

Javascript Notes - Objects

E-Applications Spring 2015

M. Lampis

Objects in javascript

- Roughly: Objects are **maps**
 - Keys are strings (not necessarily valid identifiers)
 - Each key is mapped to a **property**
- What about methods?
 - Methods are simply properties whose value happens to be a function.

Object literals

- Simplest way to initialize an object

```
var myobj = {  
    key1: "value1",  
    key2: 23,  
    method1: function() { alert("Hello!"); },  
}
```

- Note trailing comma (not necessary)

Accessing properties

- Properties can be read/written using the . operator

```
myobj.key2 // == 23
```

```
myobj.key1 = [1,2,3]; //OK, "value1" is deleted
```

```
myobj.method1(); //call the method
```

```
myobj.method1 = function() {alert("Bye");}; //OK!
```

Accessing Properties

- Properties can be written even if the key does not exist

```
myobj.key3 = "newkey"; //OK!
```

- Result -> Often we just start with empty object and then add properties at will, instead of using object literals.

Deleting Properties

- A property can be deleted using `delete`
`delete myobj.key1; //now myobj.key1==undefined`
- Not the same as setting a property to undefined
- Use sparingly...
 - Low efficiency in most implementations

The [] operator

- Object properties can also be accessed using []

```
myobj["key1"] === myobj.key1; //TRUE!
```

- Why use this instead of . ?

- Allows any string as key

```
myobj["a key with spaces"] = 74; //OK
```

- Allows computed keys

```
myobj["key"+i] = 75; //myobj.key1? myobj.key2?...
```

```
myobj["method"+1](); //what will this do?
```

this

- A special variable **this** is "always" defined
- Idea: when we call a method, this refers to the object to which the method belongs

```
var myobj = {key1: 23, method1: function() {  
    alert(this.key1); }, };
```

- (In sloppy mode) this is defined also for normal functions. It is equal to the global object (window)

Setting this

- Consider the following example

```
var myobj = {key1: 23, method1: function() {  
    alert(this.key1); }, };  
var myfunc = myobj.method1; //is this OK?  
myfunc(); //result?
```

Setting this

- The problem in the previous example is that "this" is undefined (or ==window) for myfunc
- Javascript offers some mechanisms to fix this
- call, apply and bind

Call

- The method `call` is applied to objects of type Function. Its syntax is:
`<function>.call(thisvalue,arg1,arg2,...)`
- Except for changing the value of `this`, this is the same as just calling the function
`alert("Hi"); === alert.call(this,"Hi");`

Apply

- The apply method can be applied to function expressions

`<function>.apply(thisvalue,[args]);`

- Idea: in addition to setting this, breaks down array into parameters

- Ex: `Math.max(1,2,3,2,1) == 3`

`Math.max([1,2,3,2,1]) == NaN`

`Math.max.apply(null,[1,2,3,2,1]) == 3`

Bind

- Returns a **function**, which has already been supplied some arguments

Syntax: `<function>.bind(thisvalue,arg1,arg2,...)`

- Example:

```
function add(x,y) { return x+y; }  
var plus2 = add.bind(null,2);
```

Losing "this"

- What's the deal with supplying "this" to these functions?
- "this" can easily get lost. Example:

```
var counter = {  
    count : 0,  
    inc: function () { this.count++; }  
}
```

Losing "this"

- Why `this.count++` and not `count++`?
- This works. What is the problem?

`var myfunc = counter.inc; //Valid?`

`counter.inc(); //increases counter`

`myfunc(); //should be the same, but isn't...`

Finding "this"

- The previous is a common problem when giving a "callback function" to another function
- Solution: bind

```
var myfunc = counter.inc.bind(counter);
```

- Useful with (among others)
 - `setInterval(func, time);` //call func every time ms
 - `setTimeout(func, time);` //call func after time ms

Recap

- Consider the following example: (source: <http://www.sitepoint.com/>)

```
var name = "John";  
  
var obj = {  
    name: "Colin",  
    prop: {  
        name = "Aurelio",  
        getName: function() { return this.name; }  
    }  
}  
  
alert(obj.prop.getName());  
  
var test = obj.prop.getName();  
  
alert(test.getName());
```

Inheritance

- So far, objects are little more than maps
- We would like to emulate the OOP feature of inheritance
 - Make an object a "special case" of another, more general object, sharing the same methods/properties
- In javascript this is achieved with **prototypes**

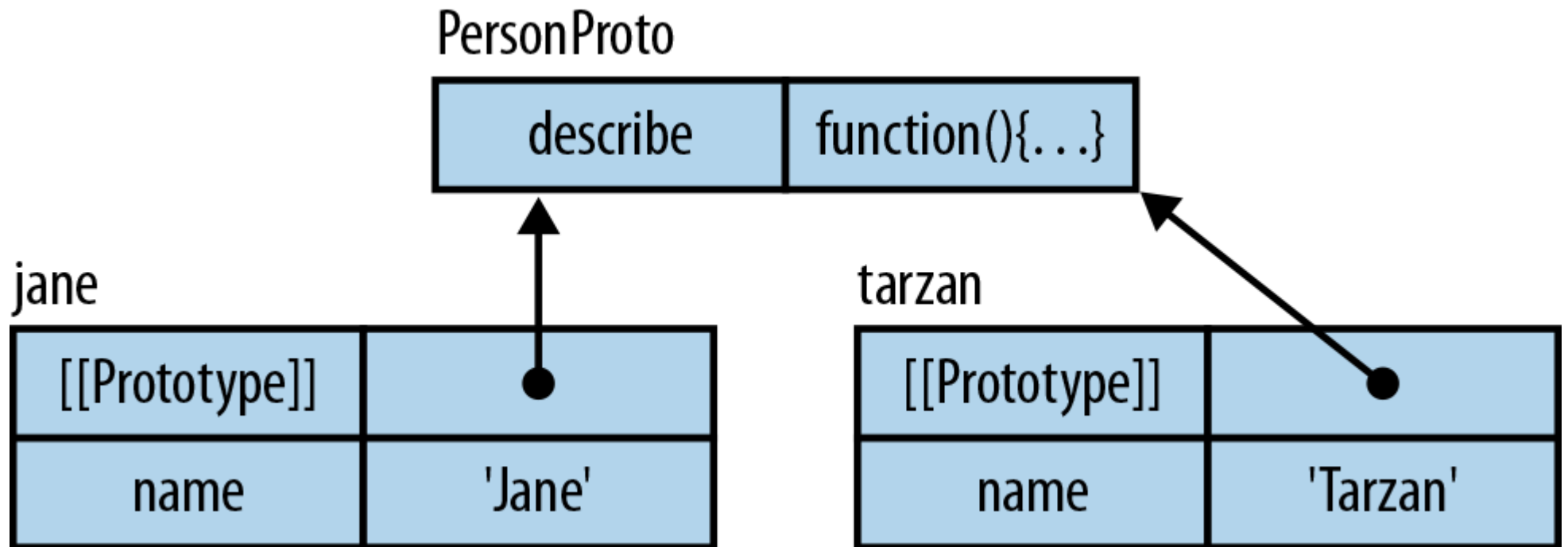
Prototypes

- How do things work?
 - Each object has a "hidden" property, called `[[Prototype]]`
 - This cannot be manipulated directly!
 - The value of this property is another "parent" object
 - When we try to access a property/method that does not exist, the search automatically contains up the prototype chain.

Prototypes

```
var PersonProto = {  
  describe: function () {  
    return 'Person named '+this.name;  
  }  
};  
  
var jane = {  
  [[Prototype]]: PersonProto, //ILLEGAL syntax!  
  name: 'Jane'  
};  
  
var tarzan = {  
  [[Prototype]]: PersonProto, //ILLEGAL syntax!  
  name: 'Tarzan'  
};
```

Prototypes



Creating child objects

- Here is the valid syntax to set [[Prototype]]

```
var PersonProto = {  
    describe: function () {  
        return 'Person named '+this.name;  
    }  
};  
  
var jane = Object.create(PersonProto);  
jane.name = "Jane"; //Now we can add the non-inherited stuff...
```

Checking inheritance relations

- The `Object.getPrototypeOf(myobj)` function returns the prototype of an object

`Object.getPrototypeOf(jane) === PersonProto`

- The `Object.isPrototypeOf(myobj)` method checks if the target object (this) is the prototype of myobj

`PersonProto.isPrototypeOf(jane) //true`

Setting and deleting

- Setting a property **over-rides** prototype properties

```
var proto = { method1: function () {alert("Hi");} };
```

```
var obj = Object.create(proto);
```

```
obj.method1(); //Result?
```

```
obj.method1() = function () {alert("Bye");};
```

```
obj.method1(); //Result?
```

```
proto.method1(); //Result?
```


Reference Semantics

- Guess the result of the following

```
var x = 5;
```

```
var y = x;
```

```
x++;
```

```
alert(y); // 5 or 6?
```

Reference Semantics

- Guess the result of the following

```
var x = {val: 5};
```

```
var y = x;
```

```
x.val++;
```

```
alert(y.val); // 5 or 6?
```

Reference semantics

- All object variables are to be treated as **references**.
- The = operator does not create a new copy of the value (as it does for numbers)
- Same is true for Arrays (Arrays are Objects)

Reference Semantics

- Guess the result of the following

```
var x = [5];
```

```
var y = x;
```

```
x[0]++;
```

```
alert(y[0]); // 5 or 6?
```

Getting around ref semantics

- Exercise: write some code that copies object a into object b
- Step 1: make sure new object has same prototype
- Step 2: copy over all local properties (recursively?)

One solution

//copy a to b

b = Object.reate(Object.getPrototype(a)); //Step1

for(i in a) b.i = a.i; //Step 2

//Naive for two reasons

//1. Also iterates over inherited properties

//2. = does not create a copy (is this what we want?)

Constructors

- A cleaner way to do OOP
 - Recall, so far we know how to use `Object.create()`
- In javascript any function can be used as a constructor for objects
 - Convention: functions meant as constructors start with capital letters
- Applying the **new** operator to a constructor creates an object, to which the constructor is applied immediately.

Constructors example

```
function Person(name){  
    this.name = name;  
}  
  
var jane = new Person("Jane Doe");  
  
//Now jane === { name: "Jane Doe" }  
  
jane instanceof Person //true  
  
jane.constructor === Person //true
```


Constructors and Prototypes

- Recall that all objects have a `[[Prototype]]` property
- All functions (also) have a `prototype` property
 - Confused?
 - Meaning: if the function `f` is used as a constructor (with `new`), `f.prototype` is the prototype used for the new object

Constructors and Prototypes

```
function Person(name){
```

```
    this.name = name;
```

```
}
```

```
Person.prototype.describe = function () {
```

```
    return "My name is "+this.name; }
```

```
var jane = new Person("Jane Doe");
```

```
jane.describe() //returns "My name is Jane Doe"
```

General idea

- Inside the prototype of the constructor we place methods that are to be shared by all the objects of this "class"
- Note that we are trying to simulate a behavior that happens automatically in Java, C++,...

More material

- Constructor Inheritance, Polymorphism...
- Data Protection
 - Everything is public
- ...