steps.

Syntax
Here’s the format of a for loop:

```
for (initialization; condition; update) {
     //statements
}
```

Examples
For example, this while loop that added numbers from 1 to 4:

```
int sum = 0;
int count = 1;
```

```
while (count < 5) {
     sum = sum + count;
     count = count + 1;
}
System.out.println("Sum: " + sum);
```

Our loop counter is `num`. Here's an equivalent for loop:

```
int sum = 0;
for (int num = 0; num <= 5; num++) {
     sum += num;
}
System.out.println("The sum is " + sum);
```

Here's another example, that prints out all odd numbers between 1 and 100. Notice that we increment the loop counter by 2 so we can immediately step to the next odd number.

```
for (int num = 1; num <= 100; num+=2) {
     System.out.println(num);
}
```

Other Loop Information

This section contains additional information for working with loops.

Infinite Loops

It is very important that we do something inside the body of the loop that will eventually make the condition false. Otherwise, the loop will execute forever – we call this an *infinite loop*. For example, consider the loop below, which is supposed to print the sum of the numbers between 1 and 4:

```
int sum = 0;
int count = 1;

//warning: this is an infinite loop!
while (count < 5) {
     sum = sum + count;
}

System.out.println("Sum: " + sum);
```

This loop will never finish executing. We told the loop to keep going while `count < 5` – but we NEVER change `count` inside the loop. Thus `count` is always 1, and the loop never ends. You can usually tell you have an infinite loop if your program just hangs without completing the execution.

Here is the corrected version of the loop:

```
int sum = 0;
int count = 1;

while (count < 5) {
     sum = sum + count;

     //update count so the condition will eventually be false
     count = count + 1;
}

System.out.println("Sum: " + sum);
```

Break
Recall that a break statement can be used to leave a switch case statement without evaluating any of the other cases. We can also use a break statement in a loop, which will let us immediately exit the loop without finishing the current iteration or checking the loop condition. For example, suppose we want to add up 10 numbers that the user types – unless the user types a 0, in which case we want to immediately stop and report the sum up to that point. Here's how:

```
int sum = 0;
int count = 0;
Scanner s = new Scanner(System.in);
while (count < 10) {
     System.out.print("Enter a number: ");
     int num = Integer.parseInt(s.nextLine());
     sum += num;

     //Exit loop if num was 0
     if (num == 0) break;
     count++;
}

System.out.println("Sum is: " + sum);
```

When we run this program, it will ask for numbers either until it has asked 10 times (and the loop ends), or until the user types a 0 (in which case we immediately leave the loop). It then reports the sum.

Continue
The `continue` statement allows us to skip the rest of the code in the loop, and immediately start on the next iteration. For example, suppose we again want to add up 10 numbers that the user types – but if they enter a negative number, we don't want to add it to our sum. Here's how:

```
int sum = 0;
int count = 0;
Scanner s = new Scanner(System.in);
while (count < 10) {
     System.out.print("Enter a number: ");
     int num = Integer.parseInt(s.nextLine());

     //Ask for next number if a negative number was entered
     if (num < 0) continue;
     sum += num;
     count++;
}

System.out.println("Sum is: " + sum);
```

This will ask for 10 numbers from the user. If a number is negative, we skip the rest of the loop (the part where we add `num` to our sum), and start on the next iteration (where we ask for another input number).

Variable Scope

There are rules about when we can use certain variables in our programs. If we declare a variable inside a set of brackets:

```
{
     //variable declaration
}
```

Then we can use that variable anywhere (after its declaration) inside those brackets. However, we cannot use the variable outside of the brackets (either before them or after them). These brackets can belong to a class, the Main method, an if-statement, or a loop – the rules are always the same.

Example

For example, if we do:

```
int sum = 0;
int count = 0;
Scanner s = new Scanner(System.in);
while (count < 100) {
     System.out.println("Enter a number: ");
     int num = Integer.parseInt(s.nextLine());
     sum = sum + num;
     count = count + 1;
}

//illegal – num is not visible here
System.out.println("The last number was: + num);
System.out.println(“The sum is: “ + sum);
```

Then we will get a compiler error. The num variable was declared inside the while loop brackets, and we cannot see it once the while loop has ended. If we did want to be able to print out the last number the user entered, we would have to declare the num variable before the while loop:

```
//now num is visible everywhere in this code fragment
int num = 0;

int sum = 0;
int count = 0;
Scanner s = new Scanner(System.in);
while (count < 100) {
     System.out.print(“Enter a number: “);
     int num = Integer.parseInt(s.nextLine());
     sum = sum + num;
     count = count + 1;
}

System.out.println(“The last number was: + num);
System.out.println(“The sum is: “ + sum);
```

Additionally, declaring the loop variable in a for loop has the same rules as declaring it inside the loop. For example:

```
for (int i = 0; i < 10; i++) {
     //do something
}
```

The `i` variable is only visible inside the for loop, just as if it was declared at the beginning of the loop.

These requirements are called *scope rules*. If we are outside the brackets where a variable was declared, then we are outside the *scope* of that variable.

Redeclaring Variables

Because we can’t always see variables that we have declared, we can reuse variable names. For example:

```
int count = 0;
while (count < 10) {
     int num;
     //do something with num
     count++;
}
```

```
count = 0;
while (count < 10) {
     int num;
     //do something with  num
     count++;
}
```

Here, we have reused the num variable. The first declaration of num is only visible to the first while loop. If we want to use the name num again in the second while loop, we must redeclare it (because we can't see the other variable). It is perfectly fine (and encouraged) to reuse variable names in this way.

We can also redeclare variables when we CAN still see the original variable, but it has a different result. For example:

```
int num = 10;
int count = 0;
while (count < 5) {
     int num = 2;
     count = count + num;
}

System.out.println(num);
```

Here, we declared num outside the while loop. Thus the num variable would have been visible inside the while loop as well. However, we declared num AGAIN inside the loop. Now there's a conflict – we can see the variable outside the loop and the new variable inside the loop. In this case, the compiler always assumes you mean the inner-most variable declaration. So in the while loop, when you add num to your count, it assumes you mean the inner-most num – the one declared inside the loop. So each time in the loop, 2 is added to your count.

Once you are outside the loop, the while-loop num is no longer visible. So the print statement prints out the original num value – 10.

This type of variable declaration is confusing and not recommended. In general, if a variable with some name is visible where you are, don't reuse the name for a different variable.

Nested Loops

We can also put one loop inside of another loop – this is called *nesting*. We tend to want a nested loop when we want to:

```
Repeat a series of instructions
     For each repetition, repeat a different series of instructions
```

In a nested loop, the inner loop is completed for each repetition of the outer loop.

Example: Printing Pattern #1
Nested loops can be tricky, so we'll start off with two simple examples. Let's say we want to print the following pattern:

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

Notice that we want 5 rows and 4 columns of stars (asterisks – above the 8 key). We will write a nested loop as if we were dealing with an array – the outer loop will step through each of the 5 rows, and the inner loop will step through each of the 4 columns. Inside the inner loop, we will print the next star (*):

```
for (int i = 0; i < 5; i++) {
     for (int j = 0; j < 4; j++) {
          System.out.print("* ");
     }
     System.out.println();
}
```

We are using the same trick with print statements that we did when printing a two dimensional array. The inner print statement is a `print`, because we are not done with the current row, and we want to print each row on a single line. The outer statement is a `println`, because we are done with the current row and want to advance to the next line for the next row.

Example: Printing Pattern #2
Let's continue our printing example, but make it a bit more complicated. Suppose this is what we want to print now:

```
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
```

This is very similar to the previous example, except that we now want 5 rows and 5 columns. Also, instead of printing stars, we want to print numbers. Notice that in each case, the number printed is the same as the column number (0 is the leftmost column, then 1, etc.). So, we can change our solution to:

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

The only change to this solution is that we are printing out the value of `j` instead of a `*`. The loop with `j` steps through the columns, so if we print `j` it will be the current column number. And for each row, it will print 0, 1, 2, 3, 4 (because those are the values that `j` steps through).

Example: Printing Pattern #3
This is our third printing pattern, and let’s make it a little more complicated. Suppose now we want to print:

```
0
0 1
0 1 2 3
0 1 2 3 4
```

We still want 5 rows, but the columns are a little different. We are still printing the current column number each time, but the number of columns varies with each row. Notice that row 0 has 1 column, row 1 has 2 columns, row 2 has 3 columns, row 3 has 4 columns, and row 4 has all 5 columns. So, the number of columns in each row is the row number + 1. In our for loops, i is keeping track of the current row number. So we can change our code as follows:

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

The only change to this solution is that the loop with j now steps while `j < i+1`. This controls how many numbers we want to print on the current row, which we decided was the row number + 1. The loop with i counts through the rows, so `i+1` is how many elements we want to print on the current row. (For the first row, `i+1` will give us 1 column. For the second row, `i+1` will give us two columns, etc.)

Example: Factoring a Number
In this example, we want to ask the user for a number, and then print out its prime factors. (If the number is prime, then we want to print that fact instead.) For example, if the user enters 20, we want to print something like this:

20 = 2*2*5

We know that we want to loop through possible factors (from 2 up to the number-1) and try seeing if they divide into the number evenly. The trick is that some values will be multiple factors – for example, we need to use 2 twice when factoring 20. So our approach will look something like this:

```
loop through all possible factors
    factor out the current number as many times as possible
```

Here is a solution:

```
import java.util.*;
public class Factor {
     public static void main(String[] args) {
          Scanner s = new Scanner(System.in);
          System.out.print("Enter a number to factor: ");
          int num = Integer.parseInt(s.nextLine());

          System.out.print(num + " = ");

          //We will divide factors out of cur as we find them
          int cur = num;

          //for each possible factor i
          for (int i = 2; i <= num-1; i++) {
               //as long as i keeps dividing in evenly
               while (cur % i == 0) {
                    //print the factor and divide it out
                    System.out.print(i + " ");
                    cur = cur / i;
                    if (cur != 1) System.out.print("* ");
               }
          }

          if (cur == num) {
               System.out.println("prime");
          }
          else {
               System.out.println();
          }
     }
}
```

Full Examples

This section includes three more examples of full programs using loops.

Example 1
In this example, we will ask the user for 10 numbers, and we will print out the smallest number entered.

```
import java.util.*;

public class Example1 {
     public static void main(String[] args) {
          Scanner s = new Scanner(System.in);
          System.out.println("Enter 10 numbers.");
          System.out.println("Press Enter after each one.");

          int min = Integer.parseInt(s.nextLine());
          for (int i = 1; i < 10; i++) {
               int next = Integer.parseInt(s.nextLine());
               if (next < min) min = next;
          }

          System.out.println("Smallest: " + min);
     }
}
```

Example 2
In the next example, we will use a loop to help compute an exponent. We will ask the user for the base (b) and the exponent (n), and will compute and print b^n.

```
     import java.util.*;

     public class Example2 {
          public static void main(String[] args) {
               Scanner s = new Scanner(System.in);

               System.out.print("Enter base: ");
               int b = Integer.parseInt(s.nextLine());
               System.out.print("Enter exponent: ");
               int n = Integer.parseInt(s.nextLine());

               int exp = 1;
               for (int i = 0; i < n; i++) {
                    exp *= b;
               }

               System.out.println(b + "^" + n + " = " + exp);
          }
     }
```

Example 3
In this example, we will ask the user for a positive integer bound (we'll call it n). Then, we will calculate and print:

1!
2!
…
n!

The ! means factorial. For example, 5! is calculated as follows:

5! = 5*4*3*2*1 = 120

This will require a nested loop. The outer loop will step through the numbers from 1 to n, and the inner loop will find the factorial of the current number.

```
import java.util.*;

public class Example3 {
     public static void main(String[] args) {
          Scanner s = new Scanner(System.in);

          System.out.print("Enter a positive integer bound: ");
          int bound = Integer.parseInt(s.nextLine());
          if (bound >=1) {
               for (int i = 1; i <= n; i++) {
                    int fact = 1;
                    for (int j = i; j >= 1; j--) {
                         fact *= j;
                    }

                    System.out.println(i + "! = " + fact);
               }
          }
          else {
               System.out.println("Bound must be positive.");
          }
     }
}
```