## Javascript Objects and Internal Representation

**Introduction**

JavaScript is template based and we can create objects without the need of having a class. Generally, an object is accompanied by a class that defines an object’s blueprints but JavaScript doesn’t require any class to be defined for objects. An object is a standalone entity, with properties and type. Compare it with a cup, for example. A cup is an object, with properties.A JavaScript object is like a real-world entity having state and behaviour, for example, a car. We can take a car as an object. It will have a state like colour and model. It will also have behaviours like accelerating and brake.

**Object():** can be called with or without new. Both create a new object.

**Syntax:**

new Object(value)

Object(value)

let object\_name = {

key\_name : value,

...

}

**JavaScript Object Properties:** The property names can be strings or numbers. In case the property names are numbers, they must be accessed using the “bracket notation” like this.

In JavaScript, an object is a collection of key-value pairs that represent properties of an entity. The keys are strings or Symbols, and the values can be any data type, including other objects. The internal representation of an object in JavaScript is called its "object structure" or "object layout." This includes information about the object's properties, such as their names, values, and types, as well as information about the object's prototype (if it has one) and any other internal data. The specifics of the object structure can vary depending on the JavaScript engine and implementation, but generally, it is designed to be fast and efficient for property lookups and other common operations on objects.

**Example;**

const car = {

brand: 'Toyota',

model: 'Camry',

year: 2022,

color: 'blue',

isElectric: false,

features: ['navigation', 'sunroof', 'leather seats'],

owner: {

name: 'Alice',

age: 30,

address: '1234 Elm Street'

},

startEngine: function() {

console.log('The car engine has started.');

},

stopEngine: function() {

console.log('The car engine has stopped.');

}

};

In this example, we have an object called car that represents a car entity. It has various properties such as brand, model, year, color, and isElectric. The features property is an array containing different features of the car. The owner property is another object with properties like name, age, and address.

The car object also contains two methods, startEngine and stopEngine, which are functions defined as properties. These methods can be called on the car object to perform specific actions.

**Here's how you can access the properties and call the methods of this object:**

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

console.log(car.features[1]); // Output: 'sunroof'

console.log(car.owner.name); // Output: 'Alice'

car.startEngine(); // Output: 'The car engine has started.'

car.stopEngine(); // Output: 'The car engine has stopped.'

Objects in JavaScript are versatile and can store various types of data, including primitive values, arrays, and even other objects. They are commonly used to represent and manipulate complex data structures in JavaScript.

**The JavaScript engine determines an object's internal representation based on several factors:**

* Object literal notation: When you create an object using object literal notation ({}), the engine creates a new object and assigns the provided key-value pairs as properties of that object. The engine analyse the object literal and constructs the internal representation accordingly.
* Constructor functions: When you create an object using a constructor function (e.g., new SomeConstructor()), the constructor function serves as a blueprint for creating new objects. The engine associates the properties and methods defined within the constructor function with the objects created from it.
* Prototypes: JavaScript uses prototypal inheritance, where objects can inherit properties and methods from other objects called prototypes. Each object has an internal reference to its prototype. When you access a property or method on an object, the engine first checks if the object itself has that property. If not, it looks up the prototype chain until it finds the property or reaches the end of the chain. The prototype chain allows for efficient memory usage and property inheritance.
* Property additions and deletions: JavaScript objects are dynamic, meaning you can add or remove properties at runtime. When you add a new property to an object, the engine updates its internal representation to include the new property. Similarly, when you delete a property, the engine removes it from the internal representation.
* Hidden classes: Some JavaScript engines, like V8, use a technique called hidden classes to optimize property access. Hidden classes are internal structures that map to an object's properties and their positions in memory. When you consistently add properties to objects in a predictable pattern, the engine can optimize property access by assigning objects with the same property layout to the same hidden class. This optimization improves performance by avoiding unnecessary property lookups.

It's important to note that the exact mechanisms and optimizations employed by JavaScript engines may differ between implementations. The internal representation of objects is an implementation detail of the engine and is not explicitly specified by the JavaScript language specification.