Shallow Copy

- A shallow copy in ECMAScript (JavaScript) is a copy of an object where only the first level (top-level properties) is duplicated, while nested objects, arrays, or references are copied by reference.
- This means that changes made to the nested objects in the original will also affect the copied object and vice versa, as they share the same references.

For a shallow copy

- Primitive data types like numbers, strings, booleans are copied directly, meaning the value is duplicated.
- Reference types (e.g., objects, arrays, functions) are copied by reference, meaning the new object still points to the same memory location as the original. This can lead to unintended side effects if changes are made to the nested data.

Key Concepts

1. **Primitive Values vs. Reference Types**
   - **Primitive Values:** When a primitive value (e.g., number, string, boolean) is copied, a new value is created, and there is no linkage between the original and copied value.
   - **Reference Types:** When copying reference types, such as arrays or objects, the shallow copy creates a new reference to the original nested objects. This means any modification to the nested object will reflect in both the original and the copied version.

2. **One-Level Copying**
   - A shallow copy duplicates the direct properties of an object.
   - If a property holds an object or array, that property will still point to the original object/array.
   - This means the copy isn't entirely independent of the original.

Methods for Shallow Copy

- **Object.assign():** Creates a shallow copy by copying enumerable properties from a source object to a target object. It does not copy nested objects or arrays by value—only by reference.

```
const obj = { name: "mohit", address: { city: "Agra" } };
const copyObj = Object.assign({}, obj);
```

- **Spread Operator (...):** It creates a new object or array by copying the top-level properties of the original object or array.

```
const arr = ["apple", "banana", { category:"fruit" }];
const copyArr = [...arr];
```

- **Array.prototype.slice():** Used to create a shallow copy of an array.

```
const arr = ["apple", "banana","pineapple", { category:"fruit" }];
const copyArr = arr.slice();
```

Behaviour of Shallow Copy

Shared References: Since reference types are only copied by reference, modifications to the nested properties in the original object or array will also impact the shallow copy.

Example

```
const original = {
  name: "Mohit",
  details: {
    profession: "Technical Trainer",
    city: "Agra"
  }
}
const shallowCopy = { ...original }
console.log(shallowCopy);
// Modifying the nested object
        original.details.city = "Gurugram"
        console.log(original.details.city)
        console.log(shallowCopy.details.city)
```

Independent Copies for Top-Level Properties: If you change a top-level property in the original object, the shallow copy of object remains unchanged because that property has been copied independently.

Example

```
const original = {
  name: "Mohit",
  details: {
    profession: "Technical Trainer",
    city: "Agra"
  }
}
const shallowCopy = { ...original }
console.log(shallowCopy);
//modifying the object
        original.name="Mohit Singh"
        console.log(original.name)
        console.log(shallowCopy.name)
```

Example: Shallow Copy of an Object with Object.assign()

```
const person = {
  name: "mohit",
  hobbies: ["mentoring", "traveling","cooking"]
};
const shallowCopy = Object.assign({}, person);

// Modify the nested array
person.hobbies.push("development");
console.log(person.hobbies);
console.log(shallowCopy.hobbies);
```

Example: Shallow Copy of an Array with Spread Operator

```
const fruits = ["apple", "banana", { category:"fruit" }];
const shallowCopy = [...fruits];
// Modify the nested object
fruits[2].category= "food";
console.log(fruits[2].category);
console.log(shallowCopy[2].category);
```

Example: Shallow Copy with Array.prototype.slice()

```
const fruits = ["apple", "banana", { type: "juicy", name: "orange" }];
const shallowCopy = fruits.slice();
// Modify the nested object
fruits[2].name = "lemon";
console.log(fruits[2].name);
console.log(shallowCopy[2].name);
```

Example: Shallow Copy of a Nested Object with Spread Operator

```
const car = {
                brand: "Suzuki",
                model: "Swift",
                specs: {
                        engine: "BS6",
                        transmission: "Manual"
                       }
            };
const shallowCopy = { ...car };
// Modify the nested object
car.specs.engine = "BS7";
console.log(car.specs.engine);
console.log(shallowCopy.specs.engine);
```

Example: Shallow Copy with Object.create()

```
const obj = { name: "mohit", address: { city: "Agra" } };
const shallowCopy = Object.create(obj);
// Modify the nested property
obj.address.city = "gurugram";
console.log(obj.address.city);
console.log(shallowCopy.address.city);
```

Example: Shallow Copy Using concat()

```
const obj = [ "mohit singh", "technical trainer", {city: "Agra" } ];
const shallowCopy = obj.concat([]);
// Modify the nested property
obj[2].city = "gurgaon";
console.log(obj[2].city);
console.log(shallowCopy[2].city);
```

Example: Shallow Copy with Array.from()

```
const array = ["mohit singh", {city: "Agra" }, "technical trainer"];
const shallowCopy = Array.from(array);
// Modify the nested object
array[1].city = "Noida";
console.log(array[1].city);
console.log(shallowCopy[1].city);
```

Example: Shallow Copy of an Array with forEach()

```
const students = [{ name:"tushar" }, { name:"nishi" }, { name:"manisha" }];
const shallowCopy = [];
students.forEach(item => {
        shallowCopy.push(item);
});
// Modify one of the nested objects
students[0].name = "saurav pandey";
console.log(students[0].name);
console.log(shallowCopy[0].name);
```

Implications of Using Shallow Copy

Advantages

- **Performance:** Shallow copies are faster and less memory-intensive than deep copies, making them efficient for copying objects that don't require complete duplication of nested data structures.
- **Use Cases:** Ideal for when only the top-level properties need to be duplicated, and nested data doesn't need to be modified independently.

Disadvantages

- **Unintended Side Effects:** If the original object or array contains complex nested data structures, changes in one of the nested properties can lead to unintended changes in the copied version.
- **Lack of Independence:** The copied version and the original share nested references, meaning they are not truly independent.

Common Mistakes

- Assuming that a shallow copy is independent, which often leads to issues like modifying a shared nested object, thus introducing bugs.
- Shallow copies are inappropriate for deeply nested data structures that require complete independence between the original and the copy.

Points to notice

- Shallow Copy is efficient and suitable for copying only the top-level structure of objects or arrays when you do not need to modify nested data independently.
- Shallow copying methods include Object.assign(), the spread operator (...), slice(), and concat().
- The main limitation is that the nested references are not duplicated, meaning that changes to nested structures in one version (either the original or the copy) will affect the other.
- For scenarios involving complex nested data, a deep copy is more appropriate to avoid shared references and unintended side effects.
- The alternatives to Shallow Copy are Deep Copy and Deep Copy Functions.

Deep Copy

- A deep copy is a complete duplication of an object or array, where all nested objects and arrays are recursively copied, creating entirely new instances for each level.
- A deep copy will recursively copy every level of an object, ensuring that the original and the copy do not share any references.
- This ensures that the copied structure is completely independent of the original, meaning changes in the original object or array do not affect the deep copy and vice versa.

Key Concepts

- **Recursive Duplication:** Unlike a shallow copy, a deep copy goes through every level of the original object or array, creating new instances for each nested element. It prevents shared references between the original and the copy.
- **Independence:** Deep copies ensure complete independence between the original and copied versions. Changes made in the nested structures of the original will not impact the deep copy.
- **Complexity:** Creating a deep copy is more complex and computationally expensive compared to a shallow copy, especially for deeply nested objects.

Methods for Creating Deep Copies

- **JSON.parse(JSON.stringify(object)):** Converts an object to a JSON string and then parses it back into a new object.

  Limitations: It cannot handle circular references, functions, undefined, Symbol, or special data types like Date or RegExp.

- **Recursive Functions:** Writing custom recursive functions to create a deep copy that can handle various types of values.

- **Libraries:** Lodash (_.cloneDeep()) and other libraries provide utility functions to handle deep copying effectively.

JSON : JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is commonly used to transfer data between a server and a web application, and its syntax is similar to JavaScript object literals.

Example

```
{
        "name": "Mohit Singh",
        "age": 34,
        "isStudent": false,
        "hobbies": ["mentoring", "traveling", "cooking"],
        "address": {
                "city": "Agra",
                "county": "India"
            }
}
```

In the example

- "name" is a string.
- "age" is a number.
- "isStudent" is a boolean.
- "hobbies" is an array of strings.
- "address" is an object containing nested properties ("city" and "country").

Example: Deep Copy with JSON.parse(JSON.stringify())

```
const original = {
        "name": "Mohit Singh",
        "address": {
                "city": "Agra",
                "county": "India"
        }
}
const deepCopy = JSON.parse(JSON.stringify(original));
// Modifying the nested property
original.address.city = "Gurugram";
console.log(original.address.city);
console.log(deepCopy.address.city);
```

Example: Limitation of JSON.parse(JSON.stringify())

```
const original = {
        "name": "Nisha Mahla",
        birthDate: new Date(),
         message: function () {
                console.log("Hello!");
         }
};

const deepCopy = JSON.parse(JSON.stringify(original));
console.log(deepCopy.birthDate instanceof Date); // Output: false (now a string)
console.log(typeof deepCopy.message); // Output: undefined
```

In above program, the Date object is converted to a string, and the message function is lost because JSON cannot handle these types.

Example: Deep Copy Using a Custom Recursive Function

```
function deepCopy(obj) {
        if (obj === null || typeof obj !== "object") {
                return obj;
        }

        let copy = Array.isArray(obj) ? [] : {};

        for (let key in obj) {
                if (obj.hasOwnProperty(key)) {
                        copy[key] = deepCopy(obj[key]);
                }
        }
        return copy;
}

const originalObject = { name: "mohit", address: { city: "Agra" } };
const copiedObject = deepCopy(originalObject);
originalObject.address.city = "Noida";
console.log(originalObject.address.city);
console.log(copiedObject.address.city);

const originalArray=["Agra",{ncrcity:"noida"}, "mumbai"]
const copiedArray=deepCopy(originalArray)
originalArray[1].ncrcity="new delhi"
console.log(originalArray[1].ncrcity)
console.log(copiedArray[1].ncrcity)
```

In above program, the custom function creates the true deep copies of the original object & original array by recursively copying each nested element.

Example: Deep Copy with Arrays

```
const array = [1, [2, 3], [4, [5, 6]]];
const deepCopy = JSON.parse(JSON.stringify(array));
array[1][0] = 21;
console.log(array);
console.log(deepCopy);
```

Example: Deep Copy with Lodash

```
const _ = require("lodash");
const original = {
  name: "Mohit Singh",
  hobbies: ["mentoring", "traveling"],
  address: {
    city: "Agra",
    state: "Uttar Pradesh"
  }
};

const deepCopy = _.cloneDeep(original);

// Modify the nested property
original.address.city = "Noida";
console.log(original.address.city);
console.log(deepCopy.address.city);
```

In the above program _.cloneDeep() from Lodash is used to create a deep copy that maintains independence from the original object.

To run the above code, you need to install Lodash library using the below command in the vs-code terminal

npm install lodash

and you can run the javascript code using command, npm javascipt_file_name.js.

note: make sure, you must have node js installed on your machine.

Example: Deep Copy of Objects with Functions

```
const _ = require("lodash");
const original = {
                    name:"Mohit",
                    method: function () {
                                console.log("My Name is :", this.name);
                     }
   };

const deepCopy = _.cloneDeep(original);

original.name = "Mohit Singh";
original.method();
deepCopy.method();
```

Difference between Shallow Copy and Deep Copy

Feature	Shallow Copy	Deep Copy
Definition	A shallow copy duplicates the top-level properties of an object but keeps references to nested objects.	A deep copy creates a completely new instance of an object, including all nested properties, resulting in independent copies.
Reference Handling	Nested objects are not duplicated; instead, references to the original objects are maintained.	Each nested object is duplicated, ensuring that changes to the original do not affect the copy.
Memory Consumption	Uses less memory since it shares references for nested structures.	Uses more memory as it creates separate instances for each nested property.
Performance	Faster to create, as it involves copying only the top-level properties.	Slower due to the recursive nature of copying every level of nested properties.
Impact of Changes	Changes to nested properties in the original object affect the shallow copy, leading to potential side effects.	Changes in the original object do not affect the deep copy, allowing for safe modifications without unintended consequences.
When to Use	Ideal when only top-level properties are relevant, or when shared references are acceptable. Common in scenarios like cloning simple objects or settings.	Best for scenarios requiring complete independence, such as when working with complex, nested data structures, or when avoiding side effects is crucial.
Example Scenario	Duplicating a configuration object where the nested data can remain shared.	Cloning a user profile with nested preferences to ensure changes do not impact the original profile.

Enhanced Object Literals

- Enhanced Object Literals in ECMAScript (ES6) provide a more concise and expressive way to create objects.
- This feature simplifies object creation, making it easier to define properties and methods.
- Enhanced object literals in ES6 make it easier and more intuitive to create objects with less code.
- They offer features like property and method shorthand, dynamic and computed property names, and more concise object creation.
- This leads to cleaner and more maintainable code.
- By leveraging enhanced object literals, you can write more expressive, compact, cleaner and more maintainable code in JavaScript.

Use Cases

- **Creating Configuration Objects:** When you need to define configuration objects with several properties.
- **Defining Classes:** When defining classes, enhanced object literals simplify the process of defining methods.
- **Dynamic Object Creation:** When property names need to be generated dynamically based on conditions or variables.

Key Features

- **Property Shorthand:** If the property name is the same as the variable name, you can omit the property name in the object literal.

  Example

```
const name = "Sunidhi Sharma";
const age = 19;
const person = {
  name, // shorthand for name: name
  age   // shorthand for age: age
};
console.log(person);
```

- **Method Shorthand:** You can define methods in object literals without the function keyword.
  Example

```
const person = {
  name: "Sunidhi Sharma",
  message() { // shorthand method
    console.log(`Hello, my name is ${this.name}.`);
  }
};

person.message();
```

- **Dynamic Property Names:** You can use expressions to compute property names in object literals using square brackets.
  Example

```
const key = "age";
const value = 19;
const person = {
                name: "Sunidhi Sharma",
                [key]: value // dynamic property name
};
console.log(person);
```

- **Computed Property Names:** When defining properties, you can use computed property names to dynamically assign property keys based on expressions.
  Example

```
const base = "item";
const obj = {
  [base + "1"]: "laptop",
  [base + "2"]: "mobile"
};
console.log(obj);
```

- **Concise Object Creation:** Enhanced object literals allow for more concise object creation, reducing boilerplate code.
  Example

```
const firstName = "Sunidhi";
const lastName = "Sharma";
const person = {
  firstName,
  lastName,
  fullName() {
    return `${this.firstName} ${this.lastName}`;
  }
};
console.log(person.fullName());
```

- **Inheritance:** Enhanced object literals can also be used in conjunction with the Object.create() method to create objects with a specific prototype.
  Example

```
const proto = {
        message() {
                console.log("Hello! Welcome to Learn2Earn Labs");
        }
};
const person = Object.create(proto, {
                        name: { value: "Sunidhi", writable: true },
                        age: { value: 19, writable: true }
});
person.message();
```

  note: The writable attribute is a powerful feature when defining properties in JavaScript objects. It allows you to control whether or not the property values can be changed, contributing to more robust and maintainable code.
  - The writable property is a boolean value (true or false) that determines whether the property's value can be changed after it has been set.
  - If writable is true, the property can be updated; if false, attempts to change the property's value will not have any effect, and in strict mode, it will throw a TypeError.