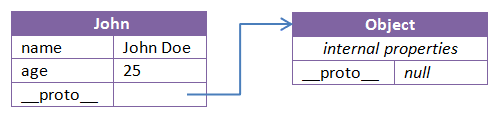
JavaScript Object and their Prototypes

Objects in JavaScript is nothing but collection of attributes (or properties to be more inline with ECMA definition). And a property is defined as a key-value pair. Object in JavaScript can be created using many ways, but for now we are gonna use a simple one.  
Lets create a Person called John:

|  |
| --- |
| var John = {               'name'  : 'John Doe',               'age'   : '25'  }; |

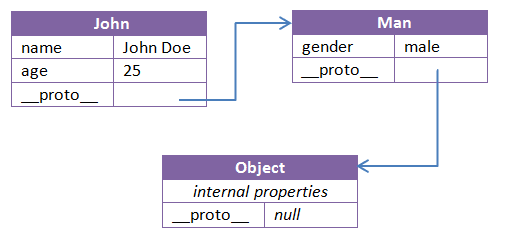
This object has two property ‘name’ and ‘age’. Note that the value of a property is not at all limited to primitive types but can be any other javascript object. So for example ‘John’ may have another property ‘address’, whose value can be an Array of different address objects. Apart from the properties defined in object, every JavaScript object has another secret internal property called ‘\_\_proto\_\_’. This property points to the prototype of the John. But what is the prototype of John? Well for John here, its JavaScript’s parent of all objects, The Object. And what is the prototype of The Object? Its *null*.  
Below image might make this more clear:



Prototype Chaining and Inheritance

We now know that each object in JavaScript has another \_\_proto\_\_ property which refers to its prototype. If I am to create an object, and explicitly assign \_\_proto\_\_ another object… what would that mean?

|  |
| --- |
| var Man = {     gender: 'male'  };  var John = {     name: 'John Doe',     age: 25,     \_\_proto\_\_: Man  } |



This would mean John *prototypically inherits* from Man. All properties of Man can be directly accessed through John, this is because whenever any property is accessed via John and if it is not found in John it will be looked in the prototype of John.

|  |
| --- |
| console.log(John.name); // John Doe  console.log(John.age); // 25  console.log(John.gender); // male |

Further, this prototypical inheritance is not limited to two levels and can be extended to any level. This sort of chaining is called **Prototype Chaining**.

The rule to resolve any property is: If a property is not found in an object, it is looked into the prototype chain of the object all the way to Object unless found earlier. If property is not found, we get a property undefined error.

Function

A function is a special type of Object which may be invoked as a routine. A function may have properties, and some lines of code which determines the behavior of the function. For example, we can have a function named isEven() which takes a number as parameter and tell us whether a number is even or not.

|  |
| --- |
| function isEven(no){     return no % 2 === 0;  } |

Further functions can be used to create Objects acting as a Constructor for the class. All that is required is to use new operator when calling that function. For example, I can write following Person function to create different Person objects:

|  |
| --- |
| function Person(name, age){     this.name = name;     this.age = age;     this.shoutYourName = function(){                                alert('My name is ' + name +'!');                          };  }  //create Objects  var John = new Person('John'25);  var Donn = new Person('Donn'29);    console.log(John.name);  //John  console.log(John.age);  //25  John.shoutYourName();   //alerts 'My name is John!' |

Now the question is what gets create and stored in memory above functions are created? What’s the prototype of it? And how is it that same functions can act as Constructors to create new objects?

To answer these question,  
A function that is called using *new* operator for creation of new objects is called a Constructor Function.  
Creating a function creates an object of type Function. That means, any new function will have its \_\_proto\_\_ point to prototype of ‘Function’.  
**As a side note:** *Function is a global internal Object, it doesn’t have any properties of its own but inherits them from Function.prototype.*  
Further, objects of type Function have another property called ‘prototype’ pointing to the prototype of the objects which will be created using this function.

In case of above example:  
1. Person is an object of type Function. And this also means Person has its \_\_proto\_\_ property pointed to Function.prototype  
2. Person has a property ‘prototype’ pointing to an object which will be assigned to \_\_proto\_\_ property of objects created by Person, in our case, John and Donn. Lets call this prototype object Person.prototype. So this would mean any objects created using Person will be prototypically inherited from Person.

|  |
| --- |
| console.log(John.\_\_proto\_\_ === Person.prototype)     //true  console.log(Donn.\_\_proto\_\_ === Person.prototype)     //true  console.log(Person.\_\_proto\_\_ === Function.prototype) //true |

Structure of Person.prototype

As already mentioned above, all objects created using Person will have their \_\_proto\_\_ pointing to Person.prototype. This means, these objects will Prototypically inherit from Person.prototype

1. \_\_proto\_\_ of Person.prototype points to Object.
2. And extra property added to Person.prototype will be prototypically available to objects created by calling new Person(). And using this reference inside properties of Person.prototype will refer to properties of Person. So, after creating the objects John and Donn, if I am to add below code:

|  |
| --- |
| Person.prototype.getAllCapsName = function(){  return this.name.toUpperCase();  }; |

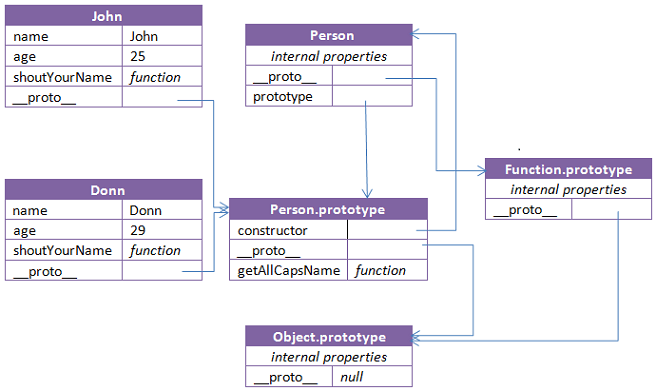
1. this property getAllCapsName will be immediately available in already created objects ‘John’ and ‘Donn’. And that’s because, saying it again, \_\_proto\_\_ of John and Donn is pointing to Person.prototype object,i.e, John and Donn prototypically-inherits from Person.

|  |
| --- |
| console.log(John.getAllCapsName()) //JOHN  console.log(John.\_\_proto\_\_.getAllCapsName()) //Can't access property of John using this.                                               //this refers to Person.prototype object  console.log(Person.prorotype.getAllCapsName()) //Can't access property using this. |

1. Person.prototype has another special property called constructor, which points to the original function object Person. And due to above point:

|  |
| --- |
| console.log(Person.prototype.constructor===Person) //true  console.log(John.constructor)// print Person function  console.log(Person.prototype.constructor)// print Person function  console.log(John.constructor===Person) //true  console.log(John.\_\_proto\_\_.constructor===Person.prototype.constructor) //true  console.log(John.\_\_proto\_\_.constructor===Person.prototype.constructor) //true |

In case things are still hazy, image below might help:



Now since behavior of all objects created using one Constructor Function will be same, it makes more sense to add the behavior in the prototype of Constructor Function than in the Constructor Function. I.e. to add functions properties inside Person.prototype than in Person. This is because any function-property in Person (say shoutYourName()) will be copied in John and Donn. But, any function-property in Person.prototype will be available to John and Donn through \_\_proto\_\_ reference.  
So if I am to change the definition of shoutYourName for John, behavior of Donn will not be affected. But it’s not the case with getAllCapsName().

Let see this working:

|  |
| --- |
| var Person = function(name, age){      this.name = name;      this.age = age;      this.shoutYourName = function(){          return 'SHOUTING '+ this.name;      };  };    var John = new Person('John',25);  var Donn = new Person('Donn',25);    console.log(John.shoutYourName()) // SHOUTING John  console.log(Donn.shoutYourName()) // SHOUTING Donn    Person.prototype.shhhYourName = function(){      return 'shhh ' + this.name;  };    console.log(John.shhhYourName()) // shhh John  console.log(Donn.shhhYourName()) // shhh Donn    //changing definition of only John  John.shoutYourName = function(){      return 'SHOUTING ' + this.name + '!!!!!';  };    //changing definition in prototype  Person.prototype.shhhYourName = function(){      return 'shhh ' + this.name + '!!!!!';  };    console.log(John.shoutYourName()) // SHOUTING John!!!!!  console.log(Donn.shoutYourName()) // SHOUTING Donn    console.log(John.shhhYourName()) // shhh John!!!!!  console.log(Donn.shhhYourName()) // shhh Donn!!!!! |

Hope it helped to have a better understanding of Objects and Functions is JavaScript.  
and The End. :)