# Prototypes in JavaScript

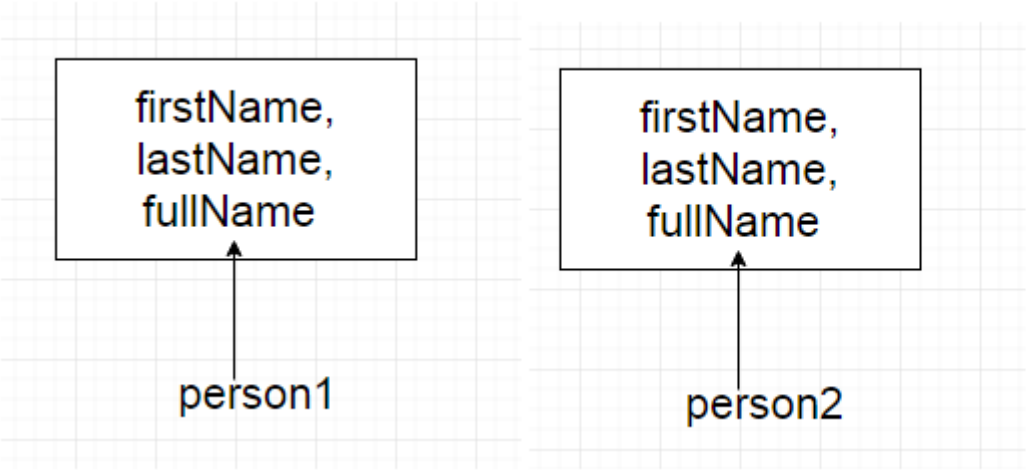
Problem with creating objects with constructor function syntax

In the previous post, we discuss about various ways of creating objects in JavaScript. One of the ways to create objects in JavaScript is using the Constructor function. Consider the construction function below:

Let's create objects *person1* and *person2* using the *Human* constructor function

```
var person1 = new Human("Virat", "Kohli");
var person2 = new Human("Sachin", "Tendulkar");
```

On executing the above code JavaScript engine will create two copy of constructor function each for *person1* and *person2*.

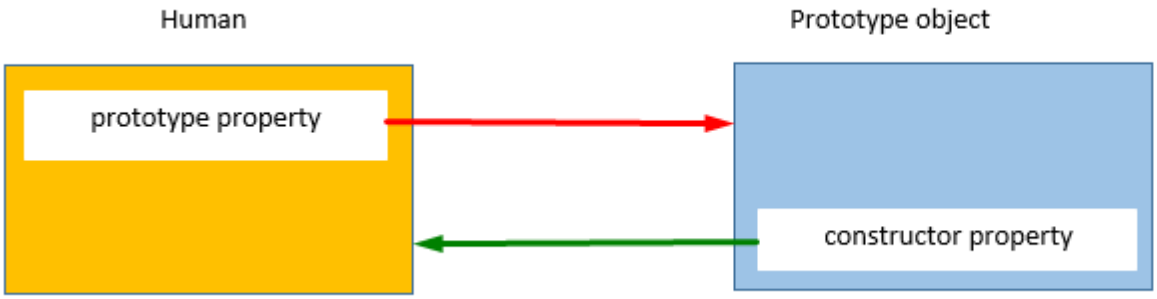



i.e. every object created using the constructor function will have it's own copy of properties and methods. It doesn't make sense to have two instances of function *fullName* that do the same thing.

Storing separate instances of function for each objects results into wastage of memory. We will see as we move forward how we can solve this issue.

## Prototypes

When a function is created in JavaScript, JavaScript engine adds a *prototype*property to the function. This *prototype* property is an object (called as prototype object) has a *constructor* property by default. *constructor* property points back to the function on which *prototype object* is a property. We can access the function's prototype property using the syntax *functionName.prototype*.



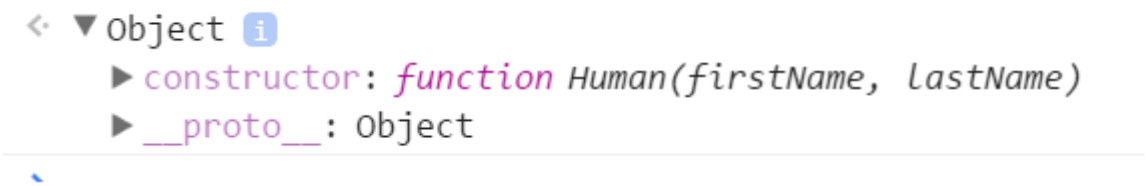

As shown in the above image, Human constructor function has a *prototype*property which points to the prototype object. The prototype object has a *constructor* property which points back to the Human constructor function. Let's see an example below:

```
console.log(Human);
```

```
▼ Human
    firstName: "Virat"
  ▶ fullName: function ()
    lastName: "Kohli"
  ▶ __proto__: Object
undefined
>
```

To access prototype property of the *Human* constructor use the below syntax:

```
console.log(Human.prototype)
```

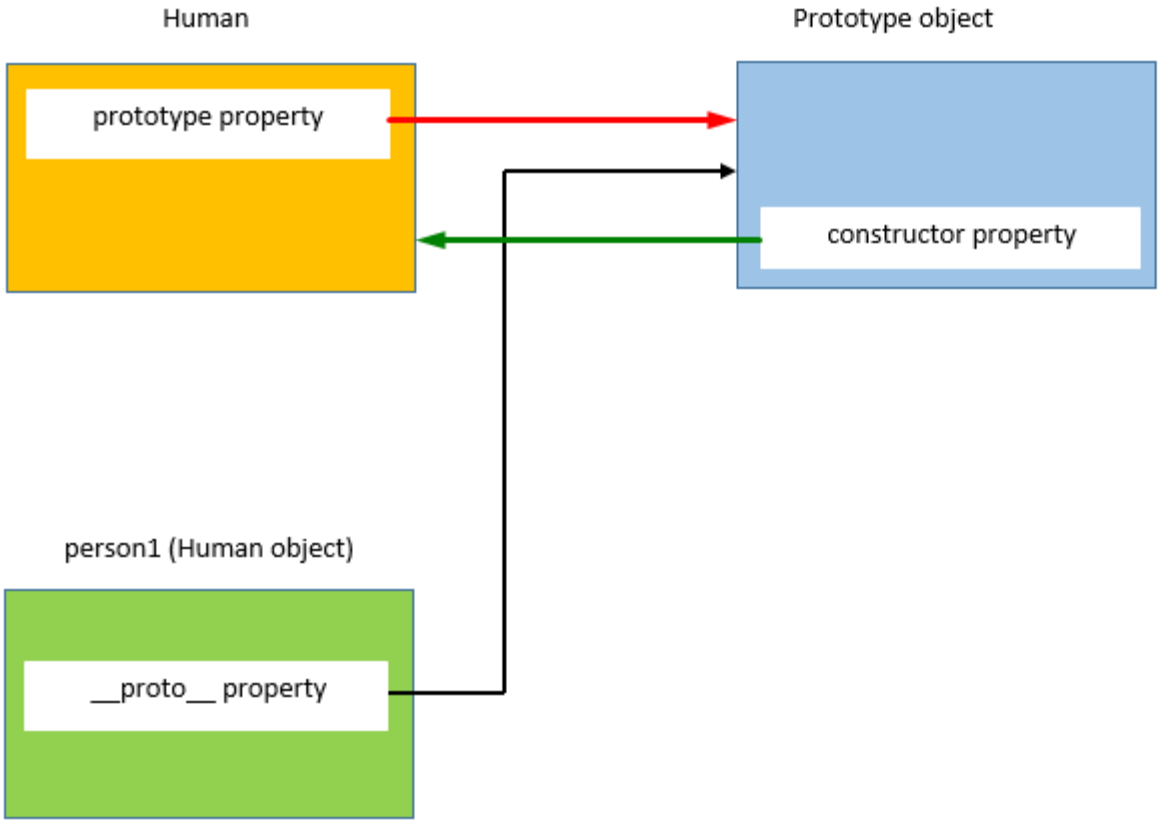



As seen from the above image *prototype* property of the function is an oject (prototype object) with two properties:

1. *constructor* property which points to *Human* function itself
2. *__proto__* property—We will discuss about this while explaining *inheritance* in JavaScript

## Creating an object using the constructor function

When an object is created in JavaScript, JavaScript engine adds a __proto__ property to the newly created object which is called as *dunder proto*. dunder proto or __proto__ points to the prototype object of the constructor function.




As shown in the above image, person1 object which is created using the Human constructor function has a dunder proto or ___proto___ property which points to the prototype object of the constructor function.

```
//Create an object person1 using the Human constructor function
var person1 = new Human("Virat", "Kohli");
```

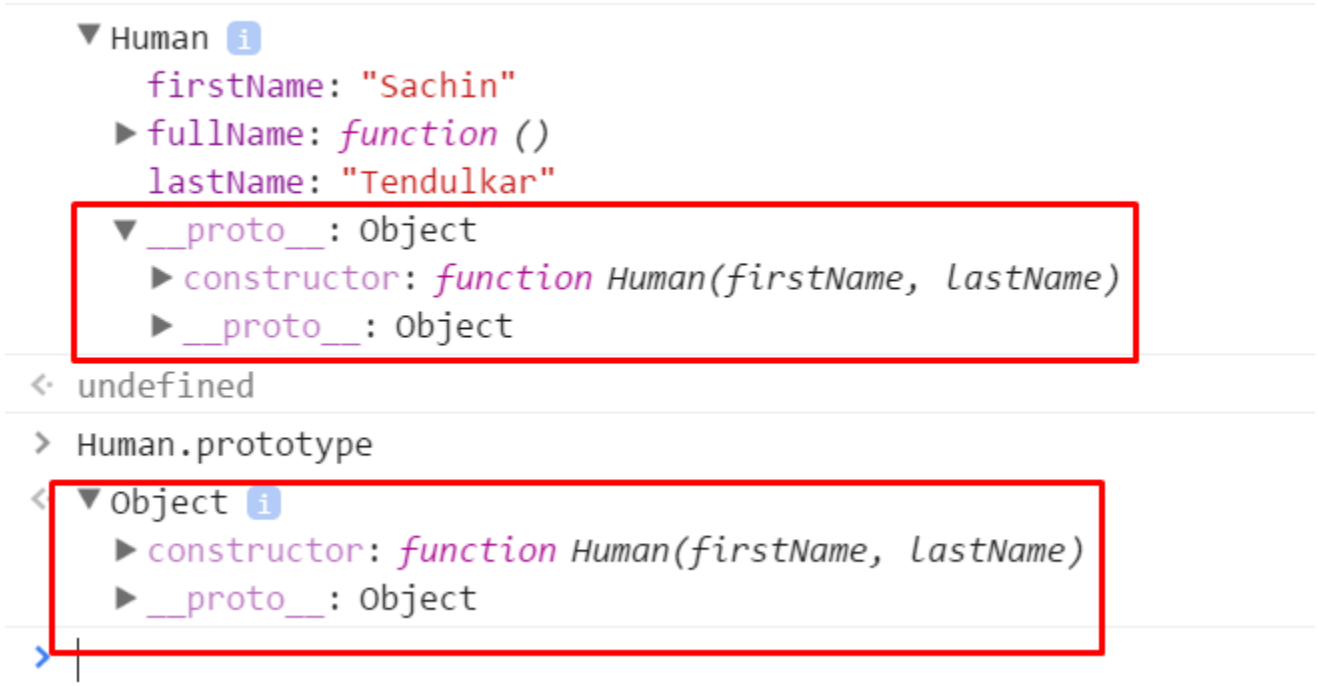

As it can be seen from the above image, both person1's *dunder proto or __proto__* property and *Human.prototype* property are equal let's check if they point at the some location using === operator

```
Human.prototype === person1.__proto__ //true
```

This shows that person1's *dunder proto* property and *Human.prototype* are pointing to the same object.

Now, lets's create an another object *person2* using the *Human* constructor function

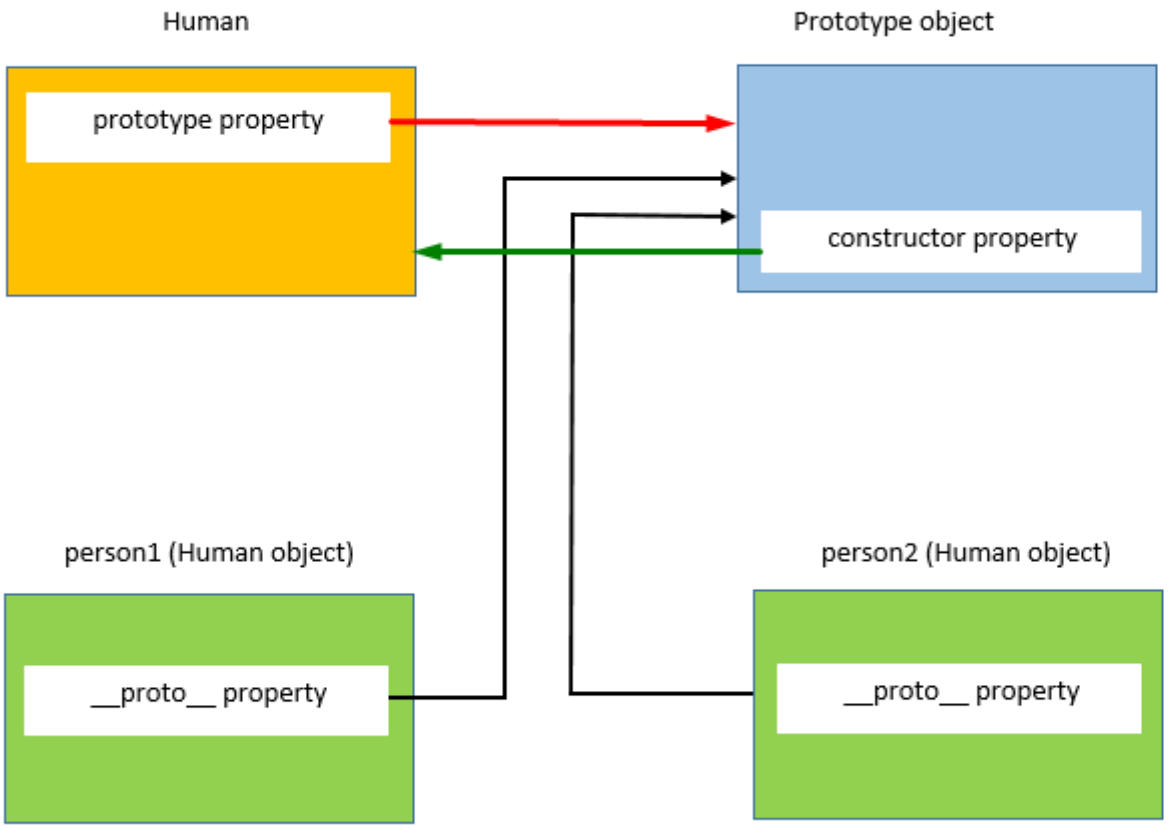
```
var person2 = new Human("Sachin", "Tendulkar");
console.log(person2);
```




Above console output shows that even person2's *dunder proto* property is equal to the *Human.prototype* property and they points to the same object.

```
Human.prototype === person2.__proto__ //true
person1.__proto__ === person2.__proto__ //true
```

Above statement proves that the *person1*'s and *person2*'s dunder proto properties points to *Human* constructor function's prototype object.




**Prototype object of the constructor function is shared among all the objects created using the constructor function.**

## Prototype Object

As prototype object is an object, we can attach properties and methods to the prototype object. Thus, enabling all the objects created using the constructor function to share those properties and methods.

New property can be added to the constructor function's prototype property using either the dot notation or square bracket notation as shown below:


*name* and *age* properties have been added to the Human prototype

**Example**

Let's analyze what happened when we did console.log(person.name) Let's check if person object has name property

```
console.log(person1);
```

As we can see that *person1* object is empty and it does not have any property except it's *dunder proto* property. So how does the output of ***console.log(person.name)*** was "Ashwin"?

When we try to access a property of an object, JavaScript engines first tries to find the property on the object; if the property is present on the object it outputs its value. But, if the property is not present on the object then it checks tries to find the property on the prototype object or *dunder proto* of the object. If the property is found the value is returned else JavaScript engine tries to find the property on the *dunder proto* of the *dunder proto* of the object. This chain continues till the *dunder proto* property is *null*. In this cases output will be *undefined*.

So, when *person1.name* is called, JavaScript engine checks if the property exists on the person object. In this case, name property was not on the *person's*object. So, now JavaScript engine checks if the *name* property exists on the *dunder proto* property or the prototype of the *person's* object. In this cases, *name* property was there on the *dunder proto* property or the prototype of *person's* object. Hence, the output was returned "Ashwin".

Let's create an another object person2 using the Person constructor function.

```
var person2 = new Person();
//Access the name property using the person2 object
console.log(person2.name)// Output: Ashwin
```

Now, let's define a property *name* on the *person1* object

```
person1.name = "Anil"
console.log(person1.name)//Output: Anil
console.log(person2.name)//Output: Ashwin
```

Here *person1.name* outputs “Anil”, because as mentioned earlier JavaScript engines first tries to find the property on the object itself as in case of *person1* the property is present on the object JavaScript engines outputs value of name property of *person1*.

In case of *person2*, name property is not present on the object. Hence, it outputs *person2’s* proptotype property *name*.

# Problems with the prototype

As prototype object is shared among all the objects created using the constructor function, it’s properties and methods are also shared among all the objects. If an object A modifies property of the prototype having primitive value, other objects will not be effected by this as A will create a property on its objects as shown below.
Here (line 1 and 2), both *person1* and *person2* does not have *name* property, hence they access the prototypes *name* property and hence the output is same for both.

When *person1* want to have different value for the name property, it creates a *name* property on its object.

## Consider another example to display the issue with proptotypes when the prototype object contains a property of reference type

In the above example, *person1* and *person2* points to the same *friends* array of the prototype

object. *person1* modifies *friends* property by adding another string in the array.

Because the *friends* array exists on *Person.prototype*, not on *person1*, the changes made in the friends property by *person1* objects 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. But here this was not the case.

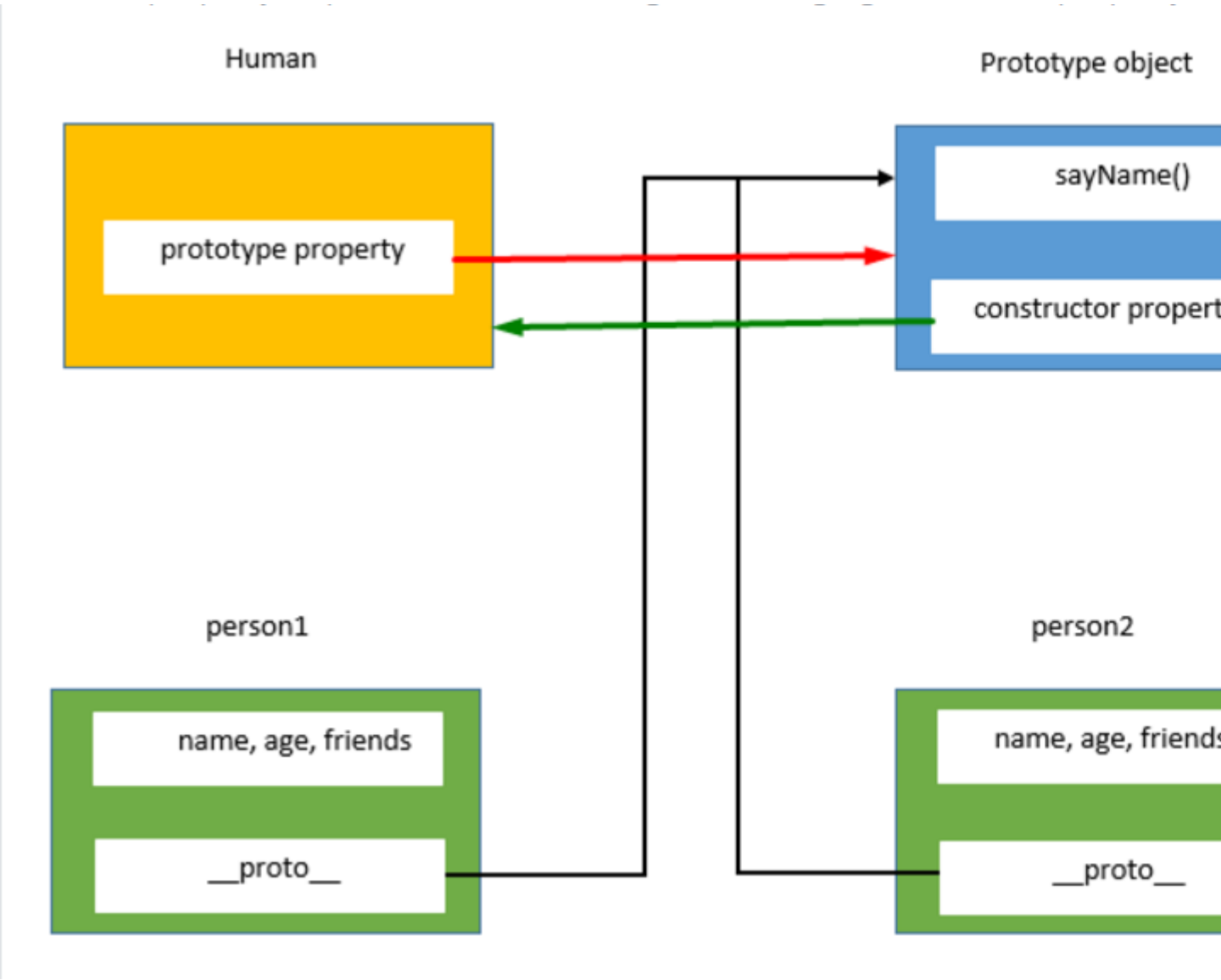
## Combine Constructor/Prototype

To solve the problems with the prototype and the problems with the constructor, we can combine both the constructor and function.

1. Problem with constructor: Every object has its own instance of the function
2. Problem with the prototype: Modifying a property using one object reflects the other object also

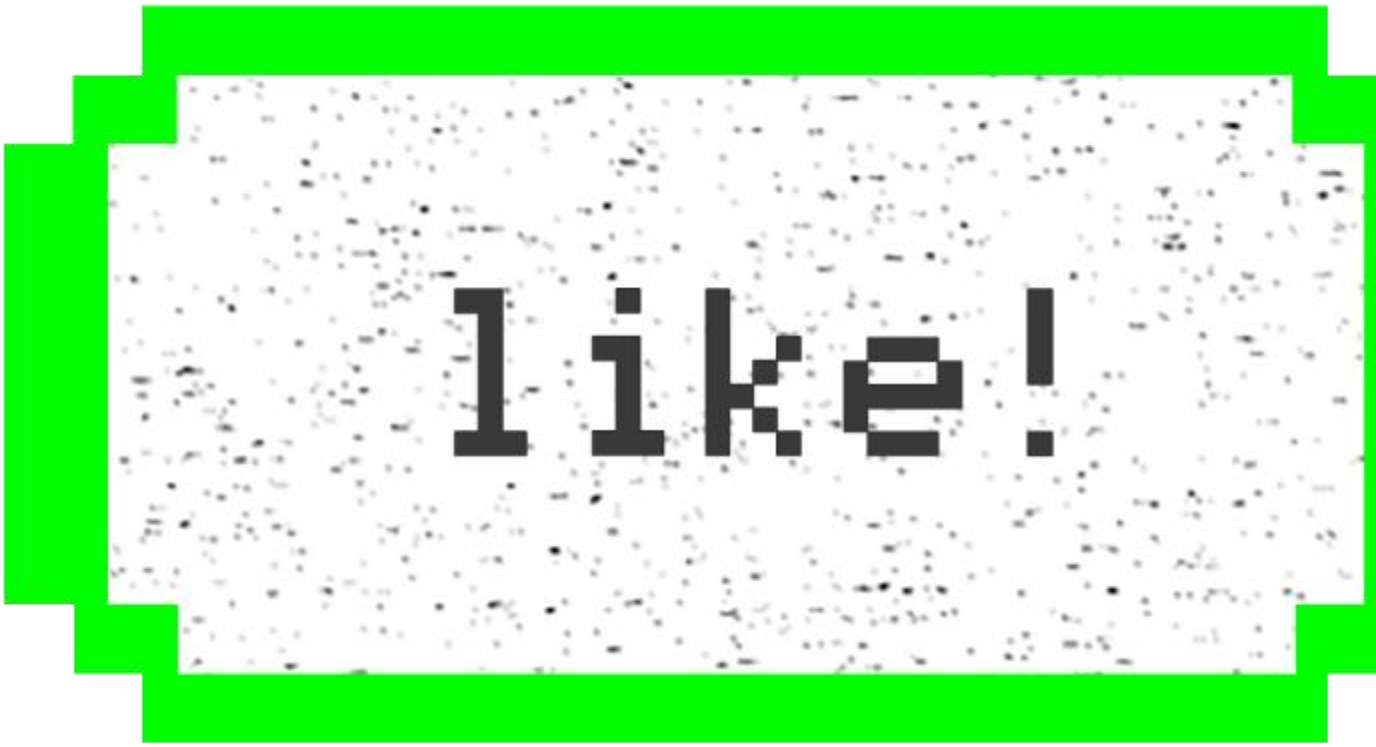
To solve above both problmens, we can define all the object specific properties inside the constructor and all shared properties and methods insdie the prototype as shown below:

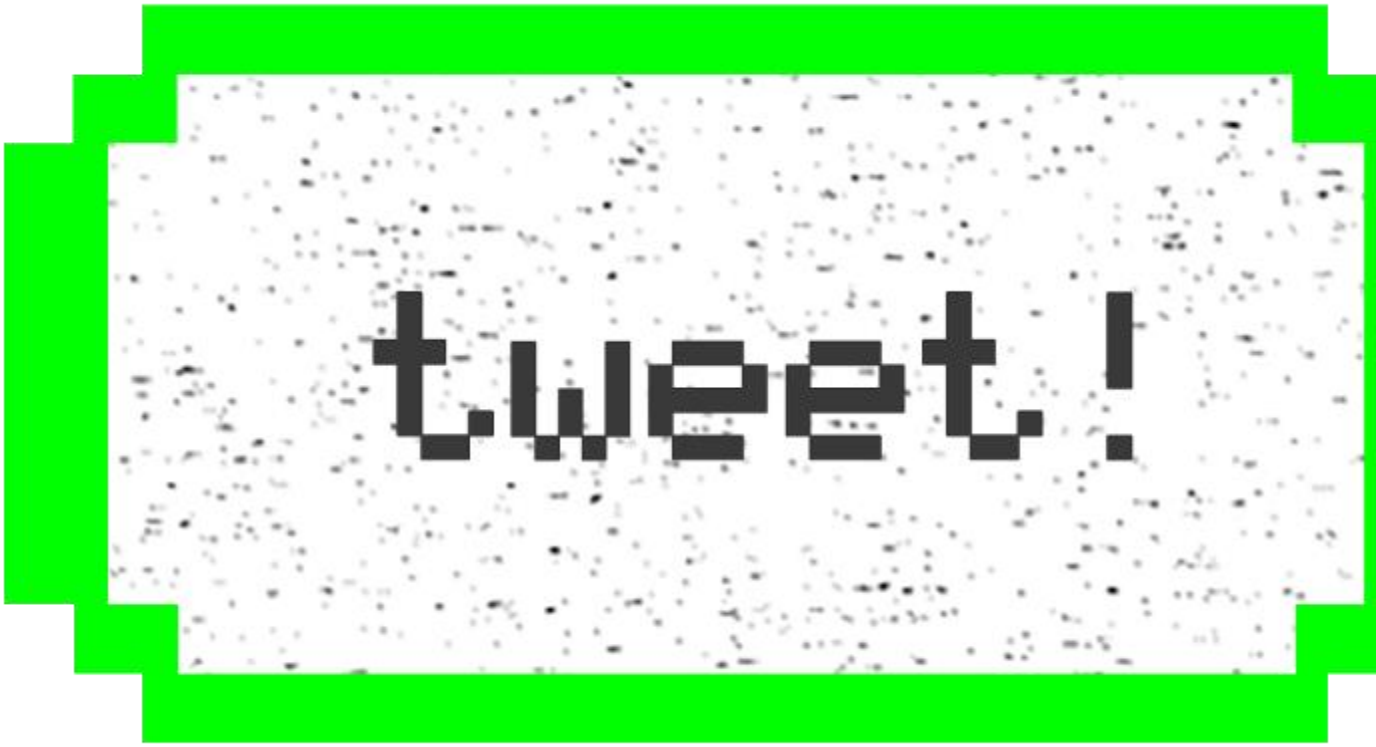
In the above example, *friends* property of *person2* did not change on changing the *friends* property of *person1*.

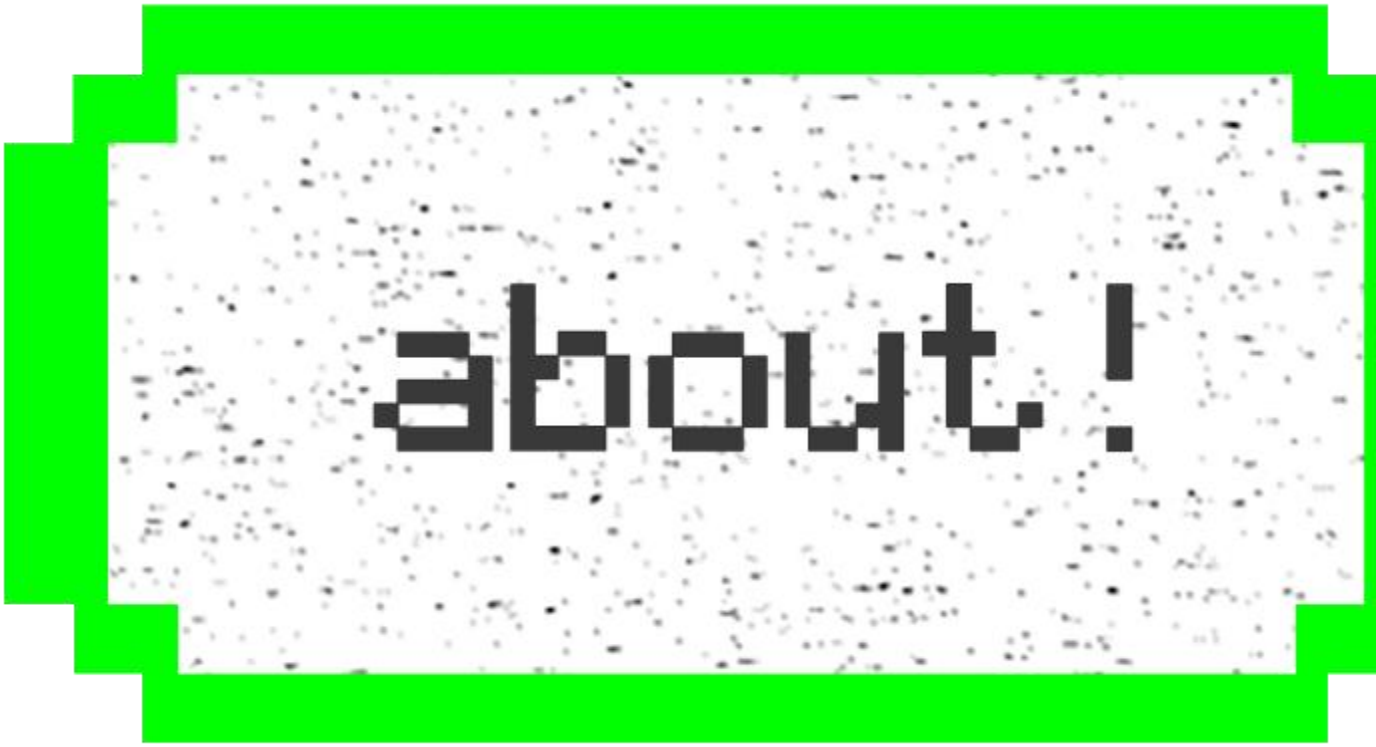



Further reads:

1. Virtual DOM in ReactJS
2. Execution Context in JavaScript
3. Inheritance in JavaScript
4. 'this' in JavaScript
5. Create Objects in JavaScript
6. Objects in JavaScript


like!


tweet!




*Hacker Noon is how hackers start their afternoons. We're a part of the @AMIfamily. We are now accepting submissions and happy to discuss advertising & sponsorship opportunities.*
*If you enjoyed this story, we recommend reading our latest tech stories and trending tech stories. Until next time, don't take the realities of the world for granted!*

- JavaScript
- Object Oriented
- Prototype

One clap, two clap, three clap, forty?
By clapping more or less, you can signal to us which stories really stand out.

384
5
- Follow

**Rupesh Mishra**

Full Stack Developer

- Follow

**Hacker Noon**

how hackers start their afternoons.

More from Hacker Noon

Ten years in, nobody has come up with a use for blockchain

Kai Stinchcombe

33K

Also tagged Object Oriented

Cultivate yourself as a coder. Adhere SOLID PRINCIPLES in your code.

Self discipline begins with the mastery of your thoughts. If you don’t control what you think, you cannot control what you do.

Shwet Mahalgi

104

Related reads
Stop reinventing architecture

H.N. Liyanage
171
**Responses**

Write a response…
Conversation with Rupesh Mishra.



Mohib Mansuri
May 13, 2017

Here (line 1 and 2), both person1 and person2 does not have name property, hence they access the prototypes name property and hence the output is same for both.

The image says otherwise.

1 response

Rupesh Mishra
May 13, 2017

Thanks for the feedback. Fixed the issue.

Show all responses

- 384
-
-
-