Chapter 4

**Recursion**

**4.1 Objectives**

* The idea behind the recursion is: Take one *problem* and doing it *over* and *over* on a *smaller piece* or on a *changing* *piece* until you *hit* *some* *endpoint* which we call the base case.
* *Define* what *recursion* is and how it can be used
* Understand the *two* essential *components* of a *recursive function*
* Visualize the *call stack* to better *debug* and *understand* recursive functions
* Use *helper method recursion* and *pure recursion* to solve more difficult problems
* For example if we can only determine the first element of an array is even or odd, we do following:

**[12, 24, 56]** : 12 is even

**[24, 56]** : 24 is even

**[56]** : 56 is even

**[]** : is empty (base case)

**4.2 Why use RECURSION**

* Recursion: A process (a function in our case) that calls *itself*. It's just a process that calls itself. And in our case this process is a javascript function.
* Recursive Function: A recursive *function* is a function that can *call* *itself*. It's kind of *repeating loop* with a *base-point* (some endpoint to stop the loop). Without a base-point its an infinite loop.
* Recursive Multiple functions: There could be multiple functions that can call each-other in a recursive way.
* Imagine that we have a function called **checkforOdd**.

We would call it and then it check for odd and would call itself again on a smaller list and then that would call check for odds over and over again until an endpoint.

**It's EVERYWHERE!**

* **JSON.parse/JSON.stringify:** Sometimes we already used it without knowing. For example JSON.parse /JSON.stringify are oftenly recursive function.
* You've made an AJAX call and got a JSON back and turned that into javascript using JSON.parse then you may have called a recursive function.
* Now JSON.parse /JSON.stringify are up to the javascript-engine to implement. So Mozilla has their own. What's it called Rhino. Their own implementation.
* It doesn't have to be written recursively but it often is.
* **document.getElementByld and DOM traversal** algorithms are also use recursion. DOM is tree structure where we have all these nested things. In HTML it could be a hundred layers nested **div**s. And if we're trying to search through all of them one popular approach is to write code that does it *recursively object traversal* which is just like traversing JSON.
* Object traversal
* We will see it with more complex data structures: When we write our own *data structures* when we create *trees* or *graphs* and then we want to *traverse* them, *search* for something across them.
* It's sometimes a cleaner alternative to iteration: Remember you don't have to do anything recursively but it's a nice. It's cleaner and simpler to understand rather than doing it without recursion. A cleaner alternative to iteration.

We'll see some examples.

We're going to write a couple of functions both ways *with* and *without* *recursion*.

**4.3 The CALL STACK**

First we have to talk about what happens *behind the scenes* when a *recursive functions* are *called* in *javascript*. And this is important for us to understand in order to *write effective recursive* *code* that doesn't break.

* JS call-stack: In almost all programming languages there is some *data structure behind* the scenes that's manages what happens when *functions are invoked*. Often are *waiting* on *other functions* to *return*.
* Functions don't just happen randomly, like if you have a function called ***first*** and then instead of that function it calls a ***second*** function. You need them to go in the correct order.
* So there's a data structure in charge of that in javascript called the Call Stack. You can think of it as a stack of papers.

|  |
| --- |
| * **The call stack** * It's a *stack* data structure - we'll talk about that later! * Any time a function is invoked it is *placed (pushed)* on the *top* of the call stack * When *JavaScript* sees the *return keyword* or when the *function ends*, the compiler will *remove* (pop) from the top. * Eg: ***F1***, ***F2***, .. ***Fn*** functions are in staked order (recursive). ***Fn*** is in the ***top*** of the ***stack***, then when function ***ends***, ***compiler*** will start ***removing*** from ***Fn***. |

* Analogy of a stack of papers on a desk is that you *put something on top*, then when you remove something you also *remove from* *the top* you don't remove from the bottom of a stack of papers. So that's the idea of the call stack.
* Example 4.1 - Callback Example:

function takeShower(){

    return "Showering!";

}

​

function eatBreakfast(){

    let meal = cookFood();

    return `Eating ${meal}`;

}

​

function cookFood(){

    let items = ["Oatmeal", "Eggs", "Protein Shake"];

    return items[Math.floor(Math.random()\*items.length)];

}

​

function wakeUp() {

    takeShower();

    eatBreakfast();

    console.log("Ok ready to go to work!");

}

​

wakeUp()

* Call stack:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| wakeUp() | takeShower()  wakeUp() | wakeUp() | eatBreakfast()  wakeUp() | cookFood()  eatBreakfast()  wakeUp() | eatBreakfast()  wakeUp() | wakeUp()  Finally execute:  console.log("..!") |

|  |
| --- |
| * Why do I care? * You're used to *functions* being pushed on the call stack and popped off when they are done * When we write *recursive functions*, we keep pushing new functions onto the call stack! That is it still a *call stack* but *with same* *function*. |

**4.4 How recursive functions work**

*Invoke* the *same function* with a different input until you *reach* your *base* case!

* Base Case: The condition when the recursion ends. This is the most important concept to understand
* Two essential parts of a recursive function!

1. *Base* *Case*
2. *Different* *Input*

* Example 4.2: Countdown number using recursion.

|  |  |
| --- | --- |
| // Recursive Version  function **countDown**(num){      if(num <= 0) {          console.**log**("All done!");          return;      }      console.**log**(num);      num--;  **countDown**(num);  }  **countDown**(3) |  |

* Notice the base case is " **if(num <= 0) { return;}**"**.** If we do not use " **return;**" it will be an *infinite* *recursion*.
* We *decrement* the number in each step " **num--;**". So in each step the input is different.
* Iterative version using for loop.

// Iterative Version

function **countDown**(num){

    for(var i = num; i > 0; i--){

        console.**log**(i);

    }

    console.**log**("All done!")

}

* Example 4.3: Calculate sum using recursion

|  |  |
| --- | --- |
| function **sumRange**(num){     if(num === 1) return 1;     return num + **sumRange**(num-1);  }  **sumRange**(4)   * Notice the base case is " **if(num === 1) return 1;**"**.** If we do not use " **return 1;**" it will be an *infinite* *recursion*. * We *decrement* the number in each step "**sumRange**(num-1)". So in each step the input is different. | Call stack:  **sumRange**(4)  return 4 + **sumRange**(3)  return 3 + **sumRange**(2)  return 2 + **sumRange**(1)  return 1  **sumRange**(4)  return 4 + **sumRange**(3)  return 3 + **sumRange**(2)  return 2 + **1**  **sumRange**(4)  return 4 + **sumRange**(3)  return 3 + 2 + **1**  **sumRange**(4)  return 4 + 3 + 2 + **1** |

* Example 4.4: Writing *Factorial Iteratively*.

function **factorial**(num){

    let total = 1;

    for(let i = num; i > 1; i--){

        total \*= i

    }

    return total;

}

* Example 4.5: Writing *Factorial Recursively*.

**Practiced Version**

// Recursively

function **factorial**(num){

        if(num === 1) return 1;

        return num\***factorial**(num-1);

}

Notice, it is identical to our previous **sumRange()**

* Notice the base case is " **if(num === 1) return 1;**"**.** If we do not use " **return 1;**" it will be an *infinite* *recursion*.
* We *decrement* the number in each step " **factorial**(num-1)". So in each step the input is different.

**4.5 Common Recursion Pitfalls**

|  |  |
| --- | --- |
| 1. No base case or wrong base case:   Forgetting base case or using wrong base case can cause infinite loop (infinite call-stack.)   * Forgetting to use base case:   if(num === 1) return 1;   * Forgetting to use "return" to end the recursion:   if(num === 1) console.log(1);   1. Returning wrong thing: Returning wrong thing can also cause infinite loop (infinite call-stack.)   return num\***factorial**(num); |  |

* Stack Overflow: All of those cases we end up with ***"VM91:4 Uncaught RangeError: Maximum call stack size exceeded"***.
* It is a "Stack Overflow Error", where too many call stack appear.
* All programming languages have a limit for "call-stack". In case of *JS* /node it is around *10000*.

**4.6 Helper Method Recursion**

* Helper Method Recursion is a design pattern that's commonly used with recursion. When a function called recursively, we cannot use local variables to store recursive values. When function is recursively called, those *local variables re-initiated* and lost the previous value.
* For these reason a *helper-function* is used to do the recursion job *inside* of another function.
* The Container function contains the variable which will store the values from the *helper-recursive-function*.

function **collectOddValues**(arr){

    let result = [];

    function **helper**(helperInput){

        if(helperInput.length === 0) {

            return;

        }

        if(helperInput[0] % 2 !== 0){

            result.**push**(helperInput[0])

        }

**helper**(helperInput.**slice**(1))

    }

**helper**(arr)

    return result;

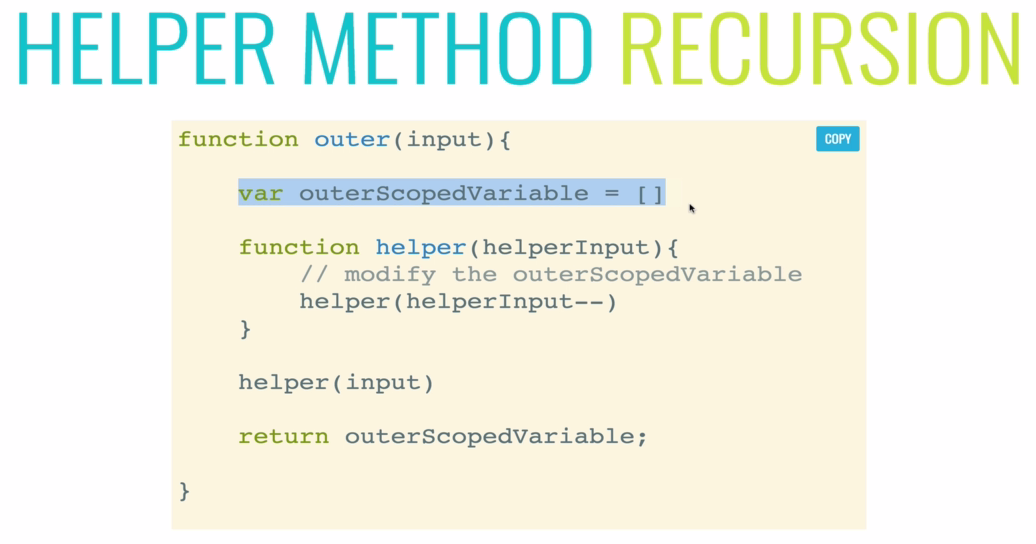
}

**collectOddValues**([1,2,3,4,5,6,7,8,9])

* We have two functions, the outer function **collectOddValues** and then inside we have our recursive function **helper** (which is called helper function).
* If we're doing something like this: *collecting all the odd values* in an array it's very easy to do helper method recursion.
* We create an empty array inside the *recursive-function*, it will be *re-initialized* every time the *function* is *called*. We need to use a *global* *variable* to our *recursive function*. That's why we defined our *array* *outside* the *recursive* *function* but *inside* the *outer* *function*.

1. **helper**(helperInput) takes an array.
2. It test "is array is empty", the base case.
3. It takes the first-value of the array, if it is odd then it put into the outer "***result***" variable.
4. Lastly it drops the first element from the array using ***helperInput.slice(1)*** and make a recursive call.

* In this example, **collectOddValues**, is the container of the *Helper-Recursive Function* **helper**. **collectOddValues** also contains the variable ***result*** which is used in **helper**.



* We'll use more *helper recursion* in *Graphs*, *Graphs-Depth-First-Search*, *traversals* etc. It is common use in different algorithms and Data-Structures.

**4.7 Pure Recursion**

We can do the previous example without using "*Helper Recursive method*", we can just use "Pure Recursive way". However using *Pure* *Recursion* is *not* the *best solution* for this case, it creates *more* *complexity*.

* Pure recursion meaning the function itself is totally self-contained and its recursive. We don't have some external data structure like we had this external ***result*** variable. We're not doing a nested function helper method recursion.
* It's just a little more challenging. So here's the code:

function **collectOddValues**(arr){

    let newArr = [];

    if(arr.length === 0) {

        return newArr;

    }

    if(arr[0] % 2 !== 0){

        newArr.**push**(arr[0]);

    }

    newArr = newArr.**concat**(**collectOddValues**(arr.**slice**(1)));

    return newArr;

}

**collectOddValues**([1,2,3,4,5])

* The main point to understand is that, when *recursion* occurs the value in **newArr** is stored in *call-stack* **newArr = [1]** is still there, but in next recursive call **newArr** is re-initialized to **[]**. But using ***concat*** makes a copy of that array ***[1]*** in the *previous* ***call-stack***. That’s why the following call-stack happens.

**Call-stack**

|  |
| --- |
|  |

* So along the way we use ***concat*** to help build up the result. If we just tried to ***push*** into new array over and over. It would be ***reassigned*** instead. **newArr** is just going to allow us to concatenate.
* However, the Helper Method Recursion is more straightforward.
* You can always do it using pure recursion you can.
* I'll give you a couple of tips when you're using *arrays* and you're trying to do a *pure recursive solution* without a helper method.

