Scope and Foreach

- Learn what loops are, and some conventions for writing loops such as using ++ and declaring variables outside the loop
- Learn what a while loop is and how to use it
- *Understand the scope within which a variable is accessible*
- *Learn what a for each loop is and how to use it.*
- *Learn what a for loop is and how to use it.*

Variable Scope

In the previous example, notice how we declare value outside the loop? If it were inside the loop (e.g. at line 10), Java would create and delete the variable in each iteration. If this were a variable which took up a large amount of memory (for instance, a large image), then creating and deleting this image might be very time consuming. By declaring value on line 03, we ensure that Java only has to allocate the space for value once. Knowing where each variable is created and deleted, and which variables are accessible from where, is called *scope*.

Scope

(definition) When you declare a variable, you also implicitly declare who can see that variable. A variable that is declared inside a loop is only visible to other code inside the same loop. A variable declared at the start of a method is visible to everything else inside that method (and not visible to any other methods). A field declared at the start of a class is visible to every method inside that class.

As an example, take a look at the following code:

```
int a = 0;
int counter = 0;
while (counter <= 10) {
   int b = 0;
   a = counter * 2;
   b = a + 5;
   counter = counter + 1;
}
```

variable **a** is accessible from inside the loop, as well as outside the loop and will maintain its value even as the loop goes through multiple iterations. However, **b** is declared *inside* the loop, and is only accessible from inside the loop. If you try to use b anywhere outside of the while loop, Java will give you an error. Why does this happen? This is because **a** has a *wider* scope than **b**, making it accessible in more places.

Now take a look at this code:

```
int c = 0;
int counter = 0;
      while (counter <= 10) {
   // do something
}
```

Notice that c is declared and assigned outside of while loop. The assignment of c with value 0 happens only once regardless of how many times the while loop iterates, because c is declared outside of the while loop. If you put line 1 (int c = 0;) inside the while loop, then both the declaration and the assignment of c happens in each iteration of the loop. In this case c would be reset to 0 during each iteration of the loop. You could also declare c outside of the loop (so that it is created only once) and assign a value inside the loop (so that the assignment happens in each iteration of the loop). An example of this is the value variable in decreaseRed() and the total variable in addTo100().

So let's try a few exercises to help you wrap your head around scoping:

did I get this

Look at the code below and answer the following questions:

```
int a = 0;
int b = 0;

while (count < 5 ) {
   int count;
   int c = 2;
   a = count + 1;
   b = a * 2;
   c = 5;
   count = count + 1;
}

System.out.println(a);
System.out.println(c);
System.out.println(count);
```

If you try to this code, you will get the error “Undefined name ‘count’”. Why does Java think count is undefined?

a. Count is never initialized?
b. Count is declared inside the while loop?
c. Count is never greater than 5?

Suppose we update the code by adding the line: int count = 0; on line 3 of the code. Try running the code again. You will get another Undefined name error. Why?

a. C is never initializes
b. C is never incremented
c. C is not accessible on line 13.

See if you can fix the remaining bugs in this code. You will need to declare c outside of the loop, and then fix an error that caused the loop to run endlessly.

Our solution

If you try to this code, you will get the error “Undefined name ‘count’”. Why does Java think count is undefined?

count is declared inside the while loop on line 5, but the first reference to count happens on line 5, which is before count is declared (and not technically inside the while loop).

Suppose we update the code by adding the line: int count = 0; on line 3 of the code. Try running the code again. You will get another Undefined name error. Why?

c is declared inside the while loop (on line 6) but referenced outside the while loop (on line 13). This means that c is out of scope when it is referenced, which generates the error.

See if you can fix the remaining bugs in this code. You will need to declare c outside of the loop, and then fix an error that caused the loop to run endlessly.

```
int a = 0;
int b = 0;
int count = 0;
int c = 2;

while (count < 5 ) {
  a = count + 1;
  b = a * 2;
  c = 5;
  count = count + 1;
}
System.out.println(a);
System.out.println(c);
System.out.println(count);
```

did I get this

Look at the code below and answer the following questions:

```
public class MyClass {
   public int myPublicValue = 0;

   public void methodA() {
      myPublicValue = 10;
   }

   public void methodB() {
      System.out.println(myPublicValue);
   }

   public void methodC() {
      int myMethodValue = 100;
   }
   public void methodD() {
      System.out.println(myMethodValue);
   }
}
```

What happens when you run methodA() followed by methodB()?

a. 0 is printed in the console
b. 10 is printed in the console
c. Nothing is printed in the console
d. An error occurs

What about if we run methodC() followed by methodD()?

a. 100 is printed to the console
b. Nothing is printed to the console
c. 0 is printed to the console
d. An error occurs.

Our solution

What happens when you run methodA() followed by methodB()?

myPublicValue is modified by methodA() and methodB() prints the result 10 to the console. (B)

What about if we run methodC() followed by methodD()?

myMethodValue is only defined in methodC(), so other methods cannot access or use that variable causing an error (D)

For-Each Loops

An alternative to a **while** loop is a **for-each** loop. This kind of loop is especially designed to do something to each element in an array. The syntax is:

for (*Type variableName:arrayName*) {
   // body of loop

```
}
```

Here, *arrayName* should be the name of the array being looped through. *variableName* should be the name of a new variable that Java can update at the start of each iteration of the loop. It holds the current array element that should be manipulated in the body of the loop.

Example: *decreaseRed()* using *foreach*

We can rewrite the decreaseRed() method from before using this construct as follows:

```
/**
 * Method decreases the amount of red in every
 * pixel of this image by half.
 */
 public void decreaseRedWithForEach() {
   Pixel[] pixelArray = this.getPixels();
   int value = 0;

   // loop through all the pixels in the array
   for (Pixel pixelObj:pixelArray) {
     // get the red value of the current pixel
     value = pixelObj.getRed();
     // decrease the red value by 50%
     value = value / 2;
     // set the red value of the current pixel to the new value
     pixelObj.setRed(value);
   }
}
```

Note...
Note that we named the new method decreaseRedWithForEach(). That way you can put both in Picture.java. An alternative would be to replace the code you used to have in decreaseRed() with this code.

Even though this type of loop does not require a counter or an explicit end condition, it functions very similarly to the while loop we described above. There are a couple of important differences, though. First of all, each time Java enters the loop, it automatically sets pixelObj to point at the next pixel in pixelArray for you. Second, when it reaches the last pixel in pixelArray, the end condition is triggered, and the loop will be terminated.

Once pixelObj is set, it can be used inside the loop. Java enters the loop and executes all of the code between the { and } and then returns to the beginning of the loop to select the next pixel (or terminate if there are none).

How do you decide whether to use a for-each loop or a while loop? In general, you would pick the solution most appropriate for your task. Since we are iterating through an array, it makes sense to use a programming construct designed for that. Also, it makes the code slightly simpler, since we don't have to keep track of a counter and end condition.

As you learn about Java, you will discover other ways in which it simplifies common tasks. For example, as discussed earlier, Java provides a special operator to increment (++) or decrement (--) a variable.

Why use a for loop?

It is easy to make small mistakes when programming. As a result, programmers like common tasks to be structured. While the foreach loop does this well for arrays, there are many examples of loops that simply count up from a starting number to an ending number but do not involve an array. The for loop is a good solution for this problem.

With the while loop, we had to use a counter to keep track of the number of times the loop was run. We also had to declare the counter variable, and initialize it before the loop. If we forgot this step, then the compiler would complain, and refuse to compile the code.

Similarly, we had to increment the counter within the loop, or the end (termination) condition would never be met, and the loop would continue to run over and over again *ad infinitum*. This mistake is so common, it has a name -- an "infinite loop". Having to find and correct these bugs can be time consuming and exasperating. Isn't there a better method of looping a set number of times? As a matter of fact, there is: the for loop.

The for loop is commonly used for looping a specific number of times, and its syntax is:

for (*initialization_area; continuation_test; change_area*)

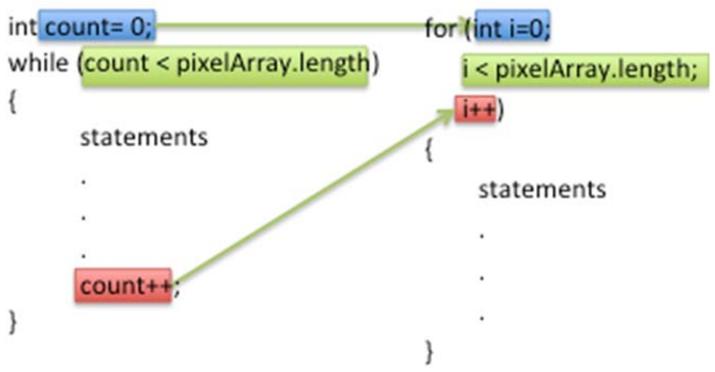
The *initialization_area, continuation_test* and *change_area* are all expressions that are evaluated as the for loop runs. Here's what each area does.

The initialization area consists of code which, you guessed it, initializes something. This is where the counter is initialized. In fact, you can both declare and initialize it here, instead of on a separate line before the loop starts as you have to do in a while loop. Since this is part of the for loop syntax, you won't forget to do it. A typical initialization area looks like: int i = 0;. This part of the for loop is evaluated only once, before the first iteration through the loop.

The continuation test area acts like the *condition* area in a while loop. It tells Java to "run the code inside the for loop until this condition is not met". Again, since it's part of the loop declaration, you're likely not to forget it. A typical continuation area checks if the counter is less than the goal (*e.g.* i < 10). In other words, it checks if the number of iterations completed so far is less than the total number needed. This is just like the *condition* part of a while loop (while (*condition*) {...}). The condition is evaluated at the start of each iteration of the for loop. If the condition evaluates to true, then the code contained in the body of the loop is executed. If it evaluates to false, the loop is terminated.

The change area, you guessed it, changes the counter. In a while loop, you would need to do this somewhere inside the body of the loop. In a for loop, since it's part of the declaration, you're not likely to forget to do it. A typical change area looks like: i++. (If you are not clear on what i++ means, refer back to our previous discussion on the increment operator).

The image below summarizes how for and while loops relate to each other. Both have the same key elements for iteration, but in a for loop, programmers can think about all of the key decisions relating to iteration when they start coding the loop. Even though the loops look different, all the pieces execute at the same time in a for loop as in a while loop.


```
int count= 0;                          for (int i=0;
while (count < pixelArray.length)          i < pixelArray.length;
{                                          i++)
        statements                     {
        .                                   statements
        .                                   .
        .                                   .
        count++;                            .
}                                      }
```

The highlighted lines are similar in a while loop and a for loop, and they are evaluated at the same time as Java iterates through the loop. The only important difference is where they are located in the code. Declaration of the counter variable (blue box) happens outside of the while loop, while it happens in the for loop. The continuation test condition (shown in green) happens in the declaration of both loops. The counter is incremented (red box) in the body of the while loop but in the declaration of the for loop.

Converting from *while* to *for*

To print 40 lines of "*****"? We implemented it with a while loop, as follows.

```
int i = 0;
while (i < 40) {
   System.out.println("*****");
   i++;
}
```

Since a while loop and a for loop only differ in the location of a few lines of code, we should easily be able to convert this from a while loop to a for loop. Can you find the initialization area above? It's on line 1, int i = 0;. Now look for the continuation test. It's on line 2, i < 40. Finally, look for the change area. It's line 5, i++. Using a for loop, the same code can be written as:

```
for (int i = 0; i < 40; i++) {
        System.out.println("*****");
}
```

In addition to being less error-prone, the for loop is more compact (has fewer lines) than a while loop. On the other hand, a while loop can do things no for loop can do. For example, a while loop might not have a counter at all.

learn by doing

Have you ever called someone back repeatedly when the phone was busy? Did you stop after a certain number of calls or when they picked up? How about search for a lost object? Do you stop looking when a certain amount of time has passed or when the object is found? These are example of “loops” that do not use a counter. Can you think of another example?

Our Solution

When practicing the piano, one might keep doing a section until they get it right more times than wrong.

Summary

In summary, scope is the part of your code in which a variable is accessible. You can think of it as analagous to the shadow of a section of code. Something is usually in scope for anything that is inside the same outermost set of curly braces ({}).

Remember to consider the scope in which a variable will be visit when you declare it. This will help to reduce bugs in your code.

We've introduced three types of loops so far -- the foreach loop, while loop, and for loop. Each is specialized for a slightly different task. **foreach** is excellent at looping through arrays. **for** is specialized for iterating a certain number of times. **while** is probably the most general of the loops as you could specify almost any end condition. In fact, some while loops don't use counters at all!

This module focused on the for loop, which has the following syntax:

```
for (initialization_area; continuation_test; change_area) {
    // body of loop
}
```

The for each loop uses:

```
for (Type variableName:arrayName) {
     // body of loop
}
```

Finally, the while loop uses:

```
while (condition) {
    // body of loop
}
```