Object initializer

It is a comma-delimited list of zero or more pairs of property names and associated values of an object, enclosed in curly braces (`{}`):

```
var myCar = {
    make: 'Ford',
    model: 'Mustang',
    year: 1969
};
```

Unassigned properties of an object are `undefined` (and not `null`).

```
myCar.color; // undefined
```

Properties of JavaScript objects can also be accessed or set using a bracket notation (for more details see property accessors). Objects are sometimes called *associative arrays*, since each property is associated with a string value that can be used to access it. So, for example, you could access the properties of the `myCar` object as follows:

```
myCar['make'] = 'Ford';
myCar['model'] = 'Mustang';
myCar['year'] = 1969;
```

An object property name can be any valid JavaScript string, or anything that can be converted to a string, including the empty string. However, any property name that is not a valid JavaScript identifier (for example, a property name that has a space or a hyphen, or that starts with a number) can only be accessed using the square bracket notation. This notation is also very useful when property names are to be dynamically determined (when the property name is not determined until runtime). Examples are as follows:

```
// four variables are created and assigned in a single go,
// separated by commas
var myObj = new Object(),
    str = 'myString',
    rand = Math.random(),
    obj = new Object();

myObj.type              = 'Dot syntax';
myObj['date created']   = 'String with space';
myObj[str]              = 'String value';
myObj[rand]             = 'Random Number';
myObj[obj]              = 'Object';
myObj['']               = 'Even an empty string';

console.log(myObj);
```

Please note that all keys in the square bracket notation are converted to string unless they're Symbols, since JavaScript object property names (keys) can only be strings or Symbols (at some point, private names will

also be added as the class fields proposal progresses, but you won't use them with `[]` form). For example, in the above code, when the key `obj` is added to the `myObj`, JavaScript will call the `obj.toString()` method, and use this result string as the new key.

You can also access properties by using a string value that is stored in a variable:

```
var propertyName = 'make';
myCar[propertyName] = 'Ford';

propertyName = 'model';
myCar[propertyName] = 'Mustang';
```

You can use the bracket notation with **for...in** to iterate over all the enumerable properties of an object. To illustrate how this works, the following function displays the properties of the object when you pass the object and the object's name as arguments to the function:

```
function showProps(obj, objName) {
  var result = ``;
  for (var i in obj) {
    // obj.hasOwnProperty() is used to filter out properties from the object's
prototype chain
    if (obj.hasOwnProperty(i)) {
      result += `${objName}.${i} = ${obj[i]}\n`;
    }
  }
  return result;
}
```

So, the function call `showProps(myCar, "myCar")` would return the following:

```
myCar.make = Ford
myCar.model = Mustang
myCar.year = 1969
```

Enumerate the properties of an object

Starting with ECMAScript 5, there are three native ways to list/traverse object properties:

- `for...in` loops
  This method traverses all enumerable properties of an object and its prototype chain
- `Object.keys(o)`
  This method returns an array with all the own (not in the prototype chain) enumerable properties' names ("keys") of an object `o`.

Alternatively, you can create an object with these two steps:

1. Define the object type by writing a constructor function. There is a strong convention, with good reason, to use a capital initial letter.
2. Create an instance of the object with `new`.

To define an object type, create a function for the object type that specifies its name, properties, and methods. For example, suppose you want to create an object type for cars. You want this type of object to be called `Car`, and you want it to have properties for make, model, and year. To do this, you would write the following function:

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

Notice the use of `this` to assign values to the object's properties based on the values passed to the function.

Now you can create an object called `mycar` as follows:

```
var mycar = new Car('Eagle', 'Talon TSi', 1993);
```

This statement creates `mycar` and assigns it the specified values for its properties. Then the value of `mycar.make` is the string "Eagle", `mycar.year` is the integer 1993, and so on.

You can create any number of `Car` objects by calls to `new`. For example,

```
var kenscar = new Car('Nissan', '300ZX', 1992);
var vpgscar = new Car('Mazda', 'Miata', 1990);
```

An object can have a property that is itself another object. For example, suppose you define an object called `person` as follows:

```
function Person(name, age, sex) {
  this.name = name;
  this.age = age;
  this.sex = sex;
}
```

and then instantiate two new `person` objects as follows:

```
var rand = new Person('Rand McKinnon', 33, 'M');
var ken = new Person('Ken Jones', 39, 'M');
```

Then, you can rewrite the definition of `Car` to include an `owner` property that takes a `person` object, as follows:

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
}
```

To instantiate the new objects, you then use the following:

```
var car1 = new Car('Eagle', 'Talon TSi', 1993, rand);
var car2 = new Car('Nissan', '300ZX', 1992, ken);
```

Notice that instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects `rand` and `ken` as the arguments for the owners. Then if you want to find out the name of the owner of car2, you can access the following property:

```
car2.owner.name
```

Note that you can always add a property to a previously defined object. For example, the statement

```
car1.color = 'black';
```

adds a property `color` to car1, and assigns it a value of "black." However, this does not affect any other objects. To add the new property to all objects of the same type, you have to add the property to the definition of the `Car` object type.

Using `this` for object references

JavaScript has a special keyword, `this`, that you can use within a method to refer to the current object. For example, suppose you have 2 objects, `Manager`and `Intern`. Each object have their own `name`, `age` and `job`. In the function `sayHi()`, notice there is `this.name`. When added to the 2 objects they can be called and returns the `'Hello, My name is'` then adds the `name` value from that specific object. As shown below.

```
const Manager = {
  name: "John",
  age: 27,
  job: "Software Engineer"
}
const Intern= {
  name: "Ben",
  age: 21,
  job: "Software Engineer Intern"
}

function sayHi() {
    console.log('Hello, my name is', this.name)
}

// add sayHi function to both objects
Manager.sayHi = sayHi;
Intern.sayHi = sayHi;

Manager.sayHi() // Hello, my name is John'
Intern.sayHi() // Hello, my name is Ben'
```

The `this` refers to the object that it is in. You can create a new function called `howOldAmI()`which logs a sentence saying how old the person is.

```
function howOldAmI (){
  console.log('I am ' + this.age + ' years old.')
}
Manager.howOldAmI = howOldAmI;
Manager.howOldAmI() // I am 27 years old.
```