# Deep Equals

# Deep Equals

## Instructions

Write a function that will take in two items of any type. The function should perform a deep equality (http://adripofjavascript.com/blog/drips/object-equality-in-javascript.html) check.

**Inputs**: Any, Any

**Output**: Boolean

## Hints

- Deep equality is used to check equivalence of objects and arrays. Because of the concept of value vs. reference (https://www.educative.io/collection/page/5679346740101120/5707702298738688/5685265389584384) in JavaScript, the equality operators ( `==` , `===` ) can't help us. They'll always return `false` for two different arrays or objects even if they contain the same items.
- If we're comparing objects or arrays, we need to go into them and check that each item is the same. If the item contains other arrays or objects, we need to go into those items as well.
- It's entirely possible to have objects or arrays nested several levels deep. Our function will have to drill all the way down into every object.

- This problem tests several JavaScript concepts:
  - Value vs. reference
  - Quirks of different data types such as `NaN` and `null`
  - Use of `typeof`
  - Ability to reuse code

```
function deepEquals(a, b) {
    // Your code here
}
```

# Solution

This problem has several parts. Let's walk through it logically.

1. comparing `NaN` with `NaN` using `==` or `===` will always return `false`. Because of this, we need to start by checking if both inputs are `NaN`.
2. If the items are of a different type, we can immediately return `false`.
3. If they're the same type and not objects, we can return the result of a simple equality operation using `===`. We have to be careful to include `null` here, as `typeof null` returns `object`. This is due to a bug in JavaScript (http://2ality.com/2013/10/typeof-null.html) itself.
4. We need to be careful to ensure that both objects have the same number of properties. If they don't, return `false`.
5. Now we need to go through every key and make sure they're the same on every object. To do this, we can loop through the first object and check to make sure that the second object has matching values.

6. If all checks have passed, return `true`

Here's all of this information in pseudocode.

```
// Ensure that a and b are deeply equivalent.
function deepEquals(a, b) {
    // 1. If both are NaN
        // return true

    // 2. If they are different types
        // return false

    // 3. If they are not objects or either one is null
        // return an equality comparison

    // 4. Ensure that both objects have the same
    // number of properties. If not
        // return false.

    // 5. Loop through all keys of the objects. Ensure that
    // the values are identical on both objects. If not
        // return false.

    // 6. Return true.
}
```

Let's transform that pseudocode into code.

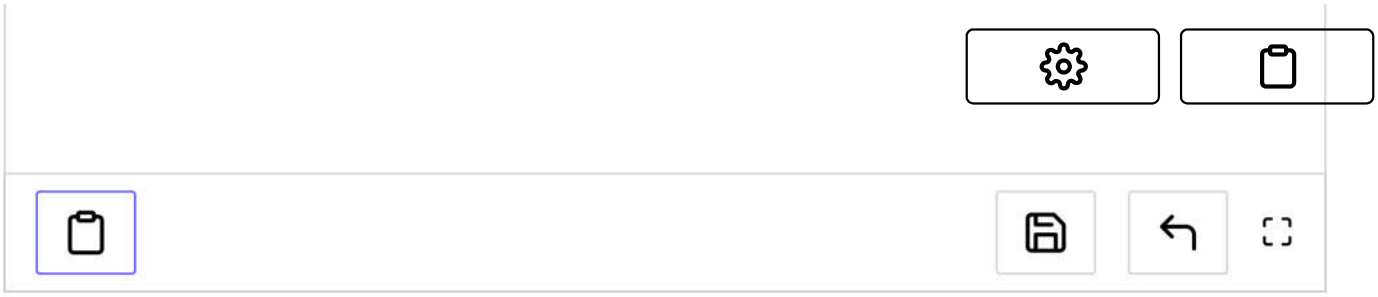
```
function deepEquals(a, b) {
    if(Number.isNaN(a) && Number.isNaN(b)) {
        return true;
    }

    if(typeof a !== typeof b) {
        return false;
    }

    if(typeof a !== 'object' || a === null || b === null) {
        return a === b;
    }

    if(Object.keys(a).length !== Object.keys(b).length) {
        return false;
```

```
    }

    for(const key in a) {
        // Ensure that the values are identical on both objects. If not
            // return false.
    }

    return true;
}
```

The only thing left to do is ensure that everything stored in the objects is identical. We need to be sure they're both deeply equal to each other.

To do this, we need to repeat steps 0 - 4 above for every property. This is a great indication that our function should be recursive.

```
function deepEquals(a, b) {
    if(Number.isNaN(a) && Number.isNaN(b)) {
        return true;
    }

    if(typeof a !== typeof b) {
        return false;
    }

    if(typeof a !== 'object' || a === null || b === null) {
        return a === b;
    }

    if(Object.keys(a).length !== Object.keys(b).length) {
        return false;
    }

    for(const key in a) {
        if(!deepEquals(a[key], b[key])) {
            return false;
        }
    }

    return true;
}
```



## Time

Our function has to go through and process every property of both items. Assuming that the two inputs are deeply equivalent, we can say that the time complexity is:

O(n).

## Space

This is a recursive function and the number of calls will generally scale with the objects' sizes, so:

O(n).

← Back

Linked List Cycles

Next →

Memoized Fibonacci

Mark as Completed

Report an Issue