Day 1: Functions

by AvmnuSng

All topics

Functions in JavaScript

Functions in JavaScript are declared using the `function` keyword. A function declaration creates a function that's a *Function* object having all the properties, methods, and behaviors of Function objects. By default, functions return the value `undefined`; to return any other value, the function must have a return statement that consists of the `return` keyword followed by the value to be returned (this can be a literal value, a variable, or even a call to a function).

- EXAMPLE

Take some time to review the code below that declares functions and prints their return values.

Input Format

The first line contains a string denoting $name$.
The second line contains two space-separated integers, a and b, to be summed.

```
'use strict';
process.stdin.on('data', function (data) {
    let input = String(data).split(new RegExp("[\n ]+"));
    main(input[0], +(input[1]), +(input[2]));
});
/**** Ignore above this line. ****/

function greetings(name) {
```

```
function sum(a, b) {
    return a + b;
}

function main(name, a, b) {
    greetings(name);
    console.log(sum(a, b));
}
```

Input

```
Julia
8 2
```

Run

Output

The Function Expression

A function expression is very similar to (and has almost the same syntax as) a function statement. The main difference between a *function expression* and a *function statement* is the function *name*, which can be omitted from a function expression to create an anonymous function. Function expressions are often used as Immediately Invoked Function Expressions (IIFEs), which run as soon as they're defined.

-    EXAMPLE

The code below demonstrates an *unnamed* function expression.

Input Format

A single integer denoting x.

```
'use strict';
process.stdin.on('data', function (data) {
    main(+(data));
});
/**** Ignore above this line. ****/

function main(input) {

    /**
    *   Defines an unnamed function and assigns it to a variable nam
    *   @param {Number} x
    *   @returns {Number} The value of argument squared.
    **/
    var square = function(x) {
        return x * x;
    };

    // Print the value returned by passing input as x to the
    // anonymous function referenced by variable square
    console.log(square(input));
}
```

Input

```
4
```

Run

Output

Named Function Expression

- EXAMPLE

The code below demonstrates a *named* function expression.

Input Format

Two space-separated integers describing the respective values of $factN$ and $fibN$.

```
'use strict';
process.stdin.on('data', function (data) {
    let input = String(data).split(new RegExp("[\n ]+"));
    main(+(input[0]), +(input[1]));
});
/**** Ignore above this line. ****/

```

```
    *   Defines a named recursive function as a property of the math
    *   @param {Number} n
    *   @returns {Number} The value of n factorial.
    **/
    var math = {
        // Declare the factorial property
        factorial:
            // Declare the function as the property's value
            function factorial(n) {
                if (n > 1) {
                    return n * factorial(n - 1);
                }
                // Returns 1 if n <= 1
                return 1;
            }
    };

    /**
    *   Defines a named recursive function referenced by the fib var
    *   @param {Number} n
    *   @returns {Number} The value of fibonacci(n).
    **/
    var fib = function fibonacci(n){
        if (n > 2) {
            return fibonacci(n - 1) + fibonacci(n - 2);
        }
        else if (n < 1) {
            return 0;
        }
        // else, return 1
        return 1;
    // function referenced by variable math:
    console.log(math.factorial(factN));
    // and by passing fibN as n to the function referenced by variab
    console.log(fib(fibN));

}
```

Input

```
5 11
```

Run

Output

- RECURSION Recommended Article

Recursion

View

Recursion

This is an extremely important 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, meaning that each subsequent modified value should bring you closer and closer to the base case; otherwise, you'll be stuck in the dreaded infinite loop!

It's important to note that any task that can be accomplished recursively can also be performed iteratively (i.e., through a sequence of repeatable steps). Recursive solutions tend to be easier to read and understand than iterative ones, but there are often performance drawbacks associated with recursive solutions that you're going to want to evaluate on a case-by-case basis. Typically, we use recursion when each recursive call significantly reduces the size of the problem (e.g., if we can halve the dataset during each recursive call). Regardless of the advisability of recursively solving a problem, it's extremely important to practice and understand *how* to recursively solve problems.

Example (Java)

Table Of Contents

OK

The code below produces the multiple of two numbers by combining addition and recursion.

Input Format
Two space-separated integers to be multiplied.

```
import java.util.*;

class Solution {
    // Multiply 'n' by 'k' using addition:
    private static int nTimesK(int n, int k) {
        // Print current value of n
        System.out.println("n: " + n);

        // Recursive Case
        if(n > 1) {
            return k + nTimesK(n - 1, k);
        }
        // Base Case n = 1
        else {
            return k;
        }
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int result = nTimesK(scanner.nextInt(), scanner.nextInt());
        scanner.close();
        System.out.println("Result: " + result);
    }
}
```

Input

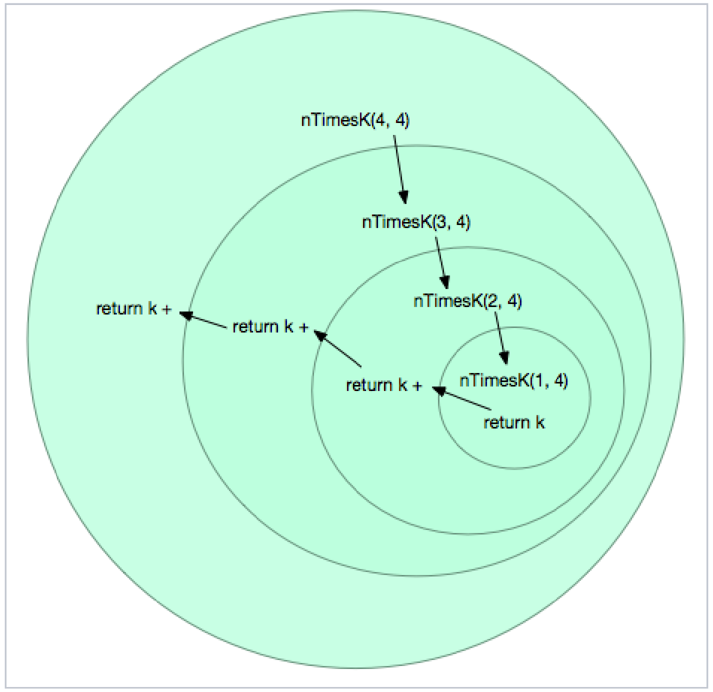
```
4 4
```

Run

Output

```
```

The diagram below depicts the execution of the code above using the default input (`4 4`). 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 $n \times k$) is returned.



Once the code hits the base case in the 4^{th} bubble, it returns k (which is 4) to the 3^{rd} bubble.
Then the 3^{rd} bubble returns $k+4$, which is 8, to the 2^{nd} bubble.
Then the 2^{nd} bubble returns $k+8$, which is 12, to the 1^{st} bubble.
Then the 1^{st} 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.

Related challenge for **Recursion**

Recursion: Fibonacci Numbers