|  |
| --- |
| [[https://myetudes.org/etudes-melete-tool/images/printer.png](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385317) Send to Printer](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385317) | [Close Window](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385317) |
|  | |
| 6. Properties  6.1. Configurable Properties in ECMAScript 5  *Copyrighted Material - subject to fair use exception*  In EcmaScript 5, properties have attributes that specify whether they can be written, enumerated, or configured.  These attributes may be set or reset with the **Object.defineProperty()** method.  When we create a property through assignment, as in myCar.color = 'blue', the property created is always writable (may be changed), enumerable (it shows up during property enumeration), and configurable (it may be deleted).  The Object.defineProperty() method gives us control over these extra details.  We can use it to create a new property or to change the attributes of an existing property.  Consider the following example:  Object.defineProperty(myCar, 'year',  { value : 2009, writable: false, enumerable: true, configurable: false});  Here we are defining a new 'frozen' property on the object myCar.  The value of year is set to 2009.  However that property is NOT **writable**, which means that we cannot change it with an assignment statement.  It is **enumerable**, which means it shows up during enumeration of the properties on the corresponding object.  It is NOT **configurable**, which means its attributes cannot be changed with another call to defineProperty, and it cannot be deleted.  Let’s see what all of that means in our example:  Object.defineProperty (myCar,  'year',  { value : 2009,  writable: false,  enumerable: true,  configurable: false  }       );  console.log(myCar);  // year is enumerable so it will show up with a value of 2009  Object { model="civic", year=2009, make="Honda", more...}  >>> myCar.hasOwnProperty('year');  true  Now let’s try to change the value of year:  >>> myCar.year = 2010;  The outcome will depend on whether or not we are using strict mode.  In strict mode, we get an exception: "year" is read-only.  **In non-strict mode, no error is generated but the assignment is ignored:**  >>> myCar.year;  2009  Let’s try to change the property to make it writable:  Object.defineProperty(myCar, 'year',  { value : 2009, **writable: true**, enumerable: true, configurable: false})  TypeError: can't redefine non-configurable property 'year'  That is not possible, because the property is not configurable.  When we introduced Object.create() we mentioned that it takes 2 arguments.  The first argument is the prototype for the new object and the optional second argument describes the properties of the new object.  So we could have also created the year and model properties together with the object as follows:  var myCar = Object.create(hondaCar, {  // hondaCar is the prototype                                          "year": { // year is a new property                                                    value : 2009,                                                    writable: false,                                                    enumerable: true,                                                    configurable: false                                                   },                                          "model": { // model is a new property                                                      value : "civic",                                                      writable: true,                                                      enumerable: true,                                                      configurable: true                                                    }                                        }                            );  Unless otherwise specified, in the context of this course, we’ll assume the default case where properties are writable, enumerable and configurable.  6.2. Deleting Properties  *Copyrighted Material - subject to fair use exception*  To remove a configurable property from an object, we can use the delete operator.  Delete will return true if the property is deleted successfully.  >>> delete myCar.model;  true  However it also returns true if the property does not exist:  >>> delete myCar.noSuchProperty;  true  The delete operator only deletes direct, own properties, not inherited ones.  However it will also return true if you attempt to delete an inherited property without actually deleting it:  >>> delete myCar.make  true  >>> myCar.make  "Honda"  >>> myCar  Object { year=2009, make="Honda", mileage=0, more...}  To delete an inherited property, we’ll need to delete it from the prototype object in which it is defined. It will then disappear from every object that inherits from that prototype.  >>>delete hondaCar.make  true  >>> myCar.make  undefined  In strict mode, attempting to delete a nonconfigurable property results in an exception. In non-strict mode, delete simply returns false in this case.  Note that in order to use strict mode, you need to run your code from Scratchpad.  The console does not support strict mode.  Remember that we had defined the year property to be non configurable.  Let’s see what happen if we try to delete it:  In strict mode, we get an exception:  delete myCar.year  Exception: property "year" is non-configurable and can't be deleted  In non-strict mode, delete just returns false in and the property is NOT deleted.  delete myCar.year  false  myCar.year  2009  6.3. Testing Properties  *Copyrighted Material - subject to fair use exception*  Sometimes it is useful to determine if an object has a given property before attempting to access it.  The in operator may be used for that.  It returns true if the object has a direct or inherited property with a given name.  >>> "mileage" in myCar;  // mileage is an inherited property of myCar  true  >>> "model" in hondaCar  // model is not a property of hondaCar  false  >>> "model" in myCar  // model is a direct own property of myCar  true  We have also already seen how to use the hasOwnProperty() method to distinguish between direct (own) and inherited properties.  The propertyIsEnumerable() is even more specific. It returns true only if the named property is a direct property and it is enumerable.  6.4. Iterating Over Properties  *Copyrighted Material - subject to fair use exception*  The for...in loop allows us to iterate **over the enumerable properties** of an object.  The general syntax is as follows:  for (property in object) {              do something with the property  }  The loop goes over the properties in **arbitrary order**.  Note that **the loop will iterate over all enumerable properties of the object itself as well as those the object inherits from its prototype.**  Let’s go back to our car example where myCar has two inherited properties, mileage and make.  We give myCar  three own properties: model, year and vin.  We make the vin (vehicle identification number) non-enumerable, so that it does not appear when we iterate over the properties.  Inside the loop, we build a string that will contain a description of the car.  It will include all the properties (own and inherited) except the vin, since the vin property is non-enumerable.  var car = {mileage: 0};  var hondaCar = Object.create(car);  // hondaCar inherits mileage from car  hondaCar.make = 'Honda';  var myCar = Object.create(hondaCar);  // myCar inherits make and mileage  myCar.model = 'civic';  Object.defineProperty(myCar, 'year',  { value : 2009, writable: false, enumerable: true, configurable: false});  Object.defineProperty(myCar, 'vin', { value: 123456789, writable: false, enumerable: false, configurable: false});  var carDescription = "";  for (**var prop in myCar)** {  // add the string property name: property value followed by a new line  carDescription += prop + ": " + **myCar[prop**] + "\n";  }  console.log(carDescription);  model: civic year: 2009 make: Honda mileage: 0  Note that at each iteration of the loop, the property name is given by the variable prop.  To access the value corresponding to the given property, we need to use the square bracket notation car[prop] (without quotes) as we have seen in module 5.3.  We cannot use car.prop because then we would be looking for a property named 'prop', which is undefined.  Keep in mind that **methods defined on an object are properties too**.  They will be iterated over unless we set them to be non-enumerable.   In our car example, let's add a drive method to the car prototype to see what happens, we'll leave everything else unchanged:  var car = {mileage: 0};  var hondaCar = Object.create(car);  // hondaCar inherits mileage from car  hondaCar.make = 'Honda';  var myCar = Object.create(hondaCar);  // myCar inherits make and mileage  myCar.model = 'civic';  Object.defineProperty(myCar, 'year',  { value : 2009, writable: false, enumerable: true, configurable: false});  Object.defineProperty(myCar, 'vin', { value: 123456789, writable: false, enumerable: false, configurable: false});  **car.drive = function (distance ) {   // define a drive method for the car object**  **this.mileage = this.mileage + distance;**  **return this;**  **};**  Let's iterate over the properties of myCar now to build the description string:  var carDescription = "";  for (**var prop in myCar)** {  // add the string property name: property value followed by a new line  carDescription += prop + ": " + **myCar[prop**] + "\n";  }  console.log(carDescription);  Here's the output.  The drive function is included in the description.  model: civic  year: 2009  make: Honda  mileage: 0  **drive: function (distance ) { // define a drive method for the car object**  **this.mileage = this.mileage + distance;**  **return this;**  **}**  So how do we avoid this?  One way is to check the type of myCar[prop] before adding it to  the description.  So we can write:  if (typeof myCar[prop] !== "function") {      carDescription += prop + ": " + myCar[prop] + "\n";  }  Now let's take this a step further.  Let's write a method, describe, that will be inherited by all cars and that will return a string describing the (enumerable, non method) properties of any car.  We define the method describe on the car prototype so that it can be inherited by any car.  However we have to be careful now to use 'this' instead of myCar or car to refer to the object.  var car = {mileage: 0};  car.drive = function (distance ) {   // define a drive method for the car object      this.mileage = this.mileage + distance;      return this;  };    var hondaCar = Object.create(car);  // hondaCar inherits mileage from car    hondaCar.make = 'Honda';    var myCar = Object.create(hondaCar);  // myCar inherits make and mileage    myCar.model = 'civic';    Object.defineProperty(myCar, 'year',  { value : 2009, writable: false, enumerable: true, configurable: false});    Object.defineProperty(myCar, 'vin', { value: 123456789, writable: false, enumerable: false, configurable: false});    car.describe = function () {      var description = "";      for (var prop in **this**) {          if (typeof **this[prop]**!== "function"){              description += prop + ": " + **this[prop]** + "\n";          }      }      return description;  };  And now we can invoke the method describe on any car object:  console.log(car.describe());  mileage: 0    console.log(hondaCar.describe());  make: Honda mileage: 0    console.log(myCar.describe());  model: civic year: 2009 make: Honda mileage: 0 |  |