## The Object

The Object type represents one of JavaScript's data types. It is used to store various keyed collections and more complex entities.

**Note:**

all objects in JavaScript are instances of Object; a typical object inherits properties (including methods) from Object.prototype, although these properties may be shadowed (overridden). The only objects that don't inherit from Object.prototype are those with null prototype, or descended from other null prototype objects.

## Object() constructor

The Object() constructor turns the input into an object. Its behavior depends on the input's type.

e.g.

```
<script>
const obj = new Object();
obj.value = 42;
console.log(o);
// { foo: 42 }
</script>
```

## empty Object object

e.g.-1

```
const o = new Object();
```

e.g.-2

```
const o = new Object(undefined);
```

e.g.-3

```
const o = new Object(null);
```

## Common Methods

**Object.keys()** : In JavaScript, the **Object.keys()** method is used to retrieve an array of all enumerable property names (keys) of an object. **It takes an object as its parameter and returns an array containing the keys of that object.**

e.g.

```
<html>
<body>
 <script>
   const obj = {
     name: 'John',
     age: 30,
     city: 'New York'
   };

   const keys = Object.keys(obj);
   console.log(keys);
```

```
  </script>
  <body>
</html>
```

**Object. Values()** : Object.values() is a built-in JavaScript method introduced in ECMAScript 2017 (ES8) that allows you to retrieve an array of the values of a given object's own enumerable properties. In other words, it returns an array containing the values associated with the properties of an object.
e.g.

```
<script>
  const person = {
    name: 'John',
    age: 30,
    profession: 'Developer'
  };

  const values = Object.values(person);
  console.log(values); // Output: ['John', 30, 'Developer']
</script>
```

**Object.entries() :** Object.entries() is a built-in JavaScript method introduced in ECMAScript 2017 (ES8) that returns an array of a given object's own enumerable string-keyed property [key, value] pairs, in the same order as that provided by a for...in loop (the difference being that a for...in loop enumerates properties in the prototype chain as well).
e.g.

```
<script>
  let television = {
    Brand: "OnePlus",
    Resolution: "8K",
    price: "30000.00",
    "Display Technology": "LED",
  };

let entries = Object.entries(television);
// console.log(entries);

for (const x of entries) {
  // console.log(x)
  // console.log(x[0])
  // console.log(x[1])
  [key,value] = x
 console.log(`Key => ${key} | Value => ${value}`);
}
</script>
```

**Object.assign():** Object.assign() is a built-in JavaScript method introduced in ECMAScript 2015 (ES6) that is used to copy the values of all enumerable own properties from one or more source objects to a target object. It returns the modified target object.
e.g.

```
<script>
const target = { a: 1, b: 2 };
const source = { b: 3, c: 4 };
const result = Object.assign(target, source);
console.log(result); // Output: { a: 1, b: 3, c: 4 }
</script>
```

**Object.create() :** Object.create() is a built-in JavaScript method that creates a new object with the specified prototype object and properties. It allows you to create an object and set its prototype at the same time. This method is particularly useful when you want to create objects with specific prototypes or when you want to create objects with no prototype at all.

**Syntax :**

```
Object.create(proto [, propertiesObject])
```

**Parameters:**

- proto: The object to be used as the prototype of the newly created object. This parameter can be null or omitted if you want to create an object with no prototype.
- propertiesObject (optional): An object whose properties are descriptors for the properties to be added to the newly created object. These properties correspond to the properties of the newly created object, and their values are descriptors.

**Use Case 1: Creating Objects with Specific Prototypes**

e.g.-1

```
<script>
// Prototype object
const animal = {
  makeSound: function() {
    console.log(this.sound);
  }
};

// Create objects with animal prototype
const dog = Object.create(animal);
dog.sound =  "Woof";
dog.makeSound(); // Output: Woof

const cat = Object.create(animal);
cat.sound = "Meow";
cat.makeSound(); // Output: Meow
</script>
```

**Use Case 2: Prototypal Inheritance**

**e.g.-2**

```
<script>
// Parent object
const vehicle = {
drive: function() {
   console.log("Driving...");
  }
};

// Child object inheriting from vehicle
```

```
const car = Object.create(vehicle);
car.honk = function() {
console.log("Beep beep!");
};

car.drive(); // Output: Driving...
car.honk(); // Output: Beep beep!
```

**Use Case 3: Creating Objects with No Prototype**
e.g.-3

```
<script>
// Creating an object with no prototype
const noPrototypeObj = Object.create(null);
noPrototypeObj.name = "John";
console.log(noPrototypeObj.name); // Output: John
// This object has no inherited properties or methods
console.log(noPrototypeObj.toString); // Output: undefined
</script>
</script>
```

**Object.defineProperty():** Object.defineProperty() is a built-in JavaScript method that allows you to define a new property directly on an object or modify an existing property on an object with more control over its behavior. It provides a way to precisely define the characteristics of a property such as its value, writability, configurability, and enumerability.

**Syntax**

Object.defineProperty(obj, prop, descriptor)

**Parameters:**

- **obj:** The object on which to define or modify the property.
- **prop:** The name or symbol of the property to be defined or modified.
- **descriptor:** An object that specifies the configuration of the property being defined or modified. It contains keys such as value, writable, enumerable, configurable, which control the behavior of the property.

**Return Value:**

The object obj that was passed to the function.

e.g.

```
<script>
const obj = {};
Object.defineProperty(obj, 'name', {
  value: 'John',
  writable: false,
  enumerable: true,
  configurable: true
});
console.log(obj.name); // Output: John
obj.name ="vinay";
console.log(obj.name);
</script>
```

**Object.freeze() :** Object.freeze() is a built-in JavaScript method that freezes an object, making it immutable. Once an object is frozen, its properties cannot be added, deleted, or modified. Any attempt to modify a frozen object will result in an error (in strict mode in ECMAScript 5 and above) or will fail silently (in non-strict mode).

**Syntax**

Object.freeze(obj)

e.g.

```
<script>
const obj = {
  prop1: 'value1',
  prop2: 'value2'
};

Object.freeze(obj);

obj.prop1 = 'new value'; // Error in strict mode or silently fails in non-strict mode

</script>
```

**Behavior:**

- Object.freeze() makes the object and its properties read-only. Any attempt to modify the object or its properties will fail.
- It prevents new properties from being added to the object.
- It prevents existing properties from being deleted from the object.
- It prevents existing properties from being reconfigured (e.g., changing their attributes such as writable, enumerable, configurable).
- If the object contains nested objects, those nested objects are not frozen and can still be modified.
- Attempting to modify a frozen object or one of its properties in strict mode throws a TypeError. In non-strict mode, the modification fails silently.

**Use Cases:**
**Data integrity**: Object.freeze() can be used to ensure that an object's properties remain unchanged after initialization, preventing accidental modification.
**Immutable objects:** Freezing an object is useful when you want to create immutable data structures, which are important in functional programming paradigms and for ensuring data integrity in concurrent programming.
**Performance optimization:** Freezing objects can also improve performance in certain scenarios, as it allows JavaScript engines to optimize memory usage and avoid unnecessary property lookups during property access.

**Object.isFrozen() :** Object.isFrozen() is a built-in JavaScript method that checks if an object is frozen or not. It returns a boolean value indicating whether the specified object is frozen or not.

**Syntax:**

Object.isFrozen(obj)

e.g.

```
<script>
const obj = {
  prop1: 'value1',
  prop2: 'value2'
};
console.log(Object.isFrozen(obj)); // Output: false
Object.freeze(obj);
```

```
console.log(Object.isFrozen(obj)); // Output: true
</script>
```

**Object.seal():** Object.seal() is a built-in JavaScript method that seals an object, preventing new properties from being added to it and marking all existing properties as non-configurable. However, the values of the existing properties can still be modified if they are writable. Essentially, sealing an object makes it immutable in terms of its structure, but mutable in terms of its values.

**Use Cases:**
**Data integrity:** Object.seal() can be used to ensure that an object's structure remains fixed after initialization, preventing new properties from being added or existing properties from being removed or reconfigured.
**Preventing accidental changes:** It can help prevent accidental modifications to an object's structure, providing additional safeguards against unexpected behavior in the code.

**Syntax:**

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

e.g.

```
<script>
const obj = {
  prop1: 'value1',
  prop2: 'value2'
};

Object.seal(obj);

obj.prop1 = 'new value'; // Allowed
obj.prop3 = 'value3';   // Not allowed in strict mode; silently fails in non-strict mode

console.log(obj.prop1)
console.log(obj.prop3)
</script>
```

**Object.isSealed():** Object.isSealed() is a built-in JavaScript method that checks if an object has been sealed using Object.seal(). It returns a boolean value indicating whether the specified object is sealed or not.

**Syntax**

```
Object.isSealed(obj)
```

**e.g.**

```
<script>
const obj = {
  prop1: 'value1',
  prop2: 'value2'
};
Object.seal(obj);
console.log(Object.isSealed(obj)); // Output: true
</script>
```

**Object.setPrototypeOf():** Object.setPrototypeOf() provides a flexible way to manipulate the prototype of an object, allowing for dynamic changes to the object's behavior and inheritance structure in JavaScript.

**Note:**

However, it's generally recommended to avoid changing prototypes dynamically due to potential performance implications and difficulties in reasoning about code.

**Syntax:**
Object.setPrototypeOf(obj, prototype)

e.g.
```
<script>
const obj = {};
const proto = {
  greet: function() {
    console.log('Hello!');
  }
};
Object.setPrototypeOf(obj, proto);
obj.greet(); // Output: Hello!
</script>
```

**Object.getOwnPropertyDescriptor() :** Object.getOwnPropertyDescriptor() is a built-in JavaScript method that returns an object containing the descriptor for a specified property of an object. The descriptor provides information about the property's attributes, such as its value, writability, enumerability, and configurability.
**Syntax:**
Object.getOwnPropertyDescriptor(obj, prop)
e.g.
```
<script>
const obj = {
  property: 'value'
};

const descriptor = Object.getOwnPropertyDescriptor(obj, 'property');

console.log(descriptor);
// Output:
// {
//  value: 'value',
//  writable: true,
//   enumerable: true,
//   configurable: true
// }
</script>
```

**Object.preventExtensions():** Object.preventExtensions() is a built-in JavaScript method that prevents new properties from being added to an object. It effectively makes an object non-extensible, meaning that it cannot have new properties added to it, but existing properties can still be modified or deleted.

**Difference between seal() & preventExtension()**
**Object.seal()** restricts both addition of new properties and modification of existing properties' attributes, making the object effectively immutable in terms of its structure.
**Object.preventExtensions()** only restricts the addition of new properties, allowing existing properties to be modified or deleted freely.

e.g.
```
<script>
const obj = {
  prop1: 'value1',
  prop2: 'value2'
};
Object.preventExtensions(obj);
obj.prop1 = 'new value'; // Allowed
obj.prop3 = 'value3'; // Not allowed in strict mode; silently fails in non-strict mode
console.log(obj.prop1)
console.log(obj.prop3)
</script>
```

**Object.is() :** Object.is() is a built-in JavaScript method that compares two values and determines whether they are the same value. It differs from the traditional equality operators (== and ===) in certain edge cases, providing a more accurate and predictable way to compare values.

**Object.is() compares two values using the SameValueZero algorithm,** which behaves similarly to the strict equality operator (===) but with some differences.
Unlike ===, Object.is() considers +0 and -0 as different values, and it also considers NaN values as the same.
For all other values, Object.is() behaves identically to ===.

**Syntax**

Object.is(value1, value2)

e.g.

```
console.log(Object.is(1, 1)); // Output: true
console.log(Object.is('foo', 'foo')); // Output: true
console.log(Object.is(0, -0)); // Output: false
onsole.log(Object.is(NaN, NaN)); // Output: true
```

**Computed property names**

Computed property names in JavaScript allow you to dynamically determine the names of object properties using expressions inside square brackets ([]) within object literals. This feature was introduced in ECMAScript 6 (ES6) to provide more flexibility in defining object properties.

Syntax

```
let propertyName = 'dynamicPropertyName';
let obj = {
  [propertyName]: propertyValue
};
```

**e.g.-1**

```
<script>
let n = "name";
var obj = {[n] :"target", course: "Btech"
};
console.log(obj);
console.log(obj.name);
</script>
```

ıigning
.net
ıng

**e.g.-2**

```
<script>
  let n = "student";
  var obj = {
    [n + "name"]: "Target",
    course: "Btech",
    detail: function () {
      return `${this.studentname} is student of ${this.course}`
    }
  }
  console.log(obj);
  console.log(obj.detail());
</script>
```

## CHECK IF A PROPERTY IS AVAILABLE

In JavaScript, you can check if a property is available in an object using various methods. Here are a few common ways to do it:

1. **Using the in operator**: The ***in operator*** checks if a property exists in an object, including properties inherited from the object's prototype chain.
   e.g.
   ```
     const myObject = { key: 'value' };
     if ('key' in myObject) {
     console.log('Property exists');
     } else {
       console.log('Property does not exist');
     }
   ```

2. **Using the hasOwnProperty method:** The hasOwnProperty method checks if a property is a direct property of the object and does not include properties inherited from the prototype chain.
   e.g.
   ```
     const myObject = { key: 'value' };
     if (myObject.hasOwnProperty('key')) {
       console.log('Property exists');
     } else {
       console.log('Property does not exist');
     }
   ```

3. **Using the undefined check:** You can also check if a property is `undefined` to determine if it exists in the object.
   e.g.
   ```
   const myObject = { key: 'value' };
   if (myObject.key !== undefined) {
     console.log('Property exists');
   } else {
      console.log('Property does not exist');
   }
   ```

4. **Using optional chaining (ES2020+):** You can use optional chaining to check for the existence of a property in a safe way.

**e.g.-1**

```
 const myObject = { key: 'value' };
 if (myObject?.key) {
 console.log('Property exists');
} else {
  console.log('Property does not exist');
}
```

**Optional chaining** is a feature introduced in ECMAScript 2020 (ES11) that provides a concise way to access properties and methods of an object without the need to explicitly check if each property exists along the path. It allows you to safely access nested properties or call methods even if some intermediate properties do not exist or are nullish (null or undefined). This helps to prevent errors such as TypeError: Cannot read property 'x' of undefined.

**e.g.-2**

```
<script>
const obj = {
foo: {
   bar: {
     baz: 'value'
   }
  }
};

console.log(obj?.foo?.bar?.baz); // Output: 'value'

const nullObj = null;
console.log(nullObj?.foo?.bar?.baz); // Output: undefined (no error)
</script>
```

```
let result;
if (obj && obj.foo &&
obj.foo.bar && obj.foo.bar.baz) {
   result = obj.foo.bar.baz;
} else {
   result = undefined;
}
```

**Akhilesh Kumar Gupta**
**(Technical Training specialist)**
**TechCarrel LLP**