

# Iterative Constructs

(CISS-110 William Jojo)

## Overview

There is a clear need for repetition in many things from our daily life. These may include cooking in the kitchen (3 meals, perhaps more), doing laundry (2 or more loads on a Saturday afternoon), broadcast television (show intro to sitcom, then alternate commercials/sitcom until end of show) or simply mowing the lawn (walk mower to end of lawn, turn, walk mower back to other end, repeat). It is very easy to see that people operate iteratively very naturally. Consider the following Java'd-up pseudocode for mowing a lawn where fictitious variables and methods have been liberally created for entertainment purposes:

```
mower.removeFromStorage();

if ( mower.oldClippings == high )
{
    mower.cleanCuttingChamber();
    mower.notifyOwnerOfBadMaintenancePractice();
}

if ( mower.bladeStatus != mowerConstants.ADEQUATE )
{
    Blade replacementBlade = new Blade();
    mower.changeBlade(replacementBlade);
}

if ( mower.fuelLevel != mowerConstants.FULL )
    mower.refuel();

while ( mower.motorRunning != true )
    mower.startMower();

do
{
    mower.walkToEndOfLawn();
    mower.turn();
    lawn.checkLawnStatus();
} while ( lawn.mowed != true );
```

Now for some fun with this code. There are several selection statements involved in the preparation of the work to be done. This is very typical of programming with *iteration* or programming with *loops*.

Once the preliminary checks are performed, there is the `while` loop to start the mower. The idea is that the `startMower()` method will set the boolean variable `motorRunning` to `true` if the mower starts and stays running. That means that while the mower is not running, we should continue to try and start it. Now, in the real world, the average person would try only a few times, especially if its a pull-start, and then begin to look for things like a flooded engine, a bad float, a fouled spark plug or simply stop to catch their breath, then try some more. That part of our program had been omitted, but could easily be added in.

Now we introduce the concept of testing before performing repetition. In the case of starting the engine, we checked to make sure it wasn't running already by virtue of the parenthesized test to the right of the `while` keyword. This constitutes a *top-test* loop meaning that we may perform the iterations of of the loop zero or more times based on the result of that test. Another way to to say it is: *while something is true, perform the work.*

There is a second loop example in our pseudo code. It is a `do/while` loop. The concept of this loop is to introduce a *bottom-test* loop. These sorts of loops are useful when we know we will perform the work inside the loop *at least once*. Only after the work is done one time will we test to see if additional applications of the same work is necessary. Now we can say: *mow a strip of lawn, turn and check lawn status while there is still lawn to be mowed.*

The mowing of the lawn lends itself well to the bottom-test concept since we started out knowing that our virtual lawn needed cutting, otherwise we wouldn't be here with our virtual mower.

It is important to note the subtlety in our two loops:

- **While** the mower is not running, start the mower.
- Mow a strip of lawn and turn **while** the mowing is incomplete.

Given different circumstances, we could have used `do-while` for the starting of the mower or a `while` for the cutting. The circumstances of the situation dictate how we will choose our looping methods.

This is carried over to programming where conceptually we would want to do the same as our mowing example. Consider the need to read three values from the user in order to derive a sum and average of those three values. We could simply write:

```
System.out.print("Enter value #1: ");
value1 = kb.nextInt();
System.out.print("Enter value #2: ");
value2 = kb.nextInt();
System.out.print("Enter value #3: ");
value3 = kb.nextInt();

sum = value1 + value2 + value3;
```

Clearly this method works. Maybe this isn't much typing for the average user, but extend the concept to 5 values. 10 values. 100 values. This suddenly becomes unwieldy the higher we go. We need to make this into a loop.

We can start by eliminating the idea of `value1`, `value2` or `valuen` and just have a single variable called `value` that we will recycle through each iteration of the loop. We will also need to take into consideration that the calculation of `sum` is no longer a job to be done after all the values are read in from the user. Instead, we will need to incorporate the idea of a *running-total* while we continue to read values.

One last piece, which is really just maintaining continuity, is incorporating the numeric value in the prompt just as our previous, brute-force method did. Consider the following `while` loop.

```
int sum = 0;
int x, value;
Scanner kb = new Scanner(System.in);

x = 1;
while ( x <= 3 )
{
    System.out.print("Enter value #" + x + ": ");
    value = kb.nextInt();
    sum = sum + value;
    x++;
}
// program continues
```

We start out by setting `sum` to zero since we have seen no values yet. This was not necessary in our previous example since `sum` was calculated after all the values were read from the user.

Next, the variable `x` is set to 1. The variable `x` will be used as our *loop control variable*. In other words, `x` is the variable we will examine to determine if more work remains or if we have completed our iterations and can stop the loop.

The loop starts with a test of `x <= 3` or, more to the point, `1 <= 3` since `x` is 1. Since that test is true we enter the loop and begin our work. The loop will perform work for values of `x` equal to 1, 2 and 3. Each time going back to the top of the loop to reevaluate the condition that allows us to stay.

When the value of `x` reaches 4, the test at the top will fail (`4 <= 3`). At that point we *fall-out* of the loop to the statement just below the loop itself and the program continues.

Note that each time the loop performs work, we update the value of `x`. The increment operator bumps the value of `x` by one prior to our retesting its value at the top.

Now that we have seen a basic example of iteration, there are three loop structures:

- The `while` loop
- The `do/while` loop
- The `for` loop.

Each of these loop structures will now be presented in detail starting with the `while` loop.

## The `while` Loop

As we saw in the overview section many repetitive things can be formed into iterative constructs. The overview touches on the concept of the top-test loop construct. Now we will look at multiple variations of the loop and how to control the work that is needed to be done.

First let us discuss the three parts of a loop. These are *initialization*, *condition* and *update*. The Initialization is the part necessary to get ready for the iterative process. The condition determines if the iterative process should continue. Finally, the update portion assists in altering the loop control variable in some way to make sure that the condition will eventually fail and cause the loop to stop, if that is indeed our goal. Some loops that never stop, know as the infinite loop, are usually unintentional, but are often used for things like printer queues and web servers.

### Range-based or Counter-based loop

The range-based loop is used when we know that something will occur a predetermined number of times and the loop control variable will be the variable keeping track of how many times the work has been done.

The previous loop example is repeated here with the details of initialization, condition and update in bold for clarity.

```
int sum = 0;
int x, value;
Scanner kb = new Scanner(System.in);

x = 1; //initialization
while ( x <= 3 ) //condition
{
    System.out.print("Enter value #" + x + ": ");
    value = kb.nextInt();
    sum = sum + value;
    x++; //update
}
// program continues
```

## Sentinel-based loop

Sentinel-based loops are used when we are expecting a specific value or values that the user must enter to indicate that the loop may cease. If we needed the user to enter -1 as a value to terminate the loop, we could craft something like this:

```
int sum = 0;
int value;
Scanner kb = new Scanner(System.in);

System.out.print("Enter integers (-1 to end): ");
value = kb.nextInt(); //initialization
while ( value != -1 ) //condition
{
    System.out.print("Enter value #" + x + ": ");
    value = kb.nextInt();
    sum = sum + value;
    System.out.print("Enter integers (-1 to end): ");
    value = kb.nextInt(); //update
}
// program continues
```

There are a few things to note about this code.

- The prompt appears before the loop and again at the bottom of the loop.
- The call to `nextInt()`, just like the prompt is located both before the loop and again at the end of the loop.
- The loop control variable is `value`.
- The same statement is used to initialize *and* to update the loop control variable.

It is necessary for us to include the prompt twice if we want the user to continue to be prompted once the loop begins since we cannot break the confines of the loop to get to the original first prompt. The prompting and reading of the next value happens at the bottom of the loop to alter the loop control variable prior to reapplying the test when we reach the top of the loop.

It is not at all unusual, as we saw in the previous example, for the update statements to be the same statements that initialized our loop control variable. In the example above, the only way to alter the loop control variable is to read another value from the user. Since the loop is driven entirely by user input, there is often little or no choice.

## Flag-based loop

The flag-based loop uses a `boolean` variable for loop control. To demonstrate let's create a complete program to ask the user to guess a random number between 1 and 20 inclusive as

shown in *Example 1*.

```
import java.util.*;

public class guess
{
    static Scanner kb = new Scanner(System.in);

    public static void main(String[] args)
    {
        int randomNumber, guess;
        boolean found;

        // random number between 1 and 20
        randomNumber = (int)(Math.random() * 20 + 1);

        System.out.println("I'm thinking of a number between 1 and 20.");

        found = false;
        while ( ! found )
        {
            System.out.print("Guess the value between 1 and 20: ");
            guess = kb.nextInt();

            if ( guess == randomNumber )
                found = true;
            else
                System.out.println("It's not " + guess + ".\n");
        }

        System.out.println("\nYou got it!\n");
    }
}
```

*Example 1: Flag-based while loop.*

A few things to note about this example are:

- The boolean variable, `found`, needs to be initialized to `false` before the loop begins.
- Using `Math.random()` coerces the result to a `double`, therefore, we cast the result to `int`.
- We are not concerned with guesses that are out of bounds or with a limit on the number of guesses a user is allowed.
- We read values from the user inside the loop with no special attention made to how or when we prompt since the `found` is the loop control variable and not `guess`.

The program in *Example 2* is reworked slightly to deal with a limit on guesses and to inform the user of the value if they are unsuccessful in guessing the correct answer.

```

import java.util.*;

public class betterguess
{
    static Scanner kb = new Scanner(System.in);
    public static void main(String[] args)
    {
        int randomNumber, guess, guessLimit=5;
        boolean found;

        // random number between 1 and 20
        randomNumber = (int)(Math.random() * 20 + 1);

        System.out.println("I'm thinking of a number between 1 and 20.");

        found = false;
        while ( ! found && guessLimit > 0 )
        {
            System.out.println("You have " + guessLimit +
                " guesses remaining.");
            System.out.print("Guess the value between 1 and 20: ");
            guess = kb.nextInt();
            guessLimit--;

            if ( guess == randomNumber )
                found = true;
            else
                System.out.println("It's not " + guess + ".\n");
        }

        if ( found )
            System.out.println("\nYou got it!\n");
        else
            System.out.println("\nThe number was " + randomNumber + ".\n");
    }
}

```

*Example 2: Improved number guessing program.*

The details of this modification are:

- The variable `guessLimit` has been set to 5 and will count down to zero.
- `guessLimit` is decremented after each guess is read from the user.
- We still lack code to deal with out-of-bounds guesses, but could easily be added after each guess. Should an out-of-bound guess count against the user?
- We've added the `&&` logical operator to account for two conditions that need to be met in order to continue prompting for guesses. The must not have discovered the answer and they must also have guesses remaining.

## EOF-based loop

The EOF-based loop works with the `hasNext()` method of the `Scanner` class. The user can enter CTRL-D in Unix or CTRL-Z in DOS/Windows to terminate the loop. Consider the next program segment to read lines of text from the user and simply display the length of each string.

```
String line;
Scanner kb = new Scanner(System.in);

System.out.print("Enter a string (CTRL-D to end): ");
while ( kb.hasNext() )
{
    line = kb.nextLine(); // read the line
    System.out.print("Input was " + line.length +
        "characters long.");
    System.out.print("Enter a string (CTRL-D to end): ");
}
```

Items to note about this code block are:

- We are prompting before the loop *and* at the bottom of the loop.
- The reading of input happens inside the loop and not automatically through the use of `hasNext()`.

## The for Loop

Another top-test loop is the `for` loop. This loop can be a bit tricky to grasp at first glance, but once you realize that it is simply a reworked `while` loop, it's a snap.

Since our loops still have the initialization, condition test and update components, the `for` loop chooses to put them all on display on the same line rather than spreading them out.

Let's take the simple `while` example from the overview again:

```
int sum = 0;
int x, value;
Scanner kb = new Scanner(System.in);

x = 1;
while ( x <= 3 )
{
    System.out.print("Enter value #" + x + ": ");
    value = kb.nextInt();
}
```

```
    sum = sum + value;
    x++;
}
// program continues
```

The `for` loop version looks like this:

```
int sum = 0;
int x, value;
Scanner kb = new Scanner(System.in);

for ( x = 1; x <= 3; x++ )
{
    System.out.print("Enter value #" + x + ": ");
    value = kb.nextInt();
    sum = sum + value;
    // the update portion happens here!
}
// program continues
```

The portions in bold show what has changed. Let's itemize the differences:

- Used the `for` reserved word in place of `while`.
- Moved *all* three loop components to the parenthesis that previously held the condition test.
- Used semicolons to separate the initialization, condition test and update.

The initialization only occurs once and before the condition test is applied. Entering the loop is still determined by the successful evaluation of the condition test just as it is with the `while` loop.

The update, however, is a little different. This will *always* happen as the very last statement prior to reapplying the condition test at the top of the loop. With our `while` loop, the update could have been any statement within the body of the loop. We chose to do it at the bottom of the `while` loop since that made the most sense.

The `for` loop enforces it at the bottom when the update is specified in the parenthesized portion.

You may leave out any or all of the parenthesized components of the `for` loop preserving the semicolons. However, you may want to reconsider why you chose the `for` loop over the other options.

## The do-while Loop

The do-while loop, often called the do loop, is a bit different than the previous loop constructs. This one is a *bottom-test* loop. Recall from the overview that this loop also differs from the other two in that we will always perform the work in the body of the loop once before applying the condition test.

```
import java.util.*;

public class menu
{
    static Scanner kb = new Scanner(System.in);

    public static void main(String[] args)
    {
        char choice;
        boolean valid;

        valid = false;
        do
        {
            System.out.println("\nPlease select a menu option.\n");

            System.out.println("Choices:");
            System.out.println("\tC) County Maps");
            System.out.println("\tM) Municipalities");
            System.out.println("\tT) Thermal Images");
            System.out.println("\tU) Utilities");
            System.out.println("\n\tQ) Quit");

            System.out.print("\n\nChoice: ");
            choice = Character.toUpperCase(kb.next().charAt(0));

            switch (choice)
            {
                case 'C': case 'M': case 'T': case 'U': case 'Q':
                    valid = true;
                    break;
                default:
                    System.out.println("The choice \"" + choice +
                                         "\" is not valid.");

                    break;
            }

        }
        while ( ! valid );
    }
}
```

*Example 3: Menu program using a do-while loop.*

The best example of the necessity of this type of loop is the menu selection program. We know we must display the menu choices once and read the menu selection from the user before we decide anything else.

One thing we could decide is that the user has chosen to quit the program through an appropriate menu choice. Another decision could be to redisplay the menu over and over until they make a *valid* selection. Let's look at the program of *Example 3*.

The first thing you should notice is the use of the `valid` boolean variable. This is used to turn our `do-while` loop into a flag-based loop. Since there is a significant amount of processing needed to determine the choice made and whether it is a valid choice, a `switch` statement has been added to simplify the use of a conditional statement to determine the validity of the value of `choice`.

The `default` clause of the `switch` statement handles error reporting. All of the `case` entries have been lumped together to treat them all equal as valid choices. If additional processing was needed beyond simple acceptance, it would likely be more appropriate to handle it in another piece of code. Remember that the intent of this is to identify a correct menu choice selection. Keeping the loop focused is essential.

A rather interesting concept to point out that it someone relevant to the validity processing is the following statement:

```
choice = Character.toUpperCase(kb.next().charAt(0));
```

With this one line of code, we read the next character from the input stream and then convert the character to upper case. Remember that `kb.next().charAt(0)` is how we read a character from the `Scanner` object and although it consumes a whole word, it is an acceptable sacrifice.

The reading of the character is nested inside a call to the method `toUpperCase()` from the `Character` wrapper class. This form of code writing is not uncommon and encouraged where appropriate.

By converting the character to upper case (or lower case if that is preferred) we do two things:

- Set the stage to accept either upper or lower case choices since they will be converted.
- If accepting both character cases is acceptable, we reduce the list of potential values to be tested by one half since we will never have to test the lower case possibilities.

The loop will continue to prompt the user with the complete menu and prompt for a choice for as long as they continue to make invalid selections.

## The `break` and `continue` statements

When dealing with loops, there are times that we wish we could either proceed to the next iteration or stop the loop immediately. We, of course, don't want to stop the program, just the loop we may be stuck in or otherwise no longer desire to be a part of.

The `break` and `continue` reserved words are actually used as complete statements. These statements allow us to do exactly what their names suggest:

- `break` out of a loop or skip the remainder of a `switch` statement.
- `continue` with the next iteration of the loop.

The `break` statement causes program execution to proceed after the end of the current compound statement block.

The `continue` statement causes the loop to proceed to the next iteration by forcing the loop to move to the condition test. For a `while` or `do-while` loop this means proceeding to the condition test. For the `for` loop, the update portion is done first, then the loop proceeds to the condition test.

## Nested Loops

The concept of placing a loop inside of another loop allows for the complete set of iterations of the inner loop for each iteration of the outer loop. Nested loops are often used for dealing with two- or three-dimensional data, but is not limited to those applications.

Consider *Example 4* while finds all of the prime numbers between 2 and 100. We will use the outer loop to process the actual numbers 2 through 100, and the inner loop will be used to find out which odd values in the range from 3 to the number-1 divide into the number.

The code in *Example 4* shows how the two loops are utilized to determine if the number is prime. Remember that a number is prime if a number is divisible only by itself and 1. We start by assuming that the number is prime until we find a value that divides evenly into `number`. The values of `range` is simply the set of numbers we will test with.

Although the compound statement is not required for the inner loop, it has been included for clarity. We also eliminate all even numbers with the `if` test prior to the entering the inner loop.

```

public class primes
{
    public static void main(String[] args)
    {
        int number, range;
        boolean prime;

        for ( number = 2; number <= 100; number++)
        {
            if ( number % 2 != 0 )
                prime = true;
            else
                prime = false;
            for ( range = 3; range < number && prime; range = range + 2 )
            {
                if ( number % range == 0 )
                    prime = false;
            }
            if ( prime )
                System.out.println(number);
        }
    }
}

```

*Example 4: Nested loop to find prime numbers from 2 to 100.*