Object.getOwnPropertyDescriptor(obj, prop)

Certainly! Let's dive into the `Object.getOwnPropertyDescriptor()` method in JavaScript. I'll explain everything step by step.

Definition:

The `Object.getOwnPropertyDescriptor()` method is used to retrieve the descriptor (metadata) of a specific property on an object. It returns an object that describes the property, including whether it is writable, enumerable, and configurable.

Syntax:

Object.getOwnPropertyDescriptor(obj, prop)

Arguments:

- **`obj`** (Required): The object that contains the property.
- **`prop`** (Required): The name of the property whose descriptor you want to retrieve.

Return Type:

It returns an **object** that represents the descriptor of the property. If the property is not found, it returns `undefined`.

The returned descriptor object contains the following properties:

- **`value`**: The value of the property.
- **`writable`**: A Boolean indicating whether the property's value can be changed.
- **`enumerable`**: A Boolean indicating whether the property will be enumerated in a `for...in` loop or `Object.keys()` method.
- **`configurable`**: A Boolean indicating whether the property can be deleted or modified.

Example 1: Basic Usage

In this example, let's fetch the descriptor for a simple object property.

```
// Creating an object with a property
const user = {
    name: "John",
    age: 30
};

// Fetching the descriptor for the 'name' property
const descriptor = Object.getOwnPropertyDescriptor(user, 'name');

console.log(descriptor);
```

Output:

```
{
  value: 'John',
  writable: true,
  enumerable: true,
  configurable: true
}
```

Explanation:

- `value`: The value of the `name` property is `'John'`.
- `writable`: This is `true` because the value of the property can be modified.
- `enumerable`: This is `true` because `name` appears in `for...in` loops or `Object.keys()`.
- `configurable`: This is `true` because the property can be deleted or its descriptor can be modified.

Example 2: Property with Non-Writable Value

In this example, let's define a property that is not writable.

```
const user = {};

Object.defineProperty(user, 'name', {
```

```
  value: 'Alice',
  writable: false,   // The value of 'name' cannot be changed
  enumerable: true,
  configurable: true
});

const descriptor = Object.getOwnPropertyDescriptor(user, 'name');
console.log(descriptor);
```

Output:

```
{
  value: 'Alice',
  writable: false,
  enumerable: true,
  configurable: true
}
```

Explanation:

- The `writable` flag is `false`, meaning you cannot change the value of the `name` property.
- The `configurable` flag is `true`, which means you can still delete or modify the property.

Example 3: Property with Non-Enumerable

Here, we will create a property that is non-enumerable, meaning it won't appear in loops like `for...in`.

```
const user = {};

Object.defineProperty(user, 'password', {
  value: '1234',
  writable: true,
  enumerable: false,  // 'password' won't show up in loops or keys
  configurable: true
});

const descriptor = Object.getOwnPropertyDescriptor(user, 'password');
```

```
console.log(descriptor);
```

Output:

```
{
  value: '1234',
  writable: true,
  enumerable: false,
  configurable: true
}
```

Explanation:

- `enumerable` is `false`, so `password` won't show up in a loop like `for...in` or in methods like `Object.keys()`.

Example 4: Property with Non-Configurable

In this example, the `configurable` flag is set to `false`, which means the property cannot be deleted or modified in terms of its descriptor.

```
const user = {};

Object.defineProperty(user, 'role', {
  value: 'admin',
  writable: true,
  enumerable: true,
  configurable: false  // Can't modify or delete this property
});

const descriptor = Object.getOwnPropertyDescriptor(user, 'role');
console.log(descriptor);
```

Output:

```
{
  value: 'admin',
  writable: true,
  enumerable: true,
  configurable: false
}
```

Explanation:

- The `configurable` flag is `false`, meaning you cannot delete this property or modify its descriptor once it's set.

Example 5: Creative Argument - Using Symbols as Property

In this example, we use a `Symbol` as a property key, which can be useful when working with private properties.

```
const uniqueSymbol = Symbol('id');
const user = {};

Object.defineProperty(user, uniqueSymbol, {
  value: 1001,
  writable: true,
  enumerable: true,
  configurable: true
});

const descriptor = Object.getOwnPropertyDescriptor(user, uniqueSymbol);
console.log(descriptor);
```

Output:

```
{
  value: 1001,
  writable: true,
  enumerable: true,
  configurable: true
}
```

Explanation:

- Using `Symbol` as a property key ensures the property has a unique name and doesn't conflict with other property names.

Conclusion:

The `Object.getOwnPropertyDescriptor()` method is a powerful tool for examining the details of a property on an object. It helps you understand whether a property is writable, enumerable, and configurable, which can be useful when dealing with object manipulations or debugging.

Let me know if you need further clarification or additional examples!

Object.getOwnPropertyNames(obj)

Certainly! Here's a detailed breakdown of the `Object.getOwnPropertyNames()` method in JavaScript with examples and explanations.

Definition:

`Object.getOwnPropertyNames(obj)` is a method that returns an array of all the properties (including non-enumerable ones) of an object `obj`. These properties are the object's own properties (not inherited properties). It does not return properties from the prototype chain, and it does not include symbol properties.

Syntax:

```
Object.getOwnPropertyNames(obj);
```

Arguments:

- **obj**: The object whose own properties are to be returned. This is the only argument the method accepts.

Return Type:

- **Array**: An array of strings, each representing a property name that is directly on the `obj` object (excluding those inherited from the prototype chain).

Examples and Explanation:

Example 1: Basic Usage

```
const person = {
  name: "John",
  age: 30,
  greet: function() {
```

```
    console.log("Hello!");
  }
};

// Get all own property names of the object
const properties = Object.getOwnPropertyNames(person);

console.log(properties);
// Output: [ 'name', 'age', 'greet' ]
```

- **Explanation**: This example demonstrates the basic usage of `Object.getOwnPropertyNames()` to get the list of all properties (`name`, `age`, `greet`) of the `person` object.

Example 2: Including Non-enumerable Properties

```
const car = {
  make: "Toyota",
  model: "Corolla"
};

Object.defineProperty(car, "year", {
  value: 2021,
  enumerable: false
});

// Get all own properties including non-enumerable ones
const carProperties = Object.getOwnPropertyNames(car);

console.log(carProperties);
// Output: [ 'make', 'model', 'year' ]
```

- **Explanation**: Even though `year` is not enumerable, it still appears in the array returned by `Object.getOwnPropertyNames()`.

Example 3: Inherited Properties Are Excluded

```
const animal = {
  species: "Dog"
};

const pet = Object.create(animal);
```

```
pet.name = "Buddy";

// Get own properties of the 'pet' object
const petProperties = Object.getOwnPropertyNames(pet);

console.log(petProperties);
// Output: [ 'name' ]
```

- **Explanation**: The `pet` object has only its own property `name`. The inherited `species` property from the `animal` object is excluded.

Example 4: Symbols Are Not Included

```
const obj = {
  prop: 1
};

const sym = Symbol('foo');
obj[sym] = "bar";

// Get own properties (symbol properties are not included)
const properties = Object.getOwnPropertyNames(obj);

console.log(properties);
// Output: [ 'prop' ]
```

- **Explanation**: Symbol properties (`sym` in this case) are not included in the result.

Example 5: Using with Arrays

```
const arr = [1, 2, 3];

const arrayProperties = Object.getOwnPropertyNames(arr);

console.log(arrayProperties);
// Output: [ '0', '1', '2', 'length' ]
```

- **Explanation**: Arrays are also objects, so `Object.getOwnPropertyNames()` will return all the properties including indices (`'0'`, `'1'`, etc.) and the `length` property.

Example 6: Object with Getter and Setter

```
const person = {
  firstName: "Jane",
  lastName: "Doe",

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },

  set fullName(name) {
    [this.firstName, this.lastName] = name.split(' ');
  }
};

const propertyNames = Object.getOwnPropertyNames(person);

console.log(propertyNames);
// Output: [ 'firstName', 'lastName', 'fullName' ]
```

- **Explanation**: The `fullName` getter and setter are also considered properties of the `person` object. They are included in the result.

Creative Way of Providing Different Arguments:

1. **Empty Object:**

```
const emptyObj = {};
console.log(Object.getOwnPropertyNames(emptyObj));
// Output: []
```

- **Explanation**: For an empty object, the result is an empty array since there are no properties.

2. **Object with a Prototype Chain:**

```
const baseObj = {
  commonProp: "I exist in all objects"
};
```

```
const derivedObj = Object.create(baseObj);
derivedObj.uniqueProp = "I am unique to this object";

console.log(Object.getOwnPropertyNames(derivedObj));
// Output: [ 'uniqueProp' ]
```

- **Explanation**: Only the properties of the `derivedObj` are returned. The `commonProp` in `baseObj` is excluded because it is not an own property.

3. **Object with Symbols and Properties:**

```
const obj = {
  regularProp: "regular value"
};
const symbol = Symbol('id');
obj[symbol] = "symbol value";

console.log(Object.getOwnPropertyNames(obj));
// Output: [ 'regularProp' ]
```

- **Explanation**: The symbol property `id` is not included in the result since `Object.getOwnPropertyNames()` excludes symbol properties.

Conclusion:

`Object.getOwnPropertyNames()` is a useful method for retrieving all the property names of an object, including non-enumerable properties. It is important to note that it does not return symbol properties, inherited properties, or prototype chain properties.

Object.getPrototypeOf(obj)

Sure! Let's dive into the `Object.getPrototypeOf(obj)` method in JavaScript with a detailed explanation, code examples, and comments.

Definition:

The `Object.getPrototypeOf(obj)` method returns the prototype (i.e., the internal `[[Prototype]]`) of the specified object `obj`. The prototype is essentially the object that is inherited from and provides the properties and methods of the object.

In simpler terms, this method tells you the "parent" of the given object.

Syntax:

Object.getPrototypeOf(obj);

- **obj**: The object whose prototype is to be returned.

Arguments:

- `obj`: This is the object whose prototype you want to retrieve. The argument is mandatory and should be an object.

Return Type:

- The return type is the prototype (i.e., an object) of the provided object, or `null` if the provided object has no prototype (e.g., `Object.create(null)`).

Example 1: Basic Usage

```
// Creating a simple object
const person = {
  name: 'Alice',
  age: 25
};

// Using Object.getPrototypeOf to get the prototype of 'person'
const prototype = Object.getPrototypeOf(person);

// Logging the prototype of 'person'
console.log(prototype); // Output: {} (the default object prototype)
```

Explanation:
In this example, we created an object `person`. When we use `Object.getPrototypeOf(person)`, it returns the prototype of `person`, which is the default `Object.prototype`.

Example 2: Prototype Chain

```
// Defining a base object
const animal = {
  eats: true
};

// Creating a new object that inherits from 'animal'
const dog = Object.create(animal);
dog.barks = true;

// Using Object.getPrototypeOf to get the prototype of 'dog'
console.log(Object.getPrototypeOf(dog)); // Output: { eats: true }
```

Explanation:
Here, `dog` inherits from `animal`. When we call `Object.getPrototypeOf(dog)`, it returns the prototype object `animal`, which has the property `eats`.

Example 3: Inheriting from a Custom Constructor

```
// Custom constructor function
function Car(make, model) {
  this.make = make;
  this.model = model;
}

// Creating a new 'Car' object
const myCar = new Car('Toyota', 'Corolla');

// Using Object.getPrototypeOf to get the prototype of 'myCar'
console.log(Object.getPrototypeOf(myCar)); // Output: Car {}

// Checking if myCar is an instance of Car
console.log(myCar instanceof Car); // Output: true
```

Explanation:
The `Car` constructor is used to create an object `myCar`. Using `Object.getPrototypeOf(myCar)` gives us the prototype of `myCar`, which is the `Car.prototype` object. This allows us to check if `myCar` is an instance of `Car`.

Example 4: Using `null` as a Prototype

```
// Creating an object with no prototype
```

```
const objWithNoProto = Object.create(null);

// Using Object.getPrototypeOf to get the prototype of 'objWithNoProto'
console.log(Object.getPrototypeOf(objWithNoProto)); // Output: null
```

Explanation:
In this example, we create an object with `null` as its prototype using `Object.create(null)`. Calling `Object.getPrototypeOf(objWithNoProto)` returns `null`, indicating that it has no prototype.

Example 5: Using `Object.getPrototypeOf` with Arrays

```
// Creating an array
const numbers = [1, 2, 3, 4, 5];

// Using Object.getPrototypeOf to get the prototype of 'numbers'
console.log(Object.getPrototypeOf(numbers)); // Output: Array.prototype
```

Explanation:
Arrays in JavaScript inherit from `Array.prototype`, so when we use `Object.getPrototypeOf` on an array, it returns `Array.prototype`.

Example 6: Prototype of a Function

```
// Defining a function
function greet() {
  console.log("Hello!");
}

// Using Object.getPrototypeOf to get the prototype of the function
console.log(Object.getPrototypeOf(greet)); // Output: [Function: Function]
```

Explanation:
In JavaScript, functions are also objects and have a prototype. Here, the `greet` function's prototype is `Function.prototype`.

Creative Example: Modifying Prototype

```
// Creating a simple object
const car = {
  wheels: 4,
};
```

```
// Creating another object that inherits from 'car'
const sportsCar = Object.create(car);
sportsCar.speed = 'fast';

// Adding a method to the prototype of 'car'
car.drive = function() {
  console.log("The car is driving!");
};

// Checking the prototype and using the method
Object.getPrototypeOf(sportsCar).drive(); // Output: The car is driving!
```

Explanation:
Here, we modify the prototype of `car` by adding a method `drive`. This method becomes available to objects inheriting from `car` (like `sportsCar`). So when we use `Object.getPrototypeOf(sportsCar).drive()`, it calls the `drive` method from the `car` prototype.

Summary:

- `Object.getPrototypeOf(obj)` returns the prototype of the specified object.
- It is useful for checking inheritance and prototype chains.
- The return value is either the prototype object or `null` if the object has no prototype.

Object.defineProperty(obj, prop, descriptor)

`Object.defineProperty()` in JavaScript

Definition: The `Object.defineProperty()` method is used to define a new property or modify an existing property on an object. It allows you to precisely configure the characteristics of the property such as whether it is writable, enumerable, and configurable.

Syntax:

```
Object.defineProperty(obj, prop, descriptor);
```

- `obj` (required): The object on which the property will be defined or modified.
- `prop` (required): The name of the property to be defined or modified.
- `descriptor` (required): The descriptor object that defines the behavior of the property.

Arguments:

1. **`obj`**: The object on which the property is to be defined.
2. **`prop`**: A string representing the name of the property.
3. **`descriptor`**: An object that describes the property. It can include the following attributes:
    - **`value`**: The value of the property. Defaults to `undefined`.
    - **`writable`**: A boolean that indicates if the property can be modified. Defaults to `false`.
    - **`enumerable`**: A boolean that indicates if the property shows up during enumeration of the object properties (e.g., in a `for...in` loop or `Object.keys()`). Defaults to `false`.
    - **`configurable`**: A boolean that indicates if the property can be deleted or if its attributes can be modified. Defaults to `false`.
    - **`get`**: A function that serves as a getter for the property.
    - **`set`**: A function that serves as a setter for the property.

Return Type:

- The method returns the object (`obj`) with the new or modified property.

Example 1: Defining a Simple Property

In this example, we define a simple property with a value and make it writable.

```
let person = {};

Object.defineProperty(person, 'name', {
  value: 'Alice',
  writable: true,
  enumerable: true,
  configurable: true
});

console.log(person.name); // Output: Alice

// Modify the 'name' property because writable is true
person.name = 'Bob';
console.log(person.name); // Output: Bob
```

Example 2: Read-Only Property (Non-Writable)

In this example, we define a property but set it as non-writable. Trying to modify it will have no effect.

```
let car = {};

Object.defineProperty(car, 'brand', {
  value: 'Toyota',
  writable: false,   // Non-writable
  enumerable: true,
  configurable: true
});

console.log(car.brand); // Output: Toyota

// Attempt to modify the 'brand' property
car.brand = 'Honda';
console.log(car.brand); // Output: Toyota (Modification didn't happen)
```

Example 3: Adding Getter and Setter

This example demonstrates how to use a getter and setter with `Object.defineProperty()`.

```
let rectangle = {
```

```
  _length: 0,
  _width: 0
};

// Define 'length' property with getter and setter
Object.defineProperty(rectangle, 'length', {
  get() {
    return this._length;
  },
  set(value) {
    if (value <= 0) {
      console.log("Length must be positive.");
    } else {
      this._length = value;
    }
  },
  enumerable: true,
  configurable: true
});

// Define 'width' property with getter and setter
Object.defineProperty(rectangle, 'width', {
  get() {
    return this._width;
  },
  set(value) {
    if (value <= 0) {
      console.log("Width must be positive.");
    } else {
      this._width = value;
    }
  },
  enumerable: true,
  configurable: true
});

rectangle.length = 10;  // Sets the value to 10
rectangle.width = 5;    // Sets the value to 5

console.log(rectangle.length);  // Output: 10
console.log(rectangle.width);   // Output: 5

rectangle.length = -2;  // Error: Length must be positive.
rectangle.width = -3;   // Error: Width must be positive.
```

Example 4: Preventing Deletion of a Property (Non-Configurable)

In this case, we define a property but set it as non-configurable. Attempting to delete the property will result in failure.

```
let user = {};

Object.defineProperty(user, 'age', {
  value: 30,
  writable: true,
  enumerable: true,
  configurable: false // Cannot delete the property
});

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

// Attempt to delete the 'age' property
delete user.age;

console.log(user.age); // Output: 30 (The property is still there)
```

Example 5: Enumerating Properties

In this example, we define a property with `enumerable: false` and show how it doesn't appear in enumerations like `for...in`.

```
let settings = {};

Object.defineProperty(settings, 'theme', {
  value: 'dark',
  writable: true,
  enumerable: false,  // Not enumerable
  configurable: true
});

Object.defineProperty(settings, 'language', {
  value: 'en',
  writable: true,
  enumerable: true,   // Enumerable
```

```
  configurable: true
});

// Enumerate properties of 'settings'
for (let key in settings) {
  console.log(key);  // Output: 'language' (theme doesn't appear)
}
```

Example 6: Using Multiple Properties with Different Configurations

Here, we define multiple properties with various configurations, demonstrating a more complex use case.

```
let book = {};

Object.defineProperty(book, 'title', {
  value: 'JavaScript Basics',
  writable: false, // Cannot change title
  enumerable: true,
  configurable: true
});

Object.defineProperty(book, 'author', {
  value: 'John Doe',
  writable: true,  // Can modify author name
  enumerable: true,
  configurable: true
});

Object.defineProperty(book, 'price', {
  value: 29.99,
  writable: true,
  enumerable: true,
  configurable: false // Cannot delete or modify the property definition
});

console.log(book.title); // Output: JavaScript Basics
console.log(book.author); // Output: John Doe
console.log(book.price); // Output: 29.99

book.title = 'Advanced JavaScript'; // Will not change as it's non-writable
book.author = 'Jane Smith';  // This will work as it's writable
```

```
book.price = 39.99;  // This will work as it's writable

console.log(book.title);  // Output: JavaScript Basics (unchanged)
console.log(book.author); // Output: Jane Smith (modified)
console.log(book.price);  // Output: 29.99 (unchanged, as configurable is false)
```

Summary:

- **Use `Object.defineProperty()`** when you need precise control over a property's behavior.
- The **`descriptor`** allows customization of properties with various flags like `writable`, `enumerable`, `configurable`, as well as `get` and `set` methods.
- **Return** is the object with the defined or modified property.

Object.defineProperties(obj, props)

Sure! Let me walk you through the `Object.defineProperties()` method in JavaScript in a detailed and beginner-friendly manner.

Definition:

`Object.defineProperties()` is a method that allows you to define multiple properties on an object or modify existing properties, including setting specific property descriptors such as `writable`, `configurable`, and `enumerable`. This method can be used to define more than one property at once on an object.

Syntax:

```
Object.defineProperties(obj, props);
```

Arguments:

1. **`obj`**: The object on which you want to define the properties.
2. **`props`**: An object that contains key-value pairs, where the key is the property name, and the value is an object that defines the property descriptor (e.g., `value`, `writable`,

`enumerable`, `configurable`).

Return Type:

- **Returns**: The same object (`obj`) with the newly defined properties.

Property Descriptors:

- **value**: The value of the property.
- **writable**: If `true`, the property value can be changed. Defaults to `false`.
- **enumerable**: If `true`, the property will show up in a `for...in` loop and `Object.keys()` method. Defaults to `false`.
- **configurable**: If `true`, the property can be deleted and its attributes (e.g., writable, enumerable, etc.) can be modified. Defaults to `false`.

Example 1: Basic Usage

```
// Define an object
const person = {};

// Use Object.defineProperties to add properties
Object.defineProperties(person, {
  name: {
    value: "John",
    writable: true,
    enumerable: true,
    configurable: true
  },
  age: {
    value: 30,
    writable: true,
    enumerable: true,
    configurable: true
  }
});

console.log(person); // { name: 'John', age: 30 }
```

Explanation: In this example, we are defining two properties `name` and `age` on the `person` object with descriptors for `writable`, `enumerable`, and `configurable`.

Example 2: Modifying Existing Properties

```
const car = {
  model: "Tesla",
  year: 2020
};

// Modify the 'year' property to be non-writable and non-enumerable
Object.defineProperties(car, {
  year: {
    writable: false,
    enumerable: false,
    configurable: true
  }
});

console.log(car.year); // 2020
car.year = 2021; // This won't change the value because 'writable' is false
console.log(car.year); // 2020
console.log(Object.keys(car)); // ['model'] (year is non-enumerable)
```

Explanation: The `year` property has been modified to be non-writable and non-enumerable. Therefore, we cannot change the `year` value, and it won't appear in the result of `Object.keys()`.

Example 3: Adding Computed Properties

```
const car = {};

// Adding a property with a getter and setter
Object.defineProperties(car, {
  brand: {
    get: function() {
      return this._brand;
    },
    set: function(value) {
      this._brand = value.toUpperCase(); // Ensure brand is stored in uppercase
    },
    enumerable: true,
    configurable: true
  }
```

```
});

car.brand = "bmw"; // Setter is called
console.log(car.brand); // "BMW" (getter is called)
```

Explanation: Here, we defined a `brand` property with a getter and setter. The setter modifies the value before storing it, and the getter retrieves the transformed value.

Example 4: Using `configurable` and `enumerable` to Delete Properties

```
const person = {
  name: "Alice",
  age: 25
};

// Define a 'city' property that is configurable
Object.defineProperties(person, {
  city: {
    value: "New York",
    enumerable: true,
    configurable: true
  }
});

delete person.city; // Deletes the 'city' property because it's configurable
console.log(person.city); // undefined
```

Explanation: Here, we defined a `city` property and then deleted it because it is configurable.

Example 5: Using Multiple Properties with `Object.defineProperties()`

```
const person = {};

Object.defineProperties(person, {
  name: {
    value: "Emma",
    writable: true,
    enumerable: true,
    configurable: true
  },
  age: {
    value: 28,
    writable: false,
```

```
    enumerable: true,
    configurable: true
  },
  greet: {
    value: function() {
      console.log(`Hello, my name is ${this.name}`);
    },
    writable: true,
    enumerable: false, // This will not show up in a for...in loop
    configurable: true
  }
});

console.log(person); // { name: 'Emma', age: 28 }
person.greet(); // Hello, my name is Emma
```

Explanation: We define three properties on the `person` object: `name`, `age`, and `greet`. `greet` is a method that prints a greeting message. It's non-enumerable, so it won't show up in a loop, but it works perfectly as a function.

Creative Ways to Provide Argument Values:

- **Dynamic Property Names**: You can use variables as keys to dynamically define properties.

```
const dynamicProp = "address";
const user = {};
Object.defineProperties(user, {
  [dynamicProp]: {
    value: "123 Street Name",
    writable: true,
    enumerable: true,
    configurable: true
  }
});

console.log(user.address); // 123 Street Name
```

- **Getter and Setter with Complex Logic**: Define a property that processes input using complex logic.

```
const calculator = {};
Object.defineProperties(calculator, {
  result: {
    get: function() {
      return this._result;
    },
    set: function(value) {
      if (value < 0) {
        console.log("Negative values are not allowed!");
      } else {
        this._result = value * value; // Always store the square of the input value
      }
    },
    enumerable: true,
    configurable: true
  }
});

calculator.result = 4;
console.log(calculator.result); // 16
calculator.result = -5; // Negative values are not allowed!
console.log(calculator.result); // 16 (previous valid value)
```

With these examples, you can see how flexible `Object.defineProperties()` is in defining and modifying object properties. It allows you to add or modify properties with custom descriptors, which can control the behavior of those properties with ease.

Object.freeze(obj)

Certainly! Let's dive into the `Object.freeze()` method in JavaScript and break it down from basic to advanced with explanations through comments in the code.

Definition:

`Object.freeze()` is a method in JavaScript used to freeze an object. When an object is frozen, it can no longer be modified: properties cannot be added, removed, or changed. This helps to ensure immutability for that particular object.

Syntax:

Object.freeze(obj);

Arguments:

- **obj**: The object that you want to freeze. The object passed as an argument will have its properties made immutable.

Return Type:

- The method returns the **frozen object**. However, all its properties are now non-writable, non-enumerable, and non-configurable.

Example 1: Basic Usage

Let's look at a basic example to understand how it works:

```
// Creating a simple object
const person = {
   name: 'John',
   age: 30
};

// Freezing the object
Object.freeze(person);

// Trying to modify a property (this won't work)
person.age = 31; // This will not update the 'age' property
person.city = 'New York'; // This will not add a new property

console.log(person); // { name: 'John', age: 30 }
```

Explanation:

- We create an object `person` with properties `name` and `age`.
- We then freeze the object using `Object.freeze()`.

- After freezing, trying to modify or add new properties to `person` will not work. The original object remains unchanged.

Example 2: Frozen Nested Objects

It's important to note that `Object.freeze()` only applies to the immediate properties of the object. If the object has nested objects, those are not frozen unless you explicitly freeze them as well.

```
const user = {
    name: 'Alice',
    address: {
        street: '123 Main St',
        city: 'Wonderland'
    }
};

// Freezing the outer object
Object.freeze(user);

// Trying to modify the outer object (this won't work)
user.name = 'Bob'; // Won't work

// But modifying the nested object is still possible (this will work)
user.address.city = 'Neverland'; // This will still work

console.log(user); // { name: 'Alice', address: { street: '123 Main St', city: 'Neverland' } }
```

Explanation:

- The outer object `user` is frozen, but its `address` object is not. This means the properties of `user` itself are immutable, but the properties inside the nested `address` object can still be modified.

Example 3: Freezing Deeply Nested Objects

To freeze deeply nested objects, you can manually freeze each nested object.

```
function deepFreeze(obj) {
    Object.freeze(obj);
    Object.keys(obj).forEach(key => {
```

```
        if (typeof obj[key] === 'object' && obj[key] !== null) {
            deepFreeze(obj[key]);
        }
    });
}

const settings = {
    theme: 'dark',
    preferences: {
        language: 'English',
        notifications: true
    }
};

deepFreeze(settings);

// Now, the entire object is frozen, including nested objects
settings.preferences.language = 'Spanish'; // This will not work

console.log(settings); // { theme: 'dark', preferences: { language: 'English', notifications: true } }
```

Explanation:

- The `deepFreeze()` function is a custom function that recursively freezes all objects inside the main object. Now, any nested objects are also immutable.

Example 4: Attempting to Modify a Frozen Object (Strict Mode)

In strict mode, attempting to modify a frozen object will throw an error:

```
'use strict';

const car = {
    brand: 'Toyota',
    model: 'Corolla'
};

Object.freeze(car);

car.model = 'Camry'; // Will not work in strict mode and will throw a TypeError

console.log(car); // { brand: 'Toyota', model: 'Corolla' }
```

Explanation:

- In `"use strict"`, JavaScript will throw an error if you attempt to modify a frozen object. This helps catch bugs in the code where you're trying to change immutable data.

Example 5: Freezing an Array

You can also freeze arrays, which prevents the array elements from being modified, added, or deleted.

```
const fruits = ['apple', 'banana', 'cherry'];

// Freezing the array
Object.freeze(fruits);

// Trying to modify the array
fruits[0] = 'orange'; // Won't work
fruits.push('grapes'); // Won't work

console.log(fruits); // ['apple', 'banana', 'cherry']
```

Explanation:

- The array `fruits` is frozen, so you cannot modify its elements or add new ones. It becomes immutable.

Summary of Key Points:

- `Object.freeze()` is used to make an object immutable.
- It freezes only the immediate properties of the object. Nested objects need to be frozen manually.
- It works on both objects and arrays.
- It returns the original frozen object.

- In strict mode, modifying a frozen object results in an error.
- Freezing is shallow, meaning it does not freeze nested objects unless you explicitly do so.

Let me know if you'd like more examples or advanced usage!

Object.seal(obj)

Absolutely! Let's break down the `Object.seal(obj)` method in JavaScript and provide a thorough explanation with examples.

Definition:

`Object.seal()` is a method in JavaScript that seals an object, preventing the addition of new properties and marking all existing properties as non-configurable. However, it doesn't make the properties immutable, meaning their values can still be changed, but their configurations (such as being deleted or redefined) cannot be modified.

Syntax:

```
Object.seal(obj);
```

Arguments:

- **`obj`**: The object to be sealed. This argument is required.

Return Value:

- **Returns** the same object (`obj`) after sealing it.

Key Points about Sealing an Object:

1. **Non-configurable properties**: Once an object is sealed, its properties cannot be removed or redefined (e.g., using `delete` or reassigning property descriptors).
2. **Writable properties**: The values of the properties can still be changed (if the properties are writable).

Examples:

1. Basic Example of Sealing an Object

```
// Define an object
let person = {
  name: 'Alice',
  age: 30
};

// Seal the object
Object.seal(person);

// Trying to add a new property (won't work)
person.gender = 'female';  // This won't be added

// Trying to delete a property (won't work)
delete person.age;  // This won't work

// Modifying an existing property (works)
person.name = 'Bob';  // This will work

console.log(person);  // Output: { name: 'Bob', age: 30 }
```

2. Sealing an Object with Multiple Properties

```
// Define an object with more properties
let car = {
  brand: 'Toyota',
  model: 'Corolla',
  year: 2021
};

// Seal the object
Object.seal(car);

// Trying to add or delete properties will fail
car.color = 'blue';  // Won't work
delete car.year;  // Won't work

// Modifying existing properties is allowed
car.model = 'Camry';  // This will work

console.log(car);  // Output: { brand: 'Toyota', model: 'Camry', year: 2021 }
```

3. Sealing an Object with a Getter/Setter Property

```
let user = {
  firstName: 'John',
  lastName: 'Doe',

  // Getter for full name
  get fullName() {
    return this.firstName + ' ' + this.lastName;
  },

  // Setter for full name
  set fullName(name) {
    [this.firstName, this.lastName] = name.split(' ');
  }
};

// Seal the object
Object.seal(user);

// Attempt to delete getter/setter (won't work)
delete user.fullName;  // This won't work

// Modifying existing properties is still allowed
user.firstName = 'Jane';  // This will work

console.log(user.fullName);  // Output: 'Jane Doe'
console.log(user);  // Output: { firstName: 'Jane', lastName: 'Doe' }
```

4. Using Object.seal() with Arrays

```
let numbers = [1, 2, 3, 4];

// Seal the array object
Object.seal(numbers);

// Adding an element won't work
numbers.push(5);  // Won't work

// Deleting an element won't work
delete numbers[1];  // Won't work

// Changing an element's value will work
numbers[2] = 99;  // This will work
```

```
console.log(numbers);  // Output: [ 1, 2, 99, 4 ]
```

Creative Use of Different Argument Values:

Sealing Nested Objects:

```
 let employee = {
  id: 101,
  department: {
    name: 'Engineering',
    location: 'Building A'
  }
};

Object.seal(employee);
employee.department.name = 'Research';  // This will work
employee.department.location = 'Building B';  // This will work

// But can't delete or add new properties
delete employee.id;  // Won't work
employee.position = 'Manager';  // Won't work

console.log(employee);
```

1.

Sealing an Object with Frozen Nested Properties:

```
 let library = {
  name: 'Central Library',
  books: [
    { title: 'Book 1', author: 'Author 1' },
    { title: 'Book 2', author: 'Author 2' }
  ]
};

Object.seal(library);
Object.freeze(library.books);

// Modifying sealed library properties is allowed
library.name = 'New Library';  // This will work

// Modifying the frozen books array is NOT allowed
library.books[0].title = 'Updated Book 1';  // This will NOT work
```

```
library.books.push({ title: 'New Book', author: 'Author 3' });  // This will NOT work

console.log(library);
```

2.

Summary:

- **Definition**: `Object.seal()` prevents new properties from being added to an object and makes all existing properties non-configurable.
- **Syntax**: `Object.seal(obj)`
- **Arguments**: A single object (`obj`) that you want to seal.
- **Return**: Returns the same object after sealing it.
- **Behavior**: Allows modification of property values but disallows adding, deleting, or changing the configuration of properties.

This should give you a solid understanding of how `Object.seal()` works and how it can be used creatively in different contexts.

Object.preventExtensions(obj)

In JavaScript, `Object.preventExtensions()` is a method used to prevent the addition of new properties to an object. However, it doesn't affect the existing properties of the object—they can still be modified or deleted.

Definition:

`Object.preventExtensions(obj)` is used to freeze the extension of an object. This means no new properties can be added to the object after this method is called, but the existing properties can still be modified or deleted.

Syntax:

```
Object.preventExtensions(obj);
```

Arguments:

- `obj`: The object that you want to prevent extensions for. It must be an object. If a non-object is passed (e.g., `null` or `undefined`), it will throw a `TypeError`.

Return Type:

- This method returns the **same object** that was passed to it.

Example:

Let's go through an example to understand how `Object.preventExtensions()` works:

```
// Define a simple object
let car = {
  brand: 'Tesla',
  model: 'Model S',
  year: 2020
};

// Prevent extensions (no new properties can be added)
Object.preventExtensions(car);

// Try adding a new property
car.color = 'red';  // This will not work, car.color will not be added

// Check the object properties
console.log(car);  // { brand: 'Tesla', model: 'Model S', year: 2020 }

// Try modifying an existing property
car.year = 2022;  // This works, year can still be modified

// Try deleting an existing property
delete car.model;  // This works, existing properties can still be deleted

console.log(car);  // { brand: 'Tesla', year: 2022 }
```

In this example:

1. After calling `Object.preventExtensions()`, we cannot add new properties (e.g., `car.color = 'red';` won't work).

2. We can still modify existing properties (e.g., `car.year = 2022;` is valid).

3. We can still delete existing properties (e.g., `delete car.model;` works).

Creative Ways to Test Different Arguments:

1. **Test with a non-object:**

```
try {
  Object.preventExtensions(null); // Throws TypeError because null is not an object
} catch (error) {
  console.log(error.message); // "Cannot convert undefined or null to object"
}
```

2. **Test with an array (arrays are objects too):**

```
let fruits = ['apple', 'banana', 'cherry'];

Object.preventExtensions(fruits);

fruits.push('date'); // Adding to an array after preventExtensions will fail
console.log(fruits); // ['apple', 'banana', 'cherry']
```

3. **Test with an object that already has properties:**

```
let person = {
  name: 'John',
  age: 30
};

Object.preventExtensions(person);

person.address = '123 Street';  // Can't add new properties
console.log(person);  // { name: 'John', age: 30 }

person.age = 31;  // Can modify existing properties
console.log(person);  // { name: 'John', age: 31 }
```

Key Points:

- **No new properties** can be added after calling `Object.preventExtensions()`.
- **Existing properties** can still be modified or deleted.
- **It is not the same as `Object.freeze()`**, which makes the object completely immutable (no addition, modification, or deletion).

This method is often used when you want to prevent further modifications to an object's structure but still allow for value changes.