Prototypal Inheritance

Single Objects

- all objects in JavaScript are maps (dictionaries) from strings to values
- A (key, value) entry in an object is called a *property*.
- Key is always a string
- The value of a property can be any JavaScript value, including a function.
- Methods are properties whose values are functions

Kinds of Properties

Properties (or named data properties)

Accessors (or named accessor properties)

Internal properties

Object Literals

- Object Literals allows us to create direct instance of `Object`

```
var js = {
      name: ‘Inheritance',
      describe: function () {
            return ‘Concept named '+this.name; // (1)
      }, // (2)
};

> js instanceof Object
true
```

Object Literals

1. Use `this` in methods to refer to the current object (also called the *receiver* of a method invocation).

2. ECMAScript 5 allows a trailing comma (after the last property) in an object literal. Alas, not all older browsers support it. A trailing comma is useful, because you can rearrange properties without having to worry which property is last.

Object Literals

You may get the impression that objects are *only* maps from strings to values ?

There exists a Prototype Relationship between Objects

Fast Access to Properties via Constructors

(.) vs []

- **Accessing Properties via Fixed Keys**
- **Accessing Properties via Computed Keys**

```
> js.name // get property `name`
‘inheritance'
> js.describe // get property `describe`
[Function]
> js.cyient //trying to access a property that does
not exist
undefined
```

Check

…(.)

```
> js.describe // call method describe
Concept named inheritance
> var obj = { hello: 'world' };
> delete obj.hello
true
> obj.hello
undefined
```

Check

…(.)

```
> var obj = { foo: 'a', bar: 'b' };
> obj.foo = undefined;
> Object.keys(obj)
[ 'foo', 'bar' ]
> delete obj.foo
true
> Object.keys(obj)
[ 'bar' ]
```

Check

...[]

The bracket operator lets you compute the key of a property, via an expression

```
> var obj = { someProperty: 'abc' };
> obj['some' + 'Property']
'abc'
> var propKey = 'someProperty';
> obj[propKey] 'abc'
```

...[]

It also allows you to access properties whose keys are not identifiers

```
> var obj = { 'not an identifier': 123 };
> obj['not an identifier']
123
```

Bracket operator coerces its interior to string

```
> var obj = { '6': 'bar' };
> obj[3+3] // key: the string '6'
'bar'
```

…[]

Calling Method via []

```
> var obj = { myMethod: function () { return true } };
> obj['myMethod']()
true
```

Setting Properties with []

```
> var obj = {};
> obj['anotherProperty'] = 'def';
> obj.anotherProperty
'def'
```

Sharing of Properties | Problem

```
var js = {
      name: ‘Inheritance',
      describe: function () {
            return ‘Concept named '+this.name;
      },
};
```

```
var ajs = {
      name: ‘Dependency Injection',
      describe: function () {
            return ‘Concept named '+this.name;
      },
};
```

Sharing of Properties | Problem

```
var ConceptProto = {
      describe: function () {
            return ‘Concept named '+this.name;
      },
};
```

Check

```
var ajs = {
      __proto__:ConceptProto,
      name: ‘Dependency Injection',
};
```

```
var js = {
      __proto__:ConceptProto,
      name: ‘inheritance',
};
```

Getting and Setting of prototype

```
var ConceptProto = {
      describe: function () {
            return ‘Concept named '+this.name;
      },
};
var js = Object.create(ConceptProto);
js.name = ‘inheritance’
```

```
> Object.getPrototypeOf (js) === ConceptProto
true
```

Check

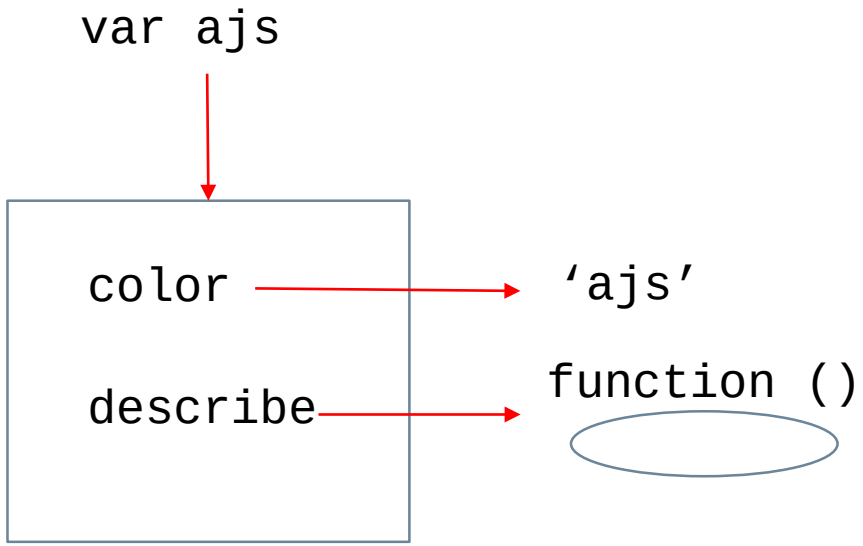
Constructors…

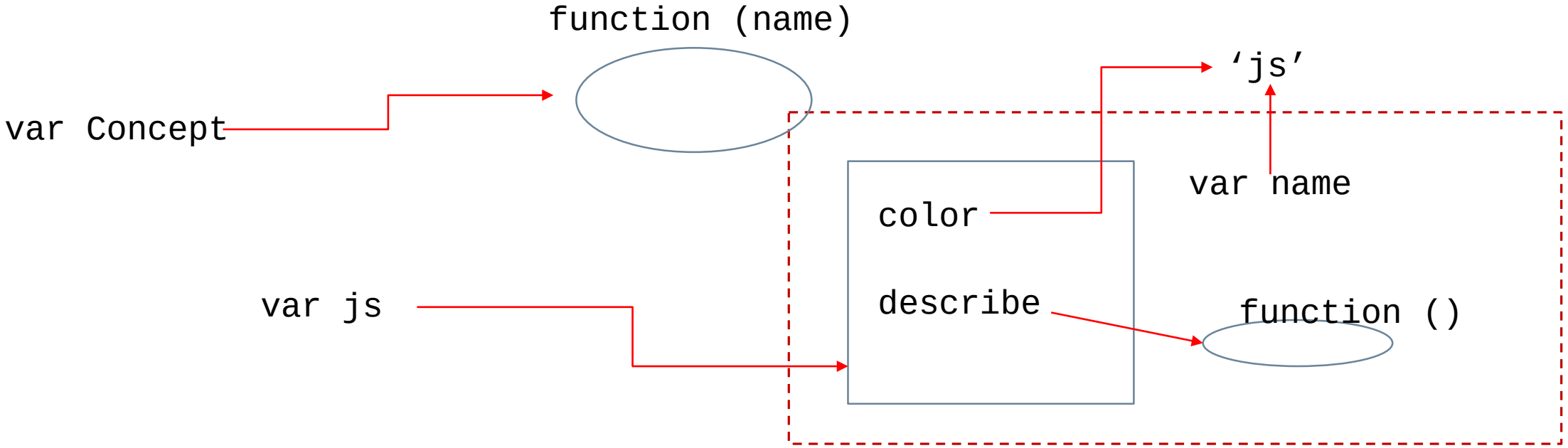
```
function Concept(name){
      this.name = name;
      this.describe = function (){
            return ‘Concept named ‘ + this.name
      };
}
var js = new Concept(‘js’);
Console.log(js instanceof Concept)
```

Check

```
true
```

```
var Concept = function(name){
        this.name = name;
        this.describe = function (){
                return 'Concept named ' +
this.name
        };
}
var js = new Concept('js');
var ajs = new Concept('ajs');
```

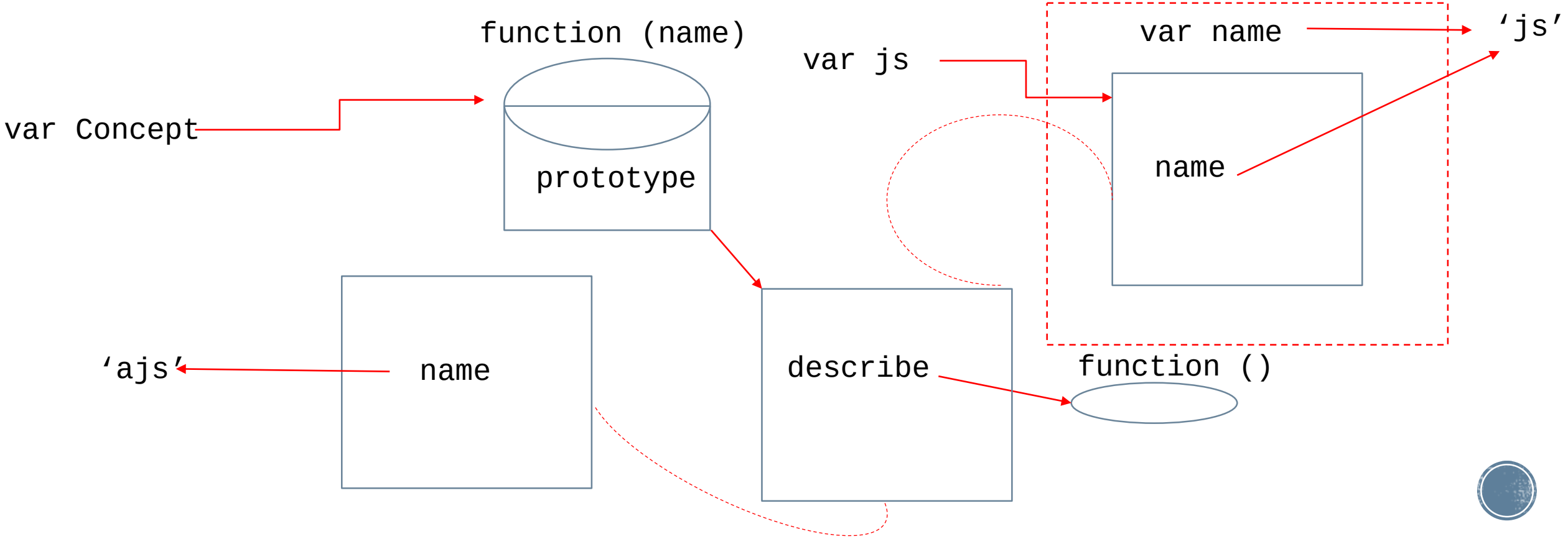




…Constructors

```
//Instance Specific Properties
function Concept(name){
      this.name = name;
}
//Shared Properties
Concept.prototype.describe = function (){
            return ‘Concept named ‘ + this.name
      };
```

```
function Concept(name){
        this.name = name;
}
Concept.prototype.describe = function (){
                return 'Concept named ' + this.name
        };
var js = new Concept('js');
var ajs = new Concept('ajs');
```



…Constructors

```
> var js = new Concept('js')
undefined
> js.name
"js"
> Object.keys(js)
["name"]
> Object.keys(Object.getPrototypeOf(js))
["describe"]
```

…Constructors | instances

Concept

prototype	

```
function Concept(name){
    this.name = name;
}
```

Concept.prototype

describe	function(){…}

js

__proto__	
name	‘js’

ajs

__proto__	
prototype	‘ajs’

...Constructors | instances

Overloaded/Confused Terminology

Prototype # 1: prototype relationship between objects
Prototype # 2: property `prototype` of Constructors (prototype of all instances)

```
> var proto = {};
> var obj = Object.create(proto);
> Object.getPrototypeOf(obj) === proto
true

> function C() {}
> Object.getPrototypeOf(new C()) === C.prototype
true
```

Derive Subject from Concept

```
function Concept(name){
      this.name = name;
}
Concept.prototype.provide = function (adjective){
            console.log(this.name + ‘provides’ + adjective
};
Concept.prototype.describe = function (){
            return ‘Concept named ‘ + this.name
};
```

Derive Subject from Concept

Subject(name, title) is like Concept, except

Additional instance property: title

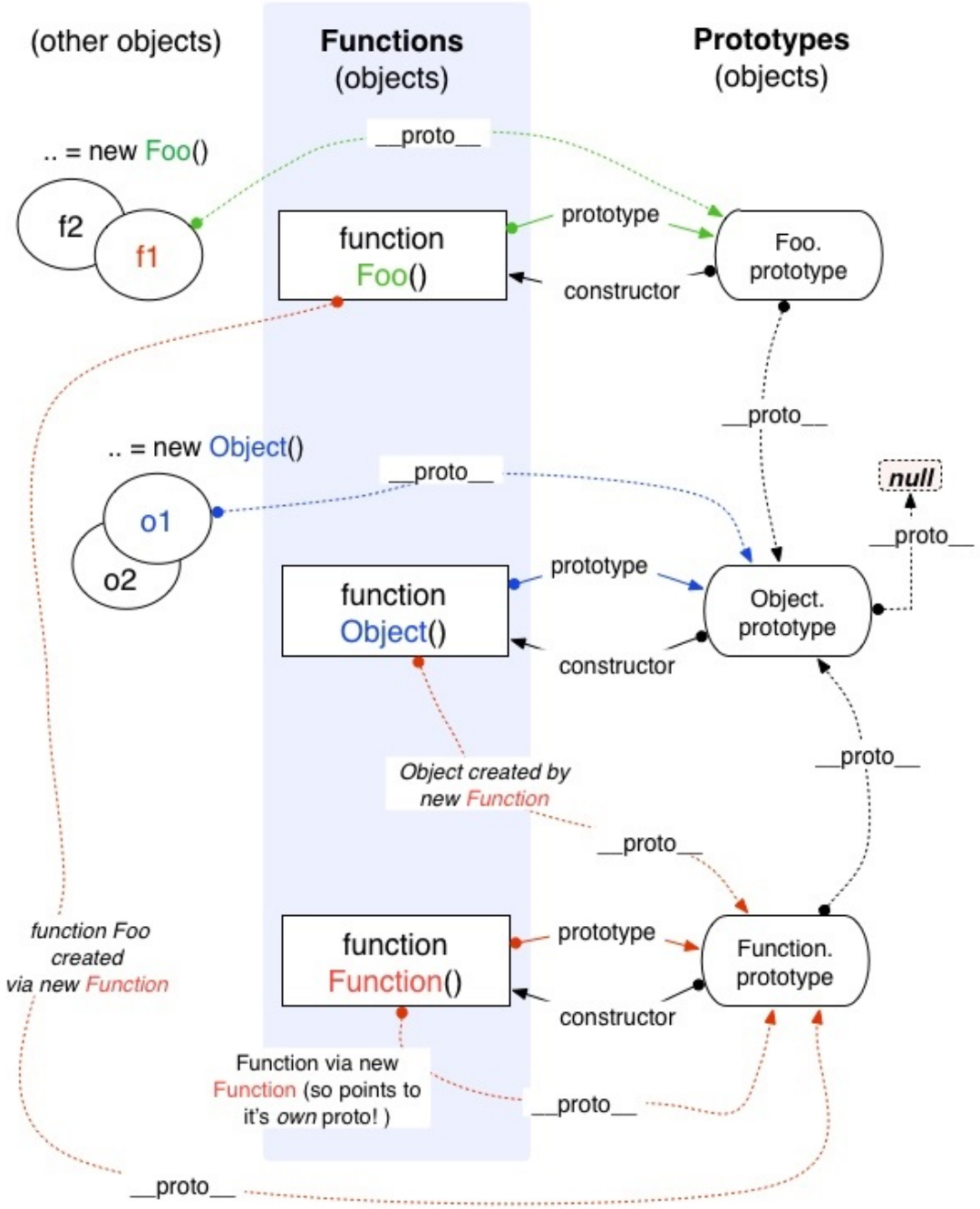
describe() returns ‘Concept named <name> (<title>’

Derive Subject from Concept

```
function Subject(name, title){
        Concept.call(this,name); // (1)
        this.title = title; //(2)
}
Subject.prototype = Object.create(Concept.prototype);//(3)
Subject.prototype.describe = function (){ // (4)
                return Concept.prototype.describe.call(this) //(5)
                        + ‘ (‘ +this.title + ’)’;
};
```

1. Inherit instance properties
2. Create the instance property title
3. Inherit prototype properties
4. Override method Concept.prototype.describe
5. Call Overridden method

Check


(other objects)
Functions
(objects)
Prototypes
(objects)
.. = new Foo()
f2
f1
__proto__
function
Foo()
prototype
constructor
Foo.
prototype
__proto__
.. = new Object()
o1
o2
__proto__
null
__proto__
function
Object()
prototype
constructor
Object.
prototype
Object created by
new Function
__proto__
__proto__
function Foo
created
via new Function
function
Function()
prototype
constructor
Function.
prototype
Function via new
Function (so points to
it's own proto!)
__proto__
__proto__