Working with objects

JavaScript is designed on a simple object-based paradigm. An object is a collection of properties, and a property is an association between a name (or key) and a value. A property's value can be a function, in which case the property is known as a method.

In JavaScript, 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 cup has a color, a design, weight, a material it is made of, etc. In the same way, JavaScript objects can have properties, which define their characteristics.

Creating new objects

You can create an object using an object initializer. Alternatively, you can first create a constructor function and then instantiate an object by invoking that function with the `new` operator.

Using object initializers

Object initializers are also called *object literals*. "Object initializer" is consistent with the terminology used by C++.

The syntax for an object using an object initializer is:

```
const obj = {
  property1: value1, // property name may be an identifier
  2: value2, // or a number
  "property n": value3, // or a string
};
```

Each property name before colons is an identifier (either a name, a number, or a string literal), and each `valueN` is an expression whose value is assigned to the property name. The property name can also be an expression.

In this example, the newly created object is assigned to a variable obj — this is optional. If you do not need to refer to this object elsewhere, you do not need to assign it to a variable.

Object initializers are expressions, and each object initializer results in a new object being created whenever the statement in which it appears is executed. Identical object initializers create distinct objects that do not compare to each other as equal.

The following example creates `myHonda` with three properties. Note that the `engine` property is also an object with its own properties.

```
const myHonda = {
  color: "red",
  wheels: 4,
  engine: { cylinders: 4, size: 2.2 },
};
```

Objects created with initializers are called *plain objects*, because they are instances of `Object`, but not any other object type. Some object types have special initializer syntaxes — for example, *array initializers* and *regex literals*.

Using a constructor function

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:

```
const 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.model` is the string `"Talon TSi"`, `myCar.year` is the integer `1993`, and so on. The order of arguments and parameters should be the same.

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

```
const kenscar = new Car("Nissan", "300ZX", 1992);
const 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:

```
const rand = new Person("Rand McKinnon", 33, "M");
const 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:

```
const car1 = new Car("Eagle", "Talon TSi", 1993, rand);
const 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;
```

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 the Object.create() method

Objects can also be created using the Object.create() method. This method can be very useful, because it allows you to choose the prototype object for the object you want to create, without having to define a constructor function:

```
// Animal properties and method encapsulation
const Animal = {
  type: "Invertebrates", // Default value of properties
  displayType() {
    // Method which will display type of Animal
    console.log(this.type);
  },
};

// Create new animal type called animal1
const animal1 = Object.create(Animal);
animal1.displayType(); // Logs: Invertebrates
```

```
// Create new animal type called fish
const fish = Object.create(Animal);
fish.type = "Fishes";
fish.displayType(); // Logs: Fishes
```

Objects and properties

You can create an object using an object initializer. Alternatively, you can first create a constructor function and then instantiate an object by invoking that function with the `new` operator.

A JavaScript object has properties associated with it. Object properties are basically the same as variables, except that they are associated with objects, not scopes. The properties of an object define the characteristics of the object.

For example, this example creates an object named `myCar`, with properties named `make`, `model`, and `year`, with their values set to "`Ford`", "`Mustang`", and `1969`:

```
const myCar = {
  make: "Ford",
  model: "Mustang",
  year: 1969,
};
```

Like JavaScript variables, property names are case-sensitive. Property names can only be strings or Symbols — all keys are converted to strings unless they are Symbols. Array indices are, in fact, properties with string keys that contain integers.

Accessing properties

You can access a property of an object by its property name. Property accessors come in two syntaxes: dot notation and bracket notation. For example, you could access the properties of the `myCar` object as follows:

```
// Dot notation
myCar.make = "Ford";
myCar.model = "Mustang";
myCar.year = 1969;
```

```
// Bracket notation
myCar["make"] = "Ford";
myCar["model"] = "Mustang";
myCar["year"] = 1969;
```

An object property name can be any JavaScript string or symbol, including an empty string. However, you cannot use dot notation to access a property whose name is not a valid JavaScript identifier. For example, a property name that has a space or a hyphen, that starts with a number, or that is held inside a variable can only be accessed using the bracket notation. This notation is also very useful when property names are to be dynamically determined, i.e. not determinable until runtime. Examples are as follows:

```
const myObj = {};
const str = "myString";
const rand = Math.random();
const anotherObj = {};

// Create additional properties on myObj
myObj.type = "Dot syntax for a key named type";
myObj["date created"] = "This key has a space";
myObj[str] = "This key is in variable str";
myObj[rand] = "A random number is the key here";
myObj[anotherObj] = "This key is object anotherObj";
myObj[""] = "This key is an empty string";

console.log(myObj);
// {
//   type: 'Dot syntax for a key named type',
//   'date created': 'This key has a space',
//   myString: 'This key is in variable str',
//   '0.6398914448618778': 'A random number is the key here',
//   '[object Object]': 'This key is object anotherObj',
//   '': 'This key is an empty string'
// }
console.log(myObj.myString); // 'This key is in variable str'
```

In the above code, the key `anotherObj` is an object, which is neither a string nor a symbol. When it is added to the `myObj`, JavaScript calls the `toString()` method of `anotherObj`, and use the resulting string as the new key.

You can also access properties with a string value stored in a variable. The variable must be passed in bracket notation. In the example above, the variable str held "`myString`" and it is "`myString`" which is the property name. Therefore, `myObj.str` will return as undefined.

```
str = "myString";
myObj[str] = "This key is in variable str";

console.log(myObj.str); // undefined

console.log(myObj[str]); // 'This key is in variable str'
console.log(myObj.myString); // 'This key is in variable str'
```

This allows accessing any property as determined at runtime:

```
let propertyName = "make";
myCar[propertyName] = "Ford";

// access different properties by changing the contents of the variable
propertyName = "model";
myCar[propertyName] = "Mustang";

console.log(myCar); // { make: 'Ford', model: 'Mustang' }
```

However, beware of using square brackets to access properties whose names are given by external input. This may make your code susceptible to object injection attacks.

Nonexistent properties of an object have value `undefined` (and not `null`).

```
myCar.nonexistentProperty; // undefined
```

Enumerating properties

There are three native ways to list/traverse object properties:

- `for...in` loops. This method traverses all of the enumerable string properties of an object as well as its prototype chain.
- `Object.keys(myObj).` This method returns an array with only the enumerable own string property names ("keys") in the object `myObj`, but not those in the prototype chain.
- `Object.getOwnPropertyNames(myObj).` This method returns an array containing all the own string property names in the object `myObj`, regardless of if they are enumerable or not.

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) {
  let result = "";
  for (const i in obj) {
    // Object.hasOwn() is used to exclude properties from the object's
    // prototype chain and only show "own properties"
    if (Object.hasOwn(obj, i)) {
      result += `${objName}.${i} = ${obj[i]}\n`;
    }
  }
  console.log(result);
}
```

So, the function call `showProps(myCar, 'myCar')` would print the following:

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

The above is equivalent to:

```
function showProps(obj, objName) {
  let result = "";
  Object.keys(obj).forEach((i) => {
    result += `${objName}.${i} = ${obj[i]}\n`;
  });
  console.log(result);
}
```

Deleting properties

You can remove a non-inherited property using the delete operator. The following code shows how to remove a property.

```

```

```
// Creates a new object, myobj, with two properties, a and b.
const myobj = new Object();
myobj.a = 5;
myobj.b = 12;

// Removes the a property, leaving myobj with only the b property.
delete myobj.a;
console.log("a" in myobj); // false
```

Inheritance

All objects in JavaScript inherit from at least one other object. The object being inherited from is known as the prototype, and the inherited properties can be found in the prototype object of the constructor. See Inheritance and the `prototype` chain for more information.

Defining properties for all objects of one type

You can add a property to all objects created through a certain constructor using the `prototype` property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object. The following code adds a `color` property to all objects of type `Car`, and then reads the property's value from an instance `car1`.

```
Car.prototype.color = "red";
console.log(car1.color); // "red"
```

Defining methods

A *method* is a function associated with an object, or, put differently, a **method is a property of an object that is a function**. Methods are defined the way normal functions are defined, except that they have to be assigned as the property of an object. See also method definitions for more details. An example is:

```
objectName.methodName = functionName;

const myObj = {
  myMethod: function (params) {
    // do something
  },
```

```
  // this works too!
  myOtherMethod(params) {
    // do something else
  },
};
```

where `objectName` is an existing object, `methodName` is the name you are assigning to the method, and `functionName` is the name of the function.

You can then call the method in the context of the object as follows:

```
object.methodName(params);
```

Methods are typically defined on the `prototype` object of the constructor, so that all objects of the same type share the same method. For example, you can define a function that formats and displays the properties of the previously-defined `Car` objects.

```
Car.prototype.displayCar = function () {
  const result = `A Beautiful ${this.year} ${this.make} ${this.model}`;
  console.log(result);
};
```

Notice the use of `this` to refer to the object to which the method belongs. Then you can call the `displayCar` method for each of the objects as follows:

```
car1.displayCar();
car2.displayCar();
```

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 has its own `name`, `age` and `job`. In the function `sayHi()`, notice the use of `this.name`. When added to the 2 objects, the same function will print the message with the name of the respective object it's attached to.

```
const Manager = {
  name: "Karina",
  age: 27,
  job: "Software Engineer",
};
const Intern = {
  name: "Tyrone",
  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 Karina
Intern.sayHi(); // Hello, my name is Tyrone
```

`this` is a "hidden parameter" of a function call that's passed in by specifying the object before the function that was called. For example, in `Manager.sayHi()`, `this` is the `Manager` object, because `Manager` comes before the function `sayHi()`. If you access the same function from another object, `this` will change as well. If you use other methods to call the function, like `Function.prototype.call()` or `Reflect.apply()`, you can explicitly pass the value of `this` as an argument.

Defining getters and setters

A **getter is a function associated with a property that gets the value of a specific property**. A **setter is a function associated with a property that sets the value of a specific property**. Together, they can indirectly represent the value of a property.

Getters and setters can be either

- defined within object initializers, or
- added later to any existing object.

Within object initializers, getters and setters are defined like regular methods, but prefixed with the keywords `get` or `set`. The getter method must not expect a parameter, while the setter method expects exactly one parameter (the new value to set). For instance:

```
const myObj = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set c(x) {
    this.a = x / 2;
  },
};

console.log(myObj.a); // 7
console.log(myObj.b); // 8, returned from the get b() method
myObj.c = 50; // Calls the set c(x) method
console.log(myObj.a); // 25
```

The `myObj` object's properties are:

- `myObj.a` — a number
- `myObj.b` — a getter that returns `myObj.a` plus 1
- `myObj.c` — a setter that sets the value of `myObj.a` to half of value `myObj.c` is being set to

Getters and setters can also be added to an object at any time after creation using the `Object.defineProperties()` method. This method's first parameter is the object on which you want to define the getter or setter. The second parameter is an object whose property names are the getter or setter names, and whose property values are objects for defining the getter or setter functions. Here's an example that defines the same getter and setter used in the previous example:

```
const myObj = { a: 0 };

Object.defineProperties(myObj, {
  b: {
    get() {
      return this.a + 1;
    },
  },
  c: {
    set(x) {
      this.a = x / 2;
```

```
    },
  },
});

myObj.c = 10; // Runs the setter, which assigns 10 / 2 (5) to the 'a' property
console.log(myObj.b); // Runs the getter, which yields a + 1 or 6
```

Which of the two forms to choose depends on your programming style and task at hand. If you can change the definition of the original object, you will probably define getters and setters through the original initializer. This form is more compact and natural. However, if you need to add getters and setters later — maybe because you did not write the particular object — then the second form is the only possible form. The second form better represents the dynamic nature of JavaScript, but it can make the code hard to read and understand.

Comparing objects

In JavaScript, objects are a reference type. Two distinct objects are never equal, even if they have the same properties. Only comparing the same object reference with itself yields true.

```
// Two variables, two distinct objects with the same properties

const fruit = { name: "apple" };
const fruitbear = { name: "apple" };

fruit == fruitbear; // return false
fruit === fruitbear; // return false
```

```
// Two variables, a single object

const fruit = { name: "apple" };
const fruitbear = fruit; // Assign fruit object reference to fruitbear

// Here fruit and fruitbear are pointing to same object

fruit == fruitbear; // return true
fruit === fruitbear; // return true

fruit.name = "grape";
console.log(fruitbear); // { name: "grape" }; not { name: "apple" }
```