VI. Repetition Control Structures

6.1 Objectives

In the previous sections, we have given examples of control structures such as if else conditional statements, ladderized if / elseif / else conditional statements and switch/case conditional statement. In this section, we will be discussing repetition control structures, which allows a program instruction that repeats some statement or sequence of statements in a specified number of times.

At the end of the lesson, the student should be able to:

• Use repetition control structures (while, do-while, for) which allow executing specific sections of code a number of times

• Use branching statements (break, continue, return) which allows redirection of program flow

6.2 Repetition Control Structures

Repetition control structures are Java statements that allows us to execute specific blocks of code a number of times. There are three types of repetition control structures, the while, do-while and for loops.

6.2.1 while loop

The while loop is a statement or block of statements that is repeated as long as some condition is satisfied.

The while statement has the form,

```
while( boolean_expression ){
statement1;
statement2;
. . .
}
```

The statements inside the while loop are executed as long as the boolean_expression evaluates to true.

For example, given the code snippet,

```
int i = 4;
while ( i > 0 ){
System.out.print(i);
i--;
}
```

The sample code shown will print 4321 on the screen. Take note that if the line containing the statement i--; is removed, this will result to an **infinite loop**, or a loop that does not terminate. Therefore, when using while loops or any kind of repetition control structures, make sure that you add some statements that will allow your loop to terminate at some point.

The following are other examples of while loops,

Example 1:

```
int x = 0;
while (x<10)
{
System.out.println(x);
x++;
}
```

Example 2:

```
//infinite loop
while(true)
System.out.println(“hello”);
```

Example 3:

```
//no loops
// statement is not even executed
while (false)
        System.out.println(“hello”);
```

6.2.2 do-while loop

The do-while loop is similar to the while-loop. The statements inside a do-while loop are executed several times as long as the condition is satisfied.

The main difference between a while and do-while loop is that, the statements inside a

do-while loop are executed **at least once.**

The do-while statement has the form,

```
do{
statement1;
statement2;
. . .
}while( boolean_expression );
```

The statements inside the do-while loop are first executed, and then the condition in the boolean_expression part is evaluated. If this evaluates to true, the statements inside the do-while loop are executed again.

Here are a few examples that uses the do-while loop:

Example 1:

```
int x = 0;
do
{
System.out.println(x);
x++;
}while (x<10);
```

This example will output 0123456789 on the screen.

Example 2:

```
//infinite loop
do{
System.out.println(“hello”);
} while (true);
```

This example will result to an infinite loop, that prints hello ***on screen.***

Example 3:

```
//one loop
// statement is executed once
do
System.out.println("hello");
while (false);
```

This example will output hello ***on the screen.***

Coding Guidelines:

1.Common programming mistakes when using the do-while loop is forgetting to write the semi-colon after the while expression.

```
do{
...
}while(boolean_expression) //WRONG->forgot semicolon ;
```

2. Just like in while loops, make sure that your do-while loops will terminate at some point.

6.2.3 for loop

The for loop, like the previous loops, allows execution of the same code a number of times.

The for loop has the form,

```
for (InitializationExpression; LoopCondition; StepExpression){
statement1;
statement2;
. . .
}
```

where,

InitializationExpression -initializes the loop variable.

LoopCondition **-** compares the loop variable to some limit value.

StepExpression **-** updates the loop variable.

A simple example of the for loop is,

```
int i;
for( i = 0; i < 10; i++ ){
System.out.print(i);
}
```

In this example, the statement i=0, first initializes our variable. After that, the condition expression i<10 is evaluated. If this evaluates to true, then the statement inside the for loop is executed. Next, the expression i++ is executed, and then the condition expression is again evaluated. This goes on and on, until the condition expression evaluates to false.

This example, is equivalent to the while loop shown below,

```
int i = 0;
while( i < 10 ){
System.out.print(i);
i++;
}
```

6.4 Branching Statements

Branching statements allows us to redirect the flow of program execution. Java offers three branching statements: break, continue and return.

6.4.1 break statement

The break statement has two forms: unlabeled (we saw its unlabeled form in the switch statement) and labeled.

6.4.1.1 Unlabeled break statement

The unlabeled break terminates the enclosing switch statement, and flow of control transfers to the statement immediately following the switch. You can also use the

unlabeled form of the break statement to terminate a for, while, or do-while loop.

For example,

```
String names[] = {"Beah", "Bianca", "Lance", "Belle",
"Nico", "Yza", "Gem", "Ethan"};
String searchName = "Yza";
boolean foundName = false;
for( int i=0; i< names.length; i++ ){
if( names[i].equals( searchName )){
foundName = true;
break;
}
}
if( foundName ){
System.out.println( searchName + " found!" );
}
else{
System.out.println( searchName + " not found." );
}
```

In this example, if the search string "Yza" is found, the for loop will stop and flow of control transfers to the statement following the for loop.

6.4.1.2 Labeled break statement

The labeled form of a break statement terminates an outer statement, which is identified by the label specified in the break statement. The following program searches for a value in a two-dimensional array. Two nested for loops traverse the array. When the value is found, a labeled break terminates the statement labeled search, which is the outer for loop.

```
int[][] numbers = {{1, 2, 3}, {4, 5, 6},
{7, 8, 9}};
int searchNum = 5;
boolean foundNum = false;
searchLabel:
for( int i=0; i<numbers.length; i++ ){
for( int j=0; j<numbers[i].length; j++ ){
```

```
if( searchNum == numbers[i][j] ){
foundNum = true;
break searchLabel;
}
}
}
if( foundNum ){
System.out.println( searchNum + " found!" );
}
else{
System.out.println( searchNum + " not found!" );
}
```

The break statement terminates the labeled statement; it does not transfer the flow of control to the label. The flow of control transfers to the statement immediately following the labeled (terminated) statement.

6.4.2 continue statement

The continue statement has two forms: unlabeled and labeled. You can use the continue statement to skip the current iteration of a for, while or do-while loop.

6.4.2.1 Unlabeled continue statement

The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop, basically skipping the remainder of this iteration of the loop.

The following example counts the number of "Beah"s in the array.

```
String names[] = {"Beah", "Bianca", "Lance", "Beah"};
int count = 0;
for( int i=0; i<names.length; i++ ){
if( !names[i].equals("Beah") ){
continue; //skip next statement
}
count++;
}
```

```
System.out.println("There are " + count + " Beahs in the
list");
```

5.4.2.2 Labeled continue statement

The labeled form of the continue statement skips the current iteration of an outer loop marked with the given label.

outerLoop:

```
for( int i=0; i<5; i++ ){
for( int j=0; j<5; j++ ){
System.out.println("Inside for(j) loop"); //message1
if( j == 2 ) continue outerLoop;
}
System.out.println("Inside for(i) loop"); //message2
}
```

In this example, message 2 never gets printed since we have the statement continue outerloop which skips the iteration.

6.4.3 return statement

The return statement is used to exit from the current method. The flow of control returns to the statement that follows the original method call. The return statement has two forms: one that returns a value and one that doesn't.

To return a value, simply put the value (or an expression that calculates the value) after the return keyword. For example,

```
return ++count;
or
return "Hello";
```

The data type of the value returned by return must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value. For example,

```
return;
```

REFERENCES:

1. Java Programming: From Problem Analysis to Program Design - Third Edition(2016)
   By: Blayne Mayfield

2. Programming Language. From Wikipedia at http://en.wikipedia.org/wiki/Programming_language

3. Gary B. Shelly, Thomas J. Cashman, Joy L. Starks. Java Programming Complete Concepts and Techniques. Course Technology Thomson Learning. 2001

4. Simple Program Logic Formulation(2016)
   By: Jerald Herrera De la Rosa