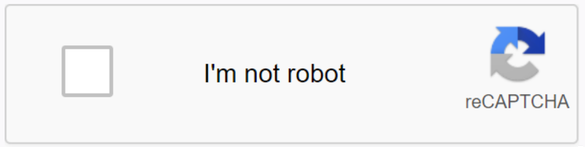

Continue

Default constructor javascript

The following is a guest post by Faraz Kekhini. Some of these things are out of my comfort zone, so I asked Kyle Simpson to check it out for me. Kyle's response (which we did during an office hours session) was very interesting. It was: 1) This article is technically sound. JavaScript doesn't really have tutorials in the traditional sense and that's the way most people shoehorn them in. 2) We may want to stop shoehorning them in. JavaScript has objects and we can use them the way they are meant to do the same kinds of things. Kyle calls it OLOO (Objects associated with other objects). Here's an introduction. I think there's value in finding out about both. Having a good understanding of manufacturers is vital to truly understand the JavaScript language. Technically, JavaScript does not, but has manufacturers and prototypes to bring similar functions to JavaScript. In fact, the class statement introduced in ES2015 merely acts as an editorial sugar above the existing original-based heritage and doesn't really add any additional functionality to the language. In this tutorial, we will explore the manufacturers in detail and see how JavaScript uses them to make objects. Creating and using Manufacturers is like regular functions, but we use them with the new keyword. There are two types of manufacturers: built-in constructions such as Array and Object, which are automatically available in the run-time environment. and custom constructions that define properties and methods for your type of object. A constructor is useful when you want to create many similar objects with the same properties and methods. It is a contract to capitalize the name of manufacturers to distinguish them from normal functions. Consider the following code: Book() function { // incomplete code } var myBook = new book(); The last line of the code creates an instance of the Book and assigns it to a variable. Although the book maker does nothing, myBook is still a case of it. As you can see, there is no difference between this function and normal operations, except that it is called with the new keyword and the function name is uppercase. Determine the type of an instance To find out if one object is an instance of another, we use the presence of the operator: myBookof Book instance // true myBook instanceof String // false Note that if the right side of the operator is not a function, it will drop an error: myBookof instance {}; TypeError: non operand 'instanceof' ({}) Another way to find the type of an instance is to use the build property. Consider the following code section: myBook.constructor ===Book? A book? true The manufacturer property of myBook shows in the Book, so that the strict equal operator returns true. Each object in JavaScript inherits a build property from its original, points to the constructor function that created the object: var s = new string(text); s.constructor ===String; true text.constructor === String; true var o = new object(); o.constructor ===Object; true var o = {}; o.constructor ===Object; true var a = new table();; a.constructor === Array; true [].constructor === Array; True Note, however, that using the build property to control the type of an instance is generally considered bad practice because it can be replaced. Custom Manufacturing Functions A manufacturer is like a cookie cutter for constructing multiple objects with the same properties and methods. Consider the following example: Book function(name, year) { this.name = name; this.year = '" + year + ')'; } The book manufacturer expects two parameters: name and year. When the manufacturer is called with the new keyword, assigns the parameters it received to the name and year property of the current instance, as shown below: var firstBook = new book (Pro CornerJS, 2014). var secondBook = new book (Secrets of JavaScript Ninja, 2013) var thirdBook = new book (JavaScript Patterns, 2010); console.log(firstBook.name, firstBook.year). console.log(secondBook.name, secondBook.year). console.log(thirdBook.name, thirdBook.year). This code records the following in the console: As you can see, we can quickly build a large number of different book objects by invoking book making with different arguments. This is exactly the same pattern that JavaScript uses on its built-in manufacturers, such as Array() and Date(). The Object.dezineProperty() method The Object.defenseProperty() method can be used within a constructor to help you perform all the necessary property settings. Consider the following constructor: Book(name) { Object.defenseProperty(this, name, { get: function() { return Book: + name; }, set: function(newName) { name = newName; }, configurable: false }); } var myBook = new book (One-page Web Applications); console.log (myBook.name). Book: Single-page Web applications // cannot delete the name property because the configurable value is set to false delete myBook.name. console.log (myBook.name). Book: Single Page Web Applications // but we can change the value of the name property myBook.name = Testable JavaScript; console.log (myBook.name). Book: JavaScript Test This code uses Object.defineProperty() to determine accessor properties. Accessor properties do not include properties or methods, but define an option to call when reading the property and a setter to call when writing the property. A table return a value while a setter receives the value assigned to the property as an argument. The above constructor returns an instance whose name property can be set or changed, but cannot be deleted. When we get the value of the name, the getter prepares the series book: to name and returns it. Eminent object notes are preferred by manufacturers JavaScript has nine built-in structures: Object(), Array(), String(), Number(), Boolean(), Date(), Function(), Error() and RegExp(). When creating prices, we are free to use either literal objects or manufacturers. However, literal object values are not only easier to read, but also faster to run, because they can be optimized when analyzing time. So for simple objects it is best to stick with literally: // an object number / / numbers have toFixed() method var obj = new object(5) ? obj.toFixed(2)· 5.00 // we can achieve the same result using literally var num = 5; num.toFixed(2)· 5.00 // a string object // strings have a slice() var obj method = new string(text). obj.slice(0,2)· te // same as the above string var = text. string.slice(0,2). te As you can see, there is almost no difference between literal objects and manufacturers. What is more intersting is that it is still possible to call methods for literally. When a method is called a literal, JavaScript automatically converts the literal to a temporary object so that the method can perform the operation. Once the temporary object is no longer needed, JavaScript rejects it. Using the new keyword is essential It is important to remember to use the new keyword before all manufacturers. If you accidentally forget news, you will modify the global object instead of the newly created object. Consider the following example: Book(name, year) { console.log(this); this.name = name; this.year = year; } var myBook = Book(js book, 2014). console.log (in the presence of myBookof Book); console.log(window.name, window.year). var myBook = new book (js book, 2014) ? console.log (in the presence of myBookof Book); console.log (myBook.name, myBook.year) Here's what this code records on the console: When we call the book maker without a new one, we actually call a function without a return statement. As a result, this within the constructor leads to the window (instead of myBook) and two global variables are created. However, when we call the function with new, the context changes from global (Window) to instance. So this shows correctly in myBook. Note that in strict mode this code will throw an error because the strict function is designed to protect the developer from accidentally calling a manufacturer without the new keyword. Field-safe manufacturers As we've seen, a manufacturer is just a so that it can be called without the new keyword. But, for inexperienced developers, this can be a source of bugs. A manufacturer with range security is designed to return the same result, regardless of whether it is called with or without a new one, so it does not suffer from these issues. Most built-in constructions, such as Object, Regex, and Array, are safe for scope. They use a special pattern to determine how the manufacturer If no new one is used, they return a correct instance of the object by calling the constructor again with a new one. Consider the following code: function Fn(argument) { // if this is not an instance of the manufacturer // does this mean that it was called without a new one if (!)(this instanceof Fn)) { // call the constructor again with a new return of a new Fn(argument)? } So a field-safe version of our manufacturer will look like this: Book mode (name, year) { if (!)(this instanceof Book)) { return new Book(name, year); } this.name = name? this.year = year; } var person1 = new book (js book, 2014); var person2 = Book(js book, 2014); console.log(person1 instances of the book) true console.log (person2 instances of the book); true Conclusion It is important to understand that the class statement introduced in ES2015 simply acts as an editorial sugar over the existing original-based legacy and adds nothing new to JavaScript. Manufacturers and originals are JavaScript's main way of defining similar and related objects. , we have taken a good look at how JavaScript manufacturers work. We learned that manufacturers are like normal functions, but they are used with the new keyword. We saw how manufacturers allow us to quickly make many similar objects with the same properties and methods, and why the presence of the operator is the safest way to determine the type of an instance. Finally, we looked at manufacturers safely, who can be called with or without new ones. Nwe.