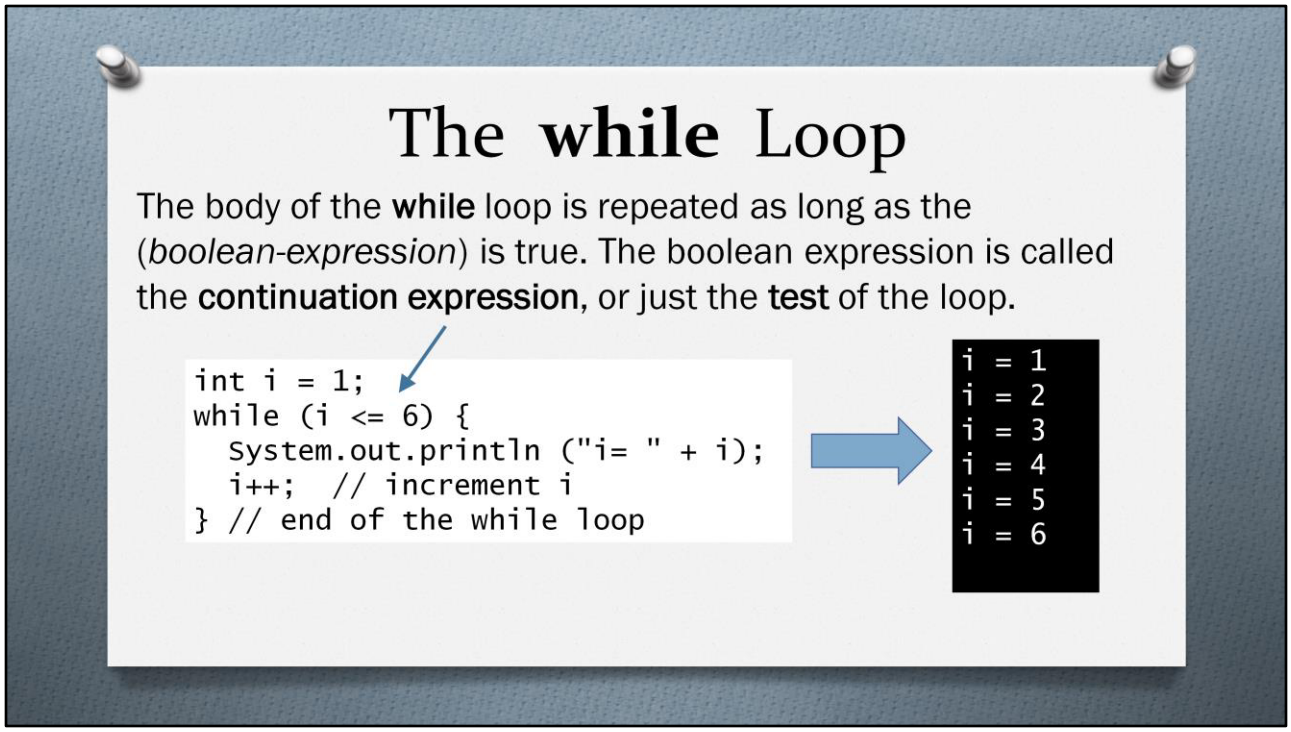
Hello programmers. A big giant welcome to the wonderful world of Java Loops!

Topics in the Discussion

- The basic **while** loop has a test at the top of the loop
- The **do...while** loop has a test at the bottom of the loop
- The **for** loop is just a fancy **while** loop - good for counting
- The **for each** loop is a fancy **for** loop good for arrays and collections

This discussion covers the basic while loop that has its test for loping at the top and the do-while loop with a test at the bottom of the loop. The for loop with its two varieties is also covered.



Let's get started with the while loop. The while loop starts off with the word **while** followed by a boolean expression inside a set of parentheses that must evaluate to either true or false. As long as the expression evaluates true, the body of the loop is executed. Although a single statement can follow the while and boolean within the parentheses, it is most common to have the body of the loop enclosed within a block of curly braces.

In this example, the variable i is initialized to 1. The while loop will execute the body of the loop as long as i is less than or equal to 6. The body of the loop has two statements, a println outputs the value of i. The second statement increments i so that i starts at 1 and increments all the way to 7. When i is a 7, the boolean expression for the while becomes a false and the body of the loop is no longer executed.

Simple Loops

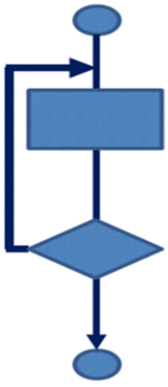
while

```
int i = 0;
while ( i < 5 ) {
   System.out.println (i);
   i++;
}
System.out.println ("Done“ );
```

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

The test for looping is done at the top of the loop. Code may not be executed if the loop condition fails.

do . . . while



```
int i = 0;
do {
   System.out.println (i);
   i++;
} while ( i < 5 ) ;
System.out.println ("Done“ );
```

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

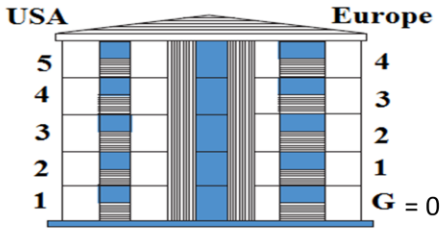
The test for looping is done at the bottom of the loop. Code will be executed at least one time.

Here are flowcharts of the while and do-while loops. So far, the results for either loop are the same. They each display the numbers 0 through 4. When the loop ends, a println statement shows "Done".

The test condition for the while loop is at the top of the loop but the test condition for the do-while loop is at the bottom of the loop. As we will see, there will be times where this can make a big difference. One more thing to look at in these examples is that i starts at 0 and the loop continues as long as i is less than 5. There are 5 passes through the loop starting at 0. 0 - 1 - 2 - 3 - 4 It is very common in programming to start counting from zero. If you are new to programming, this is something to get used to.

Counters Start from Zero

It is most common to start counting from zero because the first index (position) in an array starts with zero.



This may be confusing for most Americans who count the ground floor of a building as the 1st floor. But for the rest of the world, the 1st floor is the one that is above the ground floor.

It is most common to start counting from zero because the first index (position) in an array starts with zero.

This may be confusing for most Americans who count the ground floor of a building as the 1st floor. But for the rest of the world, the 1st floor is the one that is above the ground floor.

Repetition Without Loops

```
// Input and add three numbers (this is a code fragment)
int total = 0;
int number;
System.out.print ("Enter a number: ");  // prompt
number = stdin.nextInt( );              // input from keyboard
total += number;                        // add to total
System.out.print ("Enter a number: ");  // prompt
number = stdin.nextInt( );              // input from keyboard
total += number;                        // add to total
System.out.print ("Enter a number: ");  // prompt
number = stdin.nextInt( );              // input from keyboard
total += number;                        // add to total
System.out.println ("The total is + total);
```

Here is a program fragment. It only a piece of a full program. It is missing all the extra things that are needed to make a program run. If you wanted to run this program fragment, you would need to add any needed import statements, the class statement, the void main statement and any of their open and close curly-braces. Also needed are the statements to create the Scanner object named stdin.

This fragment inputs three numbers, one at a time, from the user and adds them into the variable total. After all three numbers have been processed, the total is displayed with a println statement. Writing a program this way is OK as long as there are not too many numbers to add up. Just think of all the code that would need to be written to input and add 20 numbers, or even 100. Another problem with coding this way is that the number of inputs may change causing a major rewrite of the program.

Repetition With at Loop

```
// Input and add three numbers
int total = 0;
int number;
int i = 0;
while (i < 3) // counting from 0
{
   System.out.print ("Enter a number: ");    // prompt
   number = stdin.nextInt( );                // input
   total += number;                          // add
   i++;                                      // next
}
System.out.println ("The total is + total);
```

Now we have the same result that uses a loop. A new variable, i has been included which is used to control the loop. The loop control counts from 0 to less than 3 which causes the program to loop three times. 0 - 1 - 2 When i becomes a 3, the loop ends without executing the body of the loop.

This time, the prompt, input from keyboard and add onto the total are only needed one time. If it is ever necessary to add a different numb of inputs than 3, the only thing that needs changing is the number 3.

Simple Loops

While

No loop body if i is 10

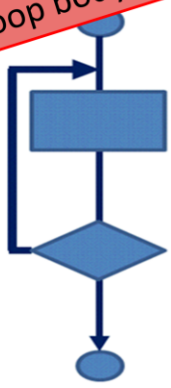
```
int i = 10;
while ( i < 5 ) {
   System.out.println (i);
   i++;
}
System.out.println ("Done“ );
```

```
Done
```

The test for looping is done at the top of the loop. Code may not be executed if the loop condition fails.

Do

Loop body executes one time



```
int i = 10;
do {
   System.out.println (i);
   i++;
} while ( i < 5 ) ;
System.out.println ("Done“ );
```

```
10
Done
```

The test for looping is done at the bottom of the loop. Code will be executed at least one time.

In these examples, the value in i is set to 10. The boolean expression for the loop control is testing for a value of less than 5 which is going to be false because 10 is not less than 5. Since the test is done at the top of the while loop, the body of the loop is never executed. The println statement after the loop is executed which displays "Done" on the output terminal.

The do-while loop always executes the code in the body at least one time because the loop test is at the bottom. In this case, the println statement in the body displays a 10, the test at the bottom fails and we don't execute the body of the loop a second time. The println statement after the loop displays "Done"

The Sum of Input Values


```
int  value = 0;  // value from keyboard
int  total = 0;  // initialize the total
int  i =  0;     // initialize loop counter
while  (i < 5) { // get 5 values
  System.out.print ("Enter a number: "); // prompt
  value = stdin.nextInt(); // input from keyboard
  total += value; //  total = total + value;
  i++;           // increment loop counter
}
System.out.println ("The total is + total);
```

This program fragment is very similar to the ones already discussed. This time 5 integers are going to be read from the keyboard and added to a total and when the loop ends, the total is displayed.

Let's look closely at this program.

Be Careful of the Semi-colon


```
int  value = 0;    // value from keyboard
int  total = 0;    // initialize the total
int  i =  0;       // initialize loop counter
while (i < 5)  {   // get 5 values
  System.out.print ("Enter a number: "); // prompt
  value = stdin.nextInt(); // input from keyboard
  total += value; //  total = total + value;
  i++;             // increment loop
}
System.out.println ("T                  total);
```

No Semi-colon here

It is very important to not put a semicolon ; character after after the closing parentheses) of the test condition or even after the open curly-brace.

Sample Execution

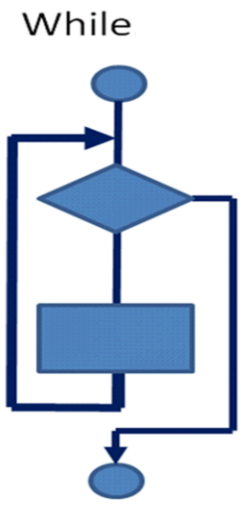

```
int  value = 0;  // value from keyboard
int  total = 0;  // initialize the total
int  i =  0;     // initialize loop counter
while  (i < 5) { // get 5 values
  System.out.print ("Enter a number: "); // prompt
  value = stdin.nextInt(); // input from keyboard
  total += value; //  total = total + value;
  i++;            // increment loop counter
}
System.out.println ("The total is " + total);
```

i =	0 1 2 3 4	5
	5 times in the loop	Done

The body of the loop is executed 5 times. When i becomes a 5, we fall out of the loop and print the total.

Just for the discussion, suppose that the user entered the following numbers **12 23 9 -6 37.** If these numbers were entered, the total would end up being 75.

Sample Execution




```
int  value = 0;  // value from keyboard
int  total = 0;  // initialize the total
int  i =  0;     // initialize loop counter
while  (i < 5) { // get 5 values
  System.out.print ("Enter a number: "); // prompt
  value = stdin.nextInt(); // input from keyboard
  total += value; //  total = total + value;
  i++;            // increment loop counter
}
System.out.println ("The total is " + total);
```

i =	0 at start of loop
i<5 (y/n)	y then execute the body
value =	12 from the user
total =	12
i++ =	1 at end of the loop

Let's look closely to see what happens as the program progresses. The initial state of total is a 0 and i is a 0. Since 0 is less than 5, the body of the loop is executed.

For example, a 12 is read from the keyboard and added to the total which becomes 12. Then i is incremented using i++, causing i to become a 1.

Sample Execution

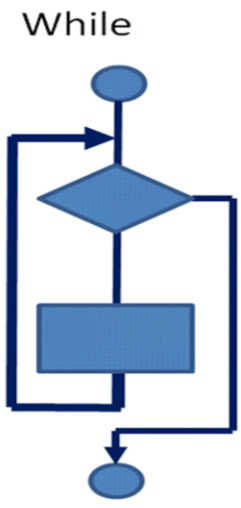

```
int  value = 0;  // value from keyboard
int  total = 0;  // initialize the total
int  i =  0;     // initialize loop counter
while  (i < 5) { // get 5 values
  System.out.print ("Enter a number: "); // prompt
  value = stdin.nextInt(); // input from keyboard
  total += value; //  total = total + value;
  i++;            // increment loop counter
}
System.out.println ("The total is " + total);
```

i =	0	1
i<5 (y/n)	y	y
value =	12	23
total =	12	35
i++ =	1	2

When starting the second pass, i is now a 1 which is less than 5, therefore the body of the loop is executed.

This time, suppose we read a 23 from the keyboard, so the total becomes 35. The last thing that happens is that i is incremented and becomes a 2.

Sample Execution

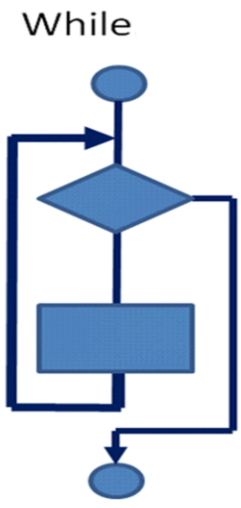



```
int  value = 0;  // value from keyboard
int  total = 0;  // initialize the total
int  i =  0;     // initialize loop counter
while  (i < 5) { // get 5 values
  System.out.print ("Enter a number: "); // prompt
  value = stdin.nextInt(); // input from keyboard
  total += value; //  total = total + value;
  i++;            // increment loop counter
}
System.out.println ("The total is " + total);
```

i =	0	1	2
i<5 (y/n)	y	y	y
value =	12	23	9
total =	12	35	44
i++ =	1	2	3

Third time's a charm. i is a 2, which is less than 5 so we do the body of the loop again. The print routine asks the user to enter a number. The nextInt method reads an integer from the keyboard which for our discussion is a 9. The 9 is added to the total which becomes 44. i is incremented using i++ and it becomes 3.

Sample Execution




```
int  value = 0;  // value from keyboard
int  total = 0;  // initialize the total
int  i =  0;     // initialize loop counter
while  (i < 5) { // get 5 values
  System.out.print ("Enter a number: "); // prompt
  value = stdin.nextInt(); // input from keyboard
  total += value; //  total = total + value;
  i++;            // increment loop counter
}
System.out.println ("The total is " + total);
```

i =	0	1	2	3	
i<5 (y/n)	y	y	y	y	
value =	12	23	9	-6	
total =	12	35	44	38	
i++ =	1	2	3	4	

On the fourth pass through the loop, i starts out at 3 - remember we are counting from zero. Since 3 is less than 5, we execute the body of the loop again. This time, the user input is -6. We are not rejecting negative numbers, so the -6 gets added to the total which changes from 44 to 38. i is incremented to 4.

Sample Execution


```
int  value = 0;  // value from keyboard
int  total = 0;  // initialize the total
int  i =  0;     // initialize loop counter
while  (i < 5) { // get 5 values
  System.out.print ("Enter a number: "); // prompt
  value = stdin.nextInt(); // input from keyboard
  total += value; //  total = total + value;
  i++;            // increment loop counter
}
System.out.println ("The total is " + total);
```

i =	0	1	2	3	4	
i<5 (y/n)	y	y	y	y	y	
value =	12	23	9	-6	37	
total =	12	35	44	38	75	
i++ =	1	2	3	4	5	

i starts out at 4 which is still less than 5. In the body of the loop we ask for and read the next number. In our example, the value read is 37. It is added to the current value in total of 38. The new value in total is 75. i is incremented to be a 5.

Sample Execution


```
int  value = 0;  // value from keyboard
int  total = 0;  // initialize the total
int  i =  0;     // initialize loop counter
while  (i < 5) { // get 5 values
  System.out.print ("Enter a number: "); // prompt
  value = stdin.nextInt(); // input from keyboard
  total += value; //  total = total + value;
  i++;            // increment loop counter
}
System.out.println ("The total is " + total);
```

i =	0	1	2	3	4	5
i<5 (y/n)	y	y	y	y	y	n
value =	12	23	9	-6	37	75
total =	12	35	44	38	75	
i++ =	1	2	3	4	5	

This time, i is a 5 at the start of the loop. We only do the body of the loop if i is less than 5, which it is not, so we exit the loop with i still equal to 5 and the total is 75. We fall out of the loop. The println statements displays, "The total is 75"

In summary, we went through the loop 5 times with i going from 0 through 4. Once i reached 5, the loop ended.

The **do...while** Loop

The **do...while** loop has the test condition at the end. The loop starts with the word **do**. The **do...while** loop has the following form:

```
do {
   statements;
while ( boolean-expression );
```

Look at the semicolon at the end. The semicolon is required. It ends the **do...while** loop.

The do-while loop has the test condition at the bottom of the loop. This loop structure is very common when the decision is not known in advance as to the number of times to go through the loop.

Coin Flip Game (it cheats)

The coin flip game asks the user to press [Enter] to start the game. The computer *flips* a coin. If the coin comes up heads, the computer claims it wins. If the coin comes up tails, it claims the player looses.

The code contains a **do...while** loop. At the end of the loop the program asks the user,

"Do you want to play again(Y/N)?"

The boolean variable **playAgain** is set **true** is the user presses [Enter] or responds with anything that starts with "Y". The **while** part of the **do...while** loop returns control back to the start of the loop if **playAgain** is **true.**

The coinFlip game has a question at the bottom of the loop, "Do you want to play again (Y/N)? "

We could play the game one time or a hundred times, anything in between or even more. Watch out for this game. The way it is written, it cheats. The computer simulates *flipping* a coin. If the coin comes up heads, the computer claims it wins. If the coin comes up tails, it claims the player loses. Then it asks if the user wants to play again.

The boolean variable **playAgain** is set **true** is the user presses [Enter] or responds with anything that starts with "Y". The **while** part of the **do...while** loop returns control back to the start of the loop if **playAgain** is **true.**

```
/*
     The program simulates the flipping of a coin and always wins by cheating.
 */
package coinflip;
import java.util.Scanner;

public class CoinFlip {
    public static void main(String[] args) {
        // create the variables and the Scanner object
        Scanner stdin = new Scanner(System.in);
        boolean playAgain;

        do {
            System.out.println ("Heads I win. Tails you lose");

            boolean coin = Math.random( ) >= 0.5; // random number >= 0.5 --> true
            if (coin == true)
                System.out.println ("It was heads. I win");
            else
                System.out.println ("It was tails. You lose");

            System.out.print ("\nDo you want to play again (Y/N)? ");
            String response = stdin.nextLine().toUpperCase(); // big or small N
            playAgain = response.length() == 0 || response.charAt(0) == 'Y';
        } while (playAgain);  // accept [Enter] or anything that starts with 'Y'

    } // end of void main
} // end of class
```

Line up the curly-braces for the do and the while. Everything within this block of code is considered the body of the do-while loop. It will be executed at least one time and more times when the test condition of the while is true.

The program creates a boolean variable named playAgain. This will be used at the bottom of the program within the while statement to determine whether the loop that contains the game should be executed again. Since this is a do-while loop, the game will be played at least once when the program starts.

Another boolean variable named coin is created. Remember that a boolean variable can only hold a value of true or false. The result of the comparison expression greater-than-or-equal gives a boolean true or. The first part of the greater-than-or equal is a call to the Math.random( ) method which in Java returns a value from 0.0 to just less than 1.0 By comparing the result Math.random( ) >= 0.5; we can let half the numbers from 0.5 up to 1.0 result in true, while anything below 0.5 results in false. So half the time the coin value is true and half the time the coin value is false. I will be using true to represent HEADS and false represent TAILS. When coin is true, the program claims it won because it always wins with heads. The else condition happens when coin is false, indicating tails. When this happens, the program says that you lose when the coin is tails. This game really cheats. You just can't win.

Rather than the cheating game. what I really want to look at though is how the while

loop can be used to determine whether or not to play the game again. The while part of do-while needs to test a boolean expression for true or false. When true, the loop is executed again from the top of the loop. We need to set up the playAgain variable so that it can be tested by the while statement.

The program asks the user, "Do you want to play again (Y/N)? " and then reads the keyboard into a String variable named response. See how the String variable named response is created just where it is needed instead of at the top of the program with the other variables.

Not only is the String variable response created at this point but it is also assigned everything that was typed on the keyboard. The nextLine( ) method returns a String containing all of the characters that were typed at the keyboard.

Even more impressive, the .toUpperCase( ) method is called to convert everything typed to upper case capital letters before being assigned into the String variable response. I wanted to convert what was typed to upper case so that it will not matter if the user types big or small letters.

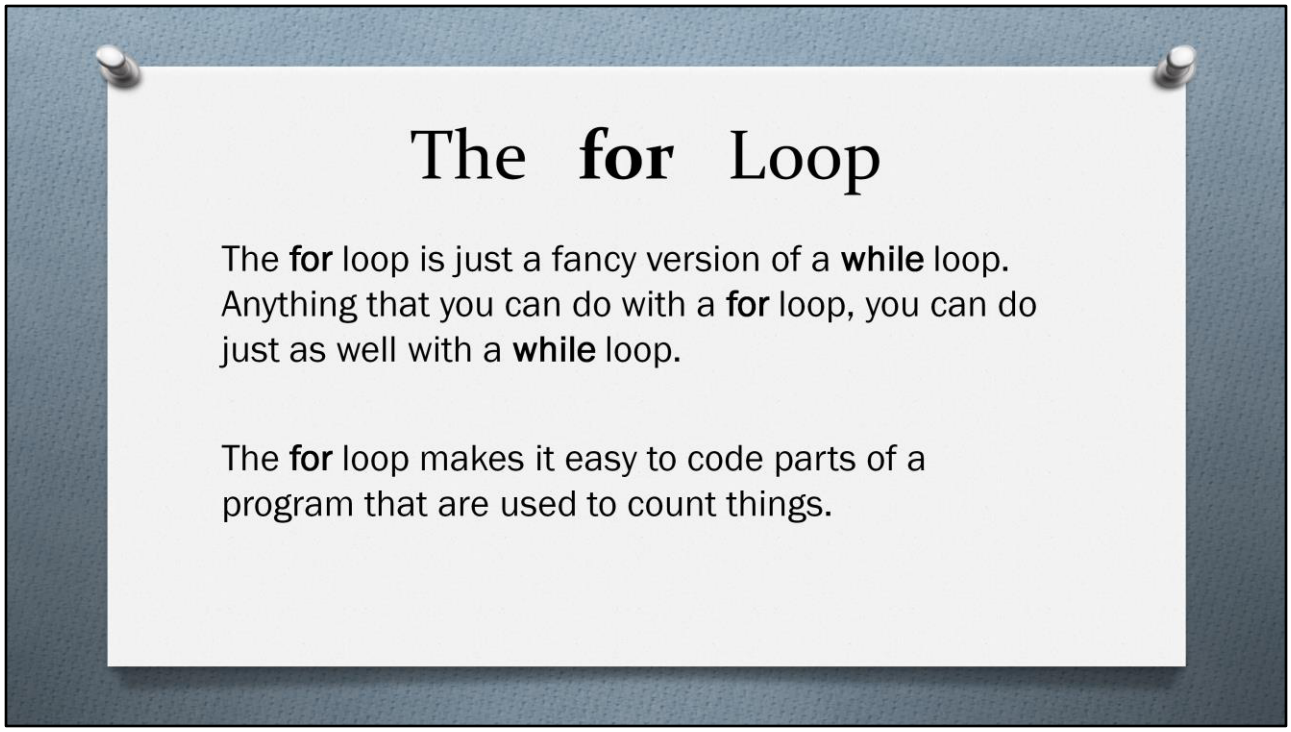
Now, look at what is placed into the boolean playAgain variable.
playAgain = response.length == 0 || response.charAt(0) == 'Y';

I wanted to be very user friendly and accept user inputs such as "Y", or "y", or "Yes", or "yes", or "yup", or "yada yada", or even pressing the [Enter] key with nothing else. Any of these will cause the program to loop and play the game again. I am going to place some additional parentheses in this expression just to make it easier to read and understand. Although parentheses are not necessary because of the order in which the expression is evaluated, sometimes it would be nice to have them there to make the program easier to understand for the next person who looks at it.

playAgain = ((response.length==0) || (response.charAt(0)=='Y'));

If somebody just pressed the [Enter] key without any other characters then the length would be 0. I will be nice and let just the [Enter] key being pressed to loop back and play the game again.

The two vertical bars || are the logical-OR operator in Java, C, C++, etc. I am going to let either just the [Enter] key or anything typed that starts with the letter 'Y' cause the game to be played again. Remember that response is the String variable that receives the characters from the keyboard, so response.charAt(0) gives us only the first character - position 0.



The **for** loop is just a fancy version of a **while** loop. Anything that you can do with a **for** loop, you can do just as well with a **while** loop.

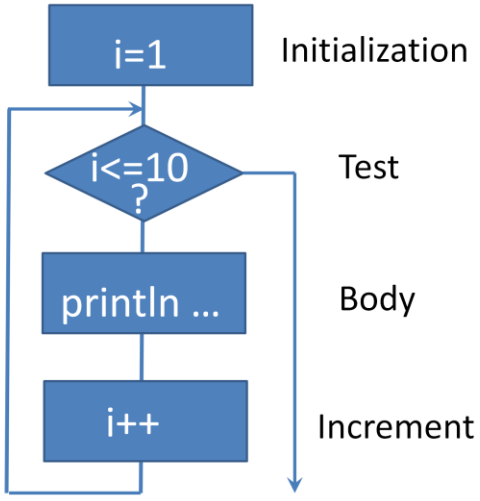
The **for** loop makes it easy to code parts of a program that are used to count things.

The **for** loop

The **for** loop in Java is a fancy **while** loop.

```
// display 1 through 10
int i;
i=1;
while (i<=10) {
  System.out.println(i);
  i++;
}
```

```
int i;
for (i=1; i<=10; i++) {
   System.out.println(i);
}
```

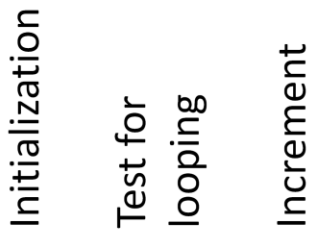


Here is a comparison between the for loop and the while loop when used to go through the loop a specified number of times.

The variable i is declared as an integer. i is initialized to 1 before the loop even starts. The first thing that happens in the loop is the test to see if the body of the loop is to be executed. If the test evaluates to true, then the body is executed. The last thing inside the loop is the increment, i++

The **for** loop

The **for** loop has three parts:




```
for (i=0; i<10; i++)
```

The initialization happens only one time and occurs before the loop even starts.

The increment happens at the bottom of the loop, even though it shows up on the same line as the for statement.

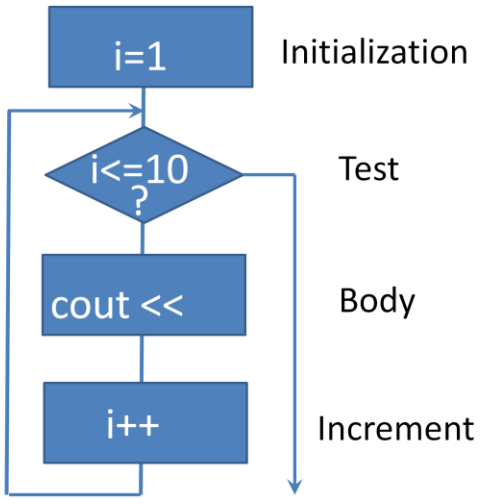
The initialization happens only one time and occurs before the loop even starts.

The increment happens at the bottom of the loop, even though it shows up on the same line as the for statement.

The parts of the **for** loop

```
// display 1 through 10
int i;
for (i=1; i<=10; i++) {
   System.out.println(i);
}
```

The increment always happens after the body of the loop is executed.



The increment always happens after the body of the loop is executed.

for With a Single Statement

The curly-braces are not needed

```
// display 1 through 10
int i;
for (i=1; i<=10; i++) {
   System.out.println(i);
}
```

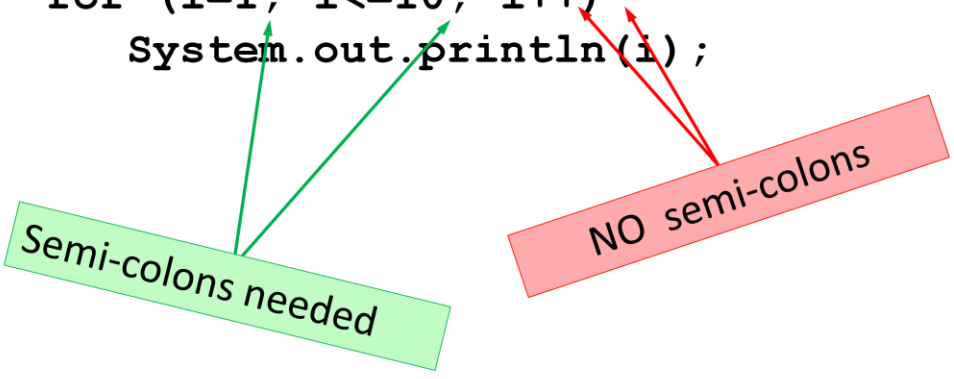
is the same as:

```
// display 1 through 10
int i;
for (i=1; i<=10; i++)
   System.out.println(i);
```

If there is only one statement for the body of the loop, the curly-braces can be eliminated.

Watch the semi-colons

```
int i;
for (i=1; i<=10; i++)
    System.out.println(i);
```



Watch out for the placement of the semicolons in the for statement. A semicolon is needed after the initialization statement and after the test statement, but not after the increment or after the closing parentheses.

for Statement on a Single Line

```
int i;
for (i=1; i<=10; i++)
  System.out.println(i);

int i;
for (i=1; i<=10; i++) System.out.println(i);

// declare and initialize i within the for
for (int i=1; i<=10; i++) System.out.println(i);
```

The System.out.println is a continuation of the for statement, and could even be placed on the same line.

Since C, C++ and Java are fairly free form when it comes to writing code, and a statement ends with a semicolon or closing curly-brace, everything could be placed on the same line if there is only one statement for the body of the loop.

Even the declaration of the variable i can be placed within the for statement.

Declare Variable Within the for Statement

```
for (int i=1; i<=10; i++)
    System.out.println(i);
```

This code declares a variable within the initialization expression. The variable **i** only exists within the for loop and is gone as soon as the loop ends. If **i** is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names **i**, **j**, and **k** are often used to control for loops. Declaring them within the initialization expression limits their life span and reduces errors.

This code declares a variable within the initialization expression. The variable **i** only exists within the for loop and is gone as soon as the loop ends. If **i** is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names **i**, **j**, and **k** are often used to control for loops, Declaring them within the initialization expression limits their life span and reduces errors.

Bad Semi-colon

```
int i;
for (i=1; i<=10; i++)
   System.out.println(i);
```

Very good. No semi-colon at the end of the line with the **for** statement.

```
int i;
for (i=1; i<=10; i++);
   System.out.println(i);
```

Bad semi-colon.
BAD SEMI-COLON !

Watch out! Do not put a semicolon at the end of the line with the for statement.

Bad Semi-colon

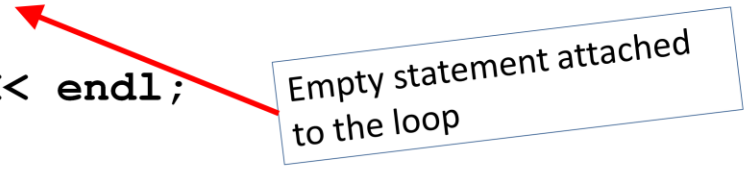
```
int i;
for (i=1; i<=10; i++) ;
  cout << i << endl;
```

This program will execute the **for loop** with an empty statement ten times and then execute the cout statement only once, even though it is indented and looks like it would be executed ten times.

If you put a semicolon at the end of the line with the for statement, you have accidently created an empty statement that becomes the body of the loop. It will be executed as many times as is programmed in the for statement and then control moves on to the next statement which will only be executed one time not the number of times you may have wanted, thinking it was the body of the for loop.

while Statement Problem

```
int i = 1;
while (i<=10) ;
{
   cout << i << endl;
   i++;
}
```



This program will execute the for loop forever. It is called an infinite loop. The while loop has no body attached. The **i++** statement is never executed and **i** never reaches the value of 10.

This one is ever more dangerous with an unexpected semicolon after the while statement. Although it appears that there is a block of code that looks like it is the body of the while statement, the actual body of the while statement is the empty statement accidently created by the semicolon.

The while statement will continue to be executed as long as i is less than 10. The i++ is not part of the body of the loop, i never gets incremented and never changes from being a 1. The while statement will execute the dummy empty statement ending with the semicolon forever, or until the program is killed.

The entire AT&T long distance network went down one day because of a problem similar to this. It took several hours to correct. Another carrier named Sprint was telling everyone to switch to Sprint because they were still up and running. They crashed a few months later with egg on their faces too.

The **enhanced for** Loop

The for statement also has another form designed for iteration through Collections and arrays This form is sometimes referred to as the ***enhanced for*** or the ***for each*** statement, and can be used to make your loops more compact and easy to read.

Java has an enhanced version of the for loop that is not available in C or C++. Sometimes it is referred to as for-each. This enhanced version is good for indexing through an array or collection, one element at a time until all elements are processed. The enhanced for statement can make loops more compact and easier to read when working with arrays or collections.

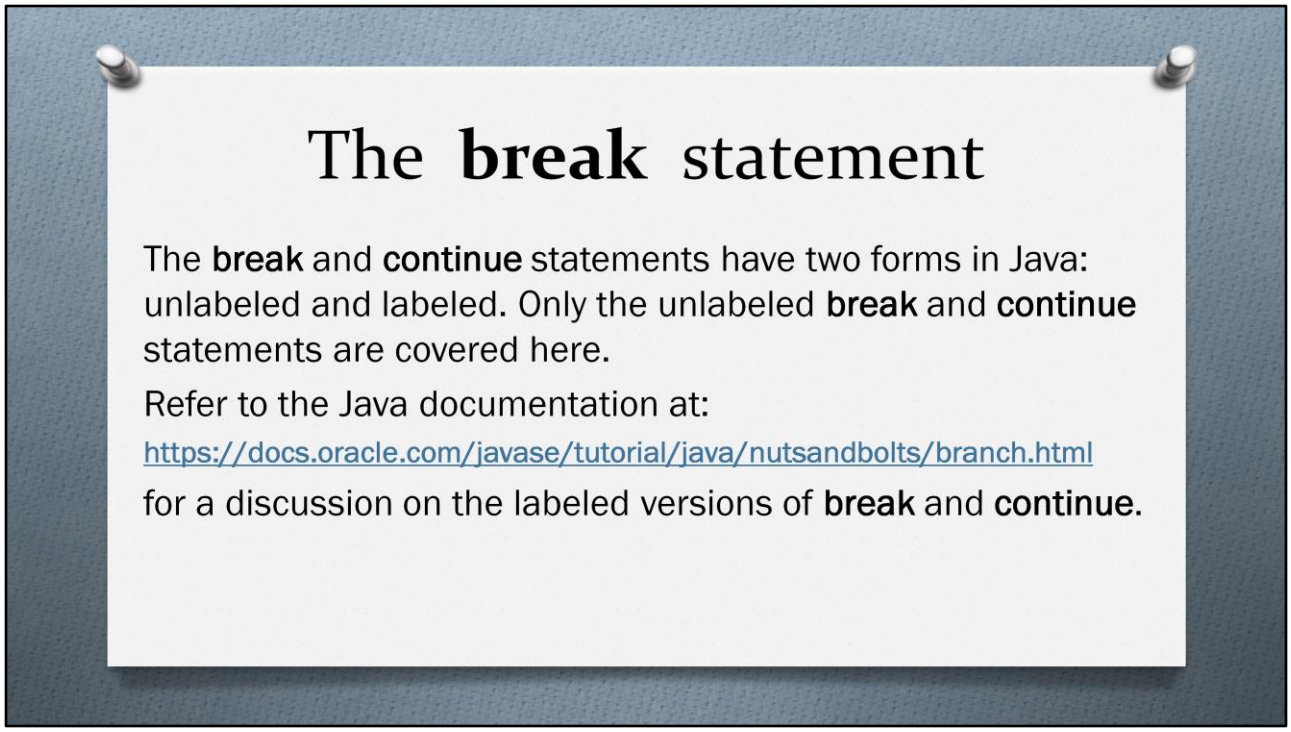
enhanced for (from Oracle)

```
class EnhancedForDemo {
    public static void main(String[] args)  {
        int[ ] numbers = {1,2,3,4,5,6,7,8,9,10};
        for (int item : numbers) {
            System.out.println("Count is: " + item);
        } // end of for
     }  // end of main
 } // end of class
```

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Use this form of the for statement instead of the general form whenever possible.

Here is an example of the enhanced for when used on an array of integers that contain the numbers 1 through 10. Actually, the name of the array is numbers. The for loop automatically steps through the array and places each value, one at a time, into the variable named item. item is declared as part of the for statement.



The break and continue statements can modify the order of how instructions are executed from within the body of the loop.

The **break** and **continue** statements have two forms in Java: unlabeled and labeled. Only the unlabeled **break** and **continue** statements are covered here.
Refer to the Java documentation at:
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html
for a discussion on the labeled versions of **break** and **continue**.

the **break** statement (from Oracle)

The **break** statement can be used in the **switch**, **for**, **while** or **do-while** statements.

In the example below, the break statement causes execution of the program to terminate the **switch**, **for**, **while** or **do-while** block of code.

The **break** statement can be used in the **switch**, **for**, **while** or **do-while** statements. In the example below, the break statement causes execution of the program to terminate the **switch**, **for**, **while** or **do-while** block of code.

```
int[] arrayOfInts =
    { 32, 87, 3, 589,
      12, 1076, 2000,
      8, 622, 127 };
int searchfor = 12;

int i;
boolean foundIt = false;

for (i = 0; i < arrayOfInts.length; i++) {
    if (arrayOfInts[i] == searchfor) {
        foundIt = true;
        break;
    }
}

if (foundIt) {
    System.out.println("Found " + searchfor + " at index " + i);
} else {
    System.out.println(searchfor + " not in the array");
}
```

This program searches for the number 12 in an array. The break statement, shown in boldface, terminates the for loop when that value is found. Control flow then transfers to the statement after the for loop. This program's output is:

Found 12 at index 4

This program searches for the number 12 in an array. The break statement, shown in boldface, terminates the for loop when that value is found. Control flow then transfers to the statement after the for loop. This program's output is:

Found 12 at index 4

The **continue** statement

The **continue** statement skips the current iteration of a **for**, **while** , or **do-while** loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop. The following program, ContinueDemo , steps through a String, counting the occurrences of the letter "p". If the current character is not a p, the **continue** statement skips the rest of the loop and proceeds to the next character. If it ***is*** a "p", the program increments the letter count.

The **continue** statement skips the current iteration of a **for**, **while** , or **do-while** loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop. The following program, ContinueDemo , steps through a String, counting the occurrences of the letter "p". If the current character is not a p, the **continue** statement skips the rest of the loop and proceeds to the next character. If it  "p", the program increments the letter count.

```
String searchMe = "peter piper picked a " + "peck of pickled peppers";
int max = searchMe.length();
int numPs = 0;

for (int i = 0; i < max; i++) {
    // interested only in p's
    if (searchMe.charAt(i) != 'p')
        continue;

    // process p's
    numPs++;
}
System.out.println("Found " + numPs + " p's in the string.");
```

Here is the output of this program:

Found 9 p's in the string.

To see this effect more clearly, try removing the continue statement and recompiling.

When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

Here is the output of this program:

Found 9 p's in the string.

To see this effect more clearly, try removing the continue statement and recompiling.

When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

I hope you find this discussion of Java loops informative and useful. I realize that there is a lot of information covered here, especially if this is your first time looking at loops. If you feel slightly overwhelmed, look it over again the next day, write some code that uses loops and maybe look over this discussion one more time.

So, as Roy Rogers and Dale Evens would sing,
"Happy trails to you until we meet again"

Bye bye