JavaScript Objects At their core, objects are just mappings from strings to other values. The values can be anything: strings, functions, other objects, etc. The string that maps to the value is called the key.

const object = { "num": 1, "str": "Hello World", "obj": { "x": 5 } }; There are three ways to access values on an object:

Dot Notation. const val = object.obj.x; console.log(val); // 5 Bracket Notation. This is used when the key isn't valid variable name. For example ".123". const val = object["obj"]["x"]; console.log(val); // 5 Destructuring Syntax. This is most useful when accessing multiple values at once. You can read more about the syntax here. const { num, str} = object; console.log(num, str); // 1 "Hello World" You can read more about objects here.

Classes and Prototypes You can also define classes in JavaScript. The classes's constructor returns an object which is an instance of that class.

class Person { constructor(name, age) { this.name = name; this.age = age; }

greet() { console.log("My name is", this.name); } }

const alice = new Person("Alice", 25); alice.greet(); // Logs: "My name is Alice" JavaScript implements classes with special objects call prototypes. All the methods (in this case greet) are functions stored on the object's prototype.

To make this concrete, the behavior of the above code could be replicated with the following code:

const alice = { name: "Alice", age: 25, **proto**: { greet: function() { console.log("My name is", this.name); } } }; alice.greet(); // Logs: "My name is Alice" Looking at this code, you might wonder "How can you access the greet method even though it's not a key on the alice object"?

The reason is that accessing keys on an object is actually slightly more complicated than just looking at the object's keys. There is actually an algorithm that traverse the prototype chain. First, JavaScript looks at the keys on the object. If the requested key wasn't found, it then looks on the keys of the prototype object. If it still wasn't found, it looks at the prototype's prototype, and so on. This is how inheritance is implemented in JavaScript!

You might also wonder why JavaScript has this strange prototype concept at all. Why not just store the functions on the object itself? The answer is efficiency. Every time a new Person is created, age and name fields are added to the object. However only a single reference to the prototype object is added. So no matter how many instances of Person are created or how many methods are on the class, only a single prototype object is generated.

You can read more about classes here.

Proxies An infrequently used but powerful feature of javascript is the proxy. They allow you to override the default behavior of objects.

For example, to implement the alice object with proxies:

const alice = new Proxy({name: "Alice", age: 25}, { get: (target, key) => { if (key === 'greet') { return () => console.log("My name is", target.name); } else { return target[key]; } }, }); alice.greet(); // Logs: "My name is Alice" Examples Here are some examples of potentially practical use-cases for proxies.

Perform validation to guarantee bad data is never entered into a form. Perform validation to guarantee bad data is never entered into a form. Code Example const validator = { set: (obj, prop, value) => { if (prop === "age") { if (typeof value !== "number" || value < 0) { throw new TypeError("Age must be a positive number"); } } obj[prop] = value; }, };

const person = new Proxy({}, validator); person.age = 25; // Works fine person.age = -5; // Throws an error

Create a log any time a key is accessed. This could be useful as a developer tool. Code Example const object = { "num": 1, "str": "Hello World", "obj": { "x": 5 } }; const proxiedObject = new Proxy(object, { get: (target, key) => { console.log("Accessing", key); return target[key]; } }); proxiedObject.num; // Logs: Accessing num

Throw an error if a an attempt was made to write to a readonly value. Code Example const READONLY_KEYS = ['name'];

const person = new Proxy({ name: "Alice", age: 25 }, { set: (target, key, value) => { if (READONLY_KEYS.includes(key)) { throw Error("Cannot write to key"); } target[key] = value; return true; } }); person.name = "Bob"; // Throws Error

Create a modified version of an immutable object by writing to it's proxy. This is implemented with the popular library immer. You can read more about proxies here. immer destructing obj obj js