SWEN-601
Software Construction

Arrays & Iteration

Next Two Weeks

WEEK 02	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ		LABOR DAY (no office hours)	Quiz #2		Quiz #3		
LECTURE			Boolean Expressions, & Conditionals		Arrays, & Iteration (`while` and `for` loops)		
HOMEWORK	Hwk 2 Due (**11:30pm**)		*Hwk 3 Assigned*		*Hwk 4 Assigned*	Hwk 3 Due (**11:30pm**)	

WEEK 01	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #4		Quiz #5		
LECTURE			Objects I: Encapsulation		Objects II: Identity, Equality, `static`		
HOMEWORK	Hwk 4 Due (**11:30pm**)		*Hwk 5 Assigned*		*Hwk 6 Assigned*	Hwk 5 Due (**11:30pm**)	

Activity: Accept the GitHub Classroom Assignment

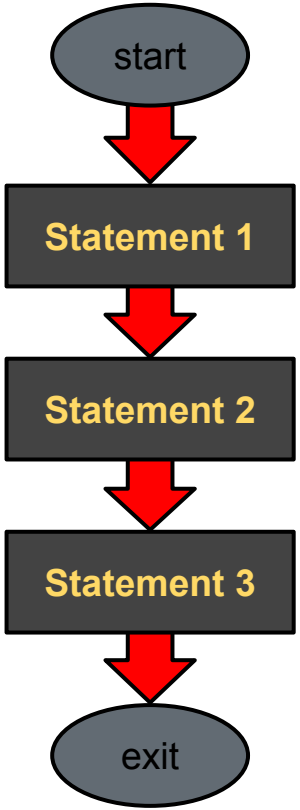
You will be asked to accept a new assignment at the start of nearly every class.

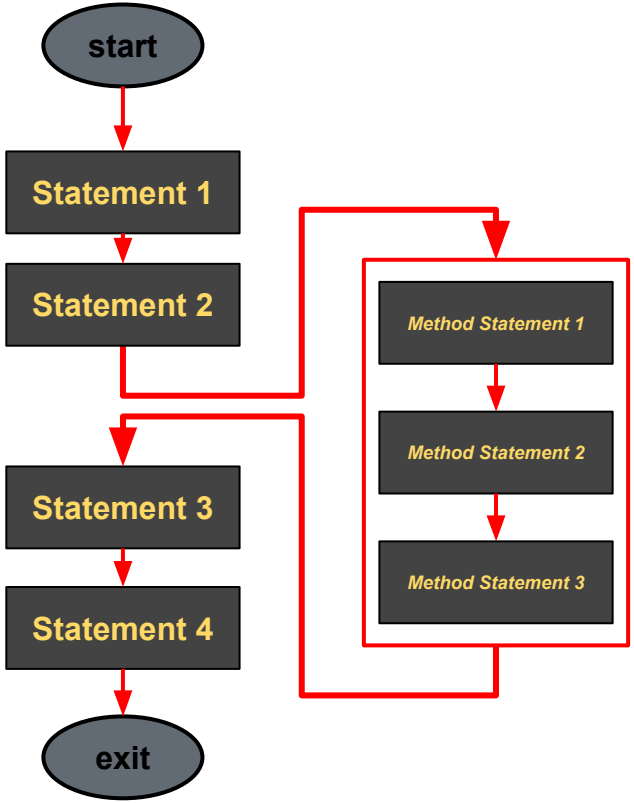
You should get used to accepting the assignment and starting your new project right after you finish your quiz each day.

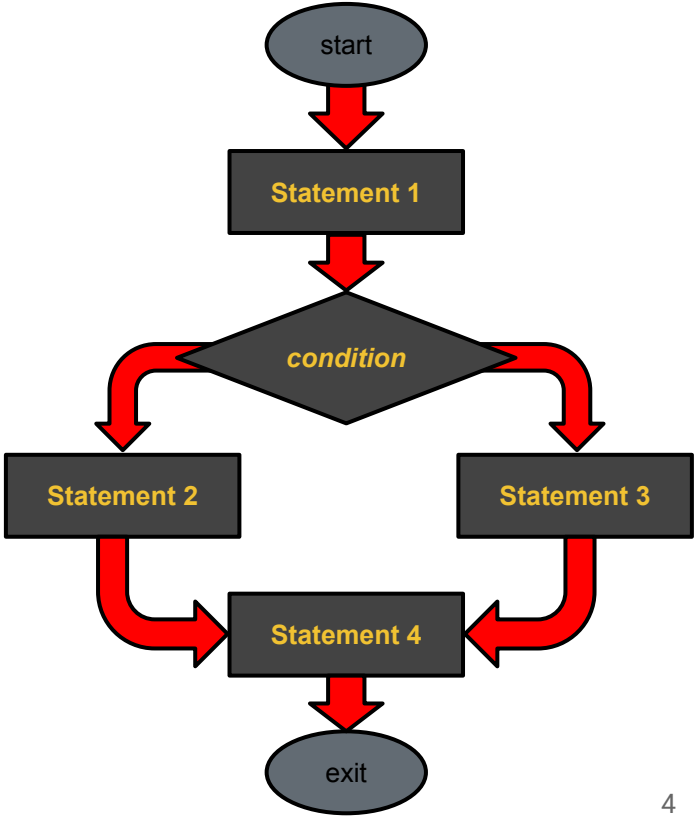
Your instructor has provided a GitHub classroom invitation. You should be able to find it under “Homework” on MyCourses.

1. Click the GitHub classroom invitation.
2. Assuming that you have already linked your GitHub account with your name in the class roster, you should be prompted to accept the assignment. Do so.
3. Once the repository is created, copy the URL.
4. Clone the repository to your local file system. As before, the repository will be empty.
5. Create a new IntelliJ Project inside the repository, and push it to GitHub.
6. Create a package named “`activities`” in your `src` folder.
7. You are now ready to begin today’s activities!

Flows of Control







A Simple (?) Problem

Q: What if I asked you to write a program that printed a countdown from 10 to 0. What would the Java code look like?

Q: What if I asked you to count down from 100 to 0? From 1000?

Q: What if I asked you to use Scanner to prompt the user to enter *any* number to count down from?

```
public static void main(String[] args) {
  System.out.println(10);
  System.out.println(9);
  System.out.println(8);
  System.out.println(7);
  System.out.println(6);
  System.out.println(5);
  System.out.println(4);
  System.out.println(3);
  System.out.println(2);
  System.out.println(1);
  System.out.println(0);
}
```

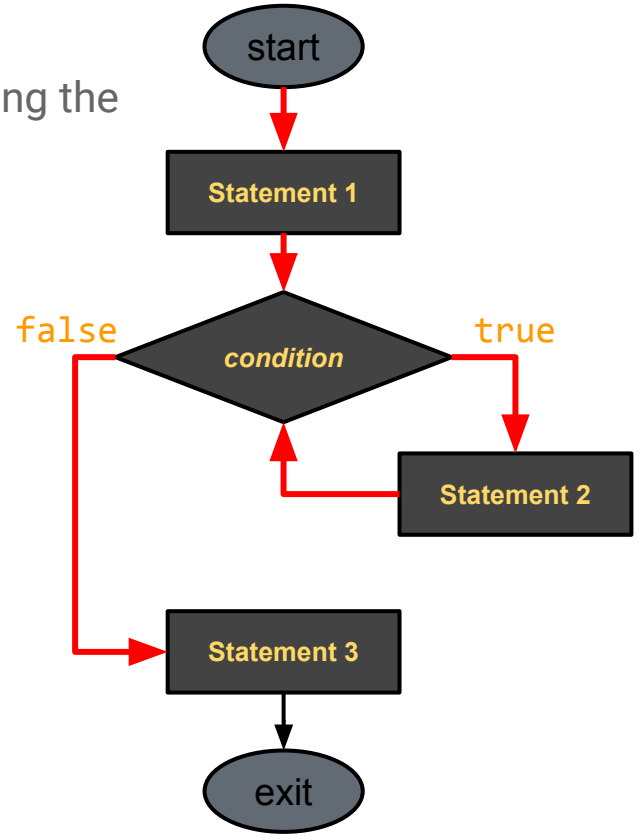
Iteration

- Sometimes a program would benefit from executing the same statement more than once.
  - Repeat ***while*** some condition is true.

If the ***condition*** is `true`, execute statement 2.

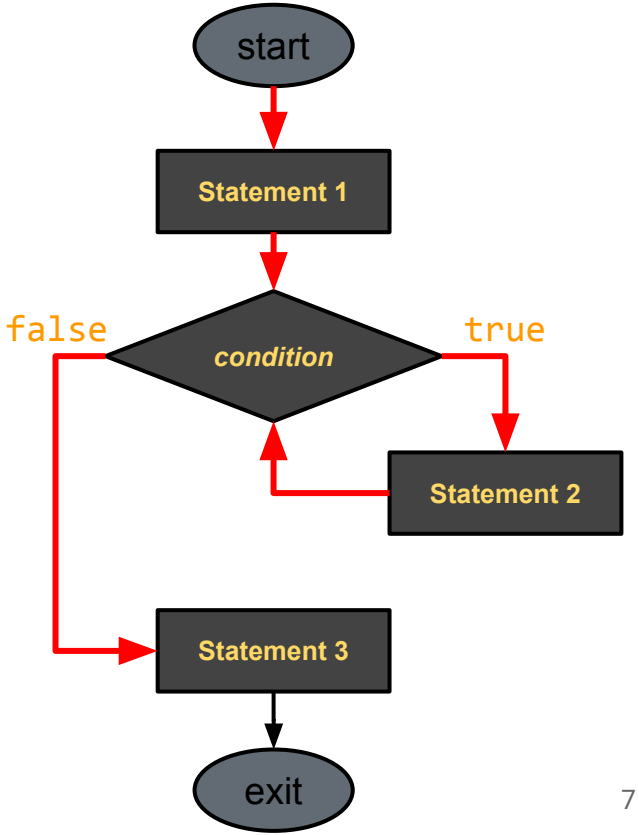
After each *iteration*, check the ***condition*** again. If it is still `true`, execute statement 2 again.

Repeat until the ***condition*** is `false`, then execute statement 3.



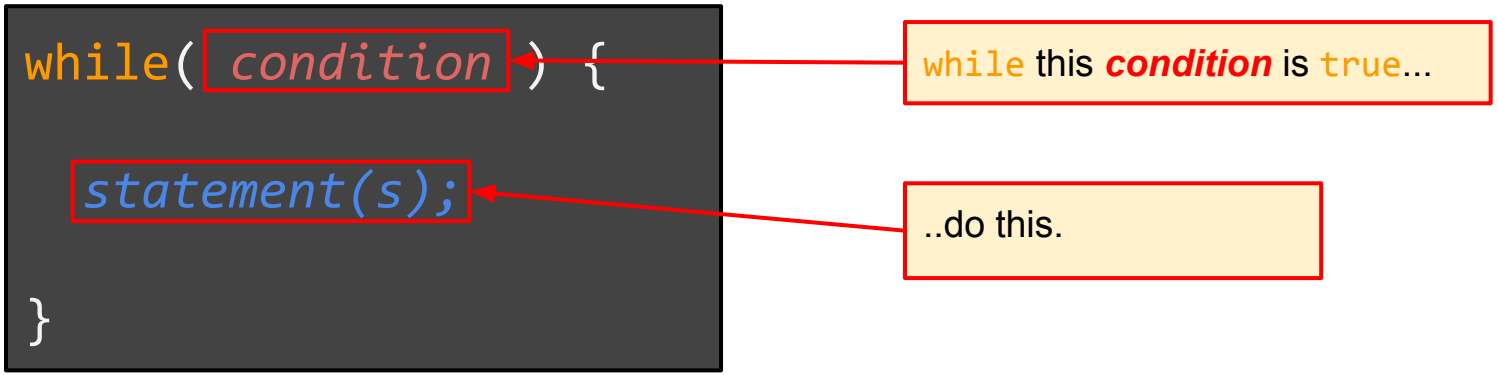
Repetition Statements

- Repetition statements allow the program to execute statements multiple times *as long as some condition is true*.
  - The are often referred to as loops.
  - Like conditional statements, they are controlled by *boolean expressions*
- Java has three kinds of repetition statements:
  - The `while` loop
  - The `do while` loop
  - The `for` loop
  - The `for` **each** loop
- The programmer should choose the right kind of loop for the situation.



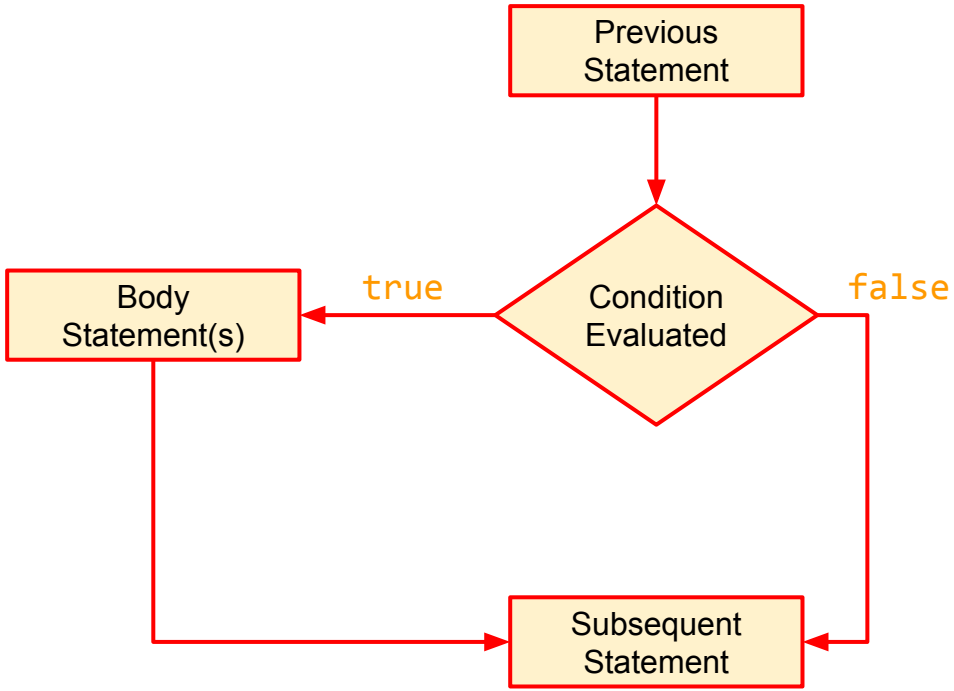
The `while` Loop

- A `while` statement has the following syntax:

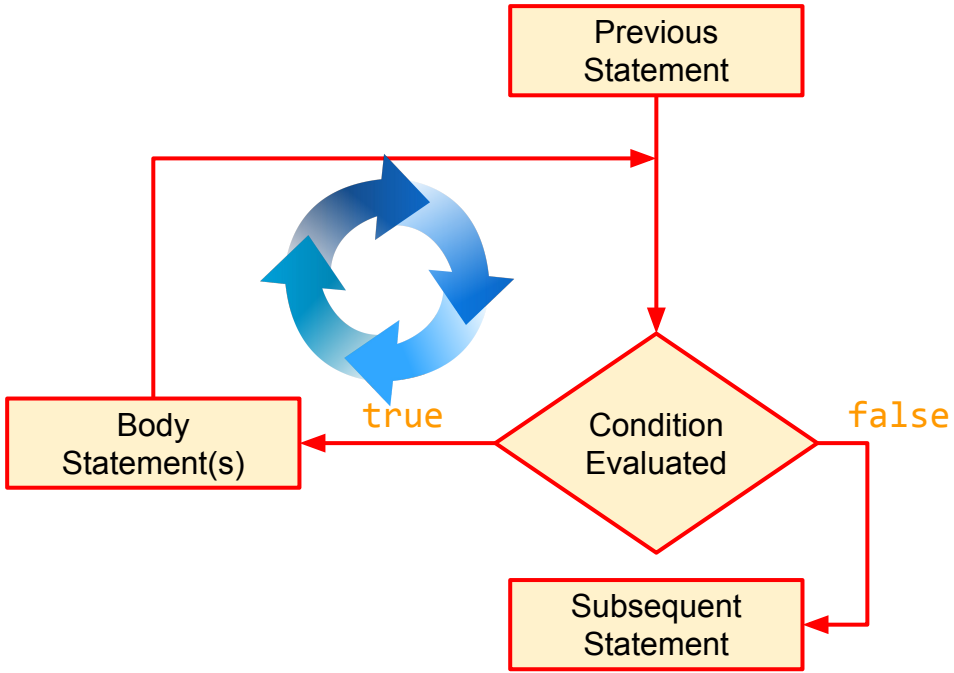


- If the *`condition`* is `true`, the `while` loop executes the *`statement(s)`*.
- Then the *`condition`* is evaluated again. If it is still `true`, the `while` loop executes the *`statement(s)`* again.
- This continues until the *`condition`* evaluates to `false`.

The Logic of an `if` Statement



The Logic of a `while` Loop



The `while` Loop

- An example:

```
int count = 1;

while(count < 4) {

    System.out.println( count );

    count = count + 1;

}

System.out.println("Done!");
```

Q: What will the output from the execution of this loop be?

A: 1
2
3
Done!

Q: Why?

A: Let's take a look...

The `while` Loop

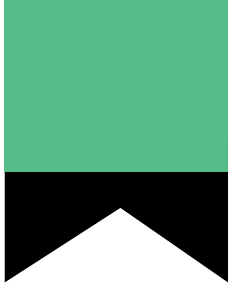
```
int count = 1;
while(count < 4 ) { // true
     System.out.println(count); // "1"
     count = count + 1; // count is now 2

while(count < 4) { // still true
     System.out.println(count); // "2"
     count = count + 1; // count is now 3

while(count < 4) { // still true
     System.out.println(count); // "3"
     count = count + 1; // count is now 4

while(count < 4) { // false

System.out.println("Done!");
```

First the previous statement is executed, assigning the value `1` to the variable `count`.

The *condition* specified in the header of the loop is evaluated; "`count < 4`" is `true`, so the program enters the loop and executes the statements in the body.

At the end of the body, the flow of control jumps back to the header. As long as the *condition* still `true`, the statements in the body are executed again, and again.

Finally the *condition* in the header is `false`. The body is skipped and the flow of control moves to the statement following the loop body.



```
Enter a number: 11
11
10
9
8
7
6
5
4
3
2
1
0
```

Activity: Counting Down

1. Create a new class, `CountDown`.
2. In the `main` method, prompt the user to enter a number.
3. Use a `while` loop to count down from that number to `0`. Print each number.

The `while` Loop

- If the condition in a `while` statement is `false` initially, the statements in the body are never executed. Not even once.
  - The condition is evaluated *before* the statements are executed.
- Therefore, the body of a `while` loop will execute *zero or more times*.

```
int count = 5;
while(count < 4) {
    System.out.println("Never gonna happen.");
}
```

Because the condition is not true the first time it is evaluated, the statements in the body of the `while` loop are never executed.

The `while` Loop

```
int count = 1;
while(count <= 3) {
    System.out.println(count);
    count = count - 1;
}
```

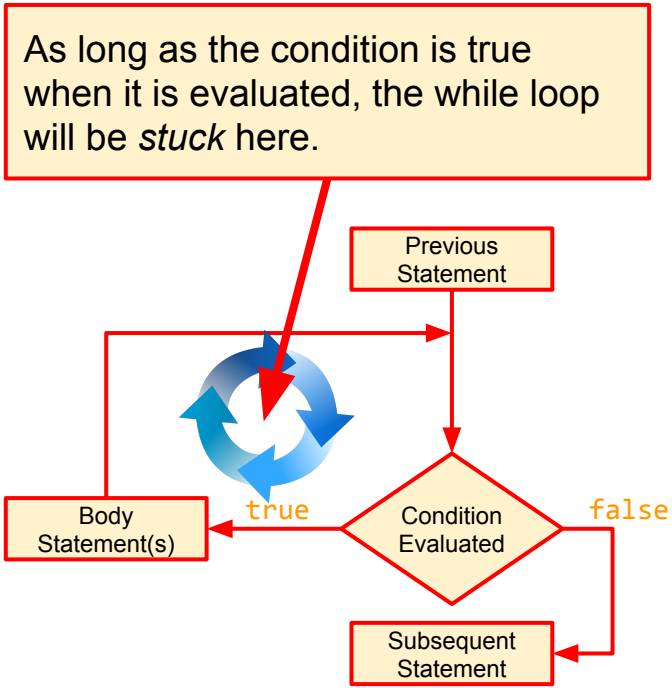
Q: What is the output of this while loop?

A: 1
0
-1
-2
-3
-4

To infinity...and beyond!

Infinite Loops

- The statements within the body of the `while` loop must eventually cause the condition to be `false`.
  - This generally means that the value used in the *`condition`* (e.g. **`count`**) is *altered* in some way by the *`statement(s)`* in the body.
  - It is common for this to be an increment, or a decrement.
- If not, an ***infinite loop*** is caused, which will execute forever or until the program is interrupted.
  - Pro tip: The little red stop button in your IntelliJ run controls will interrupt a running process.
- Writing loops with conditions that never terminate is a common semantic error.
- It is good practice to double check and test your conditions to make sure that they terminate properly.



The do while Loop

- A while loop checks the *condition first*, and if the *condition* is true, it executes the *statement(s)* in the body.
- What if the programmer wants to execute the *statement(s) at least once* regardless of the *condition*?
- One alternative: Cut and paste the *statement(s)*.

```
statement(s);
while(condition) {
    same statement(s);
}
```

Q: This is bad. Why?

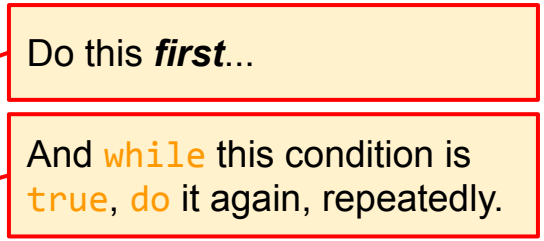
A: Code duplication is bad, if it can be avoided. Why code something twice if you can code it once? Also consider what happens *every time* the programmer needs to change the *statement(s)*.

- Another alternative: the do while loop.

The `do while` Loop

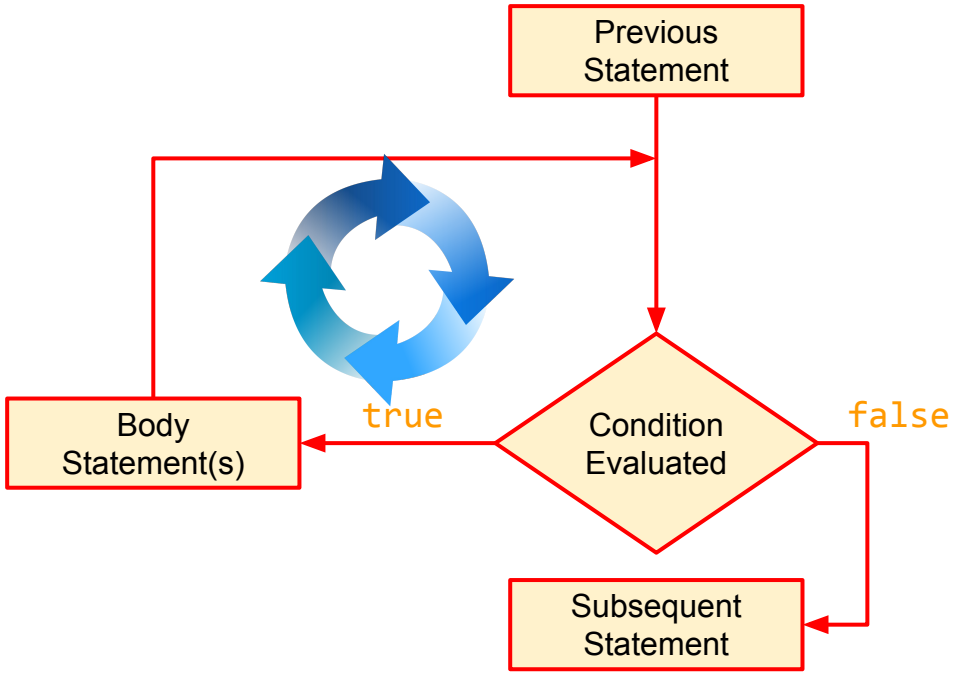
- A `do while` statement has the following syntax:

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

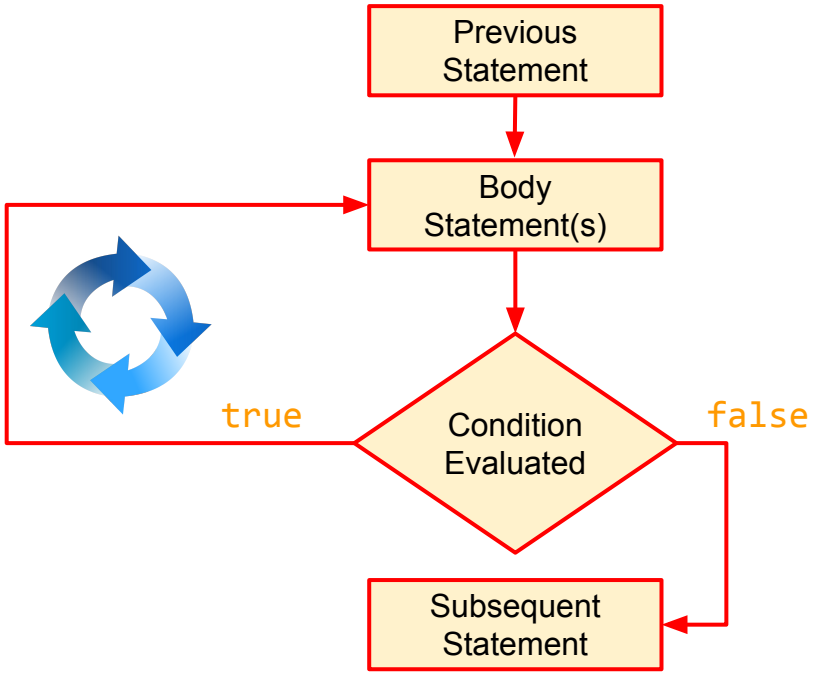


- The *`statement(s)`* are executed once *first* and *then* the *`condition`* is evaluated.
- If the condition is `true`, the *`statement(s)`* are executed again.
- This continues until the *`condition`* is `false`.
- This is also often called a `do` loop.

The Logic of a `while` Loop



The Logic of a `do while` Loop



The `do while` Loop

- An example:

```
int count = 0;

do {

    count = count + 1;

    System.out.println( count );

} while(count < 5);
```

Q: What will the output from the execution of this loop be?

A: 1
2
3
4
5

Remember, `count` is incremented to 5, *then* the value is printed, and *then* the *`condition`* is evaluated.

Activity: Evens and Odds

```
Enter a number: 24
Even.
Enter a number: 57
Odd.
Enter a number: 15
Odd.
Enter a number: 64
Even.
Enter a number: 0
Even.
Done!
```

1. Create a new class, `EvensAndOdds`.
2. Use a `do while` loop to:
   a. Prompt the user to enter a number.
   b. Print a message indicating whether the number is even or odd.
3. Stop when the user enters `0`.

A Common Loop

- Remember this?

```
int count = 1;
while(count < 5) {
    System.out.println(count);
    count = count + 1;
}
```

Initialize the variable used in the loop.

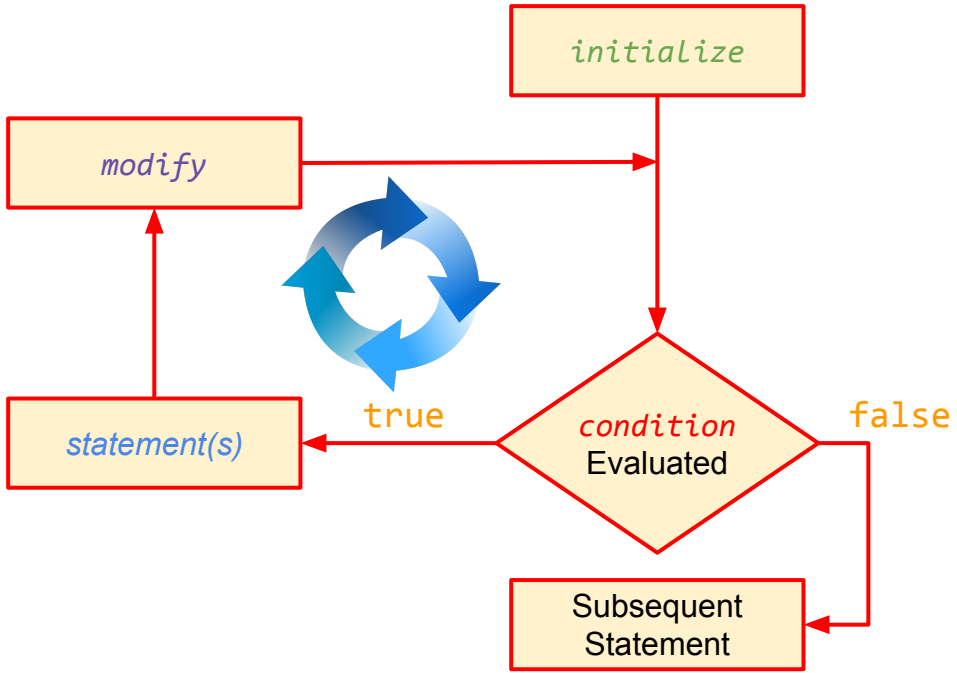
Evaluate the variable in the *condition*.

Modify the variable in some way (e.g. increment).

- This is a very common `while` loop:
  - *Initialize* the variable.
  - Evaluate the *condition*.
  - *Modify* the variable.

A Common Loop

```
initialize;
while(condition) {
    statement(s);
    modify;
}
```



The for Loop

- This kind of loop is *so* common, that a functionally equivalent shorthand exists.

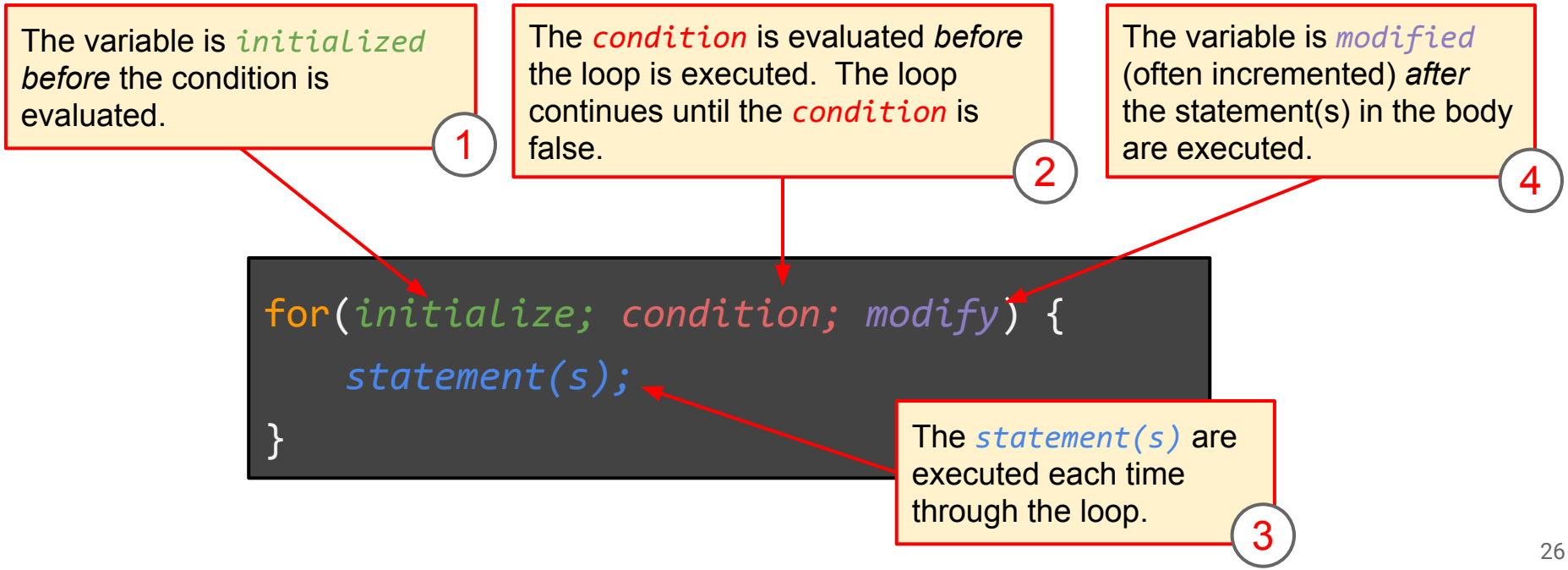
```
initialize;
while(condition) {
    statement(s);
    modify;
}
```

```
for(initialize; condition; modify) {
    statement(s);
}
```

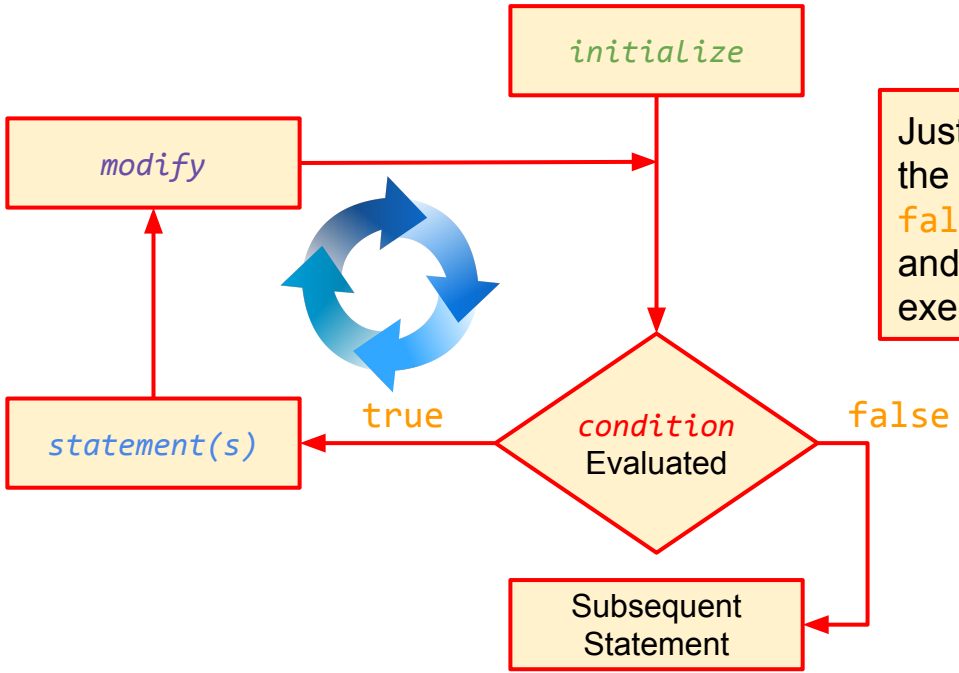
- This is called a for loop.

The `for` Loop

- Just like the common `while` loop...



Logic of a `for` Loop



Just like the `while` loop, if the *condition* is *initially* `false`, the *statement(s)* and *modify* will not be executed.

The for Loop

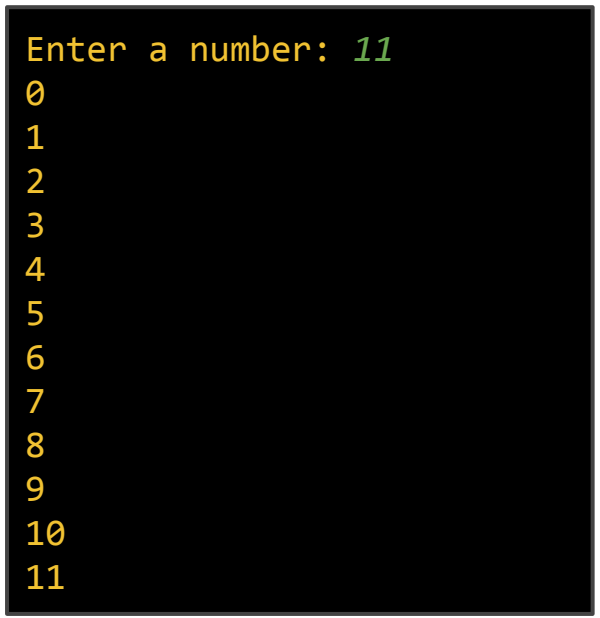
- An example:

```
for(int i=0; i<4; i++) {
    System.out.println( i );
}
```

Q: What is the output of this loop?

A: 0
1
2
3

- The *initialization* section can be used to declare a variable.
  - If a variable already exists, it can be left blank (;).
- In fact, any of the parts of the for loop header may be left blank.
  - e.g. for(;;){}
- Like a while loop, the *condition* is tested before executing the *statement(s)* in the body.
  - If initially false, the *statement(s)* are never executed.



Activity: Counting Up

1. Create a new class, `CountUp`.
2. In the `main` method, prompt the user to enter a number.
3. Use a `for` loop to count from `0` to that number. Print each number.

The `for` Loop

- While the modify section of the `for` loop header is commonly used to increment a variable, it can execute any statement. For example:

```
for(int i=0; i<100; i=i+5 )
    System.out.println(i);
```

Like the `if` statement, Java does not require curly braces around the body of a loop. The statement that immediately follows will be executed.

- Or, the modify section may do nothing at all.

```
for(boolean go=true; go;) {
    System.out.print("Keep going?");
    go = scanner.nextBoolean();
}
```

Also like an `if` statement, this is a terrible idea. Always use curly braces.

The `for` Loop

- As mentioned previously, each of the sections of the `for` loop header are optional.
  - *`initialize`* - if the variable(s) needed inside the `for` loop are declared outside of the loop, the *`initialize`* may be left blank.
  - *`condition`* - a blank *`condition`* is always considered to be `true`. This creates an infinite loop.
  - *`modify`* - if left blank, no statement is executed at the end of each loop.

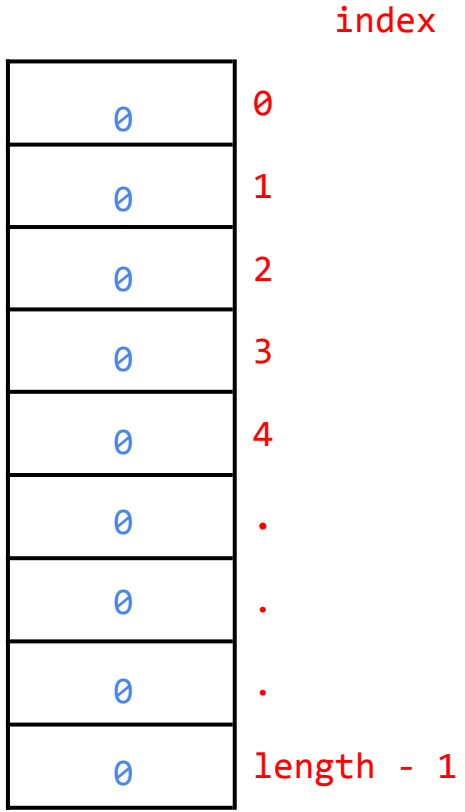
```
for( ; ; ) {
  statement(s);
}
```

...is functionally equivalent to...

```
while(true) {
  statement(s);
}
```

Arrays

- An ***array*** is a fixed length data structure.
  - The array is allocated to be some non-negative ***length***.
  - This allocates a *contiguous* block of memory.
- Once the memory is allocated, the array is filled with default values.
  - e.g. `0` for numeric types, `null` for reference types.
- Data in the array is accessed using the **index** of the desired element.
  - Indexes range from `0` to `length - 1`.
  - The memory address of the array is the same as the position of its first element.
- The default values in the array can be replaced with some other value using the index.



Arrays

```
int[] squares;

squares = new int[5];

System.out.println(squares.length);

System.out.println(squares[4]);

squares [3] = 9;
```

Default Values	
Numeric Types	`0`
char	`'\u0000'`
booleans	`false`
Reference Types	`null`

Array types are declared using any other type (including classes) with square brackets ([]).

Arrays are initialized using the `new` keyword and specifying a `length` in square brackets.

This immediately allocates a contiguous block of memory large enough to store the specified elements. The memory is filled with default values.

You can always check the `length` of an array. Once set, this will never change.

The value of a specific element is retrieved using its index.

The index is also used to modify the value using an assignment statement.

Activity: Filling an Array

```
Enter a number: 8
0
10
20
30
40
50
60
70
```

1. Write a method that accepts an integer parameter.
2. Create a new array using the parameter as the size.
3. Use a loop to fill the array with multiples of 10 beginning with `0`.
   a. e.g. if the size is `4`, the array should contain the values `0`, `10`, `20`, and `30`.
4. Your method should return the array.
5. Write a main method that prompts the user to enter an integer. Call your method and print the multiples of 10 that are returned, each on its own line.

The `for each` Loop

- In the previous problem, you probably wrote a loop that looks something like this:

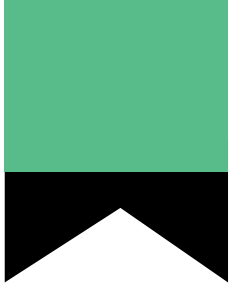
```
for(int i = 0; i<numbers.length; i++) {
    System.out.println( numbers[i] );
}
```

This kind of loop over an array is ***very*** common.

- Java provides an alternative syntax called a `for each` loop that can be used to iterate over the elements in an array.

```
for(int i : numbers) {
    System.out.println( i );
}
```

This kind of loop only works with ***iterable*** types, including arrays.



Activity: Printing an Array

```
Enter a number: 8
0
10
20
30
40
50
60
70
```

1. Modify your `main` method from the previous problem to use a `for each` loop when printing the elements of your array.

Two-Dimensional Arrays

- A normal array is only *one-dimensional*.
  - Elements are stored in a contiguous row.
- Java supports the creation of *two-dimensional* arrays as well.
  - Elements are stored in rows *and* columns.
- When a two-dimensional array is created, two different values are specified in square brackets
  - One for the length of the first dimension, and one for the length of the second.

```
int[][] table = new int[3][5];
int value = table[2][3];
```

0	0	0	0	0

A normal (one-dimensional) array.

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

A two-dimensional array.

Two indices are needed to access a value in a two dimensional array; one for each dimension.

Nested for Loops

- A single for loop is not sufficient for iterating over a two-dimensional array.
  - You can use one for loop to iterate over one dimension.
  - You need a second for loop to iterate over the other dimension.

```
for(int row=0; row<array.length; row++) {
  for(int col=0; col<array[row].length; col++) {
    System.out.print(array[row][col]);
  }
}
```

One loop iterates over the *rows*, each of which is an array of values...

...the *nested* loop iterates over each *column* in the row.

Both indices are needed to retrieve a value from the array.

Activity: Multiplication Table

```
 1  2  3  4  5  6  7  8  9 10
 2  4  6  8 10 12 14 16 18 20
 3  6  9 12 15 18 21 24 27 30
 4  8 12 16 20 24 28 32 36 40
 5 10 15 20 25 30 35 40 45 50
 6 12 18 24 30 36 42 48 54 60
 7 14 21 28 35 42 49 56 63 70
 8 16 24 32 40 48 56 64 72 80
 9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

1. Create a 10x10 two dimensional array.
2. Set each value in the array to the product of its row and column.
3. Print the array in the format shown to the left.

Flows of Control