**SHALLOW COPY V/S DEEP COPY**

In JavaScript, understanding shallow and deep copying is crucial for working with objects and arrays efficiently and avoiding unintended side effects.

**Shallow Copy**

A shallow copy of an object is a copy whose properties reference the same values as the properties in the original object. If the property values are references to other objects, just the reference addresses are copied.

**Methods to Create Shallow Copy:**

1. **Using Object.assign()**

let original = { a: 1, b: { c: 2 } };

let copy = Object.assign({}, original);

Note: This does not deeply copy nested objects.

1. **Using Spread Operator ...**

let original = { a: 1, b: { c: 2 } };

let copy = { ...original };

Note: Like Object.assign(), this does not deeply copy nested objects.

**Deep Copy**

A deep copy of an object is a copy whose properties do not reference the same values as those in the original object. Instead, copies of the values are created, including copies of any objects nested within.

**Methods to Create Deep Copy:**

1. **Using JSON stringify and parse**

let original = { a: 1, b: { c: 2 } };

let copy = JSON.parse(JSON.stringify(original));

Note: This method has limitations, such as not copying functions, undefined, Infinity, and other special values correctly.

1. **Using a 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;

}

let original = { a: 1, b: { c: 2 } };

let copy = deepCopy(original);

This method ensures all nested objects are copied correctly.

**Object Properties**

In JavaScript, objects can have different types of properties, including:

1. **Data Properties**: These properties hold values directly.

let obj = {

name: "John",

age: 30

};

1. **Accessor Properties**: These properties are defined by getter and setter functions.

let person = {

firstName: "John",

lastName: "Doe",

get fullName() {

return `${this.firstName} ${this.lastName}`;

},

set fullName(name) {

[this.firstName, this.lastName] = name.split(" ");

}

};

**Property Attributes**

Every property in a JavaScript object has certain attributes:

* **Writable**: If true, the property value can be changed.
* **Enumerable**: If true, the property is listed during enumeration of the properties.
* **Configurable**: If true, the property can be deleted and its attributes can be changed.
* **Value**: The value associated with the property (for data properties).
* **Get**: The getter function (for accessor properties).
* **Set**: The setter function (for accessor properties).

You can define or modify these attributes using Object.defineProperty:

let obj = {};

Object.defineProperty(obj, 'name', {

value: 'John',

writable: true,

enumerable: true,

configurable: true

});

Understanding these concepts is fundamental for effective JavaScript programming, especially when dealing with complex data structures and ensuring immutability when needed.

**STATIC AND NON-STACIC**

In JavaScript, the concept of "static" and "non-static" properties and methods is often discussed in the context of classes. Static properties and methods belong to the class itself, while non-static (or instance) properties and methods belong to instances of the class.

**Static Properties and Methods**

* **Static properties and methods** are defined on the class itself.
* They are not accessible from instances of the class, only from the class directly.

**Non-Static (Instance) Properties and Methods**

* **Non-static properties and methods** are defined on instances of the class.
* Each instance has its own set of these properties and methods.

Here's a basic example to illustrate the difference:

**Example:**

class Car {

// Static property

static numberOfWheels = 4;

// Constructor to define non-static properties

constructor(make, model) {

this.make = make; // Non-static property

this.model = model; // Non-static property

}

// Static method

static description() {

return `A car typically has ${Car.numberOfWheels} wheels.`;

}

// Non-static method

getDetails() {

return `This car is a ${this.make} ${this.model}.`;

}

}

// Accessing static properties and methods

console.log(Car.numberOfWheels); // Output: 4

console.log(Car.description()); // Output: A car typically has 4 wheels.

// Creating an instance of the Car class

let myCar = new Car('Toyota', 'Corolla');

// Accessing non-static properties and methods

console.log(myCar.make); // Output: Toyota

console.log(myCar.model); // Output: Corolla

console.log(myCar.getDetails()); // Output: This car is a Toyota Corolla.

// Trying to access static properties and methods from an instance (this will cause an error)

console.log(myCar.numberOfWheels); // Output: undefined

console.log(myCar.description()); // Output: TypeError: myCar.description is not a function

**Explanation:**

1. **Static Properties and Methods:**
   * Defined using the static keyword.
   * Accessible directly on the class (Car.numberOfWheels, Car.description()).
   * Not accessible on instances of the class (myCar.numberOfWheels, myCar.description() will result in errors).
2. **Non-Static (Instance) Properties and Methods:**
   * Defined inside the constructor and methods of the class without the static keyword.
   * Accessible on instances of the class (myCar.make, myCar.getDetails()).
   * Not accessible directly on the class (Car.make, Car.getDetails() will result in errors).

**Key Points:**

* **Static members** are shared among all instances of the class and belong to the class itself.
* **Non-static members** are unique to each instance and belong to the instance of the class.

By using static and non-static properties and methods appropriately, you can better structure your classes and manage their behavior in JavaScript.

**Static vs Non-Static in Simple Terms**

* **Static**: Belongs to the class itself. Think of it like a shared trait or ability of the entire class.
* **Non-Static**: Belongs to individual objects (instances) created from the class. Each object has its own copy.

**Example**

Imagine a class Dog. We can have static properties and methods that are the same for all dogs, like the number of legs (which is generally 4 for all dogs). Non-static properties and methods are specific to each individual dog, like its name and color.

**Defining the Class**

class Dog {

// Static property

static numberOfLegs = 4;

// Constructor to initialize non-static properties

constructor(name, color) {

this.name = name; // Non-static property

this.color = color; // Non-static property

}

// Static method

static bark() {

return "All dogs can bark!";

}

// Non-static method

getDescription() {

return `This dog's name is ${this.name} and it is ${this.color}.`;

}

}

// Accessing static properties and methods

console.log(Dog.numberOfLegs); // Output: 4

console.log(Dog.bark()); // Output: All dogs can bark!

// Creating an instance of the Dog class

let myDog = new Dog('Buddy', 'brown');

// Accessing non-static properties and methods

console.log(myDog.name); // Output: Buddy

console.log(myDog.color); // Output: brown

console.log(myDog.getDescription()); // Output: This dog's name is Buddy and it is brown.

// Trying to access static properties and methods from an instance

console.log(myDog.numberOfLegs); // Output: undefined (because numberOfLegs is static)

console.log(myDog.bark()); // Output: TypeError: myDog.bark is not a function (because bark is static)

**Explanation**

1. **Static Property**:
   * Dog.numberOfLegs is a static property. It is the same for all dogs and can be accessed directly using the class name.
   * You cannot access it using an instance (myDog.numberOfLegs will be undefined).
2. **Static Method**:
   * Dog.bark() is a static method. It represents an ability all dogs have and can be accessed directly using the class name.
   * You cannot call it using an instance (myDog.bark() will throw an error).
3. **Non-Static Property**:
   * name and color are non-static properties. Each dog has its own name and color, so they are defined in the constructor.
   * You access them using an instance (myDog.name and myDog.color).
4. **Non-Static Method**:
   * getDescription() is a non-static method. It provides a description specific to each dog.
   * You call it using an instance (myDog.getDescription()).

By understanding these simple differences, you can use static and non-static properties and methods effectively in your JavaScript classes.

**OBJECT ––– VAR ––– REFVAR**

1. \*\*Object\*\*: In programming, an object is a collection of data (variables) and methods (functions) that operate on the data. It's an instance of a class, which is a blueprint for creating objects. Objects allow you to model real-world entities and their interactions in your code.

2. \*\*Var\*\*: "var" is often used as a keyword in programming languages like JavaScript to declare variables. When you declare a variable using "var," the type of the variable is inferred from the assigned value. This means you don't have to explicitly specify the data type of the variable.

3. \*\*Refvar\*\*: This seems like a combination of "reference" and "variable." In programming, a reference variable is a variable that holds a reference (memory address) to an object rather than the object itself. This allows you to manipulate the same object from different parts of your code without duplicating it. Reference variables are used in languages like Java, C++, and C#.

Each of these concepts plays a role in structuring and manipulating data in programming. If you have any specific questions about them, feel free to ask!

**Variables**

A variable in JavaScript is used to store data values. You can declare a variable using let, const, or var.

**Example:**

let myNumber = 10; // Number

let myString = "Hello"; // String

**Objects**

An object is a collection of properties. Each property is a key-value pair.

**Example:**

let person = {

name: "Alice",

age: 25

};

**Reference Variables**

When you assign an object to a variable, that variable doesn't hold the actual object. Instead, it holds a reference (or address) to the location in memory where the object is stored. This is why they are called reference variables.

**Example:**

let originalPerson = {

name: "Alice",

age: 25

};

let referencePerson = originalPerson; // referencePerson now holds the same reference as originalPerson

// Modifying referencePerson also changes originalPerson

referencePerson.age = 30;

console.log(originalPerson.age); // Output: 30

console.log(referencePerson.age); // Output: 30

**Key Points**

1. **Primitive Values**:
   * Numbers, strings, booleans, null, undefined, symbol, and bigint are primitive values.
   * Variables holding primitive values store the actual data.

**Example:**

let x = 5;

let y = x; // y is a copy of x

y = 10;

console.log(x); // Output: 5

console.log(y); // Output: 10

1. **Objects and Reference Variables**:
   * Objects (including arrays and functions) are reference types.
   * When you assign an object to a variable, the variable holds a reference to the object, not the object itself.

**Example:**

let originalArray = [1, 2, 3];

let referenceArray = originalArray; // referenceArray holds the same reference as originalArray

referenceArray.push(4); // Modifying referenceArray

console.log(originalArray); // Output: [1, 2, 3, 4]

console.log(referenceArray); // Output: [1, 2, 3, 4]

1. **Copying Objects**:
   * Copying objects by reference can lead to unexpected changes. Use shallow or deep copying to create independent copies.

**Shallow Copy Example:**

let originalPerson = {

name: "Alice",

age: 25

};

let shallowCopy = Object.assign({}, originalPerson);

shallowCopy.age = 30;

console.log(originalPerson.age); // Output: 25

console.log(shallowCopy.age); // Output: 30

**Deep Copy Example:**

let originalPerson = {

name: "Alice",

age: 25,

address: {

city: "Wonderland"

}

};

let deepCopy = JSON.parse(JSON.stringify(originalPerson));

deepCopy.address.city = "New York";

console.log(originalPerson.address.city); // Output: Wonderland

console.log(deepCopy.address.city); // Output: New York

**Summary**

* **Variables**: Store data values.
* **Objects**: Collections of key-value pairs.
* **Reference Variables**: Variables that store references to objects rather than the actual objects.

By understanding these concepts, you can manage and manipulate data in JavaScript more effectively.