The secret life of objects

Tuesday, April 16, 2024

6:18 AM

**Encapsulation**

The core idea in object-oriented programming is to divide programs into smaller pieces and make each piece responsible for managing its own state.

This way, some knowledge about the way a piece of the program works can be kept local to that piece.

Different pieces of such a program interact with each other through interfaces, limited sets of functions or bindings that provide useful functionality at a more abstract level, hiding their precise implementation.

Such program pieces are modeled using objects. Their interface consists of a specific set of methods and properties. ***Properties that are part of the interface are called public***. The others, ***which outside code should not be touching, are called private***.

Yes, JavaScript provides a way to distinguish between public and private properties in a class. 
This is done using a hash prefix to denote a private property. Here's an example: 
class myC1ass { 
constructor() { 
this. publicProperty 
this. #privateProperty 
'I am public'; 
'I am private'; 
checkPrivateProperty() { 
return this.#privateproperty; 
const instance 
MyC1ass(); 
new 
console. log(instance. publicProperty); // Outputs: 
'I am public' 
console. log(instance. privateproperty); // Outputs: undefined 
console. log(instance. checkPrivateProperty()); // Outputs : 
'I am privat 
I vulnerability v 
Embedding credentials in source risks unauthorized access 
In this example, publicproperty' is a public property, which means it can be accessed directly 
from an instance of the class. #privateproperty is a private property, which means it can only 
be accessed within the class that defines it. Attempting to access it directly from an instance will 
result in undefined. However, it can be accessed through a method within the class, as 
demonstrated by the checkPrivateProperty method. 

Separating interface from implementation is a great idea. It is usually called **encapsulation**.

**Methods**

Methods are nothing more than properties that hold function values.

This is a simple method:

*let rabbit = {};*

*rabbit.speak = function(line) {*

*console.log(`The rabbit says '${line}'`);*

*};*

*rabbit.speak("I'm alive.");*

*// → The rabbit says 'I'm alive.'*

Usually a method needs to do something with the object it was called on.

When a function is called as a method as in ***object.method()*** the binding called **thi**s in its body automatically points at the object that it was called on.

*function speak(line) {*

*console.log(`The ${this.type} rabbit says '${line}'`);*

*}*

*let whiteRabbit = {type: "white", speak};*

*let hungryRabbit = {type: "hungry", speak};*

*whiteRabbit.speak("Oh my ears and whiskers, " + "how late it's getting!");*

*// → The white rabbit says 'Oh my ears and whiskers, how*

*// late it's getting!'*

*hungryRabbit.speak("I could use a carrot right now.");*

*// → The hungry rabbit says 'I could use a carrot right now.'*

You can think of **this** as an extra parameter that is passed in a different way. If you want to pass it explicitly, you can use a **function's call method**, which takes the **this** value as its first argument and treat further arguments as normal parameters.

*speak.call(hyngryRabbit, "Burp!");*

// → The hungry rabbit says 'Burp!'

**Prototypes**

In addition to their set of properties, most objects also have a ***prototype***.

A prototype is an object that is used as a fallback source of properties. When an object gets a request for a property that it does not have, its prototype will be searched for the property, then the prototype's prototype, and so on.

The prototype of an empty object is **Object.prototype**.

**Object.getPrototypeOf** returns the prototype of an object.

The prototype relations of JavaScript objects for a tree-shaped structure, and at the root sits **Object.prototype**. It provides a few methods that show up in all objects, such as **toString**, which converts an object to a string representation.

Many objects do not have **Object.prototype** as their prototype but instead have another object that provides a different set of default properties.

Functions derive from **Function.prototype**, and arrays derive from **Array .prototype.**

You can use **Object.create** to create an object with a specific prototype.

*let protoRabbit = {*

*speak(line) {*

*console.log(`The ${this.type} rabbit says '${line}'`);*

*}*

*};*

*let killerRabbit = Object.create(protoRabbit);*

*killerRabbit.type = "killer";*

*killerRabbit.speak("SKREEEE!");*

*// → The killer rabbit says 'SKREEEE!'*

A property like speak(line) in an object expression is a shorthand way of defining a method. It creates a property called speak and gives it a function as its value.

The “proto” rabbit acts as a container for the properties that are shared by all rabbits. An individual rabbit object, like the killer rabbit, contains properties that apply only to itself—in this case its type—and derives shared properties from its prototype.

**Classes**

JavaScript’s prototype system can be interpreted as a somewhat informal take on an object-oriented concept called classes. A class defines the shape of a type of object—what methods and properties it has. Such an object is called an instance of the class.

Prototypes are useful for defining properties for which all instances of a class share the same value, such as methods. Properties that differ per instance, such as our rabbits’ type property, need to be stored directly in the objects themselves.

So to create an instance of a given class, you have to make an object that derives from the proper prototype, but you also have to make sure it, itself, has the properties that instances of this class are supposed to have. This is what a **constructor** function does.

*function makeRabbit(type) {*

*let rabbit = Object.create(protoRabbit);*

*rabbit.type = type;*

*return rabbit;*

*}*