web222

WEB222 - Week 4

Suggested Readings

- Object-oriented JavaScript for beginners
- SpeakingJS, Chapter 17. Objects and Inheritance
- SpeakingJS, Chapter 20. Dates
- SpeakingJS, Chapter 21. Math

Objects in JavaScript

So far we've been working with built-in `Objects` in JavaScript. We can also create our own in order to model complex data types in our programs. There are a number of ways to do this, and we'll look at a few of them now.

An `Object` in JavaScript is a *map* (also known as an *associative array* or a *dictionary*), which is a data structure composed of a collection of *key* and *value* pairs. We call an `Object`'s key/value pairs *properties*. Imagine a JavaScript `Object` as a dynamic "bag" of properties, a property-bag. Each *key* is a unique `String`, and an `Object` can only contain a given *key* once. An `Object` can have any number of *properties*, and they can be added and removed at runtime.

Much like we did with an `Array` or `RegExp`, we can create instances of `Objects` via literals. An `Object` literal always starts with `{` and ends with `}`. In between these curly braces we can optionally include a list of any properties (comma separated) we want to attach to this `Object` instance. These properties are written using a standard `key: value` style, with the property's name `String` coming first, followed by a `:`, then its value. The value can be any JavaScript value, including functions or other `Objects`.

Here are a few examples:

```
// an empty Object, with no properties
var o = {};

// a `person` Object, with one property, `name`
var person = { name: 'Tim Wu' };

// a `campus` Object, with `name` as well as co-ordinates (`lat`, `lng`)
var campus = {
    name: 'Seneca@York',
    lat: 43.7714,
    lng: -79.4988
};

// a `menu` Object, which contains lists of menu items per meal
var menu = {
    breakfast: ['eggs', 'toast', 'banana', 'coffee'],
    lunch: ['salad', 'chicken', 'apple', 'milk'],
    dinner: ['salmon', 'rice', 'green beans']
};
```

`Object` property names are `String`s, and we can refer to them either via the *dot operator* `.name`, or using the *bracket operator* `['name']` (similar to indexing in an `Array`):

```
var person = { name: 'Tim Wu' };

// get the value of the `name` property using the . operator
console.log(person.name);

// get the value of the `name` property using the [] operator
console.log(person['name']);
```

> Why would you choose the dot operator over the bracket operator, or vice versa? The dot operator is probably more commonly used; however, the bracket operator is useful in a number of scenarios. First, if you need to use a reserved JavaScript keyword for your property key, you'll need to refer to it as a string (e.g., `obj['for']`). Second, it's sometimes useful to be able to pass a variable in order to lookup a property value for a name that will be different at runtime. For example, if you are using usernames as keys, you might do `users[currentUsername]`, where `currentUsername` is a variable holding a `String` for the logged in user.

`Object` literals allow us to define an initial set of properties on an `Object`, but we aren't limited to that set. We can easily add new ones:

```
var data = {};

data.score = 17;
data.level = 3;
data.health = '***';
```

Here we define an empty `Object`, but then add new properties. Because we can add properties after an `Object` is created, we have to deal with a property not existing. If we try to access a property that does not exist on an `Object`, there won't be an error. Instead, we will get back `undefined`:

```
var currentScore = data.score;    // `score` exists on `data`, and we get back the value `17`
var inventory = data.inventory;   // `inventory` does not exist on `data`, so we get back `undefined`
```

Because properties may or may not exist at runtime, it's important to always check for a value before trying to use it. We could rewrite the above to first check if `data` has an `inventory` property:

```
if(data.inventory) {
    // `data` has a value for `inventory`, use data.inventory here...
} else {
    // there is no `inventory` on `data`, do something else...
}
```

Using Objects: dealing with optional parameters

A very common pattern in JavaScript programs that uses this concept is optional argument passing to functions. Instead of using an unknown number of `arguments` for a function, we often use an `options` `Object`, which may contain values to be used in the function. Consider the case of starting a game and sometimes passing existing user data:

```
function initGame(options) {
    // Make sure `options` exists, and use an empty `Object` instead if it's missing.
    // If we don't do this, we'll get an error if we try to do `options.score`, since
    // we can't lookup the `score` property on `undefined`.
    options = options || {};

    // If the user already has a score, use that, otherwise default to 0
    var score = options.score || 0;
    // If the user is already on a level, use that, otherwise default to 1
    var level = options.level || 1;
    // If the user has collected an items in her inventory, use that, otherwise an empty Array
    var inventory = options.inventory || [];

    // Begin the game, passing the values we have determined above
    playGame(score, level, inventory);
}

// Define our options: we have a score and level, but no inventory
var options = {
    score: 25,
    level: 2
};
initGame(options);
```

In the code above, we have an `options` `Object` that defines some, but not all of the properties our `initGame` function might use. We wrote `initGame` using a single argument so that it was easier to call: we didn't need to worry about the order or number of arguments, and could instead just define an `Object` with all of the properties we had. The `initGame` function examined the `options` at runtime to

see which properties existed, and which were `undefined` and needed a default value instead. Recall that we can use the *logical OR* (`||`) operator to choose between two values at runtime.

Updating, Clearing, and Removing properties

We've seen that properties can be defined when declared as part of a literal and added later via the `.` or `[]` operators. We can also update or remove values after they are created:

```
var o = {};

// Add a name property
o.name = 'Tim Wu';

// Update the name property to a new value, removing the old one.
o.name = 'Mr. Timothy Wu';
```

An `Object` 's property keys are unique, and setting a value for `o.name` more than once doesn't add more properties–it overwrites the value already stored in the existing property. We can also *clear* (remove the value but not the key) or *delete* (remove the entire property from the object, key and value) things from an `Object` .

```
var o = {};

// Add a `height` property
o.height = '35 inches';

// Add an owner ID property
o.owner = '012341341';

// Clear the value of `height`. We leave the `height` key, but get rid of the '35 inches' value
o.height = null;
```

```
// Completely remove the owner property from the object (both the key and its value)
delete o.owner;
```

> Why would you choose to assign `null` vs. use `delete`? Often we want to get rid of a key's value, but will use the key again in the future (e.g., add a new value). In such cases we just *null the value* by assigning the key a value of `null`. If we know that we'll never use this key again, and we don't want to retain it on the `Object`, we can instead completely remove the property (key and value) with `delete`. You'll see both used. For the most part, setting a key's value to `null` is probably what you want.

Using Objects: creating sets to track arbitrary lists

Another common use of `Object`s, and their unique property keys, is to keep track of a sets, for example to count or keep track of an unknown number of items. Consider the following program, which tracks how many times each character appears within a `String`. The code uses the `[]` operator to allow for the keys to be created and accessed via a variable (`char`). Without an `Object` we would have to hard-code variables for each separate letter.

```
// An empty `Object`, which we'll populate with keys (letters) and values (counts)
var characterCounts = {};

var str = "The quick brown fox jumped over the lazy dog.";
var char;
var count;

// Loop through str, visiting each character
for(var i = 0; i < str.length; i++) {
    char = str[i];
    // Get the current count for this character, or use 0 if we haven't seen it before
    count = characterCounts[char] || 0;
    // Increase the count by 1, and store it in our object
    characterCounts[char] = count + 1;
}

console.log(characterCounts);
/* Our characterCounts Object now looks like this, and there were 8 spaces, 2 'h's, etc:
```

```
{ T: 1,
  h: 2,
  e: 4,
  ' ': 8,
  q: 1,
  u: 2,
  i: 1,
  c: 1,
  k: 1,
  b: 1,
  r: 2,
  o: 4,
  w: 1,
  n: 1,
  f: 1,
  x: 1,
  j: 1,
  m: 1,
  p: 1,
  d: 2,
  v: 1,
  t: 1,
  l: 1,
  a: 1,
  z: 1,
  y: 1,
  g: 1,
  '.': 1 }
*/
```

Complex Property Types: `Object`, `Function`

We said earlier that `Object` properties can be any valid JavaScript type. That includes `Number`, `String`, `Boolean`, etc., also `Object` and `Function`. A property may define a complex `Object` of its own:

```
var part = {
    id: 5,
    info: {
        name: 'inner gasket',
        shelf: 56713,
        ref: [5618, 5693]
    }
};
```

Here we define a `part`, which has an id (`part.id`) as well as a complex property named `info`, which is itself an `Object`. We access properties deep in an `Object` the same way as a simple property, for example: `part.info.ref.length` means: get the `length` of the `ref` array on the `info` property of the `part` `Object`. An `Object`'s properties can be `Object`s many levels deep, and we use the `.` or `[]` operators to access these child properties.

An `Object` property can also be a function. We call these functions *methods*. A *method* has access to other properties on the `Object` via the `this` keyword, which refers to the current `Object` instance itself. Let's add a `toString()` method to our `part` `Object` above:

```
var part = {
    id: 5,
    info: {
        name: 'inner gasket',
        shelf: 56713,
        ref: [5618, 5693]
    },
    toString: function() {
        return this.info.name + ' (#' + this.id + ')';
    }
};

console.log(part.toString()); // prints "inner gasket (#5)" to the console.
```

The `toString` property is just like any other key we've added previously, except its value is an *anonymous function*. Just as we previously bound function expressions to variables, here a function expression is bound to an `Object` 's property. When we write `part.toString` we are accessing the function stored at this key, and by adding the `()` operator, we can invoke it: `part.toString()` says *get the function stored at part.toString and call it*. Our function accesses other properties on the `part` `Object` by using `this.*` instead of `part.*` . When the function is run, `this` will be the same as `part` (i.e., a reference to *this* `Object` instance).

> The `this` keyword in JavaScript is used in different contexts, and has a different meaning depending on where and how it is used. We will return to `this` and its various meanings throughout the course.

Constructor Functions

Sometimes we need to create lots of `Objects` that have the same layout. For example, we might be defining lots of users in an application. All of our user `Objects` need to work the same way so that we can pass them around within our program, to and from functions. Every `user` needs to have the same set of properties and methods, so we decide to write a factory function that can build our `user` `Objects` for us based on some data. We call such functions a `Constructor` :

```
// Define a Constructor function, `User`
function User(id, name) {
    // Attach the id to an Object referenced by `this`
    this.id = id;
    // Attach the name to an Object referenced by `this`
    this.name = name;
}

// Create a new instance of a User (Object)
var user1 = new User(1, 'Sam Smith');
// Create another new instance of a User (Object)
var user2 = new User(2, 'Joan Winston');
```

Notice that unlike all previous functions we've defined, the `User` function starts with a capital `U` instead of a lower case `u` . We use this naming convention to indicate that `User` is special: a constructor function. A constructor function needs to be called with the extra `new`

keyword in front of it. When we say `new User(...)` we are saying, *create a new object, and pass it along to User so it can attach various things to it*.

A constructor can also add methods to an object via `this`:

```
// Define a Constructor function, `User`
function User(id, name) {
    this.id = id;
    this.name = name;

    // Add a toString method
    this.toString = function () {
        return this.name + ' (#' + this.id + ')';
    };
}

// Create a new instance of a User (Object)
var user1 = new User(1, 'Sam Smith');
console.log(user1.toString()); // 'Sam Smith (#1)
```

In the code above, we're creating a new function every time we create a new User. As we start to create lots of users, we'll also be creating lots of duplicate functions. This will cause our program to use more and more resources (memory), which can lead to issues as the program scales.

Object Prototypes

What we would really like is a way to separate the parts of a User that are different for each user (the data: `id`, `name`), but somehow share the parts that are the same (the methods: `toString`). JavaScript gives us a way to accomplish this via an `Object`'s `prototype`.

JavaScript is unique among programming languages in the way it accomplishes sharing between `Object`s. All object-oriented languages provide some mechanism for us to share or inherit things like methods in a type hierarchy. For example, C++ and Java use classes, which

can inherit from one another to define methods on parents vs. children. JavaScript uses *prototypal inheritance* and a special property called `prototype`.

In JavaScript, we always talk about `Object`s, because every object is an instance of `Object`. Notice the capital `O` in `Object`, which should give you an indication of what it is: a constructor function. In a previous week we said that an `Array` is an `Object`, and a `RegExp` is an `Object`. This is true because of JavaScript's type system, where almost everything is *chained* to `Object`.

JavaScript objects always have a prototype, which is an object to which their `.proptotype` property refers. At runtime, when we refer to an object's property, JavaScript first looks for that property on the object itself. If it doesn't find it, the prototype object is visited, and the same search is done. The process continues until the end of the prototype chain is reached at `Object`.

Let's rewrite our `User` so that the `toString` method is moved from each user instance to the prototype of all user instances:

```
// Define a Constructor function, `User`
function User(id, name) {
    this.id = id;
    this.name = name;
}

User.prototype.toString = function () {
    return this.name + ' (#' + this.id + ')';
};
```

This code looks very similar to what we originally wrote. Notice that we've moved `toString` out of the `User` function, and instead attached it to `User.prototype`. By doing so, we'll only ever need a single copy of this function: every `new User()` instance we create will also include a reference to a prototype object, which contains our function. When we use `user1.toString()`, JavaScript will do something like this:

1. does `user1` have a property called `toString`? No, we didn't add one in the constructor.
2. does `user1.prototype` have a property called `toString`? Yes, use that.

What if we'd written `user1.something()`?

1. does `user1` have a property called `something`? No, we didn't add one in the constructor.
2. does `user1.prototype` have a property called `something`? No.
3. does `user1.prototype.prototype` (i.e., `Object`) have a property called `something`? No.
4. there are no more objects in the prototype chain, throw an error

```
user1.something();
// TypeError: user1.something is not a function
```

Whenever a method is used on a prototype, we still pass the current instance so we can get access to its data. Notice in our `User.prototype.toString` method, we still referred to `this`, which will be the instance of our user, and give us access to the correct data (`name`, `id`).

> There are times when defining a method inside a constructor makes sense vs. putting it on the prototype. The prototype will only have access to *public properties* of an object instance, meaning things you explicitly add to `this` and expose to the rest of your program. Sometimes we want to define some data, but *hide* it from the rest of a program, so it can't be changed after it gets created. Consider the following example, which uses a *closure* to retain access to a variable in the scope of the constructor without exposing it:

```
function User(id, name) {
    this.id = id;
    this.name = name;

    // private variable within User function, not attached to `this`.
    // Normally this variable would go out of scope after User() completed;
    // however, we will use a closure function below to capture this scope.
    var createdAt = Date.now();

    // Return the number of ms this player has been playing
    this.playerAgeMS = function() {
        var currentTime = Date.now();

        // Access `createdAt` in the parent scope, which we retain via this closure function.
        // Calculate how many ms between createdAt and the current time.
```

```
        return (currentTime - createdAt) + " ms";
    };
}

var user = new User(1, 'Tom');
// We can access the total time this player has existed, but not modify it.
console.log(user.playerAgeMS())
// displays "4183 ms"
console.log(user.playerAgeMS())
// displays "5287 ms"
```

Object.create()

We can also define an object's prototype when we create it by using the `Object.create()` method. With `Object.create()`, we can pass an Object to use as the `prototype` for our new object. Consider the case where we want to define different characters in a game, some of which are human, and some not.

```
var person = {
    isHuman: true,
    power: 5
};

var animal = {
    isHuman: false,
    power: 3
};

var robot = {
    isHuman: false,
    power: 10
}

// Create a list of characters for the game as an Array, each of a different type
var characters = [
```

```
    Object.create(person),
    Object.create(person),
    Object.create(robot),
    Object.create(animal)
];

function calculateHitDamage(character) {
    // Double human player's hit power
    if(character.isHuman) {
        return character.power * 2;
    } else {
        return character.power;
    }
}
```

> JavaScript version note: in newer versions of JavaScript, a new `class` keyword has been implemented, allowing for a somewhat more familiar style of Object and Class definitions to be done. Underneath, prototype inheritance is still used, but this adds some new syntax.

Practice Problem: a Morse Code translator

Morse code is a system of encoding developed in the 1800s that allowed transmission of textual messages over signal systems that only supported on/off (1 and 0) notations.

Complete the program below as specified. Your program should be able to translate messages like `-- --- .-. ... ./-.-. --- -.. .` into `MORSE CODE` and vice versa. Use what you learned above about `Object`s, and also some of the built-in `Object`s we've studied, in particular `RegExp` and `String`.

Use the following limited set of morse code to use in this exercise. You could expand your program to handle more complex messages later if you want:

Letter	Morse

Letter	Morse
A	.-
B	-...
C	-.-.
D	-..
E	.
F	..-.
G	--.
H	....
I	..
J	.---
K	-.-
L	.-..
M	--
N	-.
O	---
P	.--.
Q	--.-
R	.-.

Letter	Morse
S	`...`
T	`-`
U	`..-`
V	`...-`
W	`.--`
X	`-..-`
Y	`-.--`
Z	`--..`
space	`/`

NOTE: letters are separated by a single space (`' '`) within a word, and words are separated with a `/`. For example, the words `MORSE CODE` would translate to `-- --- .-. ... ./-.-. --- -.. .`

```
// Object to provide lookup of morse code (value) for a given letter (key).
var alpha = {
    // define the mapping here as a literal
};

// Object to provide lookup of letter (value) for a given morse code (key).
var morse = {};
// Hint: use the [] operator to specify these special key values rather than a literal.

// Return `true` if all characters are morse code.  Use a RegExp.
function isMorse(characters) {

}
```

```
// Return `true` if all characters are part of the alphabet defined in `alpha`.  Use a RegExp.
// Bonus: can you rewrite it using `Object.keys()` and your `alpha` Object instead?
// https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys
function isAlpha(characters) {

}

// Given an alphabet message, convert and return in morse code.  Use your morse and/or alpha object.
// Return undefined if text is not alpha.
function textToMorse(text) {

}

// Given a morse code message, convert and return in text.  Use your morse and/or alpha object.
// Return undefined if morse is not proper code.
function morseToText(morse) {

}


// Constructor function that takes a `message` (String), which can be either morse or alpha.
// Use your functions above to decide how to store a value for `any` on `this`
function Message(any) {

}

// Return the message as morse code, converting if necessary
Message.prototype.toMorse = function() {

};

// Return the message as alpha characters, converting if necessary
Message.prototype.toAlpha = function() {

};
```

```
var msg1 = new Message('--- -... .--- . -.-. - .../.. -./.--- .- ...- .- ... -.-. .-. .. .--. -/.- .-. ./...- . .-. -.--/.--. ---
console.log(msg1.toAlpha());
console.log(msg1.toMorse());

var msg2 = new Message('I am learning how to use Objects in JavaScript');
console.log(msg2.toMorse());
console.log(msg2.toAlpha());
```

You can download the code above as well as a possible solution.

This site is open source. Improve this page.