Data structures: Objects and arrays

Saturday, March 30, 2024

4:28 PM

**Data sets**

JavaScript provides a data type specifically for storing sequences of values. It is called an ***array*** and is written as a list of values between square brackets, separated by commas.

*let listOfNumbers = [2, 3, 5, 7, 11];*

*console.log(listOfNumbers[2]);*

*// → 5*

*console.log(listOfNumbers[0]);*

*// → 2*

*console.log(listOfNumbers[2 - 1]);*

*// → 3*

**Properties**

Almost all JavaScript values have properties. The exceptions are **null** and **undefined**. If you try to access a property on one of these nonvalues, you get an error.

The two main ways to access properties in JavaScript are with a dot and with square brackets. Both value.x and value[x] access a property on value—but not necessarily the same property. The difference is in how x is interpreted. When using a dot, the word after the dot is the literal name of the property. When using square brackets, the expression between the brackets is evaluated to get the property name. Whereas value.x fetches the property of value named “x”, value[x] tries to evaluate the expression x and uses the result, converted to a string, as the property name.

**Methods**

Both string and array values contain, in addition to the length property, a number of properties that hold function values.

Properties that contain functions are generally called ***methods*** of the value they belong to.

**Objects**

Values of the type ***object*** are arbitrary collections of properties. One way to create an object is by using braces as an expression.

*let day1 = {*

*squirrel: false,*

*events: ["work", "touched tree", "pizza", "running"]*

*};*

*console.log(day1.squirrel);*

*// → false*

*console.log(day1.wolf);*

*// → undefined*

*day1.wolf = false;*

*console.log(day1.wolf);*

*// → false*

Inside the braces, there is a list of properties separated by commas. Each property has a name followed by a colon and a value. When an object is written over multiple lines, indenting it like in the example helps with readability. Properties whose names aren’t valid binding names or valid numbers have to be quoted.

This means that braces have two meanings in JavaScript:

* At the start of a statement, they start a block of statements.
* In any other position, they describe an object.

Reading a property that doesn’t exist will give you the value ***undefined***.

To briefly return to our tentacle model of bindings—property bindings are similar. They grasp values, but other bindings and properties might be holding onto those same values. You may think of objects as octopuses with any number of tentacles, each of which has a name tattooed on it.

The **delete** operator cuts off a tentacle from such an octopus. It is a unary operator that, when applied to an object property, will remove the named property from the object.

*let anObject = {left: 1, right: 2};*

*console.log(anObject.left);*

*// → 1*

*delete anObject.left;*

*console.log(anObject.left);*

*// → undefined*

*console.log("left" in anObject);*

*// → false*

*console.log("right" in anObject);*

*// → true*

The binary **in** operator, when applied to a string and an object, tells you whether that object has a property with that name.

To find out what properties an object has, you can use the **Object.keys** function. You give it an object, and it returns an array of strings—the object’s property names.

*console.log(Object.keys({x: 0, y: 0, z: 2}));*

*// → ["x", "y", "z"]*

There’s an **Object.assign** function that copies all properties from one object into another.

*let objectA = {a: 1, b: 2};*

*Object.assign(objectA, {b: 3, c: 4});*

*console.log(objectA);*

*// → {a: 1, b: 3, c: 4}*

Arrays, then, are just a kind of object specialized for storing sequences of things. If you evaluate typeof [], it produces "object".

**Mutability**

We saw that object values can be modified. The types of values discussed in earlier chapters, such as numbers, strings, and Booleans, are all ***immutable***—it is impossible to change values of those types. You can combine them and derive new values from them, but when you take a specific string value, that value will always remain the same.

Objects work differently. You can change their properties, causing a single object value to have different content at different times.

When you compare objects with JavaScript’s == operator, it compares by **identity**: it will produce true only if both objects are precisely the same value. Comparing different objects will return false, even if they have identical properties.

**Array loops**

A loop like this:

*for (let i = 0; i < JOURNAL.length; i++) {*

*let entry = JOURNAL[i];*

*// Do something with entry*

*}*

is common in Javascript. There is a simpler way to write such loops in modern JavaScript.

*for (let entry of JOURNAL) {*

*console.log(`${entry.events.length} events.`);*

*}*

When a for loop looks like this, with the word **of** after a variable definition, it will loop over the elements of the value given after of. This works not only for arrays but also for strings and some other data structures.

**Further arrayology**

We saw **push** and **pop**, which add and remove elements at the end of an array. The corresponding methods for adding and removing things at the start of an array are called **unshift** and **shift.**

*let todoList = [];*

*function remember(task) {*

*todoList.push(task);*

*}*

*function getTask() {*

*return todoList.shift();*

*}*

*function rememberUrgently(task) {*

*todoList.unshift(task);*

*}*

That program manages a queue of tasks. You add tasks to the end of the queue by calling remember("groceries"), and when you’re ready to do something, you call getTask() to get (and remove) the front item from the queue. The rememberUrgently function also adds a task but adds it to the front instead of the back of the queue.

To search for a specific value, arrays provide an **indexOf** method. The method searches through the array from the start to the end and returns the index at which the requested value was found—or -1 if it wasn’t found.

*console.log([1, 2, 3, 2, 1].indexOf(2));*

*// → 1*

Another fundamental array method is **slice,** which takes start and end indices and returns an array that has only the elements between them. The start index is inclusive, the end index exclusive.

*console.log([0, 1, 2, 3, 4].slice(2, 4));*

*// → [2, 3]*

The **concat** method can be used to glue arrays together to create a new array, similar to what the + operator does for strings.

**Strings and their properties**

Values of type string, number, and Boolean are not objects, such values are immutable and cannot be changed.

But these types do have built-in properties. Every string value has a number of methods. Some very useful ones are **slice** and **indexOf**, which resemble the array methods of the same name. The trim method removes whitespace. The **padStart** method takes the desired length and padding character as arguments to pad the string.

You can split a string on every occurrence of another string with **split** and join it again with **join**.

**Rest parameters**

It can be useful for a function to accept any number of arguments.

To write such a function, you put three dots before the function’s last parameter, like this:

*function max(...numbers) {*

*let result = -Infinity;*

*for (let number of numbers) {*

*if (number > result) result = number; }*

*return result;*

*}*

*console.log(max(4, 1, 9, -2));*

*// → 9*

When such a function is called, the rest parameter is bound to an array containing all further arguments. If there are other parameters before it, their values aren’t part of that array.

You can use a similar three-dot notation to call a function with an array of arguments.

*let numbers = [5, 1, 7];*

*console.log(max(...numbers));*

*// → 7*

**The Math object**

-

**Destructuring**

Look inside an object: If you know the value you are binding is an array, you can use square brackets to “look inside” of the value, binding its contents.

*let a, b, rest;*

*[a, b] = [10, 20];*

*console.log(a);*

*// Expected output: 10*

*console.log(b);*

*// Expected output: 20*

*[a, b, ...rest] = [10, 20, 30, 40, 50];*

*console.log(rest);*

*// Expected output: Array [30, 40, 50]*

A similar trick works for objects, using braces instead of square brackets.

*let {name} = {name: "Faraji", age: 23};*

*console.log(name);*

*// → Faraji*

**JSON**

***Serializing the data means converting them into a flat description.***

A popular serialization format is called **JSON** (pronounced “Jason”), which stands for JavaScript Object Notation. It is widely used as a data storage and communication format on the Web, even in languages other than JavaScript.

JSON looks similar to JavaScript’s way of writing arrays and objects, with a few restrictions. ***All property names have to be surrounded by double quotes,*** and only simple data expressions are allowed—no function calls, bindings, or anything that involves actual computation. Comments are not allowed in JSON.

A journal entry might look like this when represented as JSON data:

{

"squirrel": false,

"events": ["work", "touched tree", "pizza", "running"]

}

JavaScript gives us the functions **JSON.stringify** and **JSON.parse** to convert data to and from this format. The first takes a JavaScript value and returns a JSON-encoded string. The second takes such a string and converts it to the value it encodes.