2024-03-22

For Loops:

```
for (let i = 0; i < 5; i++)
         console.log(‘hello’)
```

Structure of the loop :

For (initial expression ; condition ; increment expression)

Factory functions:

a factory function is just a function that creates an object and returns it

that's it now why is this useful and why don't i just create an object myself well you can but if you have complex logic and you have to create multiple objects over and over again that have that same logic you can write that logic one time inside a function and use that function as a

Logic :

```
function factory(){

         return{……}

}
```

```
function createPerson(firstName, lastName) {
  return {
    firstName: firstName,
    lastName: lastName,
    getFullName: function() {
      return this.firstName + " " + this.lastName;
    }
  };
}

const person1 = createPerson("John", "Doe");
const person2 = createPerson("Jane", "Smith");
```

```
console.log(person1.getFullName()); // Outputs: John Doe
console.log(person2.getFullName()); // Outputs: Jane Smith
```

1. javascriptCopy code

   `function createPerson(firstName, lastName) {`

   This line defines a function named `createPerson` that takes two parameters: `firstName` and `lastName`.
2. javascriptCopy code

   `return {`

   The `return` statement begins an object literal, which will be returned by the `createPerson` function.
3. javascriptCopy code

   `firstName: firstName,`

   Here, a property named `firstName` is assigned the value of the `firstName` parameter passed to the `createPerson` function.
4. javascriptCopy code

   `lastName: lastName,`

   Similarly, a property named `lastName` is assigned the value of the `lastName` parameter passed to the `createPerson` function.
5. javascriptCopy code

   `getFullName: function() {`

   This line defines a method named `getFullName` within the object. It is a function that returns the concatenation of `firstName` and `lastName`.
6. javascriptCopy code

   `return this.firstName + " " + this.lastName;`

   Inside the `getFullName` method, `this.firstName` refers to the `firstName` property of the current object (the object being created by `createPerson`), and `this.lastName` refers to the `lastName` property of the same object.
7. javascriptCopy code

   `};`

   Closes the object literal.

8. javascriptCopy code

   ```
   const person1 = createPerson("John", "Doe");
   ```

   Invokes the `createPerson` function with the arguments `"John"` and `"Doe"`, creating a new object with `firstName` set to `"John"` and `lastName` set to `"Doe"`. This object is assigned to the variable `person1`.
9. javascriptCopy code

   ```
   const person2 = createPerson("Jane", "Smith");
   ```

   Similarly, invokes the `createPerson` function with the arguments `"Jane"` and `"Smith"`, creating a new object with `firstName` set to `"Jane"` and `lastName` set to `"Smith"`. This object is assigned to the variable `person2`.
10. javascriptCopy code

    ```
    console.log(person1.getFullName()); // Outputs: John Doe
    ```

    Calls the `getFullName` method on the `person1` object and logs the result to the console. In this case, it logs `"John Doe"`.
11. javascriptCopy code

    ```
    console.log(person2.getFullName()); // Outputs: Jane Smith
    ```

    Similarly, calls the `getFullName` method on the `person2` object and logs the result to the console. In this case, it logs `"Jane Smith"`.

In summary, the code defines a function `createPerson` that creates and returns a person object with `firstName` and `lastName` properties, along with a method `getFullName` that returns the full name by concatenating `firstName` and `lastName`. Then, it creates two person objects using this function and prints their full names to the console.

```
function createcircle(radius){
        return{
                radius: radius,
                draw(){  // this is a method that can be called
                console.log('draw');
                }

        }
}

const circle = createcircle(1);
console.log(circle);
```

```
//console ma "circle.draw()" lekhyou bhane 'draw' bhanera print huncha
```

1. `function createCircle(radius) {` : This line defines a function named `createCircle` that takes one parameter `radius` .
2. `return {` : This line starts an object literal, which is what the `createCircle` function will return.
3. `radius: radius,` : This line defines a property named `radius` within the object literal and assigns it the value of the `radius` parameter passed to the `createCircle` function.
4. `draw() {` : This line defines a method named `draw` within the object literal.
5. `console.log('draw');` : This line is the body of the `draw` method. It logs the string `'draw'` to the console.
6. `}` : This line ends the definition of the `draw` method.
7. `}` : This line ends the definition of the object literal.
8. `const circle = createCircle(1);` : This line calls the `createCircle` function with an argument of `1` , which creates a circle object with a radius of `1` and assigns it to the variable `circle` .
9. `console.log(circle);` : This line logs the `circle` object to the console, which will show the properties and methods defined within it.

Constructor functions :

The major difference between constructor and factory functions is that we use this. and new keywords in the constructor functions .

Constructor Functions: Blueprints for Objects

In JavaScript, constructor functions are special functions that serve as blueprints for creating objects. They define the properties and behaviors (methods) that will be shared by all objects created from them.

Key Characteristics:

- **Function with a Capitalized Name:** By convention, constructor function names are capitalized to distinguish them from regular functions. (e.g., `function Person() { ... }`)
- `new` **Keyword:** When you call a constructor function, you use the `new` keyword to create a new object instance based on the blueprint.

Steps Involved:

1. **Object Creation:** The `new` keyword allocates memory for a new object.
2. `this` **Binding:** Inside the constructor function, the `this` keyword refers to the newly created object. You can use `this` to assign properties and methods to the object.

3. **Implicit Return:** Constructor functions don't typically have an explicit `return` statement. By default, they return the newly created object (`this`).

Example:

JavaScript

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;

  // Method to display car information
  this.displayInfo = function() { // we are creating a method
    console.log(`${this.year} ${this.make} ${this.model}`); //`this` refers to
the new object being created.
  };
}

// Creating new Car objects
const car1 = new Car("Toyota", "Camry", 2023);
const car2 = new Car("Honda", "Civic", 2022);

car1.displayInfo(); // Output: 2023 Toyota Camry
car2.displayInfo(); // Output: 2022 Honda Civic
```

Explanation:

- The `Car` constructor function defines properties (`make`, `model`, `year`) and a method (`displayInfo`).
- When `new Car` is called with arguments (`make`, `model`, `year`), a new object is created.
- `this` inside the constructor refers to that object, and the properties and method are assigned accordingly.
- The `displayInfo` method can be called on individual `Car` objects to display their information.

Advantages of Constructor Functions:

- **Reusability:** You can create multiple objects with the same properties and behaviors from a single constructor function.
- **Code Organization:** Constructor functions help keep your code organized by encapsulating object creation logic.
- **Data Integrity:** By defining properties within the constructor, you can ensure that objects are initialized with consistent data.

In Summary:

Constructor functions are a fundamental concept in JavaScript object-oriented programming. They provide a powerful way to create objects with specific properties and methods, promoting code reusability and maintainability.

Difference between constructor and factory function

The key difference between constructor functions and factory functions in JavaScript lies in how they create objects and utilize the `new` keyword:

Constructor Functions:

- `new` **Keyword:** Invoked with the `new` keyword, which performs several actions:
  - Creates a new empty object.
  - Sets the `this` keyword within the function to refer to that new object.
  - Implicitly returns the newly created object (unless an explicit `return` is used).
- **Prototype Chaining:** Objects created with constructors inherit properties and methods from the constructor's prototype.

Factory Functions:

- **No** `new` **Keyword:** Called like regular functions, without the `new` keyword.
- **Explicit Return:** They explicitly return the created object using a `return` statement.
- **No Prototype Chaining:** Objects created by factory functions don't inherit properties from the function itself (unless you manually set up inheritance).

Here's a table summarizing the key differences:

Feature	Constructor Function	Factory Function
`new` Keyword	Required	Not Required
`this` Binding	Refers to new object	Not bound
Return	Implicit `this`	Explicit `return`
Prototype	Inherits from prototype	No inheritance

Example:

Constructor Function:

JavaScript

```
function Person(name, age) {
  this.name = name;
```

```
  this.age = age;

  // Method using this
  this.greet = function() {
    console.log(`Hi, I'm ${this.name} and I'm ${this.age} years old.`);
  };
}

const person1 = new Person("Alice", 30);
person1.greet(); // Output: Hi, I'm Alice and I'm 30 years old.
```

Factory Function:

JavaScript

```
function createPerson(name, age) {
  return {
    name,
    age,
    greet() {
      console.log(`Hi, I'm ${name} and I'm ${age} years old.`);
    }
  };
}

const person2 = createPerson("Bob", 25);
person2.greet(); // Output: Hi, I'm Bob and I'm 25 years old.
```

Choosing Between Them:

- Use constructor functions when you want to:
  - Leverage prototype chaining for inheritance.
  - Enforce a specific structure for objects (using `this`).
- Use factory functions when you:
  - Prefer a simpler syntax without the `new` keyword.
  - Want more flexibility in object creation (e.g., returning different object types).
  - Need to create objects without inheritance.

Getter and setter

we can code like below to print the name of a person :

```
const person ={
        firstname :'david',
        lastname : 'mandal'
};

console.log(person.firstname + ' ' + person.lastname);
```

or like this :

```
const person ={
        firstname :'david',
        lastname : 'mandal'
};

console.log('${person.firstname} ${person.lastname}');
```

with the above code , we would have to repeat the template literal in multiple places to print the fullname of the person

a better approach would be to call a method in the 'person' object that displays the full name of the person

```
const person ={
        firstname :'david',
        lastname : 'mandal'
        fullname(){
                return '${person.firstname} ${person.lastname}'
        }
};

console.log(person.fullname());
```

in the above code , we cant set the persons fullname from the outside . it would also be nice if we could treat *.fullname()* as a property rather than a method .
this is where the getters and the setters come in place.

Getters = access properties
Setters = change (mutate) the properties

```
const person ={
        firstname :'david',
        lastname : 'mandal'
        get fullname(){ //now this is a getter
                return '${person.firstname} ${person.lastname}'
        }
};

console.log(person.fullname);
```

```
const person ={
        firstname :'david',
        lastname : 'mandal'
        get fullname(){ //now this is a getter
                return '${person.firstname} ${person.lastname}'
        }
         set fullname(value){  // name of the property or method
                 const parts = value.split('');
                 this.firstname = parts[0];
                 this.lastname = parts[1];
         }
};
person.fullname = 'John Smith';
console.log(person);
```

explanation of the code :
defines a JavaScript object named `person` that uses getters and setters to manage its `firstname` and `lastname` properties, and provides a derived property `fullname`. Let's break it down line by line:

1. **`const person = { ... }`**
   - This line declares a constant variable named `person` and assigns an object literal to it.
2. **`firstname: 'david',` `lastname: 'mandal'`**
   - These lines define properties within the `person` object.
     - `firstname` is set to the string value `"david"`.
     - `lastname` is set to the string value `"mandal"`.
3. **`get fullname(){ ... }`**

- This defines a getter for the `fullname` property.
  - The `get` keyword indicates it's a getter.
  - The property name is `fullname`.
  - The value is an anonymous function (without a name). This function will be executed whenever you try to access the `fullname` property of the `person` object (like reading it).

4. `return '${person.firstname} ${person.lastname}'`

- This line is inside the getter function.
  - It uses template literals (enclosed in backticks) to create a string.
  - Inside the template literal, it accesses the `person.firstname` and `person.lastname` properties (assuming they exist in the `person` object) and concatenates them with spaces in between.
  - The `return` statement returns this formatted string as the value of the `fullname` property.

5. `set fullname(value){ ... }`

- This defines a setter for the `fullname` property.
  - The `set` keyword indicates it's a setter.
  - The property name is again `fullname`.
  - The value is an anonymous function that takes one argument, `value`. This function will be executed whenever you try to assign a new value to the `fullname` property of the `person` object.

6. `const parts = value.split('');`

- This line is inside the setter function.
  - It splits the `value` argument (the new value being assigned to `fullname`) into an array of strings using the `split('')` method. Each element in the array will be a single character from the original `value` string.

7. `this.firstname = parts[0]; this.lastname = parts[1];`

- These lines assume that the `value` assigned to `fullname` should have two parts (first and last name separated by a space).
  - They access the `firstname` and `lastname` properties of the `person` object using `this` (referring to the current object).
  - They assign the first element (`parts[0]`) of the `parts` array (assuming it's the first name) to the `firstname` property and the second element (`parts[1]`) (assuming it's the last name) to the `lastname` property.

8. `person.fullname = 'John Smith';`

- This line assigns the string value `"John Smith"` to the `fullname` property of the `person` object.
    - This will trigger the setter function defined earlier.

9. `console.log(person);`

- This line prints the entire `person` object to the console. However, since the getter and setter manipulate the `firstname` and `lastname` properties internally, directly accessing `person.fullname` won't reflect the changes made by the setter.

Expected Output (due to getter hiding internal changes):

```
{ firstname: 'david', lastname: 'mandal' }
```

Explanation:

The code defines a `person` object with a getter and setter for `fullname`. The getter creates a formatted string using the existing `firstname` and `lastname`, but the setter updates those underlying properties when you assign a new value to `fullname`. However, directly logging `person` shows the original values because the getter only returns a formatted string, not modifying the object itself.

In essence:

- The getter provides a way to access a formatted `fullname` without revealing the internal structure (separate `firstname` and `lastname`).
- The setter allows updating the `firstname` and `lastname` based on a provided `fullname` string, enforcing a two-part name format.

2024-03-02