WFDD Week 6

Google Camp of Tongji University

OUTLINE

JavaScript part:

- ***this*** keyword, prototypes

CSS part:

- CSS3, transitions & animations

THIS KEYWORD

In direct function call:

this === *global object*

In browsers, `window` is the global object.

THIS KEYWORD

direct function call:

```
var x = 1;
alert(window.x); // 1
```

```
function test(){
  alert(this.x);
}
test(); // 1
```

THIS KEYWORD

direct function call:

```
var x = 1;
(function(){
  var x = 0;
  function test(){
    alert(this.x);
  }
  test(); // 1
})();
```

THIS KEYWORD

In object method:

this === *the object itself*

THIS KEYWORD

object method:

```
var x = 2;
function test(){
  alert(this.x);
}
var o = {};
o.x = 1;
o.m = test;
o.m(); // 1
test(); // 2
```

THIS KEYWORD

How to access object in inner function?

```
var obj = {
  _value: 10,
  foo: function(){
    this._value = 1; // OK
    $(..).click(function(){
      // this !== obj
    });
  }
};
```

THIS KEYWORD

Solve: by using the closure feature.

```
var obj = {
  _value: 10,
  foo: function(){
    var that = this;
    this._value = 1; // OK
    $(..).click(function(){
      // that === obj
    });
  }
};
```

THIS KEYWORD

```
func.apply(obj, …)
func.call(obj, …)
```

this === *obj*

THIS KEYWORD

apply & call:

```
function test(){
    alert(this.x);
}
test.apply({x:1}, [arg1, arg2]); // 1
test.call({x:2}, arg1, arg2); // 2
```

THIS KEYWORD

In the constructor function:

this === *the new object*

THIS KEYWORD

constructor function:

```
function test(){
  this.x = 1;
}
var o = new test();
alert(o.x); // 1
```

NEW KEYWORD

Wait, what is `new`?

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

The procedure is like this:

```
var foo = {};
/* {some magic code here} */
Foo.call(foo);
```

CLASS

Let’s try to make a class and instances.

```
function People(name) {
  this.name = name;
}
var p1 = new People("foo");
var p2 = new People("bar");
p1.name // foo
p2.name // bar
```

Why it works?

CLASS

A more easily understanding rewrite:

```
function People(name) {
  this.name = name;
}
var p1 = {};
People.call(p1, "foo");
// p1.name = "foo"
var p2 = {};
People.call(p2, "bar");
// p2.name = "bar"
```

ADD METHOD

how to add methods?

```
p1.sayHi()
```

ADD METHOD

a correct way:

```
function People(name) {
  this.name = name;
  this.sayHi = function() {
    alert("hi");
  }
}
var p1 = new People("foo");
p1.sayHi() // hi
```

ADD METHOD

Problem: waste memory, no sharing.

```
var p1 = new People("foo");
var p2 = new People("bar");
```

```
p1.name === p2.name
// of course false

p1.sayHi === p2.sayHi
// false
```

ADD METHOD

incorrect way:

```
function People(name) {
  this.name = name;
}
People.sayHi = function() {
  alert("hi");
}
var p1 = new People("foo");
p1.sayHi() // Error
People.sayHi() // hi
```

QUESTION

1. How to inherit class? (subclass)

2. Can we share variables among instances of the same class? (For example, shared functions)

INHERIT

A simple solution: call constructor.

(We will see better solutions later)

```
function Animal() {
  this.species = "动物";
}
function Cat(name) {
  Animal.apply(this, arguments);
  this.name = name;
}
```

SHARE VARIABLES

A simple solution: create an object and make reference to it.

```
var sharedObj = {
  species: "动物",
  sayHi: function() { .. }
};
function Animal() {
  this.shared = sharedObj;
}
var a1 = new Animal();
var a2 = new Animal();
a1.shared.sayHi === a2.shared.sayHi
// true
```

SHARE VARIABLES

Problem1: pollute the global. Let's rewrite it.

```
function Animal() {
  this.shared = Animal.sharedObj;
}
Animal.sharedObj = {
  species: "动物",
  sayHi: function() { .. }
};
var a1 = new Animal();
var a2 = new Animal();
a1.shared.sayHi === a2.shared.sayHi
// true
```

SHARE VARIABLES

Problem2: too much code

Problem3: we should distinguish explicitly whether a variable is shared or not.

```
subclass.shared.parentMethodA();
subclass.shared.parentMethodB();
subclass.shared.parentMethodC();
subclass.shared.shared.superMethodZ();
subclass.localMethodD();
```

SHARE VARIABLES

Better solutions?

Look at the fact below first.

```
var a1 = {x:1};
var a2 = {y:2};
a1.toString === a2.toString
// true
// Notice: we are comparing functions,
not results. It means these two
toString() are the same function.
```

What's the principle behind it?

PROTOTYPE CHAIN

When **accessing** properties, JavaScript look up it through *the prototype chain*.

An experiment about accessing:

```
var foo = {};
foo.bar // undefined
foo.__proto__ = {bar:1};
foo.bar // 1 (access)
foo.bar = 2;
foo.__proto__.bar // still 1
```

PROTOTYPE CHAIN

An experiment about *prototype chain*:

```
var foo = {};
foo.bar // undefined
foo.__proto__ = {};
foo.__proto__.__proto__ = {bar:1};
foo.__proto__.bar // 1
foo.bar // 1
```

PROTOTYPE CHAIN

Let's use *prototype chain* to share variables.

```
function Animal() {
  this.__proto__ = Animal.sharedObj;
}
Animal.sharedObj = {
  species: "动物",
  sayHi: function() { .. }
};
var a1 = new Animal();
var a2 = new Animal();
// a1.shared.sayHi === a2.shared.sayHi
a1.sayHi === a2.sayHi
// true
```

__PROTO__

Wait a minute.

In fact `__proto__` should be a private property(= *[[Prototype]]*) according to ECMAScript5. We should not modify or access `__proto__` directly.

The standard provides another way:

`prototype` property

PROTOTYPE

prototype is a object property of all functions.

```
var Foo = function(){};
// Foo.prototype = {constructor:Foo}

Foo.prototype.constructor === Foo
// true

var foo = new Foo();
foo.__proto__ === Foo.prototype
// true
```

NEW KEYWORD

```
var Foo = function(){};
foo = new Foo();
```

What `new` operator really does:

```
var foo = {};
/* {some magic code here} */
Foo.call(foo);
```

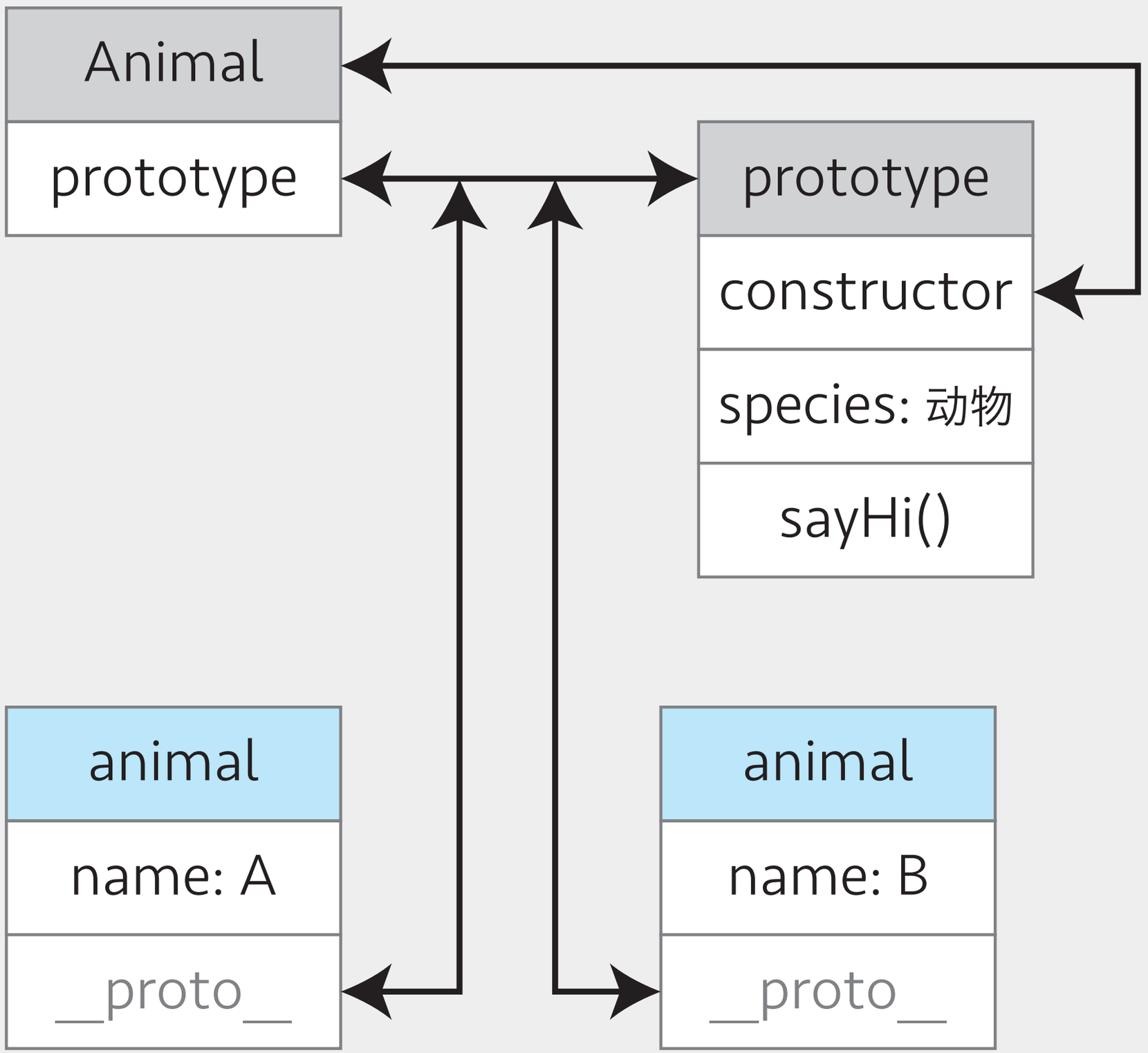
NEW KEYWORD

```
var Foo = function(){};
foo = new Foo();
```

What `new` operator really does:

```
var foo = {};
foo.__proto__ = Foo.prototype;
Foo.call(foo);
```

MIX THEM


Animal
prototype
prototype
constructor
species: 动物
sayHi()
animal
name: A
__proto__
animal
name: B
__proto__

CLASS

```
function Animal(name) {
    this.name = name;
}
Animal.prototype.species = '动物';
Animal.prototype.sayHi = function() {
    alert('Hi');
}
var a1 = new Animal('A');
var a2 = new Animal('B');
a1.sayHi === a2.sayHi // true
a1.name === a2.name   // false
```

THIS IN PROTOTYPE

```
function Animal(name) {
    this.name = name;
}
Animal.prototype.species = '动物';
Animal.prototype.whoami = function() {
    alert(this.name);
}
var a1 = new Animal('A');
var a2 = new Animal('B');
a1.whoami()   // A
a2.whoami()   // B
```

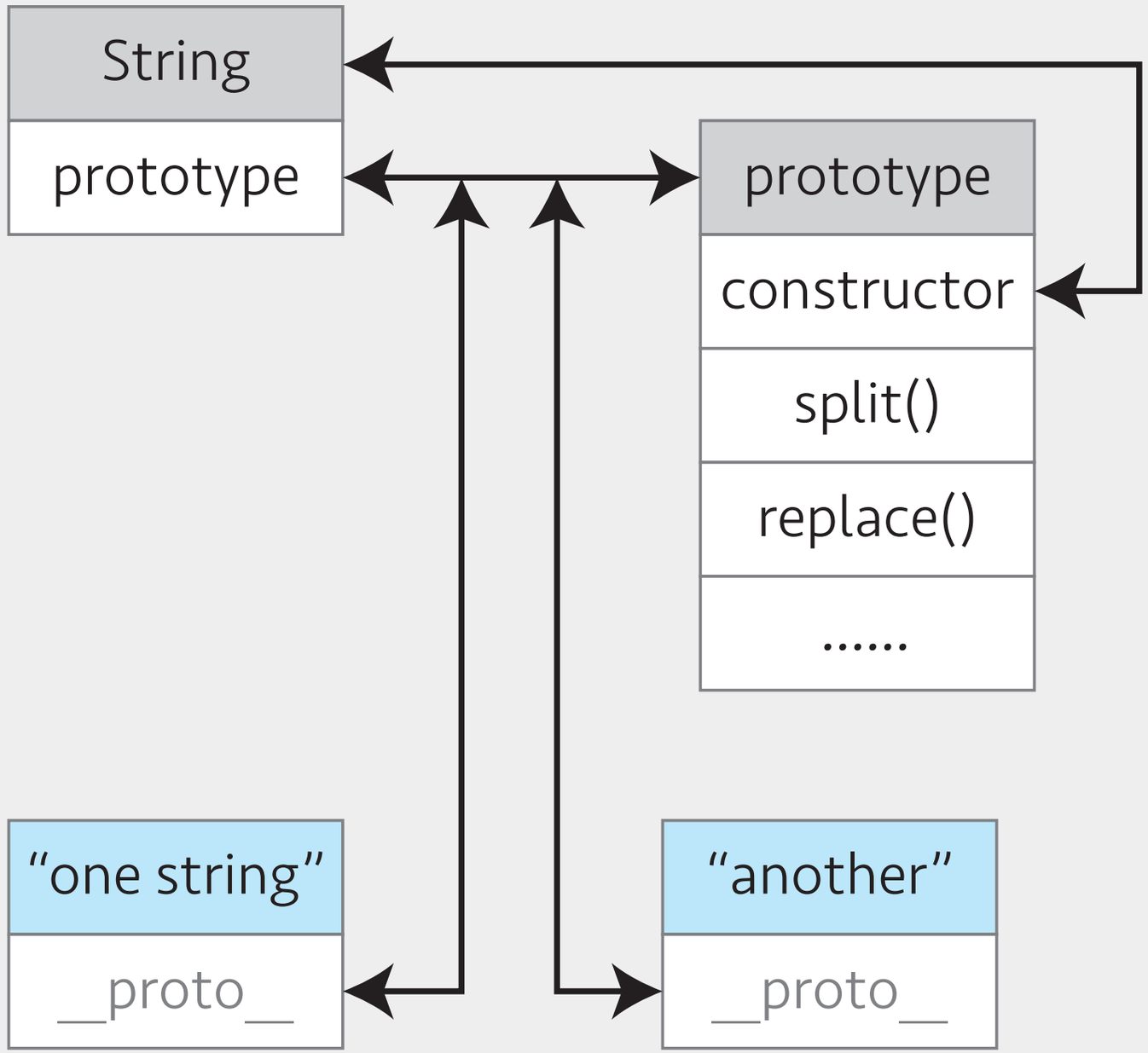
INHERIT

How to use prototype to inherit class?

→ This week’s assignment

THE MAGIC PROTOTYPE

Prototype can do much more than saving memory & sugaring syntax.



THE MAGIC PROTOTYPE

What if we modify `String.prototype`?

```
String.prototype.size = function() {
    return this.length;
}
"mystring".size() // 8
```

CSS3

border-radius

[<length> | <percentage>]{1,4} [/
[<length> | <percentage>]{1,4}]?

```
border-radius: 5px
```

CSS3

box-shadow

none | [inset? && [<offset-x> <offset-y> <blur-radius>? <spread-radius>? <color>?]]#

```
box-shadow: 0 0 10px #000
```

CSS3

text-shadow

none | [<shadow>,]* <shadow>

<shadow> is: [<color>? <offset-x> <offset-y> <blur-radius>? | <offset-x> <offset-y> <blur-radius>? <color>?]

```
text-shadow: 1px 1px 3px #000
```

transform

- translate()
- rotate()
- scale()
- skew()
- matrix()

```
transform: scale(2) rotate(30deg)
```

TRANSITION

transition

[none | <single-transition-property>] || <time> || <timing-function> || <time>

transition-property

transition-duration

transition-timing-function

transition-delay

EASING

linear

ease-in

ease-out

ease-in-out

http://easings.net

https://developer.mozilla.org/zh-CN/docs/Web/CSS/transition-timing-function

ANIMATION

→ This week’s assignment