**JavaScript Objects**

### JavaScript Objects: The Basics

In JavaScript, an **object** is a collection of **key-value pairs**. It's used to store structured data and more complex entities. Think of an object as a real-world item (like a car or a person) that has various properties (like color, make, or name, age) and can perform actions (like driving or talking).

* **Keys (or Property Names):** These are usually strings (or Symbols in ES6+) that uniquely identify a piece of data within the object.
* **Values (or Property Values):** These can be any JavaScript data type – numbers, strings, booleans, arrays, other objects, or even functions.
* **Non-primitive:** Unlike strings or numbers, objects are non-primitive data types. When you copy an object, you're copying a *reference* to the object, not the object itself.
* **Mutable:** You can change an object's properties after it's created.

**1. Creating Objects:**

* **Object Literal (Most Common and Recommended):**This is the simplest and most widely used way to create an object.

let person = {

    firstName: "Alice",

    lastName: "Smith",

    age: 30,

    isStudent: false,

    hobbies: ["reading", "hiking"],

    address: {

        street: "123 Main St",

        city: "Anytown"

    },

   }

* **new Object() Constructor :**Less common for creating simple objects, but useful for more complex scenarios or when adding properties dynamically.

let car = new Object();

car.make = "Toyota";

.model = "Camry";

car.year = 2020;

car.start = function() {

    console.log("Engine started!");

};

* **Object.create():**Creates a new object, using an existing object as the prototype of the newly created object. This is for advanced inheritance patterns.

const animal = {

    isAlive: true,

    speak: function() {

        console.log("Generic animal sound");

    }

};

const dog = Object.create(animal);

dog.name = "Buddy";

dog.breed = "Golden Retriever";

dog.speak = function() {

    console.log("Woof!");

};

console.log(dog.isAlive); // true (inherited from animal)

dog.speak();              // Woof! (overrides animal's speak)

**2. Accessing Object Properties:**

* **Dot Notation (Most Common):** Use when the property name is a known, valid identifier.

console.log(person.firstName);    // "Alice"

console.log(person.age);          // 30

console.log(person.address.city); // "Anytown"

* **Bracket Notation:** Use when the property name contains special characters (like spaces or hyphens), starts with a number, or when the property name is stored in a variable.

console.log(person["lastName"]);   // "Smith"

let propName = "age";

console.log(person[propName]);     // 30

let complexKey = "email address";

let user = {

    "email address": "test@example.com"

};

console.log(user[complexKey]); // "test@example.com"

**3. Modifying Object Properties:**

You can change existing properties or add new ones

let person = {

    firstName: "Alice",

    age: 30

};

person.age = 31;            // Modify existing property

person.country = "USA";     // Add new property

person["isMarried"] = true; // Add new property using bracket notation

console.log(person);

// Output: { firstName: 'Alice', age: 31, country: 'USA', isMarried: true }

**4. Deleting Object Properties:**

Use the delete operator.

let product = {

    name: "Laptop",

    price: 1200,

    inStock: true

};

delete product.inStock;

console.log(product); // { name: 'Laptop', price: 1200 }

### JavaScript Object Methods

An **object method** is a property of an object whose value is a **function**. Methods allow objects to perform actions related to their data.

From our person example:

let person = {

    firstName: "Alice",

    lastName: "Smith",

    greet: function() { // This is a method

        console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);

    }

};

person.greet(); // Calling the method: "Hello, my name is Alice Smith."

The this keyword inside a method refers to the object that the method is called on.

### Common Built-in Object Methods (Static Methods of the Object Constructor)

These methods are called directly on the Object *constructor* itself (Object.methodName()), not on individual object instances. They are used to inspect, manipulate, or create objects.

1. **Object.keys(obj):**
   * Returns an array of a given object's own enumerable string property names.

const user = { name: "Bob", age: 40, city: "New York" };

const keys = Object.keys(user);

console.log(keys); // ["name", "age", "city"]

1. **Object.values(obj):**
   * Returns an array of a given object's own enumerable string property values.

const user = { name: "Bob", age: 40, city: "New York" };

const values = Object.values(user);

console.log(values); // ["Bob", 40, "New York"]

1. **Object.entries(obj):**
   * Returns an array of a given object's own enumerable string property [key, value] pairs.

const user = { name: "Bob", age: 40 };

const entries = Object.entries(user);

console.log(entries); // [ ["name", "Bob"], ["age", 40] ]

// Useful for iterating

for (const [key, value] of Object.entries(user)) {

    console.log(`${key}: ${value}`);

}

// name: Bob

// age: 40

1. **Object.assign(target, ...sources):**
   * Copies all enumerable own properties from one or more source objects to a target object.
   * Returns1 the target object.
   * Useful for merging objects or creating shallow copies.

const obj1 = { a: 1, b: 2 };

const obj2 = { b: 3, c: 4 };

// Merging

const mergedObj = Object.assign({}, obj1, obj2); // {} is the target, creates a new object

console.log(mergedObj); // { a: 1, b: 3, c: 4 } (b from obj2 overwrites b from obj1)

// Shallow copy

const copyOfObj1 = Object.assign({}, obj1);

console.log(copyOfObj1); // { a: 1, b: 2 }

*Note: The spread syntax (...) is often preferred for merging and shallow copying due to its conciseness:*

const mergedWithSpread = { ...obj1, ...obj2 }; // { a: 1, b: 3, c: 4 }  
const copyWithSpread = { ...obj1 }; // { a: 1, b: 2 }

1. **Object.freeze(obj):**
   * Freezes an object. A frozen object can no longer be changed (cannot add, delete, or modify properties; its prototype cannot be changed).
   * Returns the frozen object.
   * Shallow freeze (nested objects can still be modified).

const config = { database: "mydb", port: 8080 };

Object.freeze(config);

config.port = 9000;    // This will have no effect

config.newProp = "test"; // This will have no effect

delete config.database;  // This will have no effect

console.log(config); // { database: "mydb", port: 8080 }

1. **Object.seal(obj):**
   * Seals an object. You cannot add new properties or delete existing properties, but you *can* modify existing properties.
   * Returns the sealed object.
   * Shallow seal.

const settings = { theme: "dark", notifications: true };

Object.seal(settings);

settings.notifications = false; // Works!

settings.newProp = "abc";      // Has no effect

delete settings.theme;         // Has no effect

console.log(settings); // { theme: "dark", notifications: false }

1. **Object.is(value1, value2):**
   * Determines whether two values are the same value.
   * This is a more robust comparison than == or === for certain edge cases (like NaN).

console.log(Object.is(25, 25));       // true

console.log(Object.is("foo", "foo")); // true

console.log(Object.is(NaN, NaN));     // true (unlike NaN === NaN, which is false)

console.log(Object.is(0, -0));        // false (unlike 0 === -0, which is true)