Lecture 14

'for' loops and Arrays

For Loops

```
for (initiating statement; conditional statement; next statement)
                                                  // usually incremental
{
    body statement(s);
}
```

The **for** statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied.
The general form of the for statement can be expressed as above.

Yay! You can use this instead of a counting-while-loop!!

For Loops

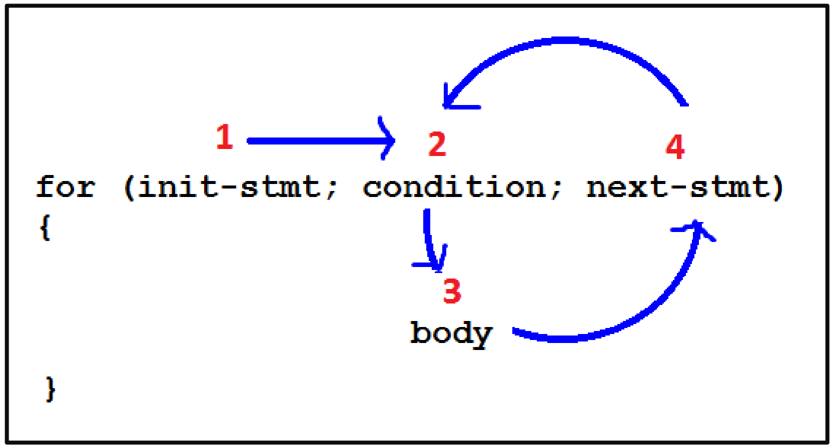
```
for (initiating statement; conditional statement; next statement)
                                                 // usually incremental
{
    body statement(s);
}
```

```
class ForLoopDemo
{
    public static void main(String[] args)
    {
         for(int count = 1; count < 11; count++)
         {
              System.out.println("Count is: " + count);
         }
    }
}
```

The output of this program is:

```
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
```

For Loops

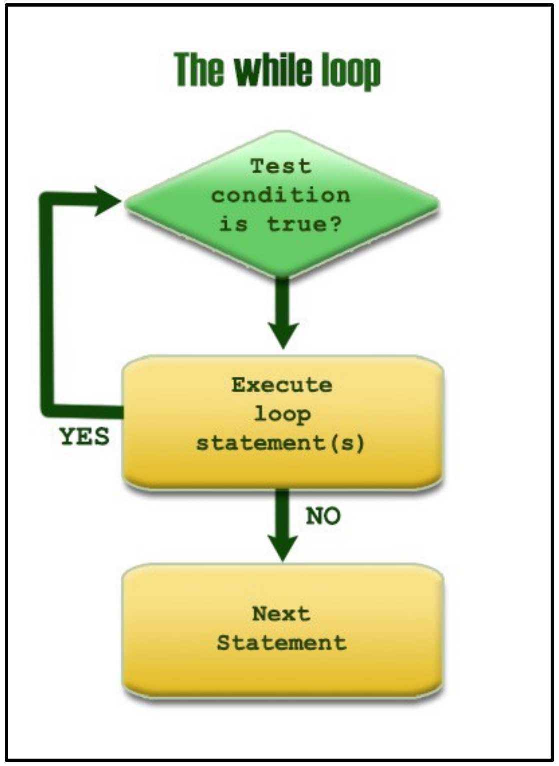


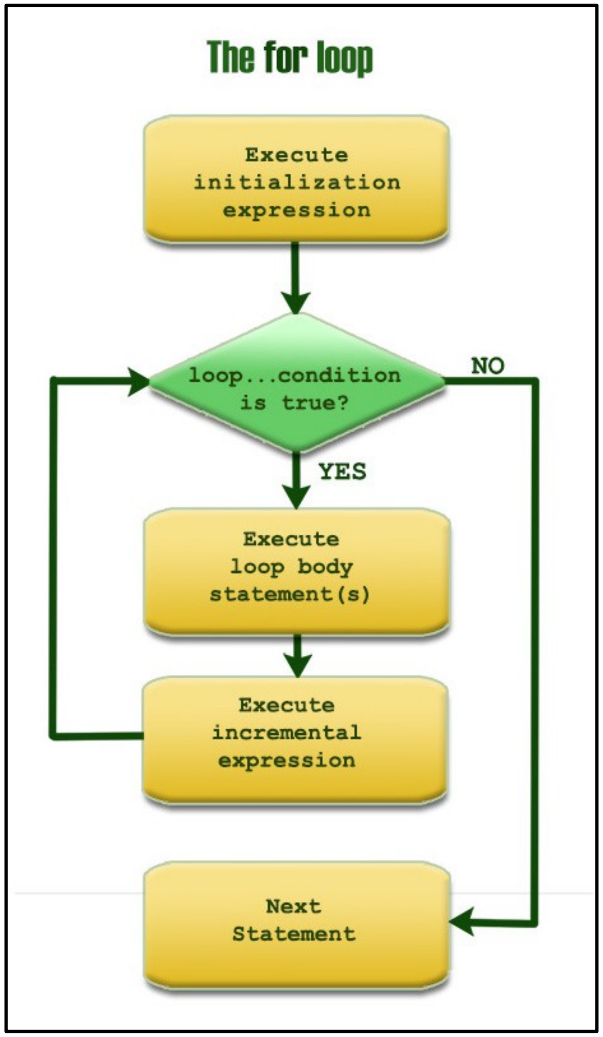
There are **three *clauses*** in the **for** statement.

- The **init-stmt** statement is done **<u>before</u>** the loop is started, usually to **initialize** an iteration variable (“counter”). After initialization this part of the loop is *no longer touched*.
- The **condition** expression is tested **<u>before</u> each time** the loop is done. The loop isn't executed if the Boolean expression is *false* (the same as the *while* loop).
- The **next-stmt** statement is done **<u>after</u>** the body is executed.
  It typically **increments** an iteration variable (“adds 1 to the counter”).

```
for(int count = 1; count < 11; count++)
{
   System.out.println("Count is: " + count);
}
```

For or While?




The for loop
Execute initialization expression
loop...condition is true?
NO
YES
Execute loop body statement(s)
Execute incremental expression
Next Statement

The for loop is shorter, and combining the intialization, test, and increment in one statement makes it easier to read and verify that it's doing what you expect.

The for loop is better when you are counting something.

If you are doing something an indefinite number of times, the while loop is be the better choice.

For or While?

```
class WhileDemo {
     public static void main(String[] args){
          int count = 1;
          while (count < 11) {
               System.out.println("Count is: " + count);
               count++;
          }
     }
}
```

```
class ForLoopDemo {
     public static void main(String[] args {
            for(int count = 1; count < 11; count++){
                  System.out.println("Count is: " + count);
            }
     }
}
```

Do-While Loops

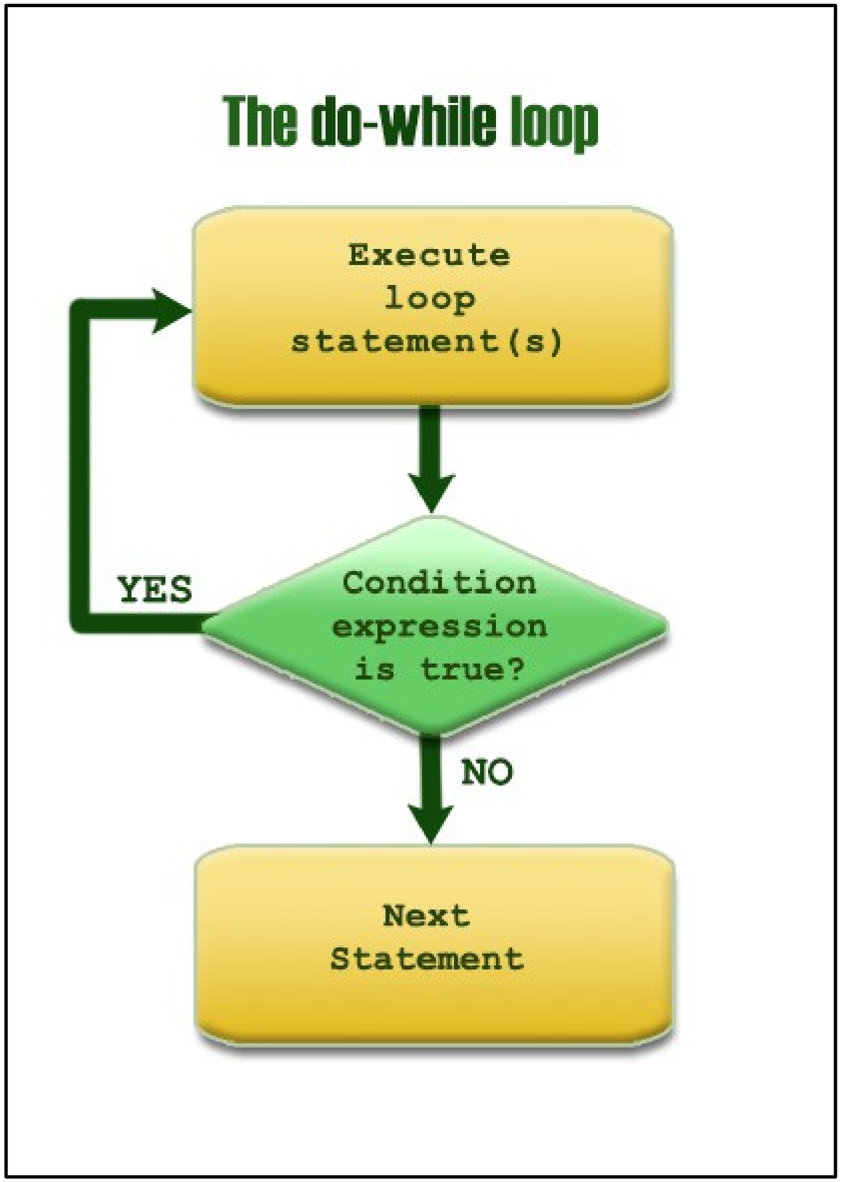
The Java programming language also provides a **do-while** statement, which can be expressed as follows:

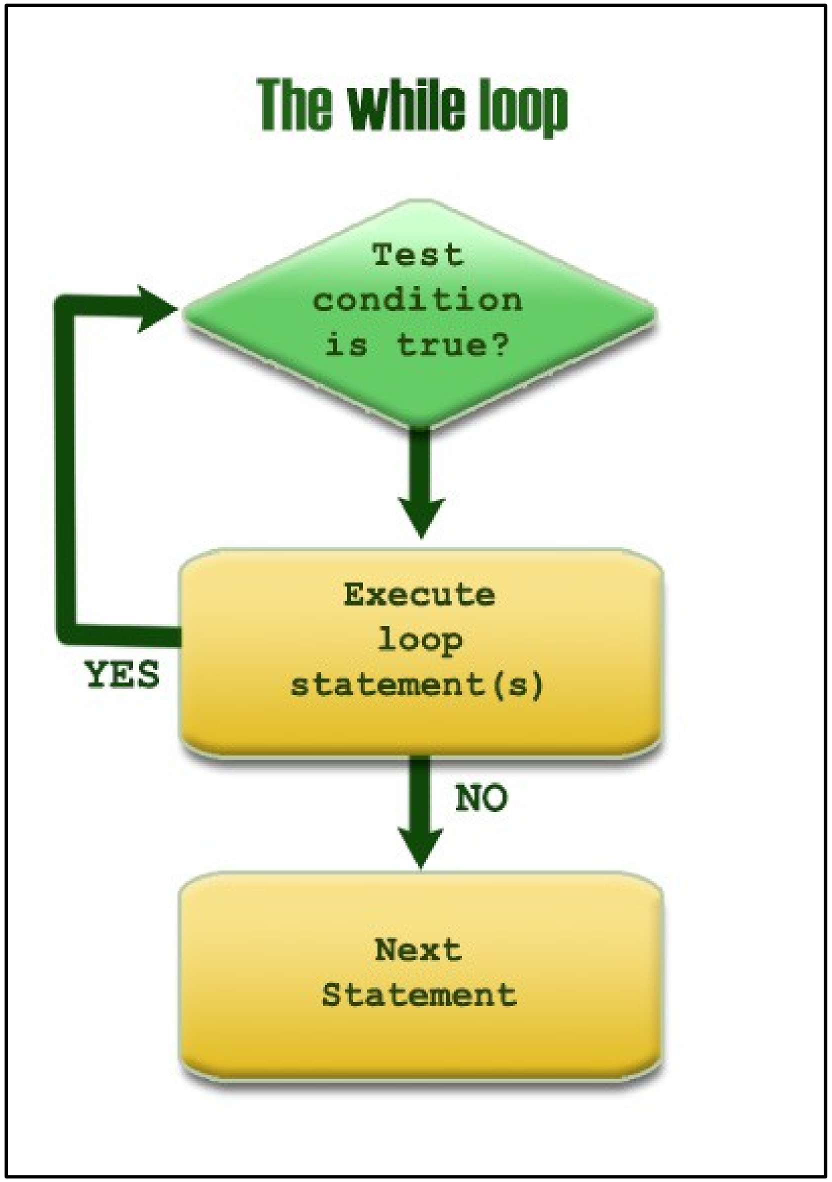
```
do {
     statement(s)
} while (expression);
```

The difference between **do-while** and **while** is that **do-while** evaluates its expression at the **bottom** of the loop instead of the **top**. **Therefore, the statements within the do block are always executed at least once**, as shown in the following **DoWhileDemo** program:

```
class DoWhileDemo
{
    public static void main(String[] args)
    {
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        } while (count < 11);
    }
}
```

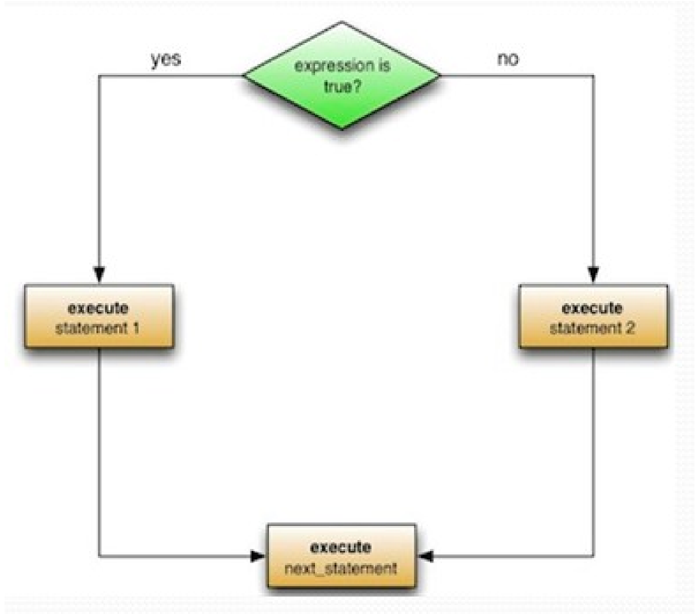
While or Do-While?





If-else Statements

```
if(Boolean_expression){
   statement 1 //Executes when true
}else{         //<-- No Conditional
   statement 2 //Executes when false
}
```



```
public class IfElseTest {
   public static void main(String args[]) {
      int x = 30;
      if( x < 20 ){
         System.out.print(“The number is less than 20.");
      }else{
         System.out.print(“The number is NOT less than 20!");
      }
   }
}
```

Several If Statements

```
int grade = 98;

if(grade >=0 && grade < 60)
{
   System.out.println("Sorry. You did not pass.");
}

if(grade >= 60 && grade < 70)
{
   System.out.println("You just squeaked by.");
}

if(grade >= 70 && grade < 80)
{
   System.out.println("You are doing better.");
}

if(grade >= 80 && grade < 90)
{
   System.out.println("Not too bad a job!");
}

if(grade >= 90 && grade <= 100)
{
   System.out.println("You are at the top of your class!");
}
```

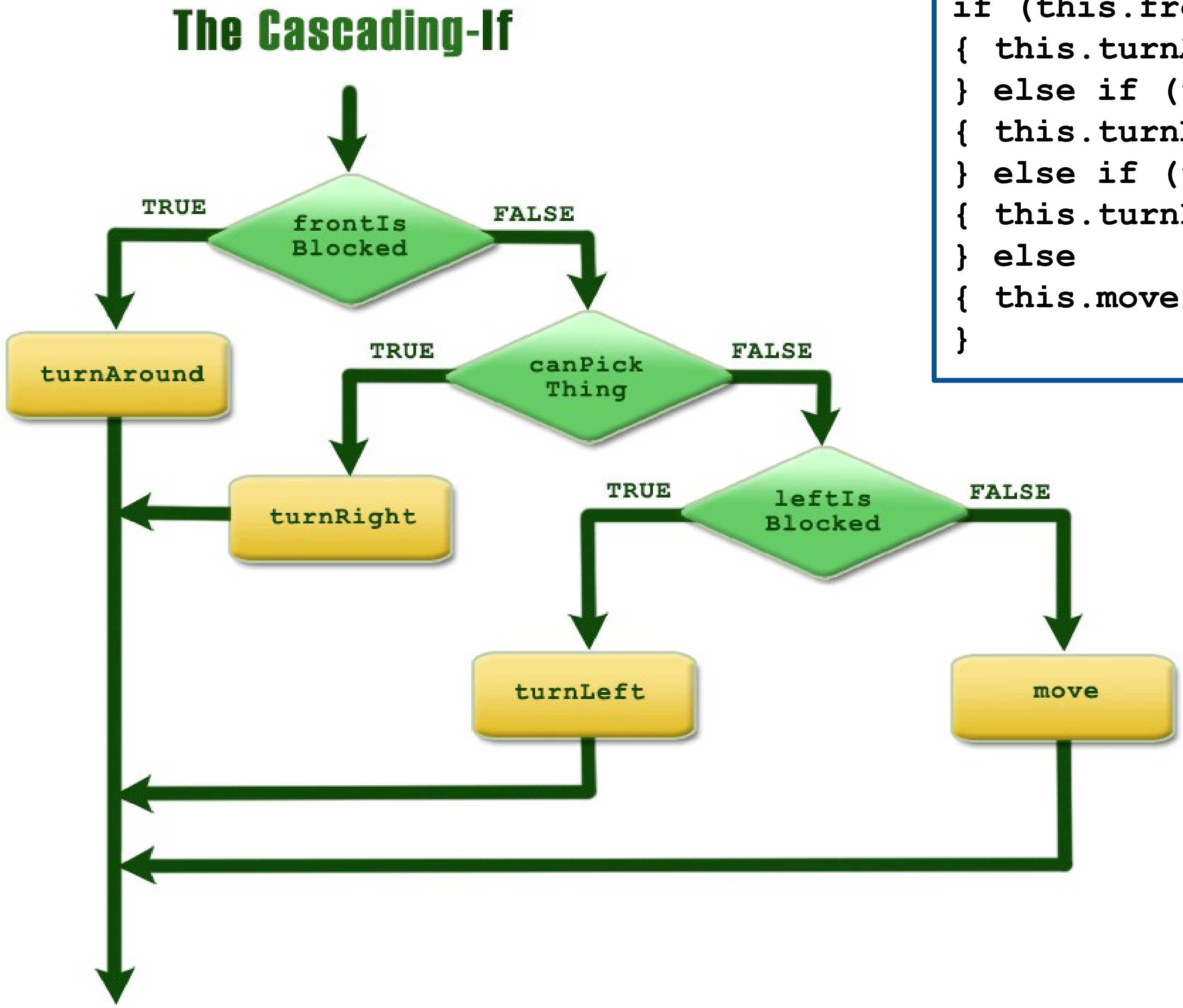
Cascading If Statements

```
int grade = 68;

  if(grade >=0 && grade < 60) {
     System.out.println("Sorry. You did not pass.");
  }
  else {
     if(grade >= 60 && grade < 70) {
        System.out.println("You just squeaked by.");
     }
     else {
        if(grade >= 70 && grade < 80) {
           System.out.println("You are doing better.");
        }
        else {
           if(grade >= 80 && grade < 90) {
              System.out.println("Not too bad a job!");
           }
           else //(grade >= 90 && grade <= 100)
           {
              System.out.println("You are at the top of your class!");
           }
        }
     }
  }
```

Cascading If Statements

```
int grade = 68;

 if(grade >=0 && grade < 60)
 {
    System.out.println("Sorry. You did not pass.");
 }
 else if(grade >= 60 && grade < 70)
 {
    System.out.println("You just squeaked by.");
 }
 else if(grade >= 70 && grade < 80)
 {
    System.out.println("You are doing better.");
 }
 else if(grade >= 80 && grade < 90)
 {
    System.out.println("Not too bad a job!");
 }
 else // <-- No conditional
 {
   System.out.println("You are at the top of your class!");
 }
```

Cascading If Statement



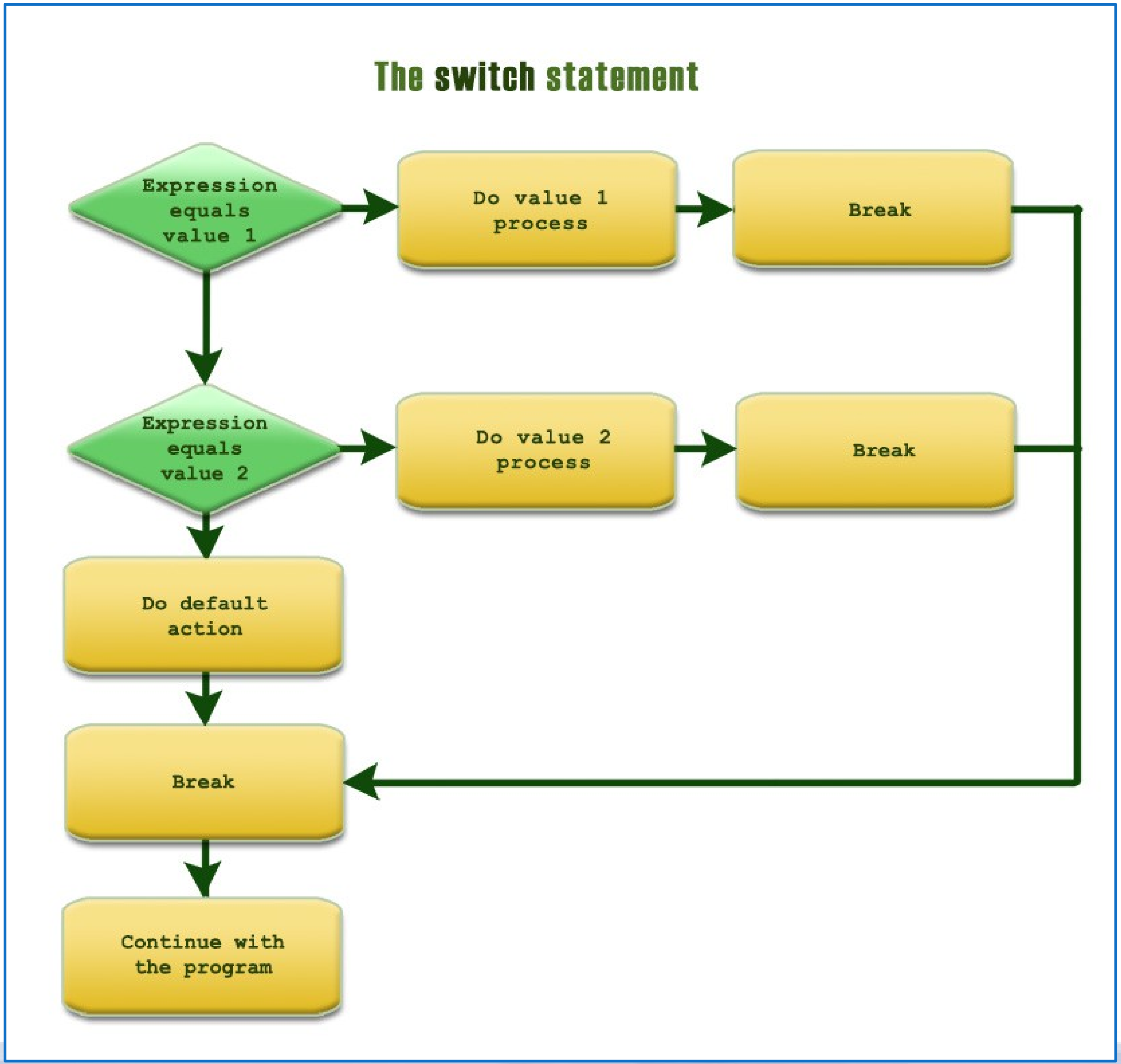
```
if (this.frontIsBlocked())
{ this.turnAround();
} else if (this.canPickThing())
{ this.turnRight();
} else if (this.leftIsBlocked())
{ this.turnLeft();
} else
{ this.move();
}
```

Switch statement

The **switch** statement is similar to the **cascading-if** statement in that both are designed to **choose one of several alternatives**. The **switch** statement is ***more restrictive*** in its use, however, because it uses a **single value** to choose the alternative to execute. The **cascading-if** can use *any* expressions desired. This restriction is also the switch statement's strength: **the coder knows that the decision is based on a single value**.

```
switch ( expression )
 {
   case value1 :
     statement(s) when expression == value1;
     break;
   case value2 :
     statement(s) when expression == value2;
     break;
   case value3 :
     statement(s) when expression == value3;
     break;
   default :
     statement(s) if no previous match;
 }
```

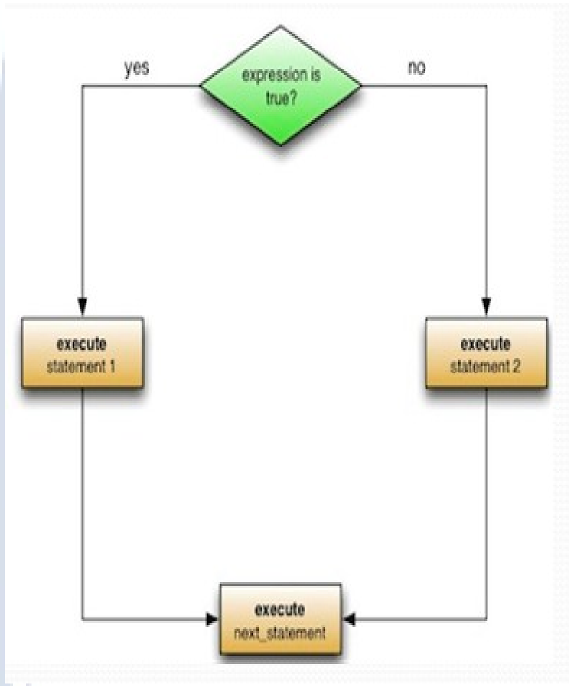
Switch Statement

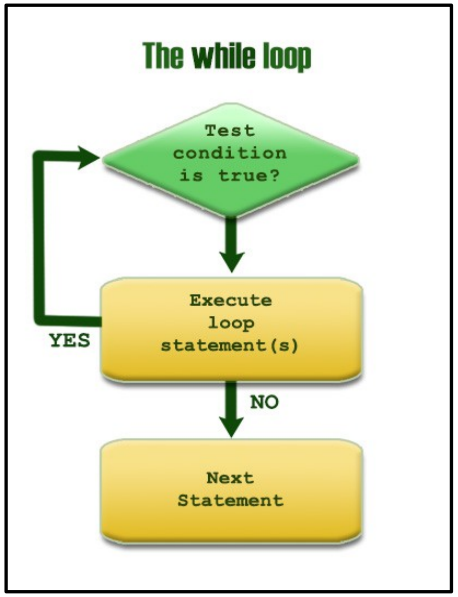


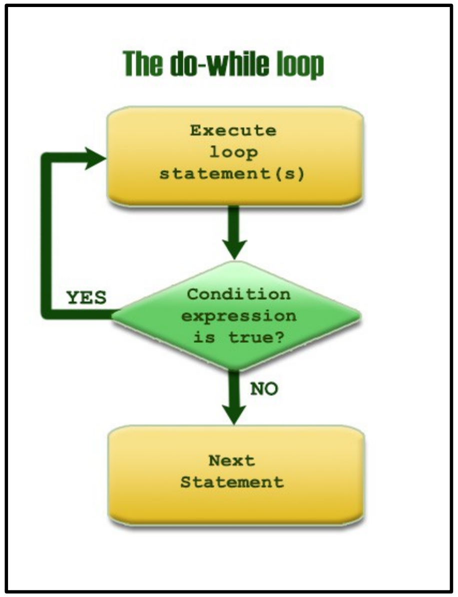
Switch Statement

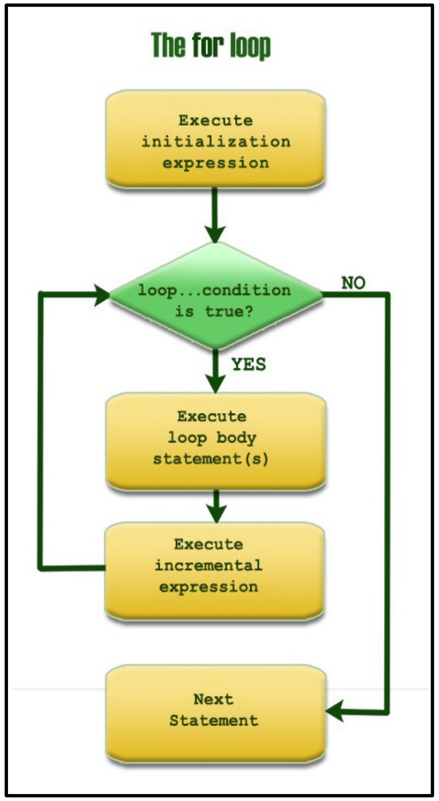
```
int score = 8;

switch (score)
{
   case 10:
      System.out.println ("Excellent.");
     // intentional fall-through
   case 9:
      System.out.println (“Well above average.");
      break;
   case 8:
      System.out.println (“Above average.");
      break;
   case 7:
   case 6:
      System.out.print ("Average. You should seek help.");
      break;
   default :
      System.out.println ("Not passing.");
}
```

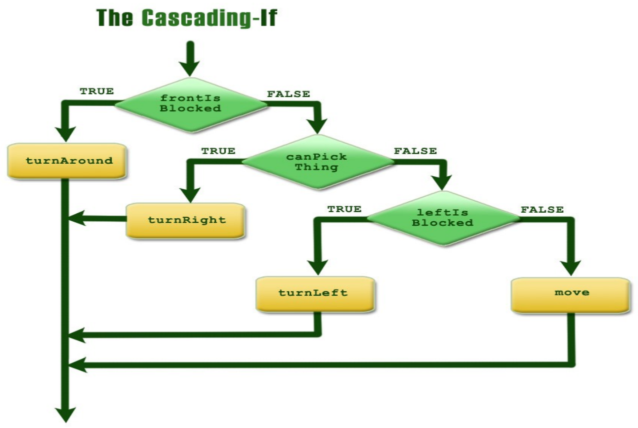
Control Statement Review

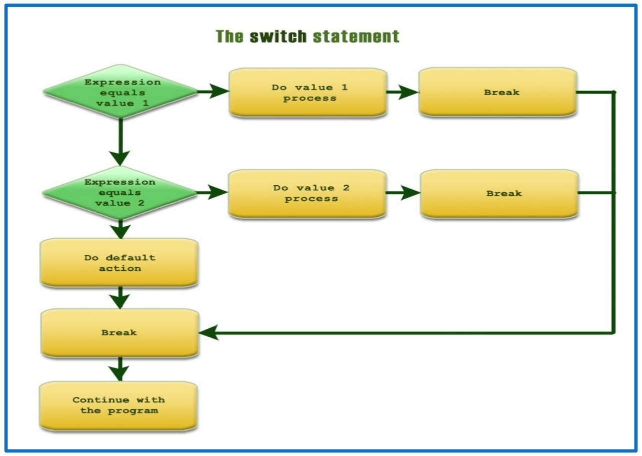







The for loop
Execute initialization expression
loop...condition is true?
NO
YES
Execute loop body statement(s)
Execute incremental expression
Next Statement





Introduction to Arrays

So far, you have been working with **variables** that hold only **one value**. The **integer** variables you have set up have held only **one number** (and *later* in the quarter we will see how **string** variables will hold **one long string** of text).

An **array** is a collection to hold *more than one value at a time*. It's like a **list** of items.

Think of an array as like the columns in a spreadsheet. You can have a spreadsheet with only one column, or several columns.

The data held in a single-list (one-dimensional) array might look like this:

	grades
0	100
1	89
2	96
3	100
4	98

Declaring Arrays

	grades
0	100
1	89
2	96
3	100
4	98

```
int[ ] grades;
```

The difference between setting up a normal integer variable and an array is a pair of **square brackets** after the **data type**. The *square brackets* tells Java that you want to set up an integer array. The name of the array above is **grades**.

The sqaure brackets don't say how many positions the array should hold. To do that, you have to set up a new array object:

```
grades  = new int[5];
```

In between the square brackets you need the *pre-defined* size of the array. The size is how many positions the array should hold. If you prefer, you can put all that on one line:

```
int[ ] grades = new int[5];
```

NOTE:
Arrays must be of the same data type, i.e., all **integers** (*whole numbers*) or all **doubles** (*floating-point numbers*) or all **strings** (*text characters*)—you **cannot** "mix-and-match" data types in an array.

Putting Values into Arrays

	grades
0	100
1	89
2	96
3	100
4	98

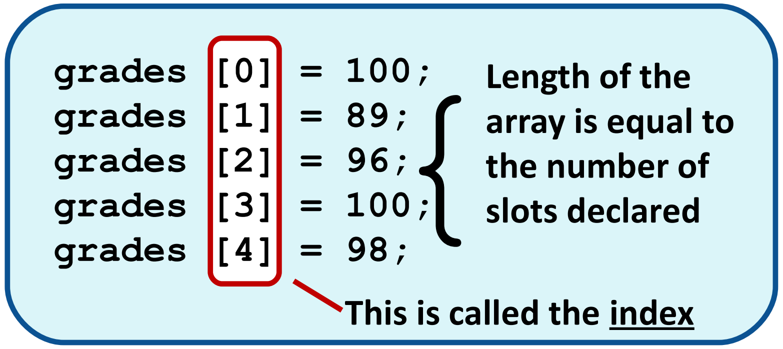
You can also declare the **int** separately and call it by its given name, like this:

```
int summer2012 = 5;
int[ ] grades = new int[summer2012];
```

We are telling Java to set up an array with 5 positions in it.

After this line is executed, Java will assign default values for the array (0 for an integer array). To assign values to the various positions in an array, you do it in the normal way:

```
grades[0] = 100;
grades[1] = 89;
grades[2] = 96;
grades[3] = 100;
grades[4] = 98;
```



```
grades [0] = 100;
grades [1] = 89;
grades [2] = 96;
grades [3] = 100;
grades [4] = 98;
```

Length of the array is equal to the number of slots declared
This is called the index

If you know what values are going to be in the array, you can set them up like this instead:

```
int[ ] grades = { 100, 89, 96, 100, 98 }; // Java treats as a
                                          // new instance
```

Array notes

- Arrays have a built in value length:
  - Int []myArray = new int [5];
  - MyArray.length returns 5
- Constructing an array with new makes space for length * dataType memory
- In Java (and most other modern languages), indices are zero based – the first value is in myArray[0], the last in myArray[length-1].
  - This means you can write loops with just a <:
  - for (int j=0; j < myArray.length; j++) {