[**Class**](https://www.hackerrank.com/challenges/30-hello-world/tutorial)   
At its most basic level, a *class* is a collection of *variables* (*fields*) and functions called *methods*. A program is a collection of classes. The basic code for declaring a Java class is as follows:

class MyClass{

// ...does cool stuff.

}

When declaring a class, the name should always start with a *capital* letter; this signifies to certain compilers (and human readers of your code) that it is a class (or other similarly-behaved structure that you'll learn about later). If you wish to use a compound phrase (e.g.: "my class") as your class name, you should write it in [CamelCase](https://en.wikipedia.org/wiki/CamelCase); this means you should capitalize each word and remove spaces between words (e.g.: "MyClass").   
  
**Note:** Class names cannot *begin* with numbers or contain any spaces.

[**Variable**](https://en.wikipedia.org/wiki/Variable_%28computer_science%29)   
Think of this as a name (identifier) that points to (references) a location in memory where information associated with that name can be stored. In Java, it is a best practice to always start variable names with a *lowercase* letter and use CamelCase for variable names composed from compound phrases. Variable names cannot contain spaces or special characters (except underscores), though they can contain (but *not* begin with) numbers.   
  
Each variable has a *data type* associated with it, which essentially restricts what that variable is allowed to reference (so your code will not run if you attempt to perform operations on your variables that aren't allowed for that data type). To *declare* a variable named myVariable having the data type DataType, we write the following:

DataType myVariable;

If we want to declare a variable of type DataType named myVar1 and *initialize* it to be value, we write:

DataType myVar1 = value;

In English, the above code is basically saying: "I'm creating a variable named myVar1; it refers to something of type DataType, and is assigned an initial value ofvalue."   
  
**Note:** The = operator is called the *assignment operator*, so you should interpret = as the English phrase "[left operand] is assigned the value of [right operand]".   
  
A *String* is a data type that holds a sequence of characters. To create a *String* variable named myString that stores the value "Hi!", write the following line of code:

String myString = "Hi!";

The compiler will interpret the characters between the two quotation marks as a *String*. Saving a reference to our it as variable myString allows us to refer to it again and again by referencing our variable name, myString.

**Note:** Some coders use lowercase letters in conjunction with underscores to simulate spaces when declaring variables (e.g.: "my\_variable"). This is not the naming convention used in Java, though there are many other languages where you might see this used frequently; however, you may see some Java coders begin some special variable names (e.g.: private class variables or constants) with an underscore to distinguish them from other variables used throughout their program.

[**Function**](https://en.wikipedia.org/wiki/Subroutine)   
A sequence of packaged instructions that perform a task.

[**Method**](https://en.wikipedia.org/wiki/Method_%28computer_programming%29)   
In Object-Oriented programming, a *method* is a type of function that operates on the fields of a class.

int myMethod(){

// ...does cool stuff.

}

void myMethod(int myInt){

// ...does cool stuff.

}

Check out [Oracle's Method documentation](https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html) to learn more.

[**Object**](https://docs.oracle.com/javase/tutorial/java/concepts/object.html)   
An *Object* is an *instance* (or *variable*) of a class.

[**Stream**](https://en.wikipedia.org/wiki/Stream_%28computing%29)   
Think of this as the flow of data from one place to another. Most of our challenges require you to read input from *System.in* (also known as *stdin*, the standard input stream), and write output to *System.out* (also known as *stdout*, the standard output stream). In Java, the [Scanner](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html) class is widely used to read input, but each language has its own mechanism for handling [IO](https://en.wikipedia.org/wiki/Input/output) (input and output).

The syntax for reading from stdin using the *Scanner* class is as follows:

Scanner scan = new Scanner(System.in);

This creates a new *Scanner* object that reads from the *System.in* stream and can be accessed using the variable name scan. To read in information from stdin, you just need to apply Scanner's methods to your scanner object. Here are two basic examples:

scan.next(); // returns the next token of input

scan.hasNext(); // returns true if there is another token of input (false otherwise)

scan.nextLine() // returns the next LINE of input

scan.hasNextLine(); // returns true if there is another line of iput

Check out the comprehensive list of [Scanner methods](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html#method_summary) to learn more.   
  
When you are finished reading from an input stream, you should *close* it to avoid a [*resource leak*](https://en.wikipedia.org/wiki/Resource_leak). The following line of code closes scan:

scan.close();

Let's say we want to assign a value received from stdin to s, and then print it. We can accomplish this with the following code:

Scanner scan = new Scanner(System.in); // open scanner

String s = scan.next(); // read the next token and save it to 's'

scan.close(); // close scanner

System.out.println(s); // print 's' to System.out, followed by a new line

If the input token is Hi!, the above code will print Hi!.   
  
**Note:** You can also print text in quotes using *System.out.println*, or combine quoted text with a variable (e.g.: System.out.println("Input received: " + s);).

**Data Types**   
Data types define and restrict what type values can be stored in a variable, as well as set the rules for what types of operations can be performed on it.

[**Primitive Data Types**](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html)   
Java has 8 [primitive data types](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html): *byte*, *short*, *int*, *long*, *float*, *double*, *boolean*, and *char*. For most challenges, you'll only need to concern yourselves with ints (e.g.: 1, 2, 3, etc.) and doubles (e.g.: 1.0, 2.5, 3.9, etc.). Another important data type we mentioned yesterday is the String class, whose objects are immutable strings of characters.

[**Scanner**](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html)   
Yesterday, we discussed Scanner's *next*, *nextLine*, *hasNext*, and *hasNextLine* methods. Scanner also has readNext/hasNext methods for different data types, which is very helpful when you know exactly what type of input you'll be reading. The *next* methods scan for *tokens* (you can think of this as a word), and the *nextLine* methods reads from the Scanner's current location until the beginning of the next line. For example, *nextInt()* will scan the next token of input as an *int*, and *nextDouble()* will scan the next token of input as a *double*.   
Each line of multi-line input contains an invisible separator indicating that the end of a line of input has been reached. When you use Scanner functions that read tokens (e.g.: *next()*, *nextInt()*, etc.), the Scanner reads and returns the next token. When you read an entire line (i.e.: *readLine()*), it reads from the current position until the beginning of the next line. Because of this, a call to *nextLine()* may return an empty string if there are no characters between the end of the last read and the beginning of the next line. For example, given the following input:

a b c

d e

f

g

The breakdown below shows how the following sequence of calls to a Scanner object, scan, will read the above input:

1. A call to scan.next(); will return the next token, a.
2. A call to scan.next(); will return the next token, b.
3. A call to scan.nextLine(); will return the next token, c. Observe that the scanner reads a space *and* a letter, because it's reading from the end of the last token until the beginning of the next line.
4. A call to scan.nextLine(); will return the contents of the next line, d e.
5. A call to scan.next(); will read and return the next token, f.
6. A call to scan.nextLine(); will return everything after f until the beginning of the next line; because there are no characters there, it returns *an empty String*.
7. A call to scan.nextLine(); will return g.

[**Additive Operator**](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op1.html)   
The + operator is used for mathematical addition and String concatenation (i.e.: combining two Strings into one new String). If you add the contents of two variables together (e.g.: a + b), you can assign their result to another variable using the *assignment operator* (=). You can also pass the result to a function instead of assigning it to a variable; for example, if a = 1 and b = 2, System.out.println(a+b); will print 3 on a new line.

[**Operators**](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html)   
These allow you to perform certain operations on your data. There are 3 basic types:

1. *Unary*: operates on 1 operand
2. *Binary*: operates on 2 operands
3. *Ternary*: operates on 3 operands

**Arithmetic Operators**   
The binary operators used for arithmetic are as follows:

* +: Additive
* -: Subtraction
* \*: Multiplication
* /: Division
* %: Remainder ([modulo](https://en.wikipedia.org/wiki/Modulo_operation))

**Additional Operators**

* +: A binary operator used for String concatenation
* ++: This unary operator is used to *preincrement* (increment by 1 before use) when prepended to a variable name or *postincrement* (increment by 1 after use) when appended to a variable.
* --: This unary operator is used to *predecrement* (decrement by 1 before use) when prepended to a variable name or *postdecrement* (decrement by 1 after use) when appended to a variable.
* !: This unary operator means *not* (negation). It's used before a variable or logical expression that evaluates to true or false.
* ==: This binary operator is used to check the *equality* of 2 primitives.
* !=: This binary operator is used to check the *inequality* of 2 primitives.
* <, >, <=, >=: These are the respective binary operators for *less than*, *greater than*, *less than or equal to*, and *greater than or equal to*, and are used to compare two operands.
* &&, ||: These are the respective binary operators used to perform *logical AND* and *logical OR* operations on two boolean (i.e.: true or false) statements.
* ? : This ternary operator is used for simple conditional statements (i.e.: if ? then : else).

[**Boolean**](https://en.wikipedia.org/wiki/Boolean_data_type)   
A logical statement that evaluates to *true* or *false*. In some languages, *true* is interchangeable with the number *1* and *false* is interchangeable with the number *0*.

[**Conditional Statements**](https://en.wikipedia.org/wiki/Conditional_%28computer_programming%29)   
These are a way of programming different workflows depending on some boolean condition. The *if-else* statement is probably the most widely used conditional in programming, and its workflow is demonstrated below:

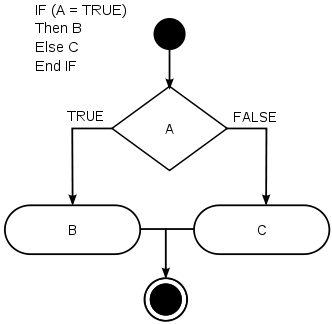


Image Source:[Wikipedia(Conditional Statements)](https://en.wikipedia.org/wiki/Conditional_%28computer_programming%29#/media/File%3aIf-Then-Else-diagram.svg)

The basic syntax used by Java (and many other languages) is:

if(condition){

// do this if 'condition' is true

}

else{

// do this if 'condition' is false

}

You can also use an if *without* an else, or follow an if(condition) with else if(secondCondition) if you have a second condition that only need be checked when condition is false. If the if (or else if) condition evaluates to true, any other sequential statements connected to it (i.e.: else or an additional else if)*will not execute*.

**Logical Operators**   
Customize your condition checks by using logical operators. Here are the three to know:

* *OR* (||), also known as logical disjunction
* *AND* (&&), also known as logical conjunction
* *NOT* (!), also known as negation

Here are some usage examples:

// if A is true and B is true, then C

if(A && B){

C;

}

// if A is true or B is true, then C

if(A || B){

C;

}

// if A is false (i.e.: not true), then B

if(!A){

B;

}

Another great operator is the *ternary* operator for conditional statements (? :). Let's say we have a variable, v, and a condition, c. If the condition is true, we want vto equal a; if the condition is false, we want v to equal b. We can write this with the following simple statement:

v = c ? a : b;

[**Switch Statement**](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html)   
This is a great control structure for when your control flow depends on a number of *known values*. Let's say we have a variable, condition, whose possible values areval0, val1, val2, and each value has an action to perform (which we will call behavior). We can *switch* between actions with the following code:

switch (condition) {

case val0: behavior0;

break;

case val1: behavior1;

break;

case val2: behavior2;

break;

default: behavior;

break;

}

**Note:** Unless you include break; at the end of each case statement, the statements will execute sequentially. Also, while it's good practice to include a default:case (even if it's just to print an error message), it's not strictly necessary.

[**Class**](https://en.wikipedia.org/wiki/Class-based_programming)   
A blueprint defining the charactaristics and behaviors of an object of that class type. Class names should be written in CamelCase, starting with a *capital* letter.

class MyClass{

...

}

Each class has two types of variables: [class variables](https://en.wikipedia.org/wiki/Class_variable) and [instance variables](https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html); class variables point to the same (static) variable across all instances of a class, and instance variables have distinct values that vary from instance to instance.

[**Class Constructor**](https://en.wikipedia.org/wiki/Constructor_%28object-oriented_programming%29)   
Creates an instance of a class (i.e.: calling the Dog constructor creates an instance of Dog). A class can have one or more constructors that build different versions of the same type of object. A constructor with no parameters is called a [default constructor](https://en.wikipedia.org/wiki/Default_constructor); it creates an object with default initial values specified by the programmer. A constructor that takes one or more parameters (i.e.: values in parentheses) is called a *parameterized constructor*. Many languages allow you to have multiple constructors, provided that each constructor takes different types of parameters; these are called [overloaded constructors](https://en.wikipedia.org/wiki/Function_overloading).

class Dog{ // class name

static String unnamed = "I need a name!"; // class variable

int weight; // instance variable

String name; // instance variable

String coatColor; // instance variable

Dog(){ // default constructor

this.weight = 0;

this.name = unnamed;

this.coatColor = "none";

}

Dog(int weight, String color){ // parameterized constructor

// initialize instance variables

this.weight = weight; // assign parameter's value to instance variable

this.name = unnamed;

this.coatColor = color;

}

Dog(String dogName, String color){ // overloaded parameterized constructor

// initialize instance variables

this.weight = 0;

this.name = dogName;

this.coatColor = color;

}

}

[**Method**](https://en.wikipedia.org/wiki/Method_%28computer_programming%29)   
A sort of named procedure associated with a class that performs a predefined action. In the sample code below, *returnType* will either be a data type or void if no value need be returned. Like a constructor, a method can have 0 or more parameters.

returnType methodName(parameterOne, ..., parameterN){

...

return variableOfReturnType; // no return statement if void

}

Most classes will have methods called *getters* and *setters* that get (return) or set the values of its instance variables. Standard getter/setter syntax:

class MyClass{

dataType instanceVariable;

...

void setInstanceVariable(int value){

this.instanceVariable = value;

}

dataType getInstanceVariable(){

return instanceVariable;

}

}

[**Parameter**](https://en.wikipedia.org/wiki/Parameter_%28computer_programming%29)   
A parenthetical variable in a function or constructor declaration (e.g.: in int methodOne(int x), the parameter is int x).

**Argument**   
The actual value of a parameter (e.g.: in methodOne(5), the argument passed as variable x is 5).

Terms you'll find helpful in completing today's challenge are outlined below, along with sample Java code (where appropriate). As you code more, you may see these loops implemented in different ways than are shown here.

[**For Loop**](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html)   
This is an iterative loop that is widely used. The basic syntax is as follows:

for (initialization; termination; increment) {

...

}

The *initialization* component is the starting point in your iteration, and your code for this section will typically be int i = 0. When we declare and initialize int i in the loop like this, we are creating a [*temporary variable*](https://en.wikipedia.org/wiki/Temporary_variable) that exists only inside this loop for the purposes of iterating through the loop; once we finish iterating and exit (or *break*) the loop, i is deleted and can be declared elsewhare in our program.

The *termination* component is the condition which, once met, you would like to exit (or *break*) the loop and proceed to the next line in your code. This is the ending point fr your loop, and is typically written as i < endValue, where i is the variable from the initialization section and endValue is some variable holding the stopping point for your iteration.

The *increment* component is executed each time the end of the code inside the loop's brackets is reached, and should generally be some modification on the initialization variable that brings it closer to the termination variable. This will typically be i++. The ++ operator is also called the [post-increment](https://en.wikipedia.org/wiki/Increment_and_decrement_operators) operator, and it will increment a variable by *1* after a line executes (for more detail and an example, see the *While* section).

To recap, this sample code:

int endOfRange = 4;

for(int i = 0; i < endOfRange; i++){

System.out.println(i);

}

produces this output:

0

1

2

3

[**While Loop**](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/while.html)   
This type of loop requires a single boolean condition and continues looping as long as that condition continues to be true. Each time the the end of the loop is reached, it loops back to the top and checks if the condition is still true. If it's true, the loop will run again; if it's false, then the program will skip over the loop and continue executing the rest of the code.

Much like in the *For* section, the code below prints the numbers *0* through *3*. Notice that we are using the *post-increment* operator on min:

int min = 0;

int max = 4;

while(min < max){

System.out.println(min++);

}

Once min ≥ max, the boolean condition (min < max) evaluates to false and the loop is broken. The line System.out.println(min++); is a compact way of writing:

System.out.println(min);

min = min + 1;

[**Do-While Loop**](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/while.html)   
This is a variation on the *While* loop where the condition is checked at the end of the brackets. Because of this, the content between the brackets is guaranteed to always be executed at least once:

do{

// this will execute once

// it will execute again each time while(condition) is true

} while(condition);

[**Unlabeled Break**](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html)   
You may recall the word break; from our previous discussion of *Switch Statements*. It will break you out of a loop even if the loop's termination condition still holds true.

[**Strings**](https://docs.oracle.com/javase/tutorial/java/data/strings.html)**and**[**Characters**](https://docs.oracle.com/javase/tutorial/java/data/characters.html)   
As we've mentioned previously, a String is a sequence of characters. In the same way that words inside double quotes signify a String, a single letter inside single quotes signifies a character. Each character has an [ASCII](http://www.asciitable.com/) value associated with it, which is essentially a numeric identifier. The code below creates a char variable with the value c, and then prints its ASCII value.

char myChar = 'c'; // create char c

System.out.println("The ASCII value of " + myChar + " is: " + (int) myChar);

Output:

The ASCII value of c is: 99

Observe the (int) before the variable name in the code above. This is called *casting*, which is a method of representing one thing as another. Putting a data type inside parentheses right before a variable is essentially saying: "The next thing after this should be represented as this data type".

To break a String down into its component characters, you can use the [String.toCharArray](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#toCharArray%28%29) method. For example, this code:

produces this output:

This is String example.

Notice that we were able to simulate printing *myString* by instead printing each individual character in the character array, *myCharArray*, created from *myString*.

[**Data Structures**](https://en.wikipedia.org/wiki/Data_structure)   
A way of organizing data that enables efficient storage, retrieval, and use.

[**Arrays**](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html)   
A type of data structure that stores elements of the same type (generally). You can think of an Array, *A*, of size *N* as a contiguous block of cells indexed from *0* to *N-1*which serve as containers for elements of the array's declared data type. To store an element, *value*, in some index, *i*, of array *A*, use the syntax A[i] and treat it as you would any other variable (i.e.: A[i] = value;). For example, the following code:

final int \_arraySize = 4; // the number of elements we want to hold

String[] stringArray = new String[\_arraySize]; // our array declaration

for(int i = 0; i < \_arraySize; i++){

stringArray[i] = "This is stored in index " + i; // set value for index i

System.out.println(stringArray[i]); // print value in index i

}

saves and then prints the values listed below in their respective indexes of stringArray:

This is stored in index 0

This is stored in index 1

This is stored in index 2

This is stored in index 3

Most languages also have a *method*, *attribute*, or *member* that allows you to retrieve the size of an array. In Java, arrays have a *length* attribute; in other words, you can get the length of some array, arrayName, by using the arrayName.length syntax.   
  
**Note:** The final keyword used in the code above is a means of protecting the variable's value by locking it to its initialized value. Any attempt to reassign (overwrite) the value of a final variable will generate an error.

**Note on Arrays in C++**   
If you want to create an array whose size is unknown at compile time (i.e.: being read as input), you need to create a [pointer](http://www.cplusplus.com/doc/tutorial/pointers/#declaration) to whatever data type you'll be declaring your array as (e.g.: char, int, double, etc.). Then you must use the [new operator](http://www.cplusplus.com/reference/new/operator%20new%5b%5d/) to set aside the space you need for your array. The example below shows how to create an array of type *DataType* and unknown size *n* that is read from stdin.

int n; // array size

cin >> n;

DataType\* arrayName = new DataType[n]; // create array of unknown size n

[**Map**](https://docs.oracle.com/javase/7/docs/api/java/util/Map.html)   
This is an [interface](https://docs.oracle.com/javase/tutorial/java/concepts/interface.html) that provides a blueprint for data structures that take *(Key, Value)* pairs and map Keys to their associated Values. The *implementation* is done by*implementing classes* such as [*HashMap*](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) or [*LinkedHashMap*](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html). Consider the following code:

Map myMap<String,String>; // declare a String to String map

myMap = new HashMap<String,String>(); // initialize it as a new String to String HashMap

myMap = new LinkedHashMap<String,String(); // change myMap to be a new (completely different) String to String LinkedHashMap instead

Here are a few Map methods you will find helpful for this challenge:

* *containsKey(Object key)*: Returns true if the map contains a mapping for *key*; returns false if there is no such mapping.
* *get(Object key)*: Returns the value to which the *key* is mapped; returns *null* if there is no such mapping.
* *put(K key, V value)*: Adds the (*Key, Value*) mapping to the Map; if the *Key* is already in the map, the *Value* is overwritten.

**Example**   
The code below:

// Create a Map of String Keys to String Values, implemented by the HashMap class

Map<String,String> myMap = new HashMap<String,String>();

myMap.put("Hi", "Bye"); // Adds ("Hi","Bye") mapping to myMap

System.out.println(myMap.get("Hi")); // Print the Value mapped to from "Hi"

myMap.put("Hi", "Bye!"); // Replaces "Bye" mapping from "Hi" with "Bye!"

System.out.println(myMap.get("Hi")); // Print the Value mapped to from "Hi"

produces the following output:

Bye

Bye!

It is not necessary to declare *myMap* as type *Map*; you can certainly declare it as a *HashMap* (the instantiated type).

Terms you'll find helpful in completing today's challenge are outlined below, along with sample Java code (where appropriate).

[**Recursion**](https://en.wikipedia.org/wiki/Recursion_%28computer_science%29)   
This is an algorithmic concept that involves splitting a problem into two parts: a base case and a recursive case. The problem is divided into smaller subproblems which are then solved recursively until such time as they are small enough and meet some base case; once the base case is met, the solutions for each subproblem are combined and their result is the answer to the entire problem.   
  
If the base case is not met, the function's recursive case calls the function again with modified values. The code must be structured in such a way that the base case is reachable after some number of iterations; otherwise, you will be stuck in the dreaded [infinite loop](https://en.wikipedia.org/wiki/Infinite_loop)!

**Example**   
The code below produces the multiple of two numbers by combining addition and recursion:

// Multiply 'n' by 'k' using addition:

private static int nTimesK(int n, int k){

System.out.println("n: " + n);

if(n > 1){ // recursive case

return k + nTimesK(n - 1, k);

}

else{ // base case n = 1

return k;

}

}

public static void main(String[] args) {

int result = nTimesK(4, 4);

System.out.println("Result: " + result);

}

When executed, this code prints:

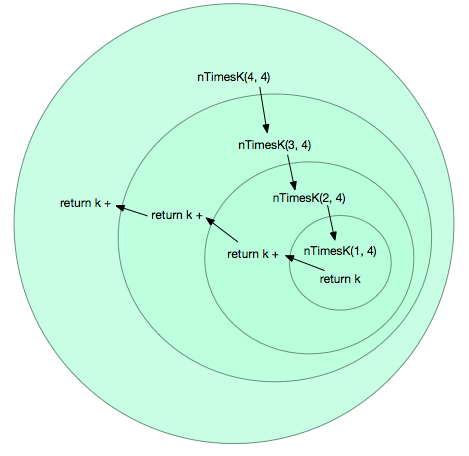
n: 4

n: 3

n: 2

n: 1

Result: 16

The diagram below depicts the execution of the code above. Each call to *nTimesK* is represented by a bubble, and each new recursive call bubble is stacked inside and on top of the bubble that was responsible for calling it. The function recursively calls itself using reduced values until it reaches the base case (*n=1*). Once it reaches the base case, it passes back the base case's return value (*k=4*) to the bubble that called it, and continues passing back k + the previously returned value until the final result (i.e.: the multiplication by addition result of *n × k*) is returned.   
  


Once the code hits the base case in the *4th* bubble, it returns *k* (which is *4*) to the *3rd* bubble.   
Then the *3rd* bubble returns k+4, which is *8*, to the *2nd* bubble.   
Then the *2nd* bubble returns k+8, which is *12*, to the *1st* bubble.   
Then the *1st* bubble returns k+12, which is *16*, to the first line in *main* as the result for *nTimesK(4, 4)*, which assigns *16* to the result variable.