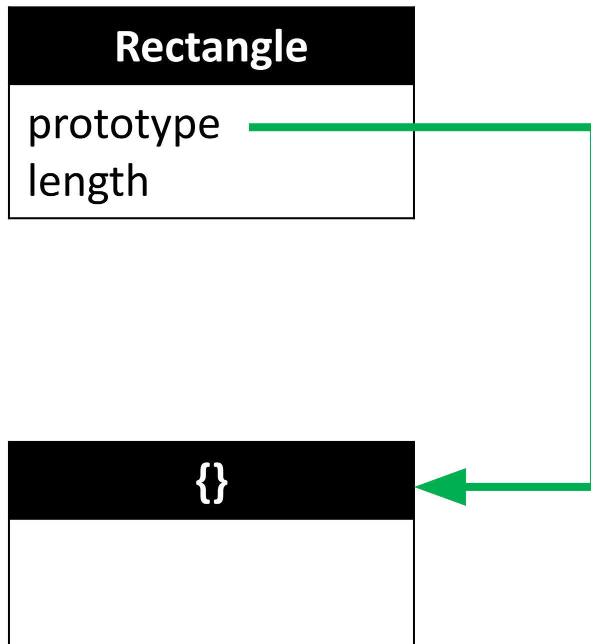
Prototype

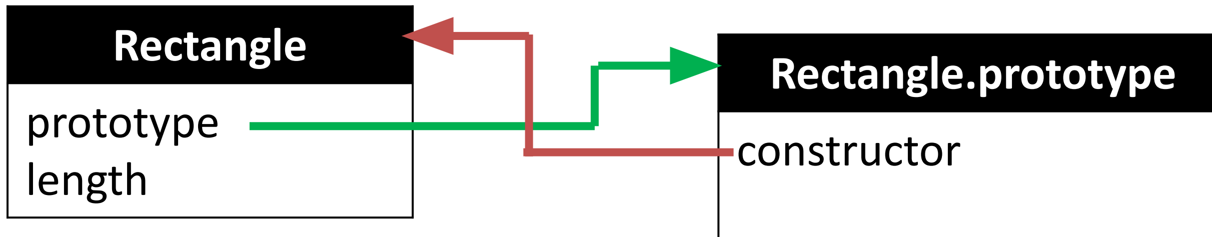
Every function has a *prototype* property and it contains an object.

```
function Rectangle(w, h)  { … }
```



Prototype

prototype **is a property that gets created as soon as you define the function. Its initial value is an object with a single** ***constructor*** **property.**



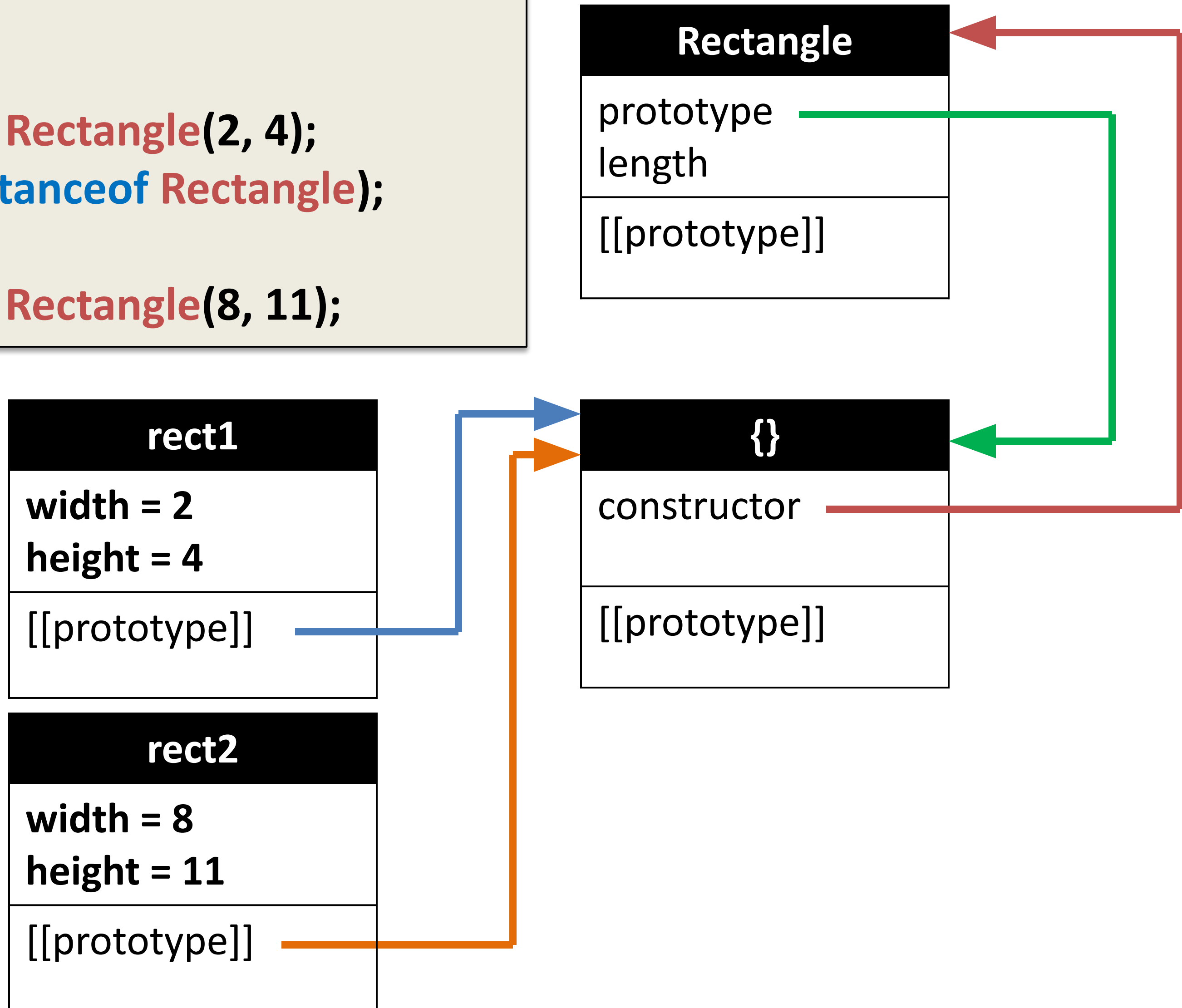

```
Assert(Rectangle.prototype.constructor === Rectangle);
```

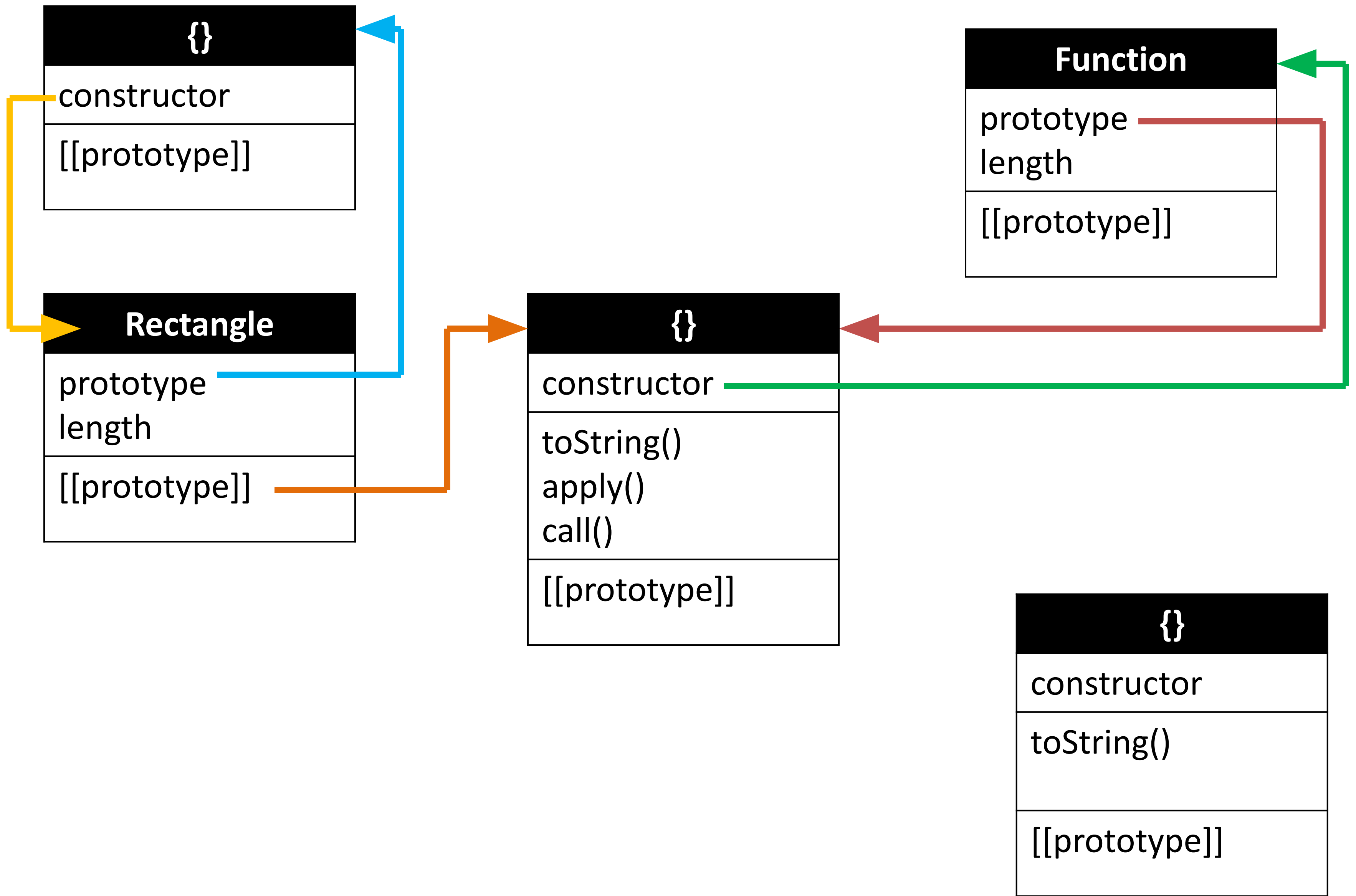
Prototype vs __proto__

The value of the *prototype* property is used to initialize the [[Prototype]] (or __proto__) property of a newly created object.

The [[Prototype]] property is an internal reference to prototype object.

```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}

var rect1 = new Rectangle(2, 4);
Assert(rect1 instanceof Rectangle);

var rect2 = new Rectangle(8, 11);
```

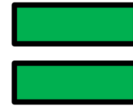



{}
constructor
[[prototype]]
Function
prototype
length
[[prototype]]
Rectangle
prototype
length
[[prototype]]
{}
constructor
toString()
apply()
call()
[[prototype]]
{}
constructor
toString()
[[prototype]]

Constructor function

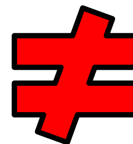
```
var rectangle = new Rectangle(2, 4);
```

```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}
```



```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
    return;
}
```

```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}
```



```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
    return {};
}
```

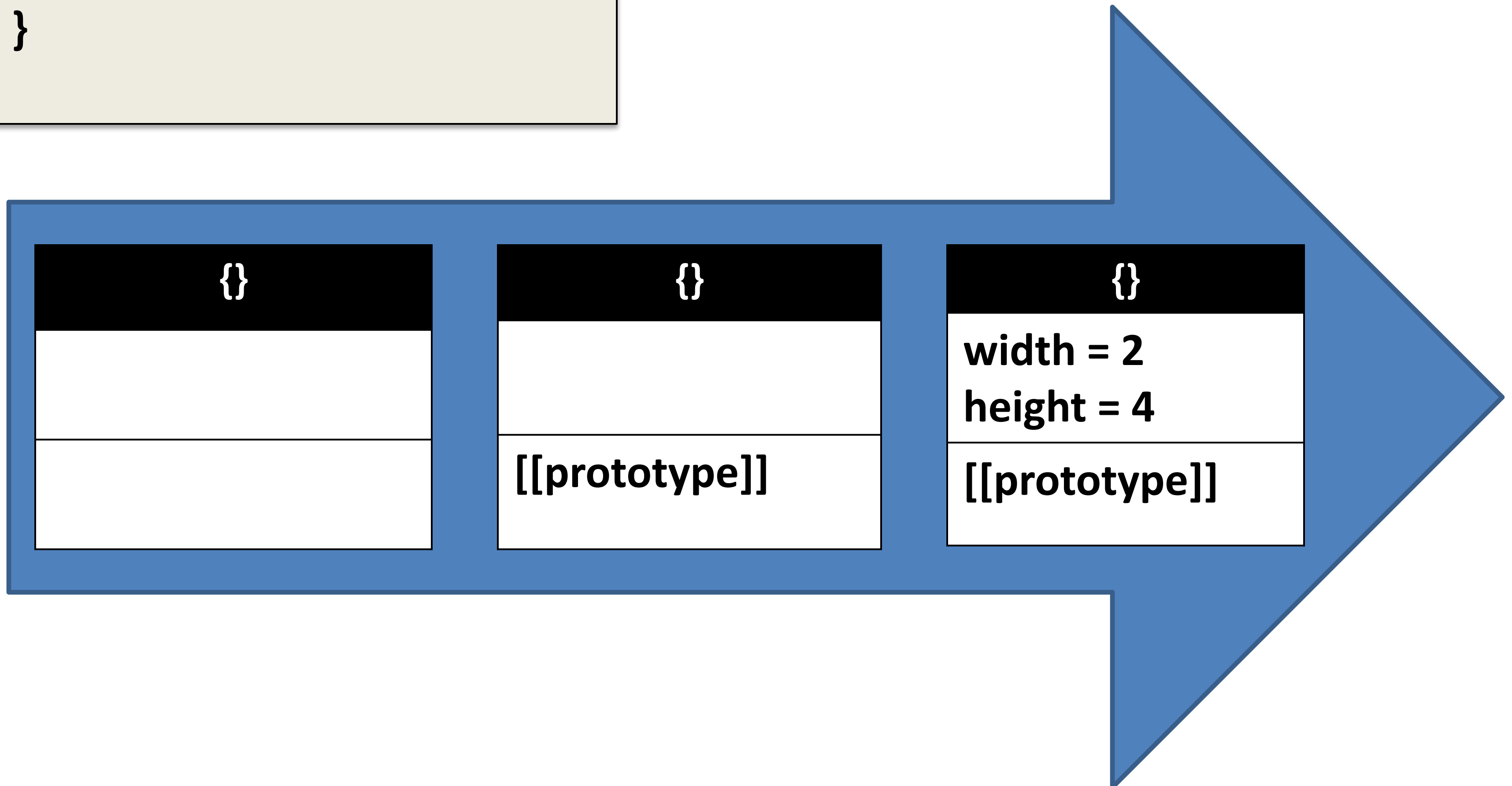
new operator

The new operator creates a new object and invokes a constructor function to initialize it.

```
var obj = new Object();
var date = new Date( );
var rectangle = new Rectangle(2, 4);
```

var rect1 = new Rectangle (2, 4);

```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}
```

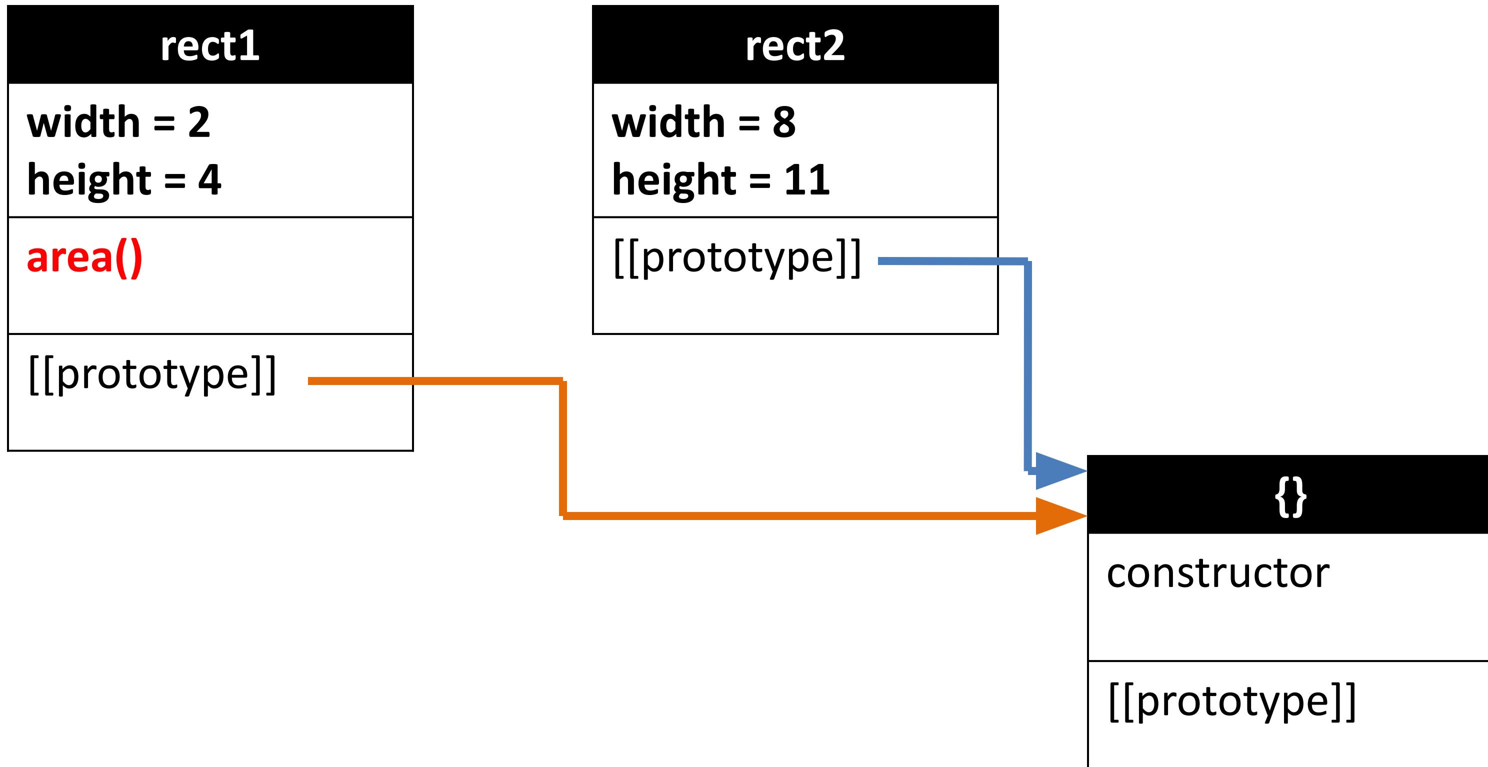


How new work?

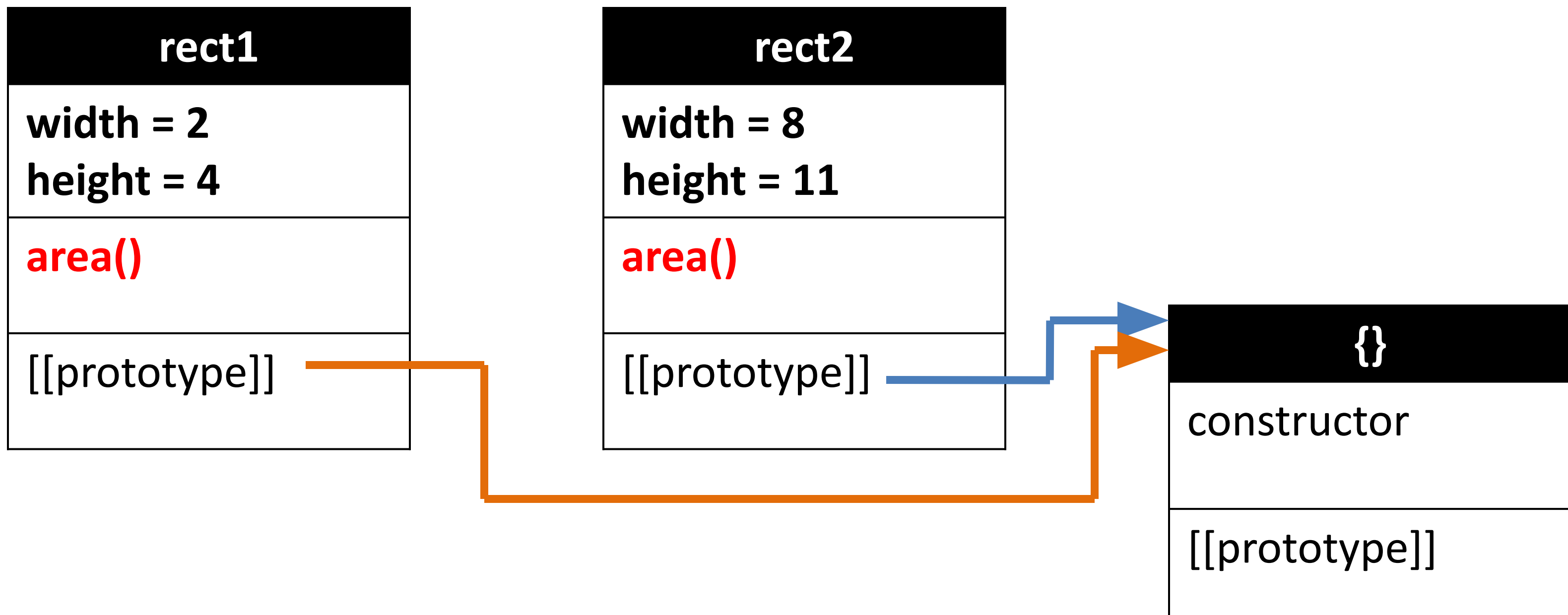
```
function newOperator(Constr, args) {
debugger;
   var thisValue = Object.create(Constr.prototype); // (1)
   var result = Constr.apply(thisValue, args);
   if (typeof result === 'object' && result !== null) {
      return result; // (2)
   }
   return thisValue;
}
```

Methods

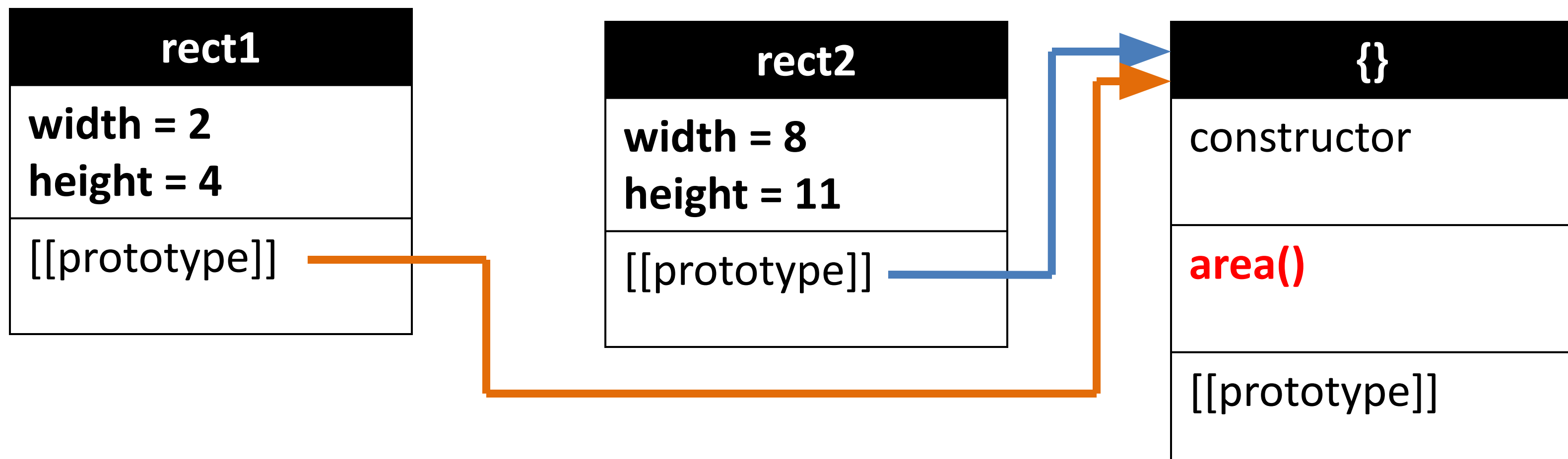
```
var rect1 = new Rectangle(2, 4);
var rect2 = new Rectangle(8, 11);

rect1.area = function() {
      return this.width * this.height;
}
```



```
var rect1 = new Rectangle(2, 4);
var rect2 = new Rectangle(8, 11);

function Rectangle(w, h) {
   this.width = w;
   this.height = h;
   this.area = function() {
      return this.width * this.height;
   }
}
```

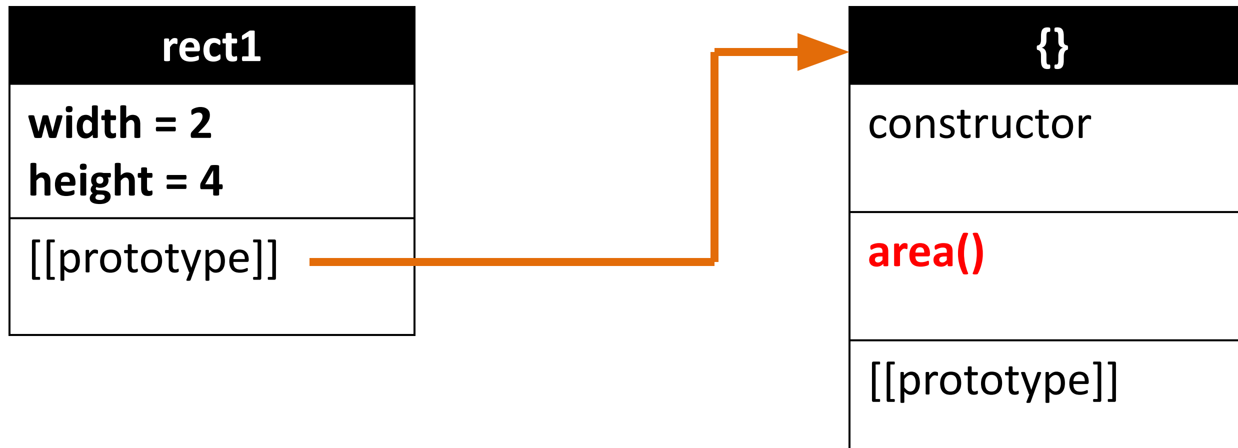


```
var rect1 = new Rectangle(2, 4);
var rect2 = new Rectangle(8, 11);

function Rectangle(w, h) {
   this.width = w;
   this.height = h;
}

Rectangle.prototype.area = function()  {
   return this.width * this.height;
}
```



Using the Prototype's Methods

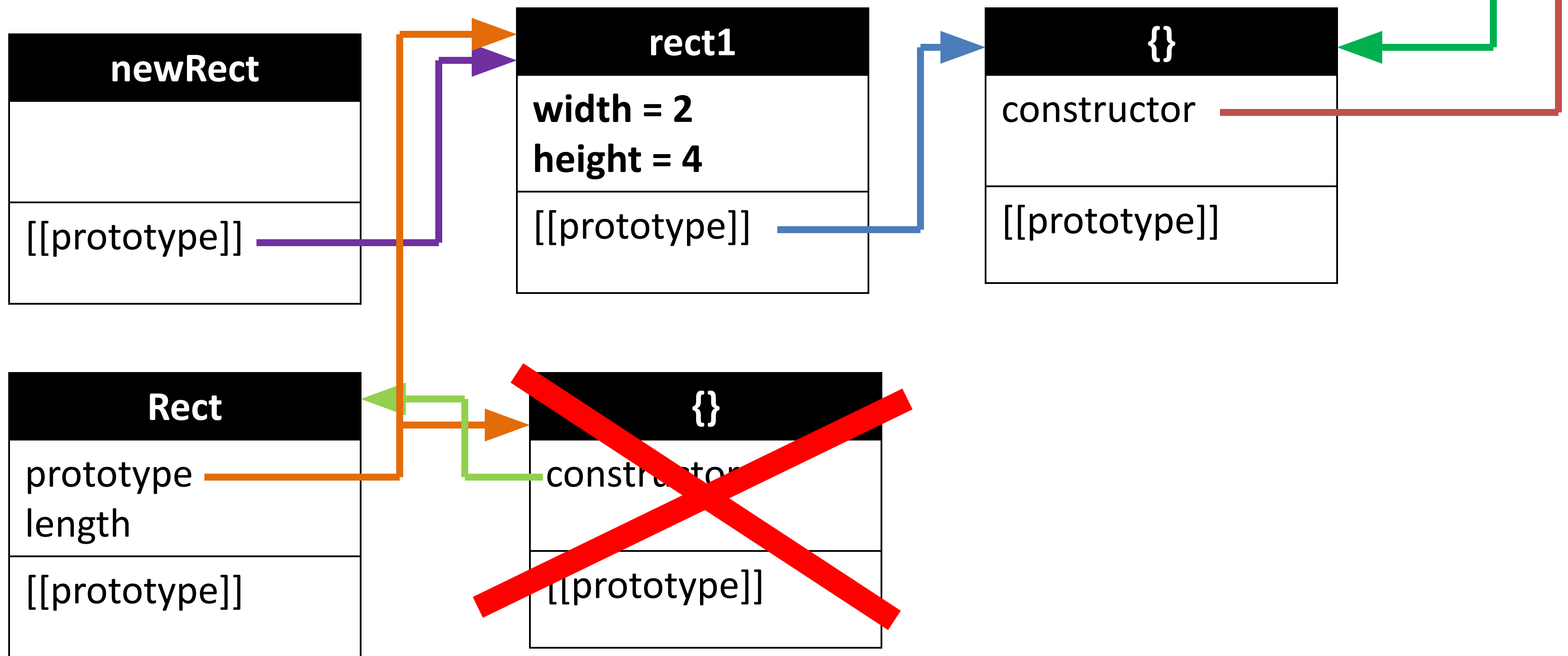
```
var rect1 = new Rectangle(2, 4);

Assert(rect1.area() == 8);
Assert(rect1.hasOwnProperty(“width") == true);
Assert(rect1.hasOwnProperty("area") == false);
Assert(Rectangle.prototype.hasOwnProperty("area") == true);
```



Inheritance features

```
function Rectangle(w, h) {
	this.width = w;
	this.height = h;
}

var rect1 = new Rectangle(2, 4);

function Rect() { };
Rect.prototype = rect1;

var newRect = new Rect();
Assert(newRect.width == 2);
Assert(newRect.height == 4);
```

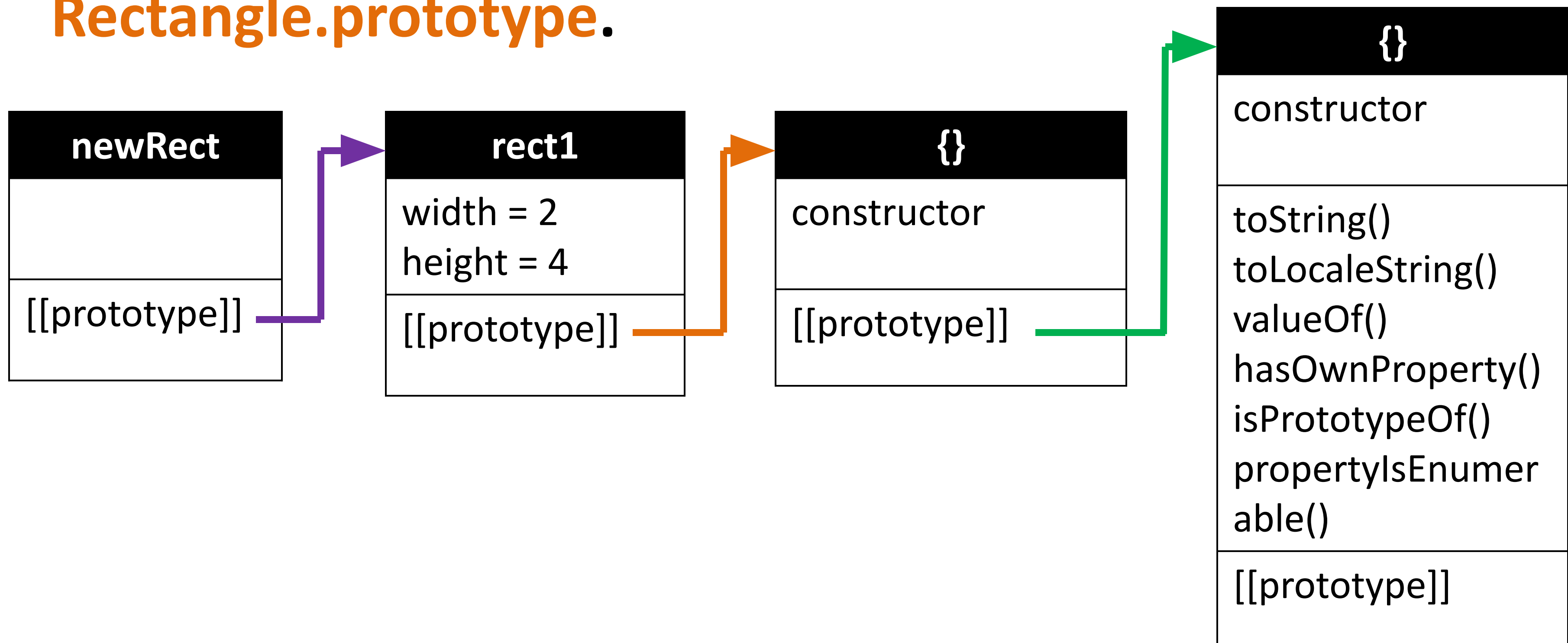
```
var rect1 = new Rectangle(2, 4);

function Rect() { };
Rect.prototype = rect1;

var newRect = new Rect();
Assert(newRect.width == 2);
Assert(newRect.height == 4);
```



Inheritance features

If access of a member of newRect fails, then search for the member in rect1.

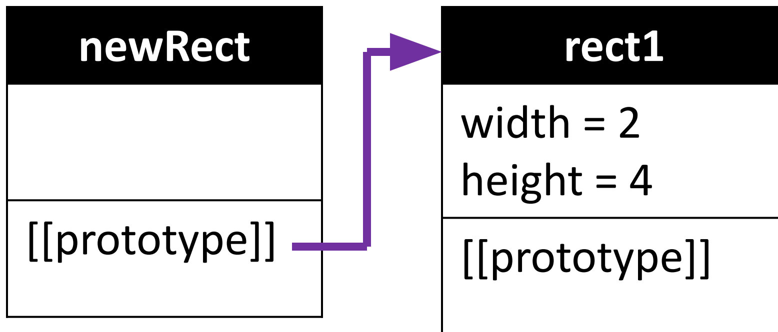
If that fails, then search for the member in Rectangle.prototype.



Inheritance features

Changes in rect1 may be immediately visible in newRect.

Changes to newRect have no effect on rect1.



Setting and Deleting Affects Only Own Properties

```
var proto = { foo: 'a' };
var obj = Object.create(proto);

obj.hasOwnProperty('foo')  // false

obj.foo = 'b';
obj.hasOwnProperty('foo')  // true

delete obj.foo;
delete obj.foo;

obj.hasOwnProperty('foo')  // ???
```

Getters and Setters

```
var obj = {
	get foo() {
		console.log('function call');
	}
};

obj.foo; // call a function without parenthesis
```

Example

instanceof

The instanceof operator tests whether an object has in its prototype chain the prototype property of a constructor.

```
function Car(make, model, year)
{
  this.make = make;
  this.model = model;
  this.year = year;
}
var mycar = new Car("Honda", "Accord", 1998);
var a = mycar instanceof Car;    // returns true
var b = mycar instanceof Object; // returns true
```

WAT

```
function A(){};
//A.prototype = {constructor: A};
var x = new A();

x instanceof A; //true

A.prototype = {};

x instanceof A; //false
```

Prototypal Inheritance

```
function object(o) {
    function f() {}

    f.prototype = o;
    return new f();
}
```

```
var obj = {
    key1: "value1",
    key2: "value2"
}

var obj2 = object(obj);
console.log(obj2.key1);
```

JS EC5

```
var obj = {
    key1: "value1",
    key2: "value2"
}

var obj2 = Object.create(obj);
console.log(obj2.key1);
```

Checklist

What is the difference between this:

```
var Foo = function () {
  this.prop = 10;
};
Foo.prototype.method = function () {
  return this.prop;
};
```

And this one:

```
var Foo = function () {
  this.prop = 10;
  this.method = function () {
    return this.prop;
  };
};
```

Checklist

What is the difference between this:

```
var Foo = function () {
  this.prop = 10;
  this.method = function () {
    return this.prop;
  };
};
```

And this one:

```
var Foo = function () {
  this.prop = 10;
};
Foo.method = function () {
  return this.prop;
};
```

Checklist

```
var Foo = function () {
  this.prop = 10;
};
Foo.prototype.method = function ()
{
  return this.prop;
};
```

What is the difference between this:

```
var foo = Foo ();
```

And this one:

```
var foo = new Foo ();
```

Checklist

```
var Foo = function () {
  var prop = 10;
  return {
    prop: 10
  };
};
Foo.prototype.method = function () {
 // do something
};
```

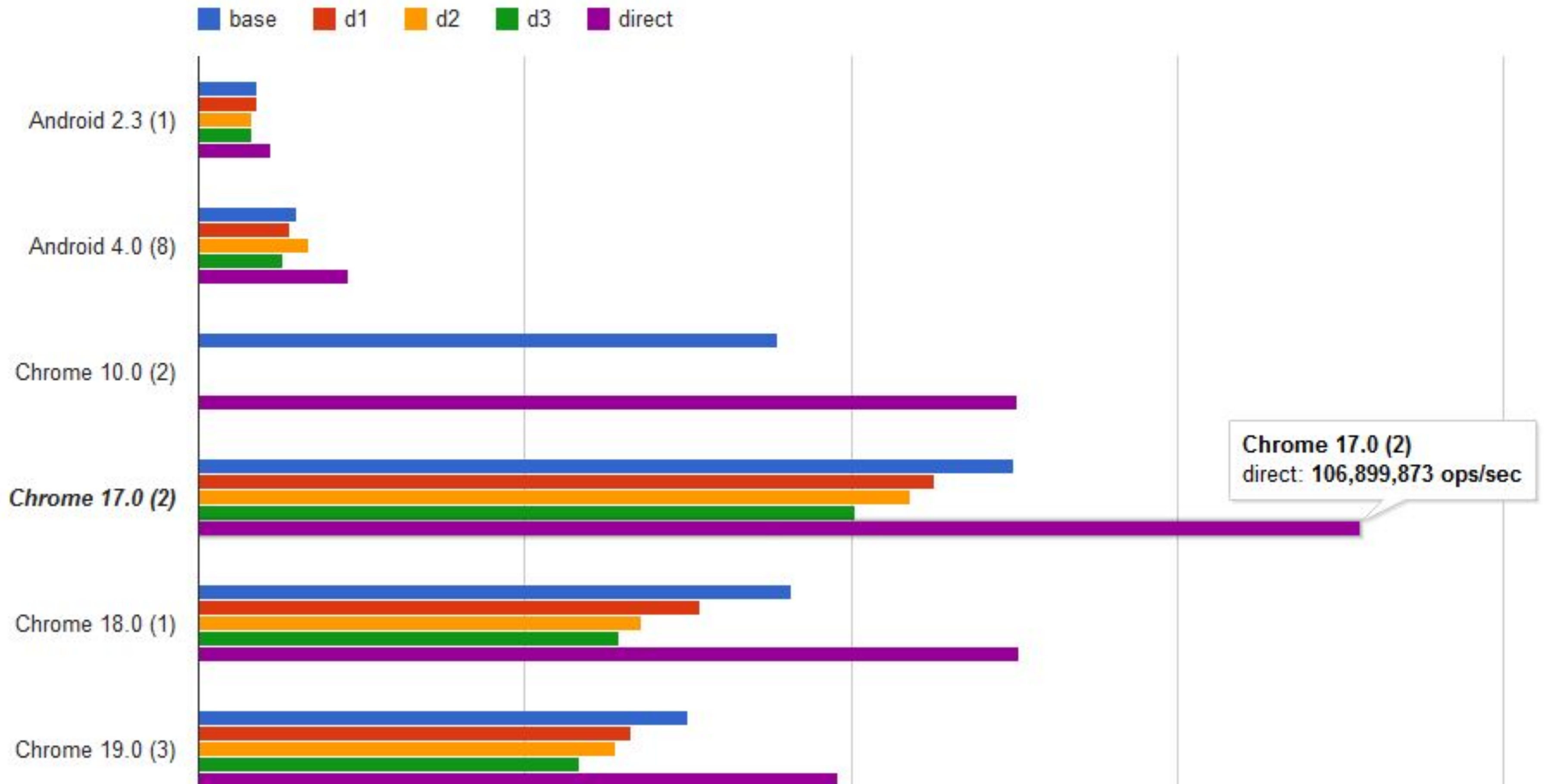
What is the difference between this:

```
var foo = Foo ();
```

And this one:

```
var foo = new Foo ();
```

prototype lookup



http://habrahabr.ru/blogs/javascript/108915/

Pseudoclassical inheritance

Pseudoclassical inheritance

```
function Phone(model, color) {
	this.model = model;
	this.color = color;
}

Phone.prototype.makeCall = function() {…}
Phone.prototype.answer = function() {…}

function MobilePhone(model, color, ringtone) {
	…
}

MobilePhone.prototype = new Phone(???); //wrong
```

```
MobilePhone.prototype = Phone.prototype; //wrong
```

```
extend(MobilePhone,  Phone); //old school
```

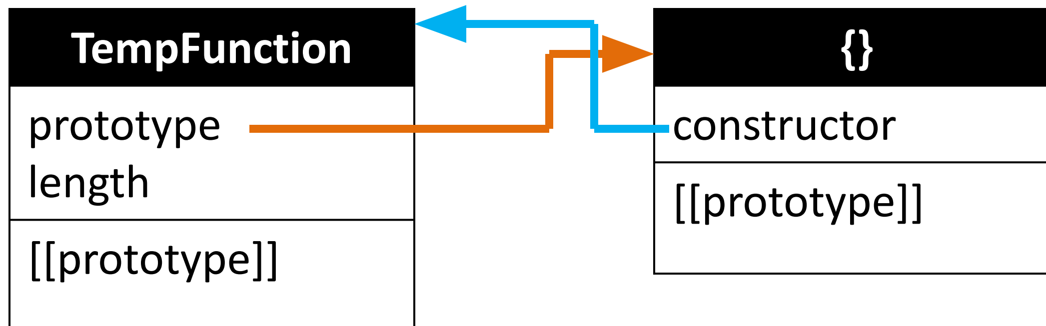
Extend function

```
function extend(MobilePhone, Phone) {
	var TempFunction = function() { };
	TempFunction.prototype = Phone.prototype;
	MobilePhone.prototype = new TempFunction();

	MobilePhone.prototype.constructor = MobilePhone;
	MobilePhone.superclass = Phone.prototype;
}
```

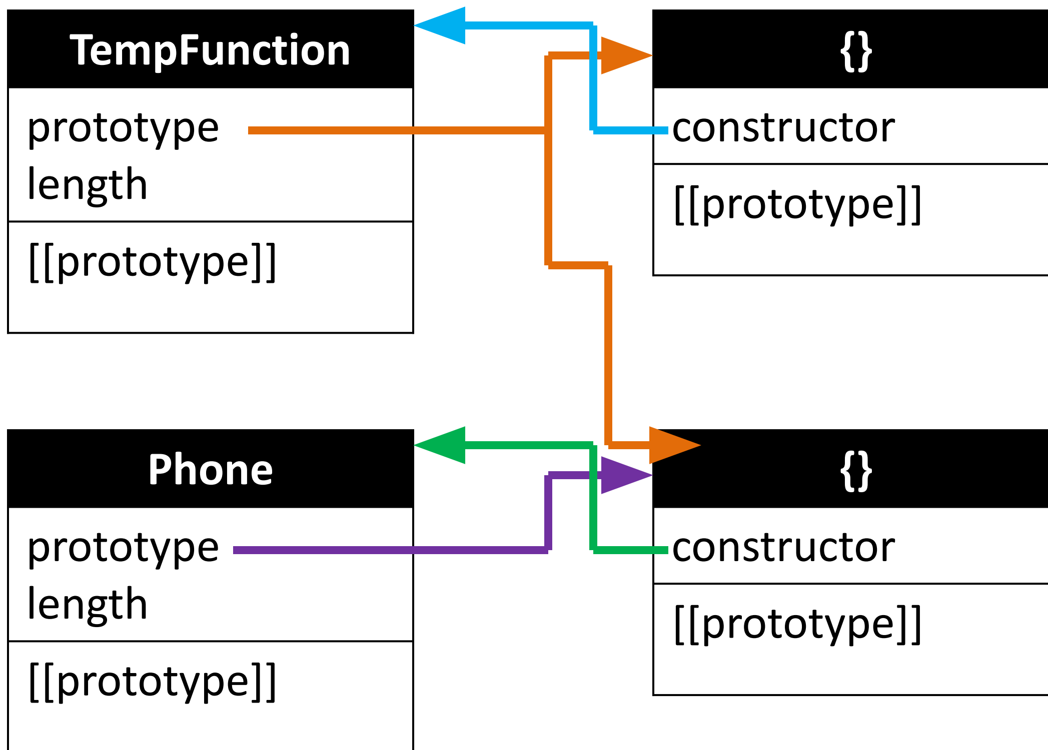
```
extend(MobilePhone, Phone);
```

Extend function

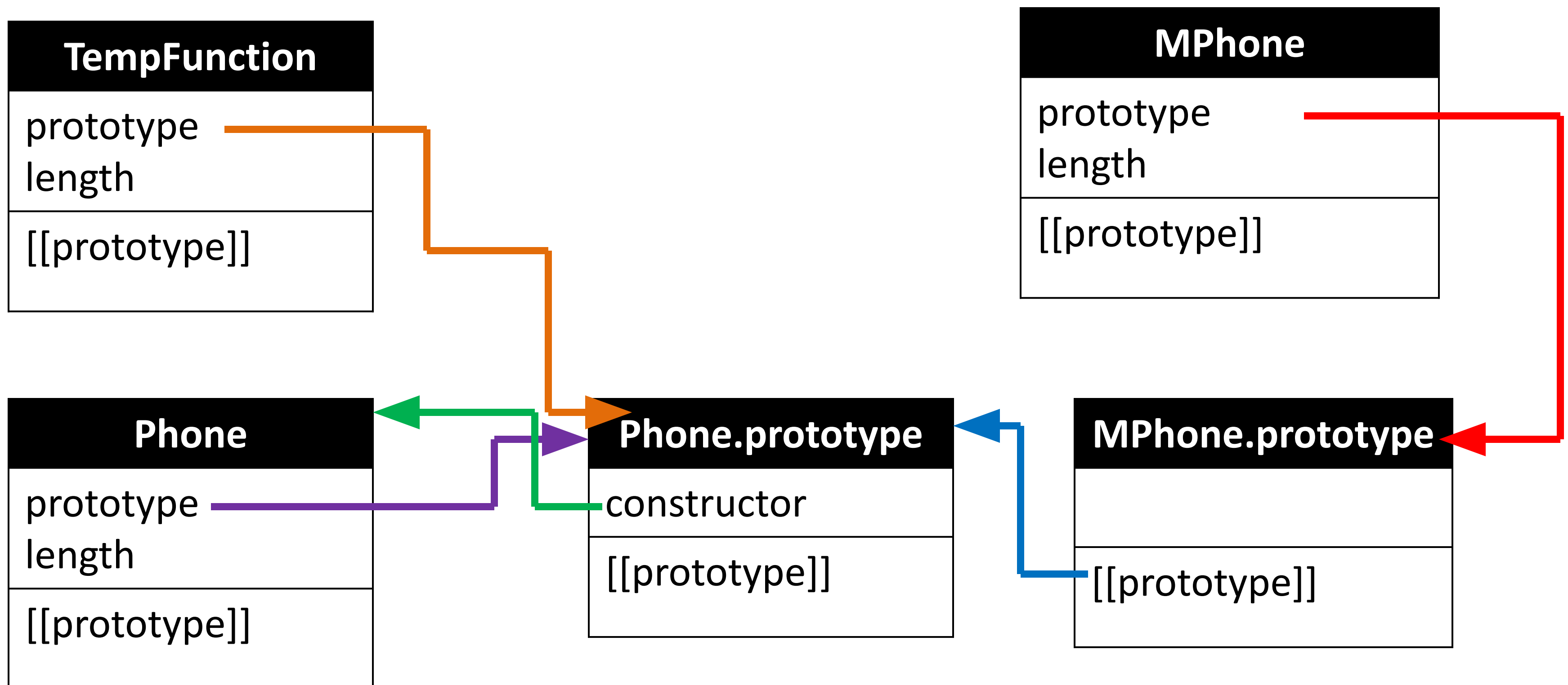
```
function extend(MobilePhone, Phone) {
      var TempFunction = function() { };

      ...
}
```



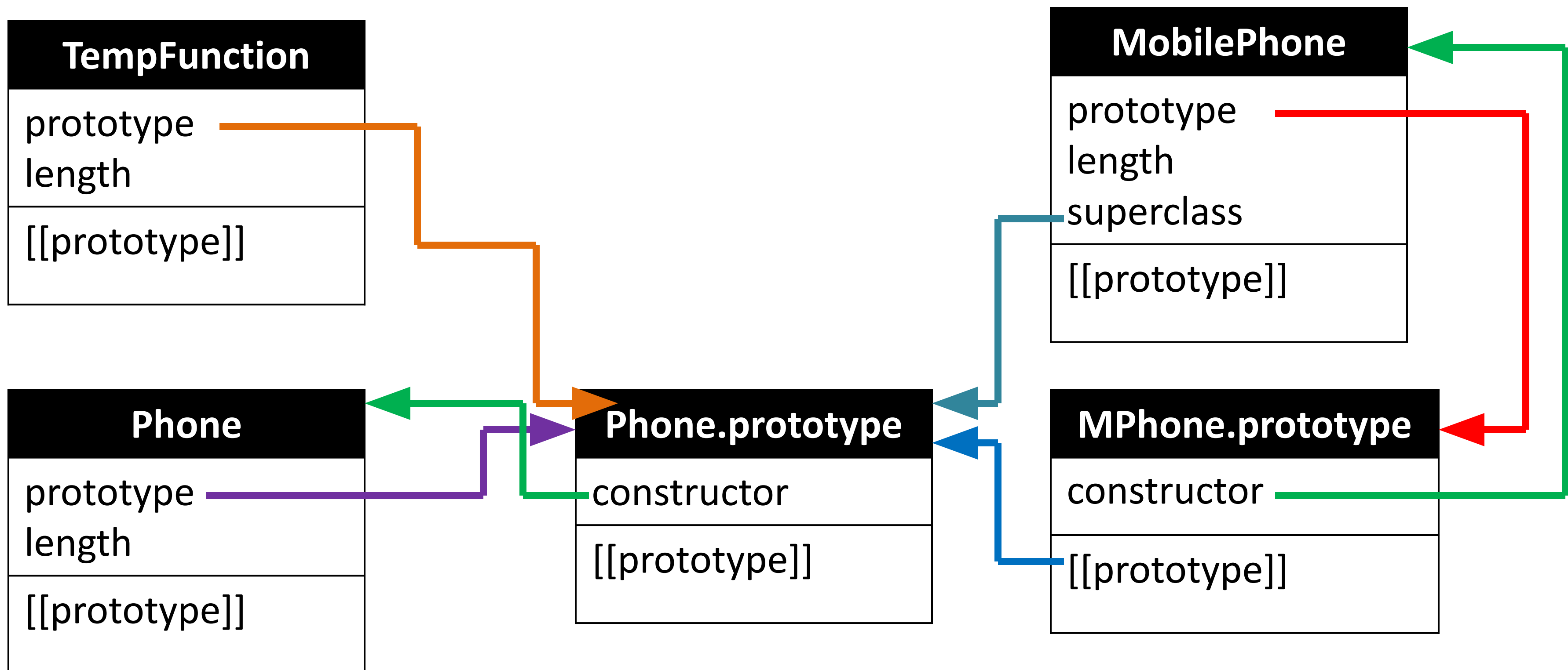
```
function extend(MobilePhone, Phone) {
	var TempFunction = function() { };
	TempFunction.prototype = Phone.prototype;
	...


}
```



```
function extend(MobilePhone, Phone) {
    var TempFunction = function() { };
    TempFunction.prototype = Phone.prototype;
    MobilePhone.prototype = new TempFunction();
    ...
}
```



```
function extend(MobilePhone, Phone) {
    var TempFunction = function() { };
    TempFunction.prototype = Phone.prototype;
    MobilePhone.prototype = new TempFunction();
    MobilePhone.prototype.constructor = MobilePhone;
    MobilePhone.superclass = Phone.prototype;
}
```

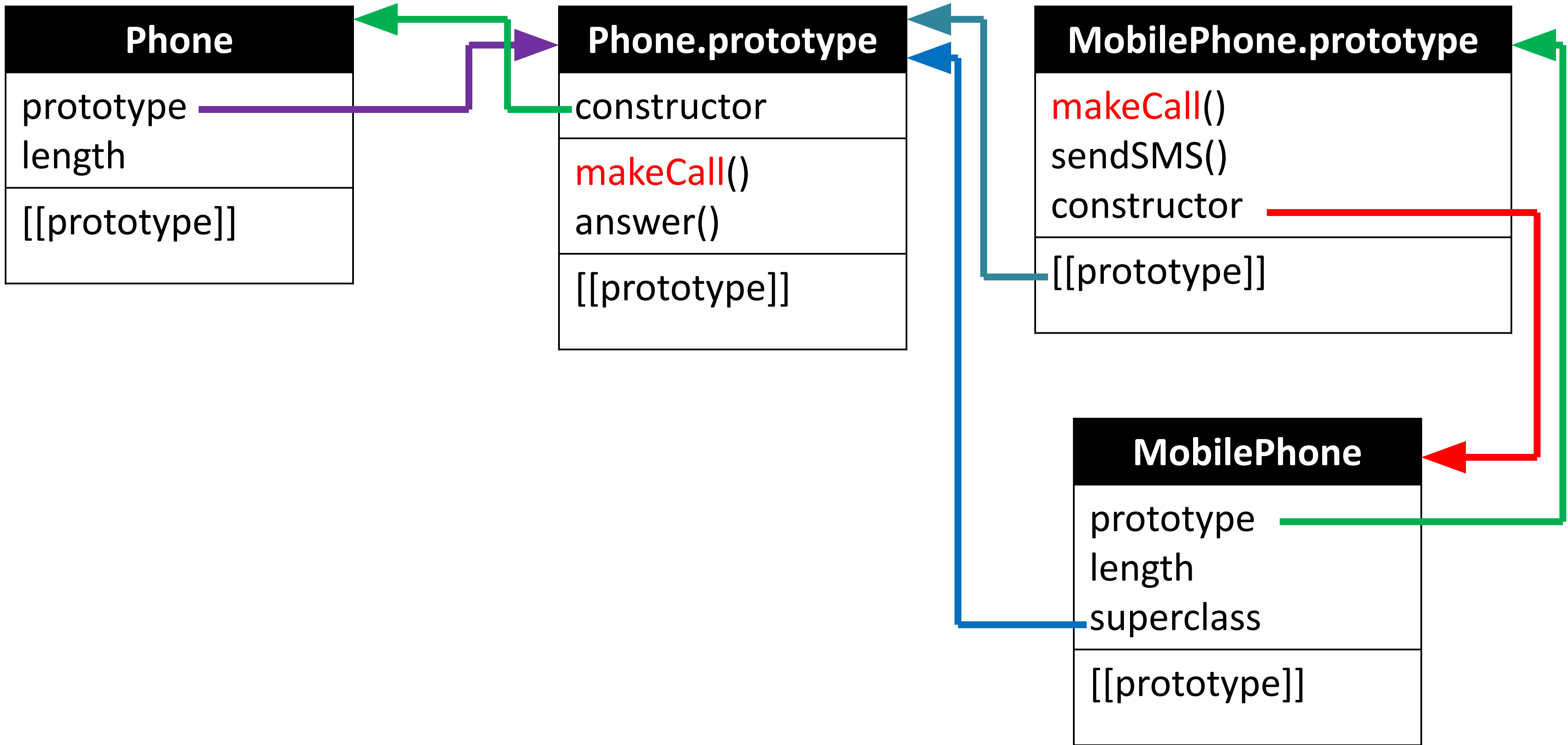


ES5 Extend function

```
function extend(MobilePhone, Phone) {
	MobilePhone.prototype = Object.create(Phone.prototype);

	MobilePhone.prototype.constructor = MobilePhone;
	MobilePhone.superclass = Phone.prototype;
}
```

Pseudoclassical inheritance

```
function Phone(model, color) {
	this.model = model;
	this.color = color;
}
Phone.prototype.makeCall = function() {…}
Phone.prototype.increaseVolume= function() {…}

function MobilePhone(model, color, ringtone) {
	MobilePhone.superclass.constructor.call(this, color, ringtone);
	this. ringtone = ringtone;
}

MobilePhone.prototype.increaseVolume = function(newVolume) {
	MobilePhone.superclass.increaseVolume.call(this, newVolume);
	//…
}

extend(MobilePhone,  Phone);
```


Phone
prototype
length
[[prototype]]
Phone.prototype
constructor
makeCall()
answer()
[[prototype]]
MobilePhone.prototype
makeCall()
sendSMS()
constructor
[[prototype]]
MobilePhone
prototype
length
superclass
[[prototype]]

Private members

```
function Phone(model, color) {
	this.model = model;
	this.color = color;
	var  volume ;
	this.getVolume = function() {return volume;}
	this.setVolume = function(v) {volume  = v;}
}
```

Extending Without Inheriting

```
function borrowMethods(borrowFrom, addTo) {
    var from = borrowFrom.prototype;
    var to = addTo.prototype;
    for (m in from) {
        if (typeof from[m] != "function") continue;
        to[m] = from[m];
    }
}

borrowMethods(Stopwatch , MobilePhone);
```

Mixins

Parasitic inheritance (functional pattern)

Parasitic inheritance

new

constructor

prototype

instanceof

Example

QUESTIONS?

Prefer containment (composition) over inheritance?

Think of containment as a has a relationship. A car "has an" engine, a person "has a" name, etc.

Think of inheritance as an is a relationship. A car "is a" vehicle, a person "is a" mammal, etc.

http://stackoverflow.com/questions/49002/prefer-composition-over-inheritance
http://en.wikipedia.org/wiki/Composition_over_inheritance