OOPS Theoretical Understanding:

**Something about execution context**

In JavaScript, the execution context is a concept that refers to the environment in which a piece of code is executed. It consists of two main components: the Variable Object and the Scope Chain.

1. \*\*Variable Object (VO):\*\* The Variable Object is where the variables, function declarations, and function parameters are stored during the execution of a function. It acts as a container for these entities within a particular execution context.

2. \*\*Scope Chain:\*\* The Scope Chain is a list or chain of variable objects that are used to resolve variables during code execution. It is created based on the lexical scope of the code, which is determined by the physical placement of the code in the source.

There are **three types of execution contexts** in JavaScript:

- \*\*Global Execution Context:\*\* The default or outermost context. It represents the environment in which the JavaScript code is executed when it is not inside any function.

- \*\*Function Execution Context:\*\* Created whenever a function is invoked. Each function call results in a new execution context, and these contexts can be nested when functions are called within other functions.

- \*\*Eval Function Execution Context:\*\* Created when the eval function is executed. This context is able to modify its own variable object, which can have implications for variable scope.

Additionally, each execution context has a reference to its outer (enclosing) execution context, forming a chain known as the scope chain. This chain is crucial for variable resolution, as it allows the interpreter to look up variables in outer scopes if they are not found in the current scope.

Understanding execution contexts is essential for comprehending variable scope, hoisting, and the flow of code execution in JavaScript.

**Something about resolving variables:**

In the context of programming, "resolving variables" refers to the process of determining the value of a variable during the execution of a program. When a variable is encountered in the code, the programming language needs to find the corresponding value that the variable represents. This process involves searching for the variable within the current scope and, if necessary, in outer scopes through the scope chain.

The scope chain is a hierarchical structure that represents the nested scopes in a program. When a variable is referenced, the interpreter looks for the variable's value in the current scope. If the variable is not found in the current scope, it then looks in the outer (enclosing) scope, and this process continues until the global scope is reached. If the variable is not found in any scope, an error is usually raised, indicating that the variable is not defined.

Here's a simple example in JavaScript to illustrate variable resolution:

```javascript

var globalVar = "I am global";

function exampleFunction() {

var localVar = "I am local";

console.log(globalVar); // Resolves 'globalVar' in the global scope

console.log(localVar); // Resolves 'localVar' in the current function's scope

}

exampleFunction();

```

In this example, when `exampleFunction` is called, it has access to both `globalVar` (a global variable) and `localVar` (a variable declared within the function). The process of resolving variables involves determining the values associated with these variables based on their scope. The interpreter first looks for the variables in the function's scope and then, if necessary, in the global scope.

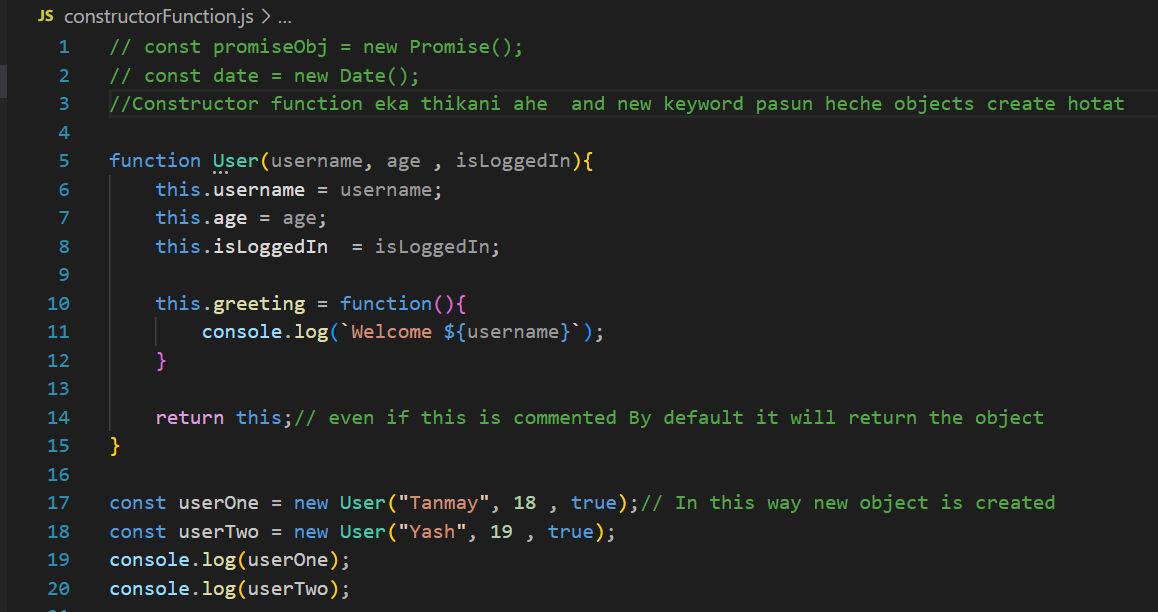
**The nature of Javascript:**

Javascript is a prototype based language…It has prototype based inheritance mechanisms

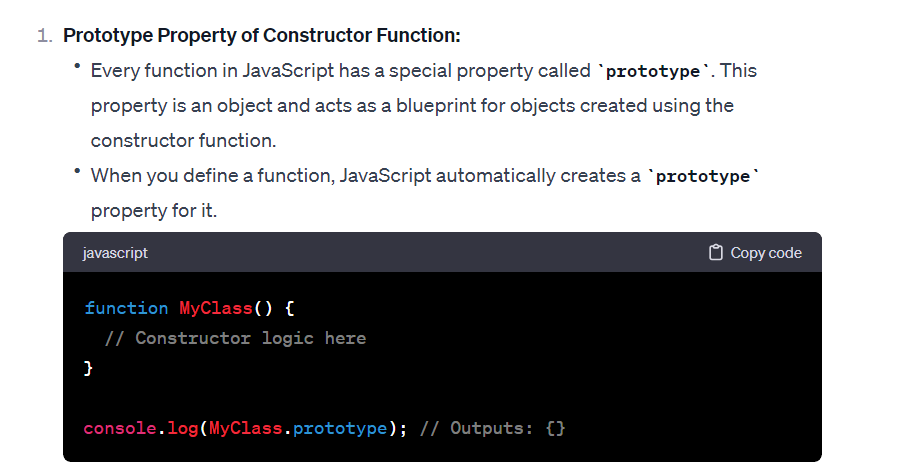
Suruvatila hecha object oriented feature je hota te veglya hena implement kela gela hota (using prototype idea)

Traditionally constructor functions cha use karun objects tayar kele jaat hote like for example when we create a date object or a promise object , There objects are created using constructor functions

Constructor functions: It is a way of creating an object



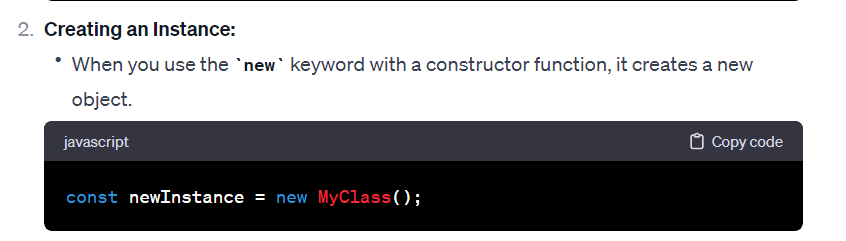
**The prototype property of Constructor functions:JS is responsible for creating the prototype object**

****

**The reason why this prototype wala object is created is to establish a prototype inheritance with the ancestor objects**

**The objects created with the prototype wale functions:**

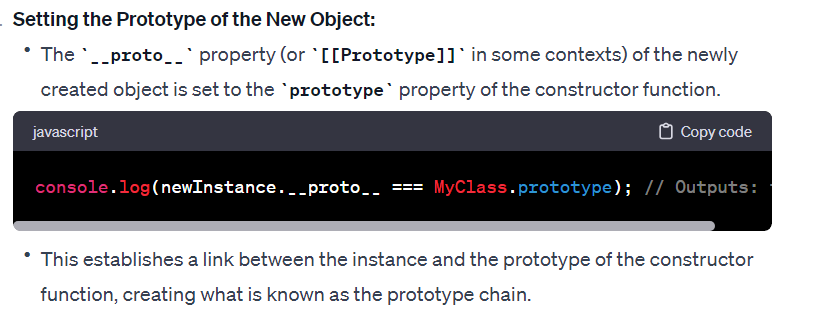
**Lets call them newInstance**

****

The indetail working of new keyword is mentioned later

**The relationship between newInstance and prototype object:**

The prototype object has the access to ancestor functionalities and properties … the newInstance should also have access to them … in case if it doesn’t have the functionalities and properties



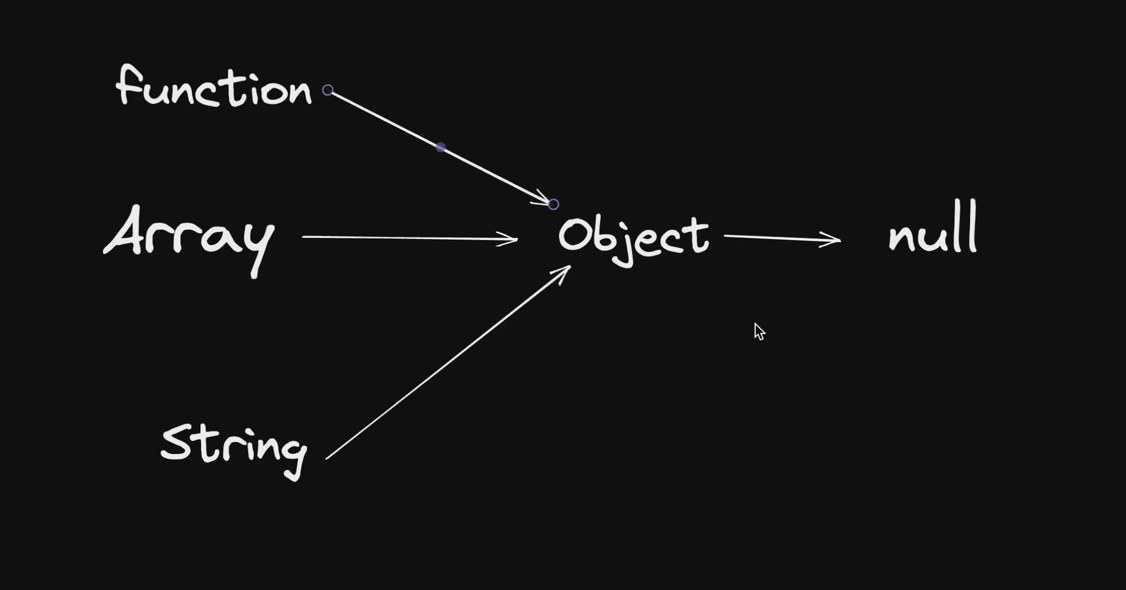
The output for above code is true

**Prototypal Behaviour**

**EVERYTHING IN JS IS AN OBJECT**

(Analogy )

:: JS kadhi haar manat nahi suppose ek object ahe tyachi functionality tyala mahit nahi ahe tr te techya parent madhe javoon functionality check karte ani jr parent madhe milali nahi tr grandparent madhe javoon functionality check karte…tevda steps maga jate joparyant tyala null milat nahi…(ithe nahi milala tr varti javoo , ithe pn nahi milala tr techya pn varti javoo)



Because of this prototypal behaviour new keyword , this keyword , prototype inheritance is defined

Function can behave as a function as well and function can behave as an object as well.

**What exactly is a prototype?**

In JavaScript, a prototype is an object from which other objects inherit properties and methods. Every object in JavaScript is associated with a prototype object, which acts as a template for that object. The prototype is essentially a mechanism for object inheritance.

Here are the key points about prototypes:

1. \*\*Prototype as a Blueprint:\*\*

- Think of a prototype as a blueprint or a template. Objects in JavaScript can be created based on this prototype.

2. \*\*Object Linkage:\*\*

- Each object in JavaScript is linked to a prototype object. This linkage is established when the object is created.

3. \*\*Property and Method Inheritance:\*\*

- Objects inherit properties and methods from their prototypes. If a property or method is not found in an object, JavaScript looks for it in the object's prototype and follows this chain until it finds what it's looking for.

4. \*\*`prototype` Property:\*\*

- Every function in JavaScript has a special property called `prototype`. When you create an object using a constructor function (using the `new` keyword), the object inherits properties and methods from the constructor function's `prototype`.

5. \*\*`Object.prototype`:\*\*

- At the top of the prototype chain is the `Object.prototype`. All objects in JavaScript, whether created through constructor functions or literals, ultimately inherit from `Object.prototype`.

Here's a simple example:

```javascript

// Constructor function

function Animal(name) {

this.name = name;

}

// Adding a method to the prototype

Animal.prototype.sayHello = function() {

console.log("Hello, I'm " + this.name);

};

// Creating an object using the constructor

var cat = new Animal("Whiskers");

// The 'cat' object inherits from Animal.prototype

cat.sayHello(); // Outputs: Hello, I'm Whiskers

```

In this example, `Animal.prototype` is the prototype of objects created with the `Animal` constructor. The `cat` object inherits the `sayHello` method from its prototype, allowing it to call `sayHello()` even though the method is not directly defined in the `cat` object.

**Prototype Chain in JavaScript:**

In JavaScript, this process is like the "prototype chain." Each object has a reference to another object, and if something is missing in one object, it can look up the chain to find it in another object.

For example, if an object doesn't have a particular method or property, it can check its prototype (connected object) to see if that prototype has what it needs. This chain can continue, allowing objects to share and inherit features from each other.

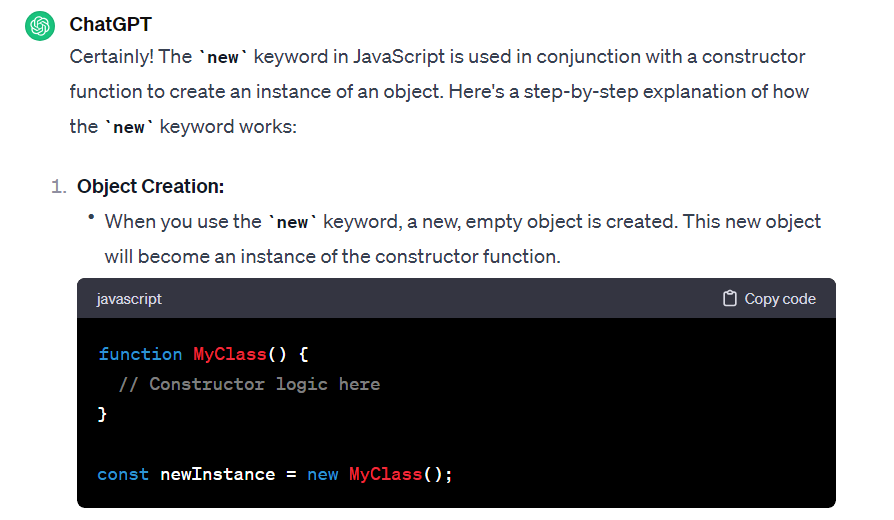
In essence, the prototype chain is like a reference system that objects use to borrow or share information, creating a network of interconnected objects in JavaScript.

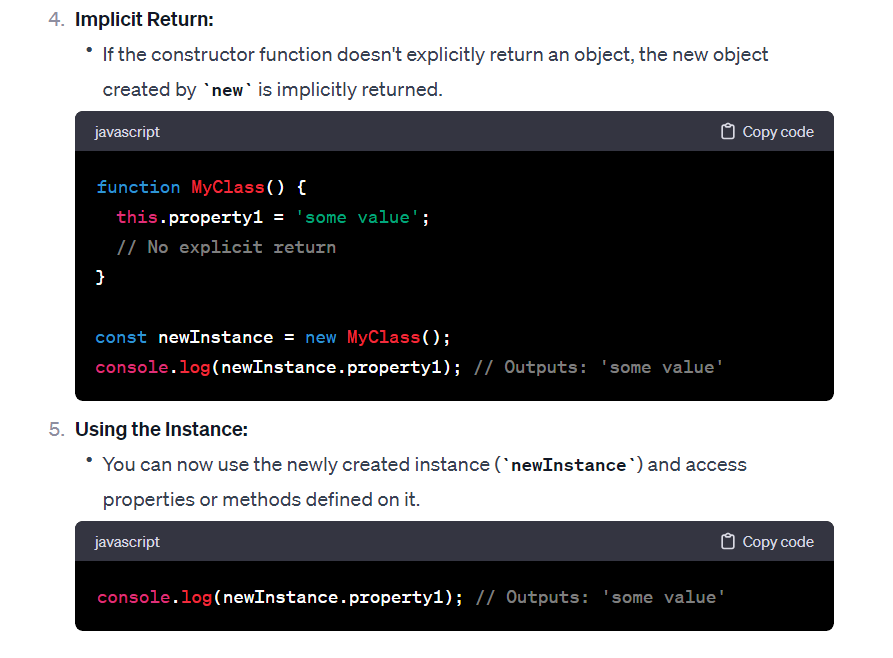
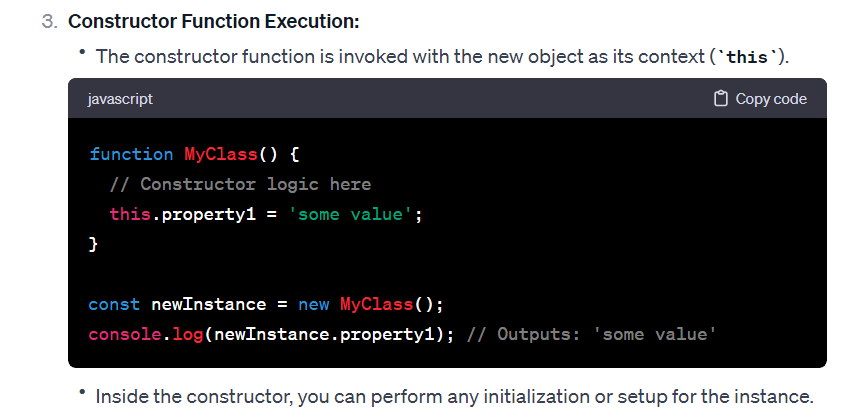
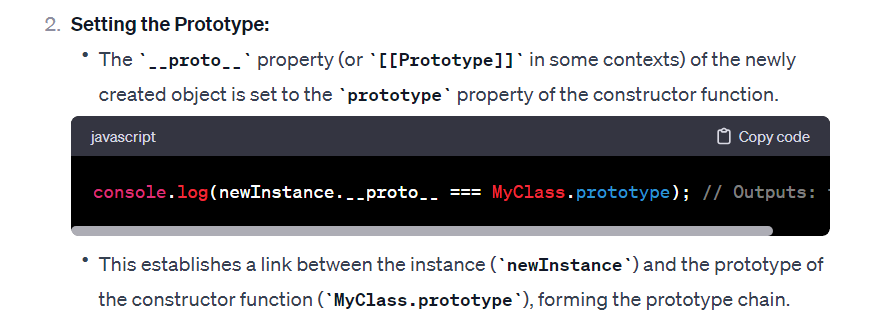
**The importance of new Keyword and how actually it works?**

**The new keyword triggers a process of linking the newInstance and its prototype objects**

**4 steps :**

1. **Create an empty object**
2. **Link it with its prototype using newInstance.\_\_proto\_\_ = constructorFun.prototype**
3. **Run the constructor functions to initialize the properties written by the user**
4. **Return the object**





/\*

Here's what happens behind the scenes when the new keyword is used:

A new object is created: The new keyword initiates the creation of a new JavaScript object.

A prototype is linked: The newly created object gets linked to the prototype property of the constructor

function. This means that it has access to properties and methods defined on the constructor's prototype.

The constructor is called: The constructor function is called with the specified arguments and this is

 bound to the newly created object. If no explicit return value is specified from the constructor,

 JavaScript assumes this, the newly created object, to be the intended return value.

The new object is returned: After the constructor function has been called, if it doesn't return

 a non-primitive value (object, array, function, etc.), the newly created object is returned.

**if i write something like Array.prototype what exactly does it mean?**

**Here the array created by you(newInstance) is different and the arrays prototype object created by JS is different… there is implicit linking between the two so our newInstance can access the properties of the prototype like length , map, filter , reduce**

-> When you write `Array.prototype` in JavaScript, you are referring to the prototype object associated with the `Array` constructor function. Here's what this means:

1. \*\*`Array` is a Constructor Function:\*\*

- In JavaScript, `Array` is a built-in constructor function used to create arrays.

2. \*\*`prototype` Property:\*\*

- Every function in JavaScript has a special property called `prototype`. This property is an object, and it's used as the prototype for objects created with that constructor function.

3. \*\*`Array.prototype`:\*\*

- Therefore, `Array.prototype` refers to the prototype object associated with the `Array` constructor. It's a template or blueprint that defines properties and methods that are shared among all arrays created using the `Array` constructor.

4. \*\*Shared Functionality:\*\*

- Any method or property added to `Array.prototype` becomes available to all arrays in JavaScript. This is an example of prototype-based inheritance, where objects inherit properties and methods from their constructor's prototype.

Here's a simple illustration:

```javascript

// Adding a custom method to Array.prototype

Array.prototype.customMethod = function() {

console.log("This is a custom method for arrays!");

};

// Creating an array

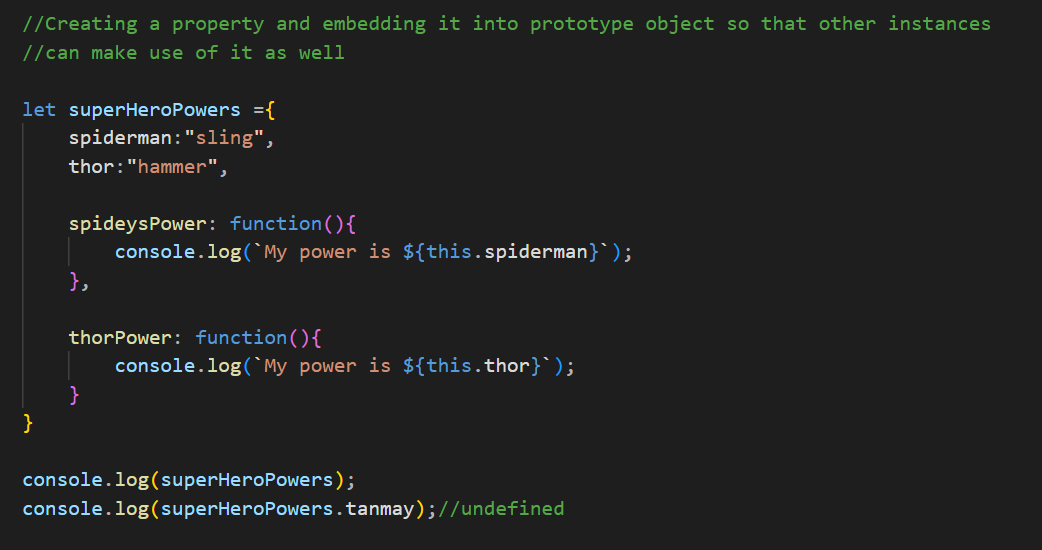
var myArray = [1, 2, 3];

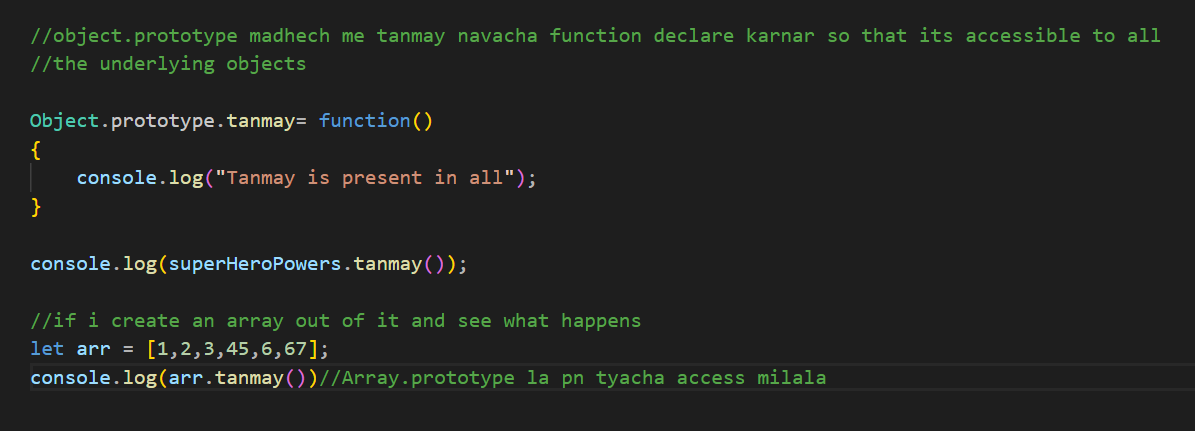
// Now, myArray inherits the customMethod from Array.prototype

myArray.customMethod(); // Outputs: This is a custom method for arrays!

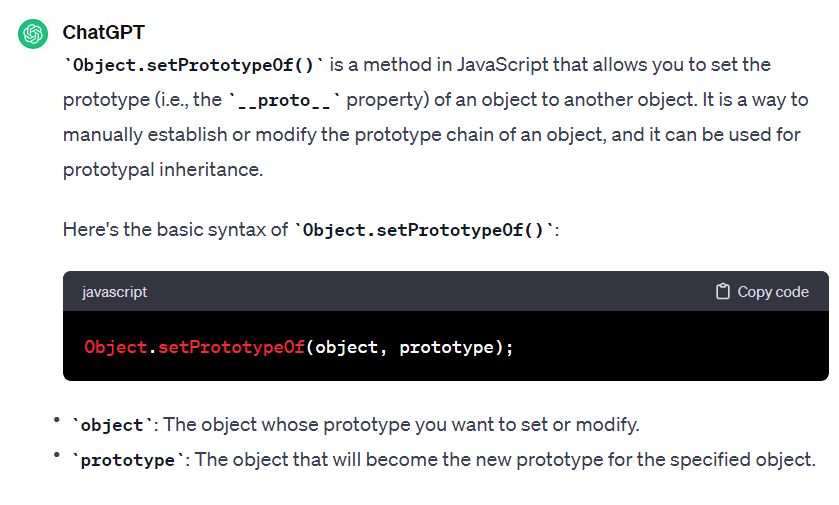
```

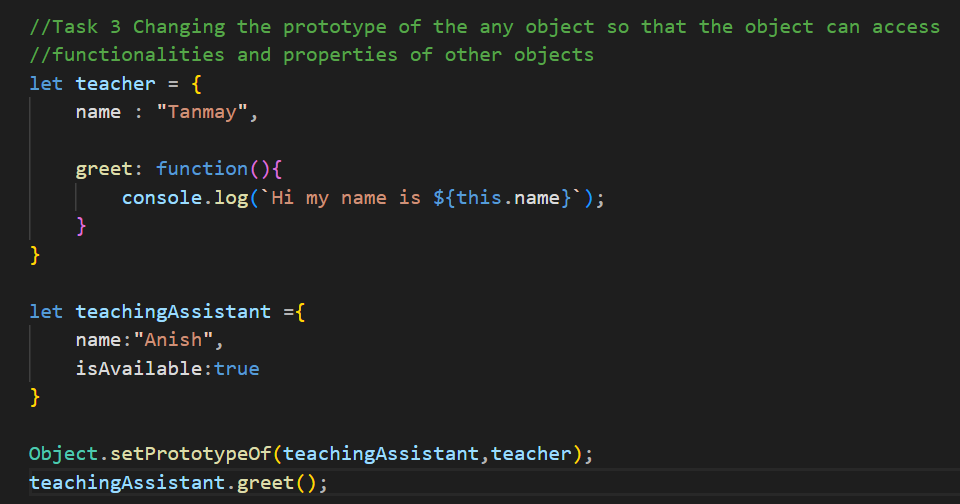
In this example, `customMethod` is added to `Array.prototype`, and because of the prototype chain, all arrays in JavaScript (including `myArray`) can now access and use this custom method.

**Task 1 : Creating a property and embedding it into prototype object so that other instances can make use of it as well** ****

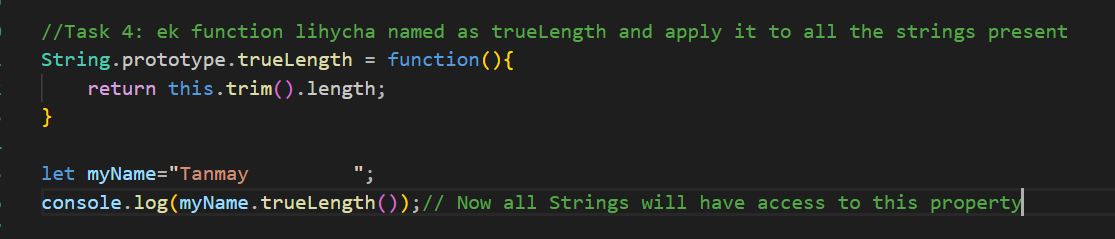
****

**Task 2: Changing the prototype of the any object so that the object can access functionalities and properties of other objects**

****

****

**Task 4:**

****