COP 3330 – Object Oriented Programming in Java
Loops

While Loop

The while loop is nearly the same in every language. Do note that since Java has a boolean type and C doesn't, that would be the only observed difference between the two in those two languages. Python has a boolean type so this loop runs exactly the same in Java as it does in Python. Since it's rare to have a single statement in a while loop, we can present its most typical form:

```
while (boolean expression) {
    stmt1;
    stmt2;
    stmt3;
    ...
    stmtk;
}
```

We first evaluate the boolean expression. If it is true, we do all of the statements inside the block of statements from stmt1 through stmtk. This is identical to the if statement. BUT, after we complete executing the block of statements, we go back to evaluating the boolean expression in the while loop, which allows for repetition.

Typically speaking, while loops are good for situations where we want to continue executing the set of statements until something condition has been met, as opposed to a situation where we want to repeat things a fixed number of times.

Imagine a situation where we are collecting donations for individuals and our goal is $100. We don't know how many people we'll need to ask, but we know when we can stop asking people. Here is a code snippet for this situation:

```
Scanner stdin = new Scanner(System.in);

int money = 0;
while (money < 100) {
    System.out.println("How much money will you donate?");
    int donation = stdin.nextInt();
    money += donation;
}
```

This loop will continue to run until money < 100 is false, meaning until we get at least $100.

One standard situation to use the while loop is for controlling a menu driven program. A menu driven program gives uses choices to execute, and then once the user chooses, executes their choice before providing the menu again. This continues on and on, until the user decides to quit. On the next page is a framework for the main method of a user driven program:

```
public static void main(String[] args) {

    Scanner stdin = new Scanner(System.in);

    // Present menu choices via prints
    int choice = stdin.nextInt();

    while (choice != QUIT) {

        if (choice == 1) {
            // Execute choice 1
        }
        else if (choice == 2) {
            // Execute choice 2
        }
        ...
        else {
            // Print ERROR message choice must be in between
            // 1 and QUIT.
        }

        // Present menu choices via prints.
        choice = stdin.nextInt();
    }
}
```

Here we can see that we get the user's initial menu choice and then start our iteration (word for repetition). Within the loop, we have code corresponding to each valid choice, and also one option if the user made an invalid selection. Once we complete executing their desired choice within the loop we present the menu again and read in the user choice. This construct allows the user to make as many choices as they would like until they want to quit the program.

Included in the code examples are the following:

1. Donation Example
2. Bank Example to show a menu driven program
3. Half-life example (where # of atoms in a sample divides by 2 each time period)
4. Car Payment Example (showing payments to pay off a car loan)

For Loop

The for loop in Java and C are nearly the same (with the exception of C not having a boolean type), but the for loop in Java is different than the one in Python, in that Java allows for more flexibility in its loop than Python does. Still, in actual use, the two are fairly similar. (Most people don't utilize the flexibility that Java provides for for loops and instead use them just as they are used in Python.)

The general syntax of a for loop with a block of statements is this:

```
for (init stmt; boolean expression; inc stmt) {
    stmt1;
    stmt2;
    stmt3;
    ...
    stmtk;
}
```

This is equivalent to the following:

```
{
    init stmt;
    while (boolean expression) {
        stmt1;
        stmt2;
        stmt3;
        ...
        stmtk;
        inc stmt;
    }
}
```

The outer braces are only necessary due to a technicality: if the initialization statement declares a variable (which it usually does), then that variable's scope is only valid during the loop and that variable can NOT be used after the loop is completed.

But as this translation shows, here is what happens:

1. Execute the initial statement.
2. Evaluate the boolean expression.
3. If true, then execute all of the statements in the block, followed by the increment statement.
4. Go back to step 2.

This set up lends itself to being able to repeat a set of statements a fixed number of times:

```
for (int i=0; i<n; i++) {
    // stmts
}
```

Assuming that n has been declared as an integer before and stores a positive value, then this mold will repeat the statements inside of the block exactly n times, **provided that the value of i is NOT changed by the statements inside the loop.**

This is identical to C but different than Python. In Python, even if you change the value of a loop variable in the loop, it reverts back to as if you hadn't changed it, forcing the loop to run exactly n times no matter what. (This was a design likely to reduce inadvertent programming bugs, but one that is truly illogical, in my opinion.)

Here is a simple snippet of code where we gather donations from the number of people entered by the user:

```
Scanner stdin = new Scanner(System.in);

int money = 0;

System.out.println("How many people will be donating?");
int numpeople = stdin.nextInt();

for (int i=0; i<numpeople; i++) {
    System.out.println("How much is donation #"+(i+1)+"?");
    int donation = stdin.nextInt();
    money += donation;
}
```

Here is a code snipped from one more example which calculates an exponent:

```
Scanner stdin = new Scanner(System.in);

System.out.println("Enter x.");
int x = stdin.nextInt();
System.out.println("Enter n.");
int n = stdin.nextInt();

int ans = 1;
for (int i=1; i<=n; i++) {
    ans = ans*x;
}

System.out.println(x+" raised to the "+n+" equals "+ans);
```

do-while loop
While this loop is rarely used, it's a valid part of Java syntax, with the following syntax (shown with a block of statements again since it almost never appears with a single statement inside of it):

```
do {
    stmt1;
    stmt2;
    stmt3;
    ...
    stmtk;
} while (boolean expression);
```

This is the same as a regular while, except for the expression is evaluated at the end, instead of the beginning. This ensures that the loop runs at least once no matter what. This isn't true of either the while or for loop since it's possible that the boolean expression evaluated in the beginning isn't true.

You can use this for the menu driven program as follows:

```
public static void main(String[] args) {

    Scanner stdin = new Scanner(System.in);
    int choice = 0;

    do {
        // Present menu choices via prints
        choice = stdin.nextInt();

        if (choice == 1) {
            // Execute choice 1
        }
        else if (choice == 2) {
            // Execute choice 2
        }
        ...
        else {
            // Print ERROR message choice must be in between
            // 1 and QUIT.
        }
    } while (choice != QUIT);
}
```

The Bank program is included with this framework as Bank3.java

Nested Loops
Once we have loops and ifs, we can nest them in any structure we want. Thus, it's entirely possible to have a loop inside of a loop. Two examples are presented here:

1. Multiplication Table
2. Triangle of Stars

In a multiplication table, we have several rows, and each row has some number of entries in it. Here is a code snippet that does the main work for printing out a multiplication table:

```
for (int row_num=1; row_num<=rows; row_num++) {

    for (int col_num=1; col_num <= rows; col_num++)
        System.out.print("\t"+(row_num*col_num));

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

The outer loop goes through the rows. Then, for each row, we want to print rows number of entries. Each entry is just the 1-based row number times the 1-based column number. After completing a row, a new line has to be printed before we start printing the next row.

Here is a code snippet for the triangle of stars, which prints 1 star on the first row, 2 stars on the second row and so forth:

```
for (int linecnt = 1; linecnt <= numstars; linecnt++) {

    for (int starcnt = 1; starcnt <= linecnt; starcnt++) {
        System.out.print("*");
    }

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

In structure, this is nearly identical to the previous example, but the key difference is that the inner loop runs linecnt number of times, and linecnt itself is changing, so the first time the inner loop runs once, the second time it runs twice, etc.

Loop control: break

Sometimes, it's useful to be able to immediately exit a loop instead of waiting until the boolean expression at the top of the loop gets evaluated. This is exactly what a break statement does. As soon as it's encountered, it takes execution from the current point to the first statement outside of the nearest loop within which you were previously. For example, if we are checking if a number is prime or not, as soon as we find a divisor of a number, say the fact that 3 is a divisor of 105, then we don't have to look further; we can immediately report that 105 is composite (not prime). Here is a code snippet with a break statement that allows us to stop searching after we find a divisor of a number. Assume n was read in from the user before this code executes (and n is a positive integer 2 or greater.)

```
boolean isPrime = true;
for (int i=2; i*i<=n; i++) {
    if (n%i == 0) {
        isPrime = false;
        break;
    }
}
```

The first divisor to check is 2 and we use the mod operator to check divisibility. If n is ever divisible by i, then the if statement triggers, isPrime is set to false, and we immediately break out of the nearest loop, the for loop, so that we no longer check future values of i.

Loop control: continue

Finally, there is the continue statement. Sometimes, while we are in a loop, we don't want the loop itself to stop like the break situation, but we want to skip over the rest of that particular loop iteration. This is what the continue statement does. If the continue statement is executed, it takes control immediately back to the top of the loop within which it resides. Here is a code snippet from a program that takes the average of all valid test scores in between 0 and 100, inclusive. Any scores entered that are not in range are "skipped" via the continue statement:

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

    System.out.println("Enter the next score.");
    int score = stdin.nextInt();

    if (score<0 || score>100) {
        System.out.println("Sorry not in range, so not counted.");
        continue;
    }

    sum += score;
    valid++;
}
```

We only get to the last two statements when the score entered is valid. When it's not, we pop back up to the top of the for loop and go to process the next score.