1- Understanding object

What is Object?

Before we talk about Object, maybe we should talk a lil bit about ECMA-262, Because we will maybe use this word many times in this presentation.

This is a **standard** defines the ECMAScript scripting language, and for someone who is still have no idea what is ECMAScript :

It is the scripting language standardized by Ecma International in the ECMA-262 and for more information u can visit these sources for more details :

<http://en.wikipedia.org/wiki/ECMAScript>

<http://www.ecma-international.org/memento/index.html>

ECMA-262 defines an object as an “unordered collection of properties each of which contains a primitive value, object, or function.”. Each property or method is identified by a name that is mapped to a value.

Actually, It’s nothing more than a grouping of name-value pairs where the value may be data or a function.

Like this example :

var person = {

name: "Hans",

age: 26,

job: "Super Junior Developer",

sayName: function(){

console.log(this.name);

}

};

But why does this work? I mean, what actually happened underground when we add a name-value pair ( a property ) to an object and the result is the object has the property with name and value inside?

When a property is first added to an object, JavaScript uses an internal method called [[Put]] on the object.

The [[Put]] method creates a spot in the object to store the property. The result of calling [[Put]] is the creation of an *own property* (remember this)on the object.

But there is a difference if that is the first time or not. When a new value is assigned to an existing property, a separate operation called [[Set]] takes place. This operation replaces the current value of the property with the new one.

Where do [[Put]] method and [[Set]] operation come from ? May be in Javascript Engine or in Universe. Try to figure it out after this presentation ( if u r so fucking curious … and I love it ).

The **person** variable doesn’t contain the object actually, it contains a point, and this point points to the actual object. So when a variable is assignee an object like this :

var object1, object2;

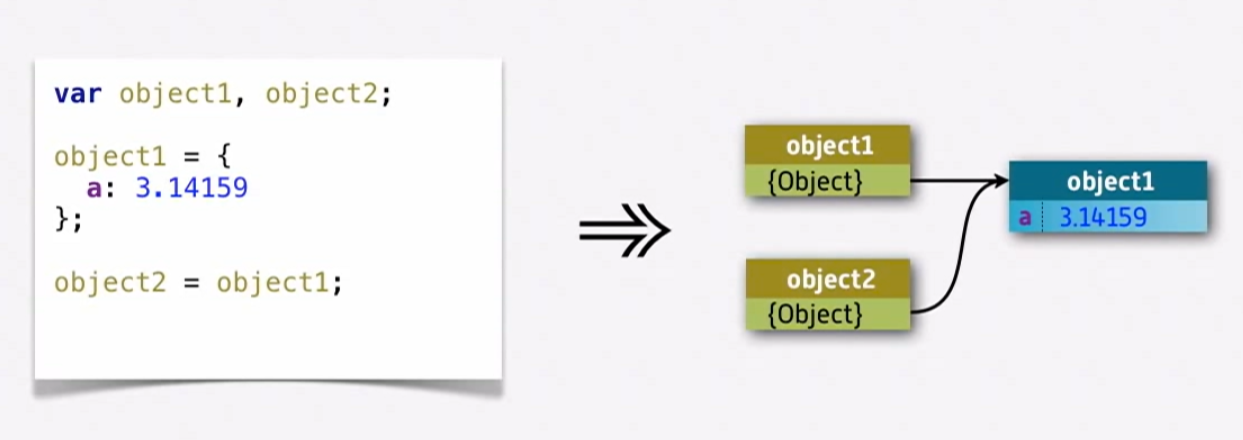
object1 = {

a: 3.14159

}

object2 = object1;

The real picture is :



Each variable contains a point which points to the same object because of this line of code:

object2 = object1;

This one means the variables share a point to point to an object.

So remember when we interact with an object, it doesn’t mean that we are interacting it directly.



“Control is an illusion”

But remember Javascript Engine creates this illusion in this case, and we are out of The Matrix, that Matrix at least, we are Neo and Trinity, luckily.

By the way, I think we should clear confused definitions like reference, instance. What is instance ? What is reference ? Like I said before, I used point, but I should use reference instead.

Here :

- A variable will hold a reference to an instance of an object.

- The actual object is an instance.

So let’s get back to the example :

Now we say it again : both object1 and object2 hold references to the same object with data : { a: 3.14159 }

Now we go deep inside the properties :

There are two types of properties: Data properties and Accessor properties.

**Data properties :**

* Data properties contain a single location for a data value. Values are read from and written to this location.
* Data properties have four attributes describing their behavior:
* [[Configurable]] : Indicates if the property may be redefined by removing the property via delete, changing the property’s attributes, or changing the property into an accessor property. By default, this is true for all properties defined directly on an object, as in the previous example.
* [[Enumerable]] : Indicates if the property will be returned in a for-in loop. By default, this is true for all properties defined directly on an object, as in the previous example.
* [[Writable]] — Indicates if the property’s value can be changed. By default, this is true for all properties defined directly on an object, as in the previous example.
* [[Value]] — Contains the actual data value for the property. This is the location from which the property’s value is read and the location to which new values are saved. The default value for this attribute is undefined.

When a property is added to an object as in the previous examples, [[Configurable]], [[Enumerable]], and [[Writable]] are all set to true while the [[Value]] attribute is set to the assigned value. So in the previous example:

* object1 = {
* a: 3.14159
* }

The property called **a** is created and a value of **3.14159** is assigned. That means  [[Value]] is set to **3.14159**, and any changes to that value are stored in this location.

But how about the other internal attributes? Because you are Neo so you can change them though a method of ECMAScript 5: Object.defineProperty().

This method accepts three arguments: the object on which  the property should be added or modified, the name of the property, and a descriptor object… like this :

var person1 = {};

Object.defineProperty(person1, "name", {

value: "Neo",

enumerable: true,

configurable: true,

writable: true

});

That’s it, but there is something you should know that setting configurable to false means that the property cannot be removed from the object. Calling delete on the property has no effect in nonstrict mode and throws an error in strict mode. Additionally, once a property has been defined as nonconfigurable, it cannot become configurable again. Any attempt to call Object.defineProperty() and change any attribute other than writable causes an error:

var person = {};

Object.defineProperty(person, “name”, {

configurable: false,

value: “Neo”

});

Object.defineProperty(person, “name”, {

configurable: true, // 🡨 error here

value: “Trinity”

}); // this will throw an error

That means you can call Object.defineProperty() multiple times for the same property, but there are limits once configurable has been set to false.

Note : Internet Explorer 8 was the first version to implement Object .defineProperty(). Unfortunately, the implementation is extremely limited. This method can be used only on DOM objects and can create only accessor properties.

**Accessor Properties :**

- Accessor Properties contain a combination of a getter function and a setter function (though both are not necessary). When an accessor property is read from, the getter function is called, and it’s the function’s responsibility to return a valid value; when an accessor property is written to, a function is called with the new value, and that function must decide how to react to the data.

- Accessor properties have four attributes:

* [[Configurable]] — Indicates if the property may be redefined by removing the property via delete, changing the property’s attributes, or changing the property into a data property. By default, this is true for all properties defined directly on an object.
* [[Enumerable]] — Indicates if the property will be returned in a for-in loop. By default, this is true for all properties defined directly on an object.
* [[Get]] — The function to call when the property is read from. The default value is undefined.
* [[Set]] — The function to call when the property is written to. The default value is undefined.

It is not possible to define an accessor property explicitly so you must use Object.defineProperty().

var book = {

\_year: 2004,

edition: 1

};

Object.defineProperty(book, “year”,

{

get: function(){

return this.\_year;

},

set: function(newValue){

if (newValue > 2004) {

this.\_year = newValue; this.edition += newValue - 2004;

}

}

});

book.year = 2005;

alert(book.edition); //2

So there are some javascript frameworks like EmberJS and BackboneJS both use accessors (getters and setters) on wrapped objects for listening events when the object have changes, but other ones use dirty-checking, but we should discuss this in another time cause I dun wanna go so far away from our topic, OOP.

Source:

<http://blog.bguiz.com/post/57373805814/accessors-vs-dirty-checking-in-javascript-frameworks/>

What if we wanna define multiple properties ? ECMAScript 5 still has another function for this need : Object.defineProperties().

There are two arguments: the object on which to add or modify the properties and an object whose property names correspond to the properties’ names to add or modify. For example:

var book = {};

Object.defineProperties(book,

{

\_year: { value: 2004 },

edition: { value: 1},

year: {

get: function(){

return this.\_year;

},

set: function(newValue){

if (newValue > 2004) {

this.\_year = newValue;

this.edition += newValue - 2004;

}

}

}

});

It’s not necessary to assign both a getter and a setter. Assigning just a getter means that the property cannot be written to and attempts to do so will be ignored.

It’s also possible to retrieve the property descriptor for a given property by using the ECMAScript 5 Object.getOwnPropertyDescriptor() method. This method accepts two arguments: the object on which the property resides and the name of the property whose descriptor should be retrieved. The return value is an object with properties for configurable, enumerable, get, and set for accessor properties or configurable, enumerable, writable, and value for data properties.

Note : The Object.getOwnPropertyDescriptor() method can be used on any object in JavaScript, including DOM and BOM objects. This method is supported in Internet Explorer 9+, Firefox 4+, Safari 5+, Opera 12+, and Chrome.

Let’s get the property descriptor in above example :

var descriptor = Object.getOwnPropertyDescriptor(book, “\_year”);

alert(descriptor.value); //2004

alert(descriptor.configurable); //false

alert(typeof descriptor.get); //”undefined”

var descriptor = Object.getOwnPropertyDescriptor(book, “year”);

alert(descriptor.value); //undefined

alert(descriptor.enumerable); //false

alert(typeof descriptor.get); //”function”

Why enumerable and configurable properties are false, why aren’t they true like the information above ? Remember this line : “this is true for all properties defined directly on an object.” In the nearest example, we didn’t define it directly on an object, we defined it through Object.defineProperties method, and that’s why they are false. Just console it out :

console.log(descriptor);

Object {get: function, set: function, enumerable: false, configurable: false}

See ? except get and set are functions, enumerable and configurable are false.

2- OBJECT CREATION

There are so many patterns for creating object, but we will just discuss only popular patterns.

Patterns :

1. The Factory Pattern : The factory pattern is a well-known design pattern used in software engineering to abstract away the process of creating specific objects

function Person(name, age, job) {

return {

name = name,

age = age,

job = job,

run: function() {

alert(name + " is running!");

}

}

}

var person1 = Person("Hans")

person1.run();

Here, the function Person() accepts arguments with which to build an object with a method run and three properties. The function can be called any number of times with different arguments and will still return an object that has three properties and one method. But like what you see, we don’t know what is person1 really, the issue of object identification (what type of object an object is).

person1 instanceof Object; //true

person1 instanceof Person; //false

Oh my god, I’m not a person ... but I can run. What am I really ?

1. The Constructor Pattern : constructors in ECMAScript are used to create specific types of objects. There are native constructors, such as Object and Array, which are available automatically in the execution environment at runtime. It is also possible to define custom constructors that define properties and methods for your own type of object.

function Person(name, age, job){

this.name = name;

this.age = age;

this.job = job;

this.sayName = function(){

alert(this.name);

};

}

var person1 = new Person(“Le Roi”, 27, “Super Star Singer”);

var person2 = new Person(“Le Hai Thinh”, 26, “Psychology Doctor”);

The differences between the function Person above and the function Person in previous example are :

* There is no object being created explicitly.
* The properties and method are assigned directly onto the this object.
* There is no return statement.

By convention, constructor functions always begin with an uppercase letter, whereas non-constructor functions begin with a lowercase letter. This convention is borrowed from other OO languages and helps to distinguish function use in ECMAScript, since constructors are simply functions that create objects.

What happened when we use the operator **new**:

1. Create a new object.

2. Assign the **this** value of the constructor to the new object (so **this** points to the new object which we are creating).

3. Execute the code inside the constructor (adds properties to the new object).

4. Return the new object.

At the end of the preceding example, person1 and person2 are each filled with a different instance of Person. Each of these objects has a constructor property that points back to Person, as follows:

alert(person1.constructor == Person); //true

alert(person2.constructor == Person); //true

The constructor property was originally intended for use in identifying the object type. However, the instanceof operator is considered to be a safer way of determining type. Each of the objects in this example is considered to be both an instance of Object and an instance of Person, as indicated by using the instanceof operator like this:

alert(person1 instanceof Object); //true

alert(person1 instanceof Person); //true

alert(person2 instanceof Object); //true

alert(person2 instanceof Person); //true

Defining your own constructors ensures that instances can be identified as a particular type later on, which is a great advantage over the factory pattern. In this example, person1 and person2 are considered to be instances of Object, because all custom objects inherit from Object.

The only difference between constructor functions and other functions is the way in which they are called. Constructors are, after all, just functions; there’s no special syntax to define a constructor that automatically makes it behave as such. Any function that is called with the new operator acts as a constructor, whereas any function called without it acts just as you would expect a normal function call to act.

//use as a constructor

var person = new Person(“Le Roi”, 26, “Singer”);

person.sayName(); //”Le Roi”

//call as a function

Person(“Le Tuon Roi”, 27, “Graphic Designer of NAU”); //adds to window

window.sayName(); //”Le Tuon Roi”

//call in the scope of another object

var o = new Object();

Person.call(o, “Kristen”, 25, “Nurse”);

o.sayName(); //”Kristen”

At this example, calling a function with call method has something to talk about.

o variable now has every properties and methods like person, but when we check what it is with instanceof :

o instanceof Person // “false”

Why is this? Just because this is a kind of The Factory Pattern but not in a factory function.

The Constructor Pattern works well, but there is a problem with it.

Function Person can be changed like this :

function Person(name, age, job){ this.name = name;

this.age = age;

this.job = job;

this.sayName = new Function(“alert(this.name)”); //logical equivalent

}

So, in the previous example, both person1 and person2 have a method called sayName(), but those methods are not the same instance of Function. Because everytimes we create an instance of Person, we create difference instances of Function object which is stored in each sayName property. So when we check these methods like this :

alert(person1.sayName == person2.sayName); //false

It doesn’t make sense to have two instances of Function that do the same thing, so we can move the function definition outside of the constructor, as follows:

function Person(name, age, job){

this.name = name;

this.age = age;

this.job = job;

this.sayName = sayName;

}

function sayName(){

alert(this.name);

}

This solves the problem of having duplicate functions that do the same thing but also creates some clutter in the global scope by introducing a function that can realistically be used only in relation to an object. If the object needed multiple methods, that would mean multiple global functions, and perhaps that is a mess and not nice.

1. The Prototype Pattern

Each function is created with a prototype property, which is an object containing properties and methods that should be available to instances of a particular reference type. This object is a prototype for the object to be created once the constructor is called. The benefit of using the prototype is that all of its properties and methods are shared among object instances. Instead of assigning object information in the constructor, they can be assigned directly to the prototype, as in this example:

function Person(){ }

Person.prototype.name = “Hans”;

Person.prototype.age = 26;

Person.prototype.job = “Super Junior”;

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

person1.sayName(); //”Hans”

￼￼var person2 = new Person();

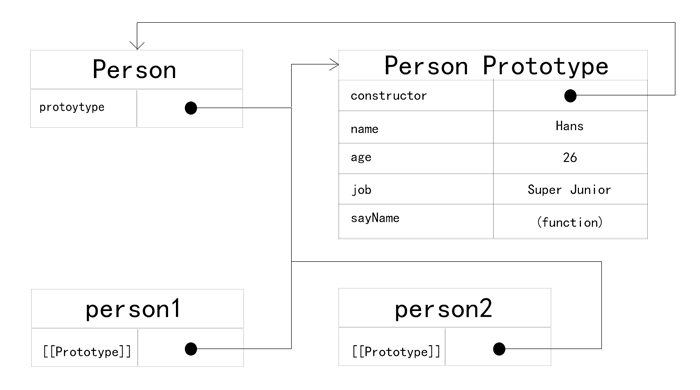
person2.sayName(); //”Hans”

alert(person1.sayName == person2.sayName); //true

Unlike the constructor pattern, the properties and methods are all shared among instances, so person1 and person2 are both accessing the same set of properties and the same sayName() function.

Whenever a function is created, its prototype property is also. By default, all prototypes automatically get a property called constructor that points back to the function on which it is a property. In the previous example, for instance, Person.prototype .constructor points to Person and all other methods are inherited from Object. Each time the constructor is called to create a new instance, that instance has an internal pointer to the constructor’s prototype. In ECMA-262 fifth edition, this is called [[Prototype]]. There is no standard way to access [[Prototype]] from script, but Firefox, Safari, and Chrome all support a property on every object called \_\_proto\_\_.; in other implementations, this property is completely hidden from script.

Consider the previous example using the Person constructor and Person.prototype. The relationship between the objects in the example is shown in the picture under.



Each instance of Person, person1, and person2 has internal properties that point back to Person.prototype only. That Person.prototype points to the prototype object but Person.prototype.constructor points back to Person. The prototype contains the constructor property and the other properties that were added.

Whenever a property is accessed for reading on an object, a search is started to find a property with that name. The search begins on the object instance itself. If a property with the given name is found on the instance, then that value is returned; if the property is not found, then the search continues up the pointer to the prototype, and the prototype is searched for a property with the same name. If the property is found on the prototype, then that value is returned. So, when person1.sayName() is called, a two-step process happens. First, the JavaScript engine asks, “Does the instance person1 have a property called sayName?” The answer is no, so it continues the search and asks, “Does the person1 prototype have a property called sayName?” The answer is yes, so the function stored on the prototype is accessed. When person2.sayName() is called, the same search executes, ending with the same result. This is how prototypes are used to share properties and methods among multiple object instances.

function Person(){ }

Person.prototype.name = “Thang”;

Person.prototype.age = 26;

Person.prototype.job = “Minior”;

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

var person2 = new Person();

person1.name = “Le Roi”;

alert(person1.name); //”Le Roi” - from instance

alert(person2.name); //” Thang” - from prototype

Once a property is added to the object instance, it shadows any properties of the same name on the prototype, which means that it blocks access to the property on the prototype without altering it. And when we remove the property in instance, it still looks for the property in the prototype.

delete person1.name;

alert(person1.name); //”Thang” - from the prototype

Yeah, we should remove Le Roi, I dun like that property.

So, to check a propery from instance or from prototype, we have a method named hasOwnProperty()

alert(person1.hasOwnProperty(“name”)); //false

person1.name = “Trang”;

alert(person1.name); //”Trang” - from instance alert(person1.hasOwnProperty(“name”)); //true

Another way to define properties on prototype is simply overwrite the prototype with an object literal that contains all of the properties and methods, as in this example:

function Person(){

}

Person.prototype = {

name : “Nicholas”,

age : 29,

job : “Software Engineer”,

sayName : function () {

alert(this.name);

}

};

But there is a problem, the constructor property no longer points to Person, it points to Object exactly because we overwrite with an object and now its constructor is the object constructor.

var friend = new Person();

alert(friend instanceof Object); //true

alert(friend instanceof Person); //true

alert(friend.constructor == Person); //false

alert(friend.constructor == Object); //true

If the constructor’s value is important, you can set it specifically back to the appropriate value, as shown here:

￼￼￼function Person(){ }

Person.prototype = {

constructor: Person,

name : “Nicholas”,

age : 29,

job : “Software Engineer”,

sayName : function () {

alert(this.name);

}

};

Native constructor properties are not enumerable by default, so if you’re using an ECMAScript 5–compliant JavaScript engine, you may wish to use Object.defineProperty() instead:

function Person(){ }

Person.prototype = {

name : “Nicholas”,

age : 29,

job : “Software Engineer”,

sayName : function () {

alert(this.name);

}

};

//ECMAScript 5 only – restore the constructor

Object.defineProperty(Person.prototype, “constructor”, {

enumerable: false,

value: Person

});

function Person(){ }

var friend = new Person();

Person.prototype = {

constructor: Person,

name : “Nicholas”,

age : 29,

job : “Software Engineer”,

sayName : function () {

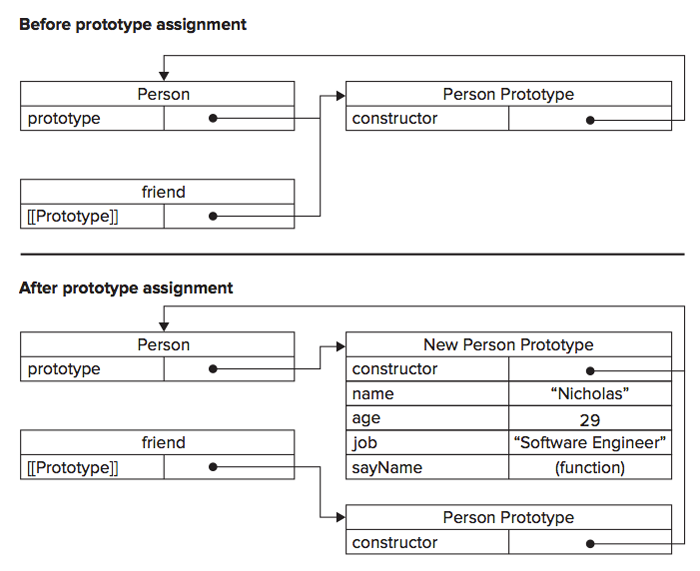
alert(this.name);

}

};

friend.sayName(); //error

￼In this example, a new instance of Person is created before the prototype object is overwritten. When friend.sayName() is called, it causes an error, because the prototype that friend points to doesn’t contain a property of that name. This is what really happens :



Overwriting the prototype on the constructor means that new instances will reference the new prototype while any previously existing object instances still reference the old prototype.

Although possible to do this :

String.prototype.startsWith = function (text) {

return this.indexOf(text) == 0;

};

var msg = “Hello world!”;

alert(msg.startsWith(“Hello”)); //true

It is not recommended to modify native object prototypes.

All properties on the prototype are shared among instances. Properties that contain primitive values also tend to work well, as shown in the previous example, where it’s possible to hide the prototype property by assigning a property of the same name to the instance. The real problem occurs when a property contains a reference value. Consider the following example:

function Person(){ }

Person.prototype = {

constructor: Person,

name : “Le Roi”,

age : 29,

job : “Singer”,

friends : [“Thuy TOP”, “Phuong Trinh”],

sayName : function () {

alert(this.name);

}

};

var person1 = new Person();

var person2 = new Person();

person1.friends.push(“Phi Thanh Van”);

alert(person1.friends); // ”Thuy TOP,Phuong Trinh,Phi Thanh Van”

alert(person2.friends); // ”Thuy TOP,Phuong Trinh,Phi Thanh Van”

alert(person1.friends === person2.friends); //true

Because the friends array exists on Person.prototype, not on person1, the changes made are also reflected on person2.friends (which points to the same array). If the intention is to have an array shared by all instances, then this outcome is okay. Typically, though, instances want to have their own copies of all properties. This is why the prototype pattern is rarely used on its own.

1. Combination Constructor/Prototype Pattern

The most common way of defining custom types is to combine the constructor and prototype patterns. The constructor pattern defines instance properties, whereas the prototype pattern defines methods and shared properties. With this approach, each instance ends up with its own copy of the instance properties, but they all share references to methods, conserving memory. This pattern allows arguments to be passed into the constructor as well, effectively combining the best parts of each pattern. The previous example can now be rewritten as follows:

function Person(name, age, job){

this.name = name;

this.age = age;

this.job = job;

￼this.friends = [“Trang”, “Vu”];

}

Person.prototype = {

constructor: Person,

sayName : function () {

alert(this.name);

}

};

var person1 = new Person(“Hans”, 26, “Terrorist”);

var person2 = new Person(“Thang”, 23, “Cop”);

person1.friends.push(“Thang”);

alert(person1.friends); //” Trang,Vu,Thang”

alert(person2.friends); //” Trang,Vu”

alert(person1.friends === person2.friends); //false

alert(person1.sayName === person2.sayName); //true

The hybrid constructor/prototype pattern is the most widely used and accepted practice for defining custom reference types in ECMAScript. Generally speaking, this is the default pattern to use for defining reference types.

There are many patterns but we don’t have time so I think we should stop here and if you’re still curious about these patterns, there is a online book you can take a look.

Resources :

<http://addyosmani.com/resources/essentialjsdesignpatterns/book/>

3- INHERITANCE

1. Prototype Chaining :

ECMA-262 describes prototype chaining as the primary method of inheritance in ECMAScript. The basic idea is to use the concept of prototypes to inherit properties and methods between two reference types. Recall the relationship between constructors, prototypes, and instances: each constructor has a prototype object that points back to the constructor, and instances have an internal pointer to the prototype. What if the prototype were actually an instance of another type? That would mean the prototype itself would have a pointer to a different prototype that, in turn, would have a pointer to another constructor. If that prototype were also an instance of another type, then the pattern would continue, forming a chain between instances and prototypes. This is the basic idea behind prototype chaining.

Example :

function SuperType(){

this.property = true;

}

SuperType.prototype.getSuperValue = function(){

return this.property;

};

function SubType(){

this.subproperty = false;

}

//inherit from SuperType

SubType.prototype = new SuperType();

SubType.prototype.getSubValue = function (){

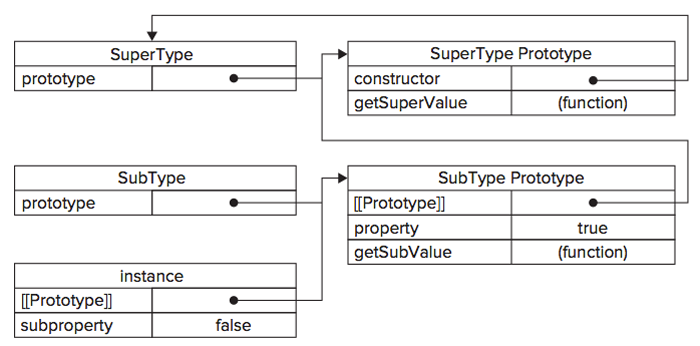
return this.subproperty;

};

var instance = new SubType();

alert(instance.getSuperValue()); //true

We transform above example to a diagram :

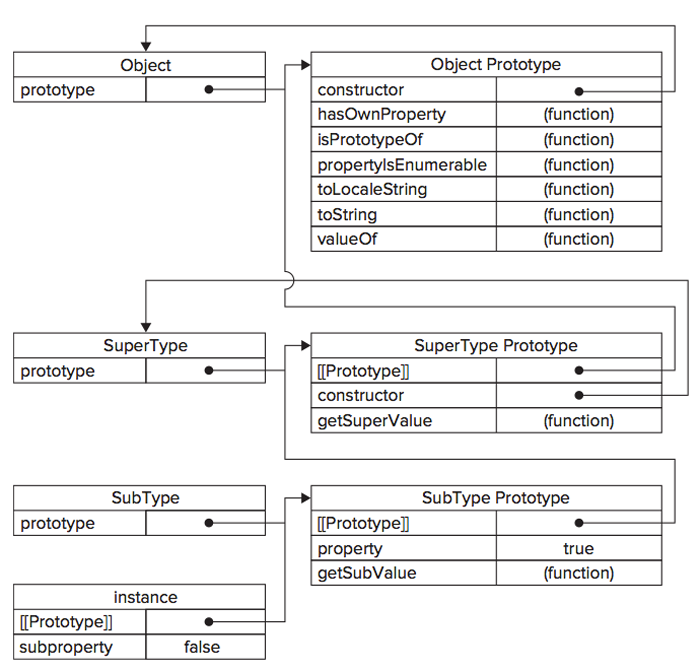


This code defines two types: SuperType and SubType. Each type has a single property and a single method. The main difference between the two is that SubType inherits from SuperType by creating a new instance of SuperType and assigning it to SubType.prototype. This overwrites the original prototype and replaces it with a new object, which means that all properties and methods that typically exist on an instance of SuperType now also exist on SubType.prototype. After the inheritance takes place, a method is assigned to SubType.prototype, adding a new method on top of what was inherited from SuperType.

Instead of using the default prototype of SubType, a new prototype is assigned. That new prototype happens to be an instance of SuperType, so it not only gets the properties and methods of a SuperType instance but also points back to the SuperType’s prototype. So instance points to SubType.prototype, and SubType.prototype points to SuperType.prototype. Note that the getSuperValue() method remains on the SuperType.prototype object, but property ends up on SubType.prototype. That’s because getSuperValue() is a prototype method, and property is an instance property. SubType.prototype is now an instance of SuperType, so property is stored there. Also note that instance.constructor points to SuperType, because the constructor property on the SubType.prototype was overwritten.

Prototype chaining extends to the prototype search mechanism described earlier. As you may recall, when a property is accessed in read mode on an instance, the property is first searched for on the instance. If the property is not found, then the search continues to the prototype. When inheritance has been implemented via prototype chaining, that search can continue up the prototype chain. In the previous example, for instance, a call to instance.getSuperValue() results in a three-step search: 1) the instance, 2) SubType.prototype, and 3) SuperType.prototype, where the method is found. The search for properties and methods always continues until the end of the prototype chain is reached.

In reality, there is another step in the prototype chain. All reference types inherit from Object by default, which is accomplished through prototype chaining. The default prototype for any function is an instance of Object, meaning that its internal prototype pointer points to Object .prototype.



SubType inherits from SuperType, and SuperType inherits from Object. When instance .toString() is called, the method being called actually exists on Object.prototype.

Often a subtype will need to either override a supertype method or introduce new methods that don’t exist on the supertype. To accomplish this, the methods must be added to the prototype **after the prototype has been assigned**. Consider this example:

function SuperType(){

this.property = true;

}

SuperType.prototype.getSuperValue = function(){

return this.property;

};

function SubType(){

this.subproperty = false;

}

//inherit from SuperType

SubType.prototype = new SuperType();

//new method

SubType.prototype.getSubValue = function (){

return this.subproperty;

};

//override existing method

SubType.prototype.getSuperValue = function (){

return false;

};

var instance = new SubType();

alert(instance.getSuperValue()); //false

Another important thing to understand is that the object literal approach to creating prototype methods cannot be used with prototype chaining, because you end up overwriting the chain. Here’s an example:

function SuperType(){

this.property = true;

}

SuperType.prototype.getSuperValue = function(){

return this.property;

};

function SubType(){

this.subproperty = false;

}

//inherit from SuperType SubType.prototype = new SuperType();

//try to add new methods - this nullifies the previous line

**SubType.prototype = {**

**￼getSubValue : function (){**

**return this.subproperty;**

**},**

**someOtherMethod : function (){**

**return false;**

**}**

**};**

var instance = new SubType();

**alert(instance.getSuperValue()); //error!**

In this code, the prototype is reassigned to be an object literal after it was already assigned to be an instance of SuperType. The prototype now contains a new instance of Object instead of an instance of SuperType, so the prototype chain has been broken — there is no relationship between SubType and SuperType.

Even though prototype chaining is a powerful tool for inheritance, it still has problems.

function SubType(){ }

//inherit from SuperType

SubType.prototype = new SuperType();

var instance1 = new SubType();

instance1.colors.push(“black”);

alert(instance1.colors); //”red,blue,green,black”

var instance2 = new SubType();

alert(instance2.colors); //”red,blue,green,black”

1. Constructor Stealing

Developers began using a technique called constructor stealing (also sometimes called classical inheritance) to solve the inheritance problem with reference values on prototypes. The basic idea is quite simple: call the supertype constructor from within the subtype constructor, the apply() and call() methods can be used to execute a constructor on the newly created object, as in this example:

function SuperType(){

this.colors = [“red”, “blue”, “green”];

}

function SubType(){

//inherit from SuperType

**SuperType.call(this);**

}

var instance1 = new SubType();

instance1.colors.push(“black”);

alert(instance1.colors); //”red,blue,green,black”

var instance2 = new SubType();

alert(instance2.colors); //”red,blue,green”

One advantage that constructor stealing offers over prototype chaining is the ability to pass arguments into the supertype constructor from within the subtype constructor. Consider the following:

function SuperType(name){

this.name = name;

}

function SubType(){

//inherit from SuperType passing in an argument

SuperType.call(this, “Le Roi”);

//instance property

this.age = 29;

}

var instance = new SubType();

alert(instance.name); //”Le Roi”;

alert(instance.age); //29

The downside to using constructor stealing exclusively is that it introduces the same problems as the constructor pattern for custom types: methods must be defined inside the constructor, so there’s no function reuse. Furthermore, methods defined on the supertype’s prototype are not accessible on the subtype, so all types can use only the constructor pattern. Because of these issues, constructor stealing is rarely used on its own.

1. Combination Inheritance

Combination inheritance (sometimes also called pseudoclassical inheritance) combines prototype chaining and constructor stealing to get the best of each approach. The basic idea is to use prototype chaining to inherit properties and methods on the prototype and to use constructor stealing to inherit instance properties. This allows function reuse by defining methods on the prototype and allows each instance to have its own properties. Consider the following:

function SuperType(name){

this.name = name;

this.colors = [“red”, “blue”, “green”];

}

SuperType.prototype.sayName = function(){

alert(this.name);

};

function SubType(name, age){

//inherit properties

SuperType.call(this, name);

this.age = age;

}

//inherit methods

SubType.prototype = new SuperType();

SubType.prototype.sayAge = function(){

alert(this.age);

};

var instance1 = new SubType(“Le Roi”, 26);

instance1.colors.push(“black”);

alert(instance1.colors); //”red,blue,green,black”

instance1.sayName(); //”Le Roi”;

instance1.sayAge(); //26

var instance2 = new SubType(“Le Coc”, 27);

alert(instance2.colors); //”red,blue,green”

instance2.sayName(); //”Le Coc”;

instance2.sayAge(); //27

Addressing the downsides of both prototype chaining and constructor stealing, combination inheritance is the most frequently used inheritance pattern in JavaScript. It also preserves the behavior of instanceof and isPrototypeOf() for identifying the composition of objects.

1. Prototypal Inheritance

In 2006, Douglas Crockford,



“Hello, I’m Douglas Crockford, Hans’s friend”

wrote an article titled “Prototypal Inheritance in JavaScript” in which he introduced a method of inheritance that didn’t involve the use of strictly defined constructors. His premise was that prototypes allow you to create new objects based on existing objects without the need for defining custom types. The function he introduced to this end is as follows:

function object(o){

function F(){}

F.prototype = o;

return new F();

}

var person = {

name: “Douglas”,

friends: [“Quy”, “Thanh”, “Thinh”]

};

var anotherPerson = object(person);

anotherPerson.name = “Le Roi”;

anotherPerson.friends.push(“Vu”);

var yetAnotherPerson = object(person);

yetAnotherPerson.name = “Linda Kieu”;

yetAnotherPerson.friends.push(“Thang”);

alert(person.friends); //”Quy,Thanh,Thinh,Vu,Thang”

ECMAScript 5 formalized the concept of prototypal inheritance by adding the Object.create() method. This method accepts two arguments, an object to use as the prototype for a new object and ￼an optional object defining additional properties to apply to the new object. When used with one argument, Object.create() behaves the same as the object() method:

var person = {

name: “Douglas”,

friends: [“Quy”, “Thanh”, “Thinh”]

};

var anotherPerson = Object.create(person);

anotherPerson.name = “Le Roi”;

anotherPerson.friends.push(“Vu”);

var yetAnotherPerson = Object.create(person);

yetAnotherPerson.name = “Linda Kieu”;

yetAnotherPerson.friends.push(“Thao”);

alert(person.friends); //” Quy,Thanh,Thinh,Vu,Thao”

Sources:

<http://javascript.crockford.com/prototypal.html>

I think we should read this article to understand Prototypal Inheritance.

1. Others: Parasitic Inheritance, Parasitic Combination Inheritance …

Because we don’t have enough time for these techniques so I just introduce these ones for you and prefer some sources for more details.