|  |
| --- |
| [[https://myetudes.org/etudes-melete-tool/images/printer.png](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385316) Send to Printer](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385316) | [Close Window](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385316) |
|  | |
| 5. Objects  5.1. What is an object?  *Copyrighted Material - subject to fair use exception*  Almost everything in JavaScript is an Object:  Functions, Arrays, even primitive datatypes such as strings can be treated as objects.  We can think of an object as an **unordered collection of properties**, each of which has a name and a value.  We have already seen the length property of a string:  >>> var greeting = 'Hello';  >>> greeting.length  5  >>> greeting['length']  // another way to access the length property  5  There are 2  syntaxes for accessing the property of an object:  objectName.propertyName  or  objectName['propertyName']  Objects also have a certain behavior associated with them.  This behavior is described by the methods that can be invoked on the objects.  We have already seen how to invoke the toLowerCase() method on a string:  >>> var greeting = 'Hello';  >>> var newGreeting = greeting.toLowerCase();  >>> newGreeting  "hello"  The general syntax for invoking a method on an object is:  objectName.methodName()  5.2. Creating JavaScript Objects  *Copyrighted Material - subject to fair use exception*  We can define and create our own objects.  There are 3 ways to do that.  We can simply use **an object literal**.  **An object literal is a list of colon-separated name:value pairs, enclosed within curly braces.**  For the Python programmers out there, it looks like a dictionary.  Example:  >>> var hisCar = {make: 'Honda', model: 'Civic'};  // object literal  A property name is a valid JavaScript identifier or a string literal (the empty string is allowed).  make and model are property names.  A property value is any JavaScript expression.  The strings 'Honda' and 'Civic' are property values.  The following examples are all valid objects:  >>> var emptyObject = {};  This is just an empty object with no properties.  >>> var herAccount = {'account holder': 'Alice', balance: 300};  The property name ‘account holder’ includes a space, so it is not a valid JavaScript identifier but it still can be a property name:  we just enclose it in quotes.  String literals are valid property names.  Once we have created the object, we can access it and access its properties:  >>> console.log(herAccount);  Object { account holder="Alice", balance=300}  >>> console.log(herAccount.balance);  300  >>> console.log(herAccount['account holder']);  Alice  The account holder property includes a space character so we can’t access it with the dot notation but we can always access it with the square bracket notation.  Objects may also be nested in other objects.  >>> var person = {idNo: 8989000, account: herAccount};  The value of the account property of person is another object.  >>> console.log(person);  Object { idNo=8989000, account={...}}  >>> console.log(person.account);  Object { account holder="Alice", balance=300}  5.3. Dot vs Square Bracket Notation  *Copyrighted Material - subject to fair use exception*  We have seen that there are two syntaxes for accessing the property of an object:  objectName.propertyName  or  objectName['propertyName']  When we use the dot notation, objectName.propertyName, propertyName has to be a valid identifier.  The square bracket notation, objectName['propertyName'],  allows us to use non valid identifiers (such as strings that include space) as a property name.  The square bracket notation also **allows us to use a** **computed property name.**  **The value within the brackets is an expression that evaluates to a string that is the property name.** This is very useful when we need to loop through the properties of an object.  Let’s make sense of this with an example:  Consider the following object describing the population of various cities:  population = { 'San Francisco': 825863, 'San Jose': 982765, 'Santa Clara': 119311, Saratoga: 30677}  To access the population of Saratoga, we can write:  population.Saratoga or population['Saratoga'] or even  population['Sara' + 'toga']  In the last case the value within the brackets ('Sara' + 'toga') is evaluated to a string first and that string is used as the property name.  Because San Francisco includes a space character, we cannot use the dot notation to access its population, but we can still write:  population['San Francisco' ].  But what if the city’s name is saved in a variable, city, and we need to access that city’s population?  If we write population.city,  it is as if we are looking for a property whose name is 'city'.  The object population does not have a property city.  population.city is undefined.  The variable city does not get replaced by its value here.  However if we write population[city] (**without quotes around city)**, the interpreter will **evaluate**city first and then  will look for**a property whose name is the value of city** (which is 'San Jose' in the example below.)  function getPopulation(city) {      // return the population of a given city      var population = {          'San Francisco': 825863,          'San Jose': 982765,          'Santa Clara': 119311,          Saratoga: 30677      }      return population[city];  }  var city = 'San Jose';  console.log(city, ': ',  getPopulation(city));  San Jose: 982765  5.4. Methods  *Copyrighted Material - subject to fair use exception*  Methods are used to describe the behavior of an object. They operate on the object or perform object-specific computations.  We say that a method is invoked on a particular object.  **In JavaScript, methods are functions that are attached to objects through a property.**  Consider the following example:  function depositAmount(amount) {      // we’ll fill in the body of the function a little later  }  var account = {name: 'Alice', balance: 0, **deposit: depositAmount**};  console.log(account);  Object { name="Alice", balance=0, deposit=depositAmount()}  The deposit property of the object account is a function that has been defined earlier and named depositAmount.  It takes one parameter, amount.  **The syntax for invoking a method is different from the syntax for calling a function.**  The general syntax is:  objectName.methodName(arguments);  To invoke the deposit method on the object account, we write:  account.deposit(20);  **In general, when a method is invoked on an object, it has access to the object through the keyword this.**  We say that the object becomes its **invocation context**.  Note that **this** is passed implicitly to the method:  it does not have to be listed as a parameter.  Now we can go back and fill in the body of our method.  We need to update the balance property of the object to reflect the new amount deposited.  We have access to that property through **this.balance**.  So we can write:  function depositAmount(amount) {  **this.balance = this.balance + amount**; // update the balance      return this;  // return the updated object  }  Putting it all together, we can now write the following (try it in Scratchpad):  function depositAmount(amount) {      this.balance = this.balance + amount; // update the balance      return this;  // return the updated object  }  var account = {name: 'Alice', balance: 0, **deposit: depositAmount**};  console.log(account);  // balance is 0  account.deposit(20);  console.log(account); // balance is 0 + 20  account.deposit(50);  console.log(account); // balance is 20 + 50  The following should be displayed on the Firebug console:  Object { name="Alice", **balance=0,** deposit=depositAmount()}  Object { name="Alice", **balance=20**, deposit=depositAmount()}  Object { name="Alice", **balance=70**, deposit=depositAmount()}    In the example above, the function was defined and given a name first, then attached to the object. **We could have also defined the method directly on the object as follows** **using a function expression**:  var account = {      name: 'Alice',      balance: 0,      deposit: **function (amount) {**  **this.balance = this.balance + amount;**  **return this;**  **}**  };  Note that deposit is just a property of account, so we can also invoke it using the square bracket notation:  account["deposit"](50)  **Chaining:**  The above method returns the updated object. When we write methods that return updated objects, we can chain method invocations so that the object is operated on by several methods sequentially.  For instance, in our example, we could invoke the deposit method multiple times on account as follows:  account.deposit(20).deposit(100);  5.5. Prototype  *Copyrighted Material - subject to fair use exception*  Every object is linked to **a prototype object from which it can inherit properties**.  When we create a new object from object literals ({property:value, …}), we don’t have the option to specify the prototype.  All objects created from object literals have the same prototype, Object.prototype, an object that comes standard with JavaScript.  It is denoted by Object {}.  We can think of it as the ultimate prototype of all objects, sitting at the top of the inheritance hierarchy.  All JavaScript objects eventually inherit from it.  In ECMAScript 5, we can get the prototype of any object by using Object.getPrototypeOf().  >>> var herAccount = {'account holder': 'Alice', balance: 300};  >>> console.log(**Object.getPrototypeOf**(herAccount));  Object { }  >>> var person = {idNo: 8989000, account:herAccount};  >>> console.log(Object.getPrototypeOf(person));  Object { }  Object.prototype is one of the rare objects that has no prototype: it does not inherit any properties.  >>> console.log(Object.getPrototypeOf(Object.prototype));  null  We’ll see that there are two other ways to create objects and these two ways allow us to specify the prototype.  5.6. Creating Objects with Object.create()  *Copyrighted Material - subject to fair use exception*  In ECMAScript 5, we can use the method Object.create() to  create a new object.  We specify the prototype for the new object as the first argument to the method.  **This is the recommended way to create objects in JavaScript.**  Object.create() also takes an optional second argument that describes the properties of the new object. We’ll talk about the second argument in a later section.  Here are some examples on how to use Object.create():  >>> var car = {mileage: 0};  The variable car is created using an object literal.  Its prototype is automatically set to Object.prototype (denoted by Object {} below).  This is equivalent to calling Object.create as follows:  var car = Object.create(Object.prototype);  and then adding the mileage property:  car.mileage = 0;  >>> console.log(car);  Object { mileage=0}  >>> console.log(Object.getPrototypeOf(car));  Object { }  Once we have created this generic car object with a mileage property, we can go on to create more specific car objects that will inherit from car.  To create a hondaCar object that will inherit from car, we specify car as the prototype as follows:  >>> var hondaCar = Object.create(**car**);  The variable hondaCar is created using the Object.create() method.  Its prototype is **explicitly set** to the object referred to by car (denoted by Object{mileage=0} below).  >>> console.log(Object.getPrototypeOf(hondaCar));  Object { mileage=0}  The object referred to by hondaCar has **inherited**the mileage property from its prototype car.  >>> console.log(hondaCar);  Object { mileage=0}  We can add a new property to hondaCar.  **To add a new property, we just assign a value to it.**  >>> hondaCar.make = 'Honda';  >>> console.log(hondaCar);  Object { make="Honda", mileage=0}  This does NOT change the prototype car.  **The property is only added to the object itself but not to its prototype.**  >>> console.log(car);  Object { mileage=0}  We can create a new object myCar with hondaCar as its prototype:  >>> var myCar = Object.create(**hondaCar**);  >>> console.log(Object.getPrototypeOf(myCar));  Object { make="Honda", mileage=0}  myCar inherits ALL the properties of hondaCar, including those that hondaCar itself inherited from car:  >>> console.log(myCar);  Object { make="Honda", mileage=0}  We can add properties to myCar without affecting its prototype.  >>> myCar.model = 'civic';  >>> console.log(myCar);  Object { model="civic", make="Honda", mileage=0}  >>> console.log(hondaCar);  Object { make="Honda", mileage=0}  It is important to note that the **prototype relationship here is a dynamic relationship.**  **We can add, remove or modify properties dynamically to the prototype thus affecting an entire set of objects.**  To illustrate that, let’s add a drive method to the car object.  Remember that a method is a function assigned to the property of an object.  We can define the drive method using a function expression as follows:  **car.drive** = function (distance ) {   // define a drive method for the car object      this.mileage = this.mileage + distance;      return this;  }  We saw that when a method is invoked on an object, the method has access to that object (and its properties) through **this**.  It is essential to use **this** here (and not car) so that when drive is invoked on a object that inherits from car (such as myCar), the mileage of that object is updated and not the mileage of the prototype car.  **The drive method is now** visible in all of the objects that are prototyped after the car object  – even those that were created prior to this addition.  We can write:  myCar.drive(20);  console.log(myCar);  Object { model="civic", **mileage=20,**make="Honda", more...}  One last thing to note here is that changing the mileage property of myCar does not affect the mileage of its prototype.  >>> console.log(hondaCar);  Object { make="Honda", **mileage=0**}  >>> console.log(car);  Object { **mileage=0**}  5.7. Creating Objects with Constructors  *Copyrighted Material - subject to fair use exception*  We can also create and initialize an object with the **new**keyword. The **new** keyword must be followed by a function invocation. A function used in this way is called a **constructor.**  JavaScript includes some built-in constructors: Object, Date and Array are some of the built-in constructors.  The following example creates a new instance of an object using the Object constructor.  >>> var herAccount = new Object();  Once the object is created, we can add properties to it:  >>> herAccount.name = "Alice";  >>> herAccount.balance = 300;  >>> console.log(herAccount);  Object { name="Alice", balance=300}  Because herAccount was created with the Object constructor, its prototype is Object.prototype:  >>> console.log(Object.getPrototypeOf(herAccount));  Object { }  We can also define our own constructor functions to initialize newly created objects.  The following example defines a **constructor function** Car to create an object and initialize it with the given property, make:  function Car(make) {  **this**.make= make;  }  **this** refers to the new object being created.  It is equivalent to the self parameter in Python.  **It is automatically passed to the constructor.  It does not have to be listed as a parameter.**  The convention is to **start the names of constructors with a capital letter**. This makes it easy to distinguish them from other functions.  We can also define properties and methods for the **prototype**:  these properties and methods will be inherited by all **the objects created with this constructor.  These properties and methods have to be defined under the constructor property ‘prototype’:**  Car.**prototype**.mileage = 0;  // define a mileage property for the prototype  Now all objects created with the Car constructor will inherit the mileage property from Car.prototype.  It is important here to emphasize the following:   1. Car here is the constructor function NOT the prototype object. 2. The prototype for all objects created with the Car constructor is Car.prototype.  Remember Object.prototype?  There is also Array.prototype that is the prototype of all objects created with the Array constructor.   We can also define methods for the prototype.  Here we define a function (with a function expression) and assign it to the drive  property of Car.prototype. **All objects created with the Car constructor will then inherit the drive method from Car.prototype.**  Car.prototype.drive = function (distance ) {   // define a drive method for the prototype      this.mileage = this.mileage + distance;  }  this.mileage is the mileage corresponding to the object that the method is invoked on.  Once we have defined the Car constructor, we can create new objects with the new keyword as follows:  var myCar = **new** Car(**"Honda"**);  "Honda" will be passed to the Car constructor function as the make parameter and will be set as the value of the make property for the object denoted by 'this' which in this case is myCar:  function Car(**make**) {      this.make= **make**;  }  console.log(myCar);  Car { make=**"Honda",** mileage=0, drive=function()}  console.log(myCar.make);  Honda  var yourCar = **new**Car(**"Porsche"**);  Here "Porsche" will be passed to the Car constructor function as the make parameter and will be set as the value of the make property for the object denoted by 'this' which in this case is yourCar:  console.log(yourCar);  Car { make=**"Porsche"**, mileage=0, drive=function()}  console.log(yourCar.make);  Porsche  console.log(Object.getPrototypeOf(myCar));  Car { mileage=0, drive=function()}  console.log(Object.getPrototypeOf(yourCar));  Car { mileage=0, drive=function()}  Note that myCar and yourCar are 2 different objects.  They have different values for their make property.  They have the same prototype and they both inherited mileage and drive from it.  myCar.drive(10);  console.log(myCar.mileage);  10  myCar.drive(10);  console.log(myCar.mileage);  20  We can invoke the drive method on myCar and as a result its mileage property is updated.  The mileage property of yourCar is left unchanged.  console.log(yourCar.mileage);  0  5.8. Properties and Inheritance  *Copyrighted Material - subject to fair use exception*  We have seen that we can access the properties of an object using the dot notation as well as the square bracket notation:  account.balance  account['balance']  With inheritance, things are a little more complicated as to which properties we are really accessing.  Each object has its own direct properties and it also inherits properties from the prototype object.  We can use the **hasOwnProperty()** method on any object to distinguish between direct (own) and inherited properties.  Consider the following example:  >>> var car = {mileage: 0};   // create car using an object literal >>> var hondaCar = Object.create(car);  // car is the prototype of hondaCar >>> hondaCar.make = 'Honda';  // set the make property of hondaCar to 'Honda' >>> var myCar = Object.create(hondaCar);  // hondaCar is the prototype of myCar >>> myCar.model = 'civic';  // set the model property of myCar to 'civic'  >>> myCar.hasOwnProperty('mileage')   //mileage is **not a direct property**of myCar  false   >>> hondaCar.hasOwnProperty('mileage') //mileage is **not a direct property** of hondaCar  false   >>> car.hasOwnProperty('mileage') //mileage is a direct property of car  true  >>> myCar.mileage   // the inherited mileage property is accessible through myCar  0  So even though mileage is a property of car and not a direct property of myCar, I can still access it and get its (inherited) value through myCar.  In general, if the object does not have a direct property, JavaScript looks for the property in the prototype.  If the prototype object does not have a direct property by that name either but has a prototype itself, then JavaScript looks for that property in the prototype of the prototype. This continues until the property is found or until an object with a null prototype is searched.  >>> myCar.color  undefined  In this case, none of the objects in the prototype chain has a color property, so undefined is returned.  Now what happens if we assign a value to myCar.mileage?  >>> myCar.mileage = 10000;  // create a new property on myCar  10000  This assignment actually creates a new property on myCar.  >>> car.mileage  0  >>> hondaCar.mileage  0  The mileage property of the prototype(s) is unchanged.  >>> myCar.hasOwnProperty('mileage');  true  Since myCar does not have a direct property mileage, the assignment myCar.mileage = 10000 creates a new property mileage on myCar. **The mileage property inherited from car is now hidden by the newly created direct property with the same name.  This mechanism allows us to selectively override inherited properties.**    5.9. Prototype vs Class  *Copyrighted Material - subject to fair use exception*  **JavaScript is an object-based language based on prototypes, rather than class.**  All objects are instances and you construct an object hierarchy by assigning another object as its prototype.  Instead of inheriting properties by following the class chain,**JavaScript objects inherit properties by following the prototype chain.**  The major advantage lies in the following distinction.  The class relationship is a static relationship.  A class definition specifies all properties of all instances of a class at once.  In general, we cannot add properties dynamically at run time.  The prototype relationship is a dynamic relationship. **The prototype specifies an initial set of properties. We can add or remove properties dynamically to individual objects or to the prototype thus affecting an entire set of objects.**When we add a new property to a prototype, that property is immediately visible in all of the objects that are based on that prototype – even those that were created prior to this addition. |  |