JAVASCRIPT OBJECTS - BASICS

Object Literals is a data structure type that is defined by key value pairs. (remember the use of CRUD: CREATE, READ, UPDATE, DELETE)

(CREATE) Example of an object literal:

```
const person = {
  name: ‘David Alan’,
  age:30,
  occupation: ‘welder’
  address: ‘123 Alphabet Street’
}
```

objects work by the structure **key: value**.

The keys name, age, and occupation are **properties** of an object literal. ‘David Alan’, 30, and ‘welder’ are **values** within an object literal.

(READ) These can be accessed using variables and console.log

```
const games = {
  soulslike: ‘bloodborne’,
  rpg: ‘Fallout_4’
  shooter: ‘Borderlands’
}

let gamelist = games;
console.log(gamelist);
console.log(gamelist.soulslike);
```

(UPDATE) **add** a property and a value:

```
games.platformer = ‘Mario Bros’;
console.log(gamelist);
```

(UPDATE) **remove** a property or value:
delete games.rpg;
console.log(gamelist);
// THE DELETE OPERATOR SPECIFICALLY REMOVES A PROPERTY ONLY

(DELETE) completely throw an object into garbage collection:

```
games = null;
console.log(gamelist);
```

Adding a Function to an Object in Javascript:

Creating a function within an object in JavaScript can be achieved using a few different methods, primarily through standard property assignment or using the shorthand method syntax.

1. Using Standard Property Assignment (Function Expressions)

This is a common and straightforward method where you assign a function expression as the value of an object property [1].

```
const myObject = {
  // Define a property and assign a function expression as its value
  myFunction: function(param1, param2) {
    return param1 + param2;
  }
};

// Call the function
console.log(myObject.myFunction(5, 10)); // Output: 15
```

2. Using the Shorthand Method Syntax

ES6 (ECMAScript 2015) introduced a cleaner, more concise syntax for defining methods inside objects. This is the preferred modern approach [1].

```
const myObject = {
  // Shorthand syntax: omit the "function" keyword and the colon
  myFunction(param1, param2) {
    return param1 + param2;
  }
};

// Call the function
console.log(myObject.myFunction(5, 10)); // Output: 15
```

3. Using Arrow Functions (Function Expressions)

You can also use arrow functions for methods, although caution is needed with the this keyword, as arrow functions do not bind their own this but rather inherit it from the surrounding scope [1, 2].

```
const myObject = {
  // Define a property and assign an arrow function as its value
  myFunction: (param1, param2) => {
    return param1 + param2;
  }
};

// Call the function
console.log(myObject.myFunction(5, 10)); // Output: 15
```

4. Adding a User-defined Function to an Existing Object

If you already have an object and want to add a function later, you can simply assign it like any other property.

```
const myObject = {
  name: "Example"
};

// Add a new method to the existing object
myObject.sayHello = function() {
  console.log("Hello, " + this.name);
};

// Call the function
myObject.sayHello(); // Output: Hello, Example
```

Usually the **shorthand method** is considered to be the most concise.

The **this** keyword may be used to access a property from within the function.

Object Methods:

Common **object methods** in JavaScript are primarily static methods available on the global Object constructor, used for tasks like property access, iteration, copying, and protecting objects.

Here are some of the most frequently used methods:

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

```
const person = { name: 'John', age: 30 };
console.log(Object.keys(person)); // Output: ['name', 'age']
```

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

```
const person = { name: 'John', age: 30 };
console.log(Object.values(person)); // Output: ['John', 30]
```

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

```
const person = { name: 'John', age: 30 };
console.log(Object.entries(person)); // Output: [['name', 'John'], ['age',
30]]
```

Object.assign(target, source): Copies the values of all enumerable own properties from one or more source objects to a target object.

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };
const returnedTarget = Object.assign(target, source);
console.log(target); // Output: { a: 1, b: 4, c: 5 }
```

Object.create(proto): Creates a new object, using an existing object as the prototype of the newly created object.

```
const animal = {
  speak() {
    console.log('Moo');
  }
};
const cow = Object.create(animal);
cow.speak(); // Output: Moo
```

Object.freeze(obj): "Freezes" an object, preventing new properties from being added, existing properties from being removed, and the writability or configurability of existing properties from being changed.
javascript

```
const obj = { prop: 1 };
Object.freeze(obj);
obj.prop = 2; // Fails silently or throws a TypeError in strict mode
console.log(obj.prop); // Output: 1
```

Object.hasOwn(obj, prop): A safer and more modern way to check if an object has a specified own property (as opposed to one inherited via the prototype chain).

```
const person = { name: 'John' };
console.log(Object.hasOwn(person, 'name')); // Output: true
console.log(Object.hasOwn(person, 'age'));  // Output: false
```