[Lesson 2]

Roi Yehoshua 2018

[DAN.IT EDUCATION]

What we learnt last time?

- Destructuring
- Closures
- Function declaration
- Named functional expressions
- Immediately invoked functional expressions
- Garbage collection

[Our targets for today]

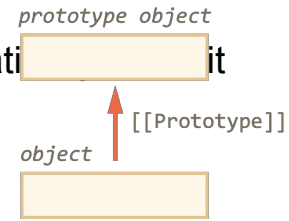
- Propotypes
- Prototype inheritance
- Native prototypes

[Prototype]

→ In JavaScript, objects have a special hidden property [[Prototype]], that is either null or references another object which is called prototype

→ When we want to read a property from object, and it's missing, JavaScript automatically gets it from the prototype

→ This is called "prototypal inheritance"



→ The property [[Prototype]] is internal and hidden, but there are many ways to set it

→ One of them is to use __proto__, like this:

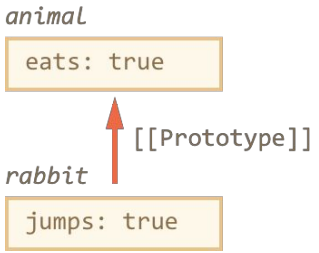
```
let animal = {  eats: true
};
let rabbit = {
    jumps: true
};

rabbit.__proto__= animal;
```

[DAN.IT] EDUCATION

[Prototype]

→ If we look for a property in rabbit, and it's missing, JavaScript automatically takes it from animal:

```
// we can find both properties in rabbit now:
alert(rabbit.eats); // true
alert(rabbit.jumps); // true
```



→ We say that "animal is the prototype of rabbit"

→ So if animal has a lot of useful properties and methods, then they become automatically available in rabbit

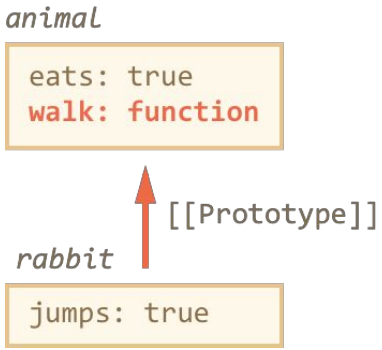
→ Such properties are called "inherited"

[DAN.IT] EDUCATION

[Prototype]

→ If we have a method in animal, it can be called on rabbit:

```
let animal = {  eats: true,  walk() {
        alert("Animal walk");
    }
};
let rabbit = {
    jumps: true
};

rabbit.__proto     = animal;

// walk is taken from the prototype
rabbit.walk(); // Animal walk
```

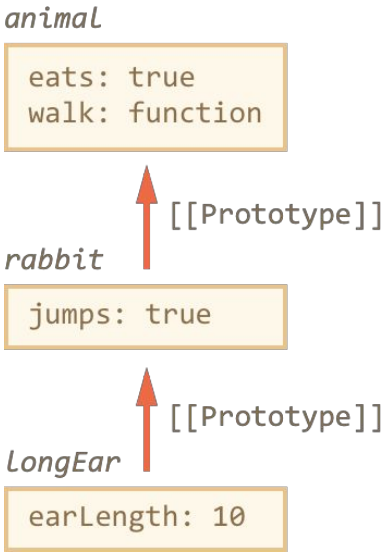


[DAN.IT] EDUCATION

[Prototype]

→ The prototype chain can be longer:

```
let animal = {  eats: true,  walk() {
        alert("Animal walk");
    }
};

let rabbit = {  jumps: true,
    ___proto_: animal
};

let longEar = {  earLength: 10,
    ___proto_: rabbit
}

// walk is taken from the prototype chain  longEar.walk();
// Animal walk  alert(longEar.jumps); // true (from rabbit)
```



[Read/Write Rules]

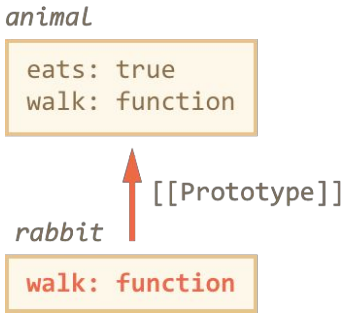
→ The prototype is only used for reading properties

→ Write/delete operations work directly with the object

→ In the example below, we assign its own walk method to rabbit

→ From that point, rabbit.walk() call finds the method immediately in the object and executes it, without using the prototype

```
let animal = {
    eats: true,
    walk() {       /* this method won't be used by rabbit */
    }
};
let rabbit = {
    __proto_: animal
}

rabbit.walk = function () {  alert("Rabbit! Bounce-bounce!");
};
rabbit.walk(); // Rabbit! Bounce-bounce!
```



[DAN.IT] EDUCATION

The value of “this”

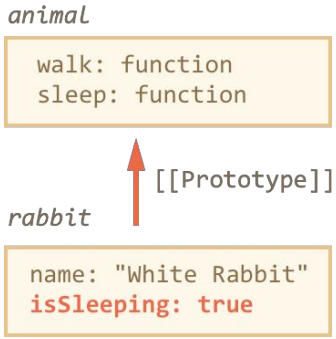
→ If we call obj.method(), and the method is taken from the prototype, ”this” still references obj

→ So methods always work with the current object even if they are inherited

→ In the example below, the call rabbit.sleep() sets this.isSleeping on the rabbit object:

```
let animal = {  walk() {
        if (!this.isSleeping) {
            alert('I walk');
        }
    },
    sleep() {
        this.isSleeping = true;
    }
};

let rabbit = {
    name: "White Rabbit",
    __proto__: animal
};
```

```
// modifies rabbit.isSleeping
rabbit.sleep();

alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // undefined  (no such property in the prototype)
```



[Exercise (1)]

→ We have two hamsters: speedy and lazy inheriting from the general hamster object

→ When we feed one of them, the other one is also full. Why? How to fix it?

```
let hamster = {  stomach: [],  eat(food) {
        this.stomach.push(food);
    }
};
let speedy = {
    ___proto_: hamster
};
let lazy = {
    ___proto_: hamster
};

// This one found the food  speedy.eat("apple");
alert(speedy.stomach); // apple
// This one also has it, why? fix please.
alert(lazy.stomach); // apple
```

[DAN.IT] EDUCATION

[The “prototype” Property]

→ As we know already, new F() creates a new object

→ When a new object is created with new F(), the object’s [[Prototype]] is set to F.prototype

  → Note that F.prototype here means a regular property named "prototype" on F

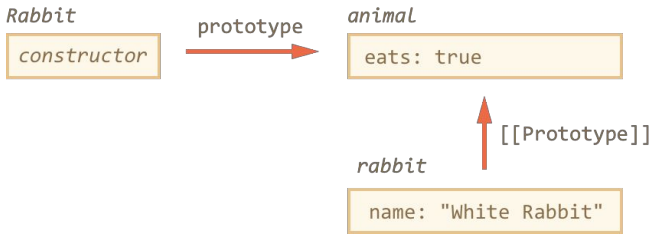
→ In other words, functions have a **prototype** property, and when you invoke functions with new, they will construct an object having a [[Prototype]] identical to the constructor function's prototype property

[DAN.IT EDUCATION]

The “prototype” Property

```
let animal = {
      eats: true
};

function Rabbit(name) {
      this.name = name;
}
Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); //  rabbit.__proto      == animal  alert(rabbit.eats);
// true
```

→ Setting Rabbit.prototype = animal literally states the following: "When a new Rabbit is created, assign its [[Prototype]] to animal"

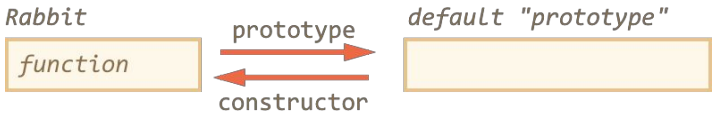


Default F.prototype

→ Every function has the "prototype" property even if we don't supply it

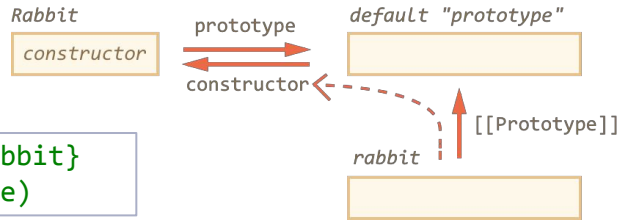
→ The default "prototype" is an object with the only property **constructor**, that points back to the function itself

```
function Rabbit() { }
// by default:
// Rabbit.prototype = { constructor: Rabbit };

alert(Rabbit.prototype.constructor === Rabbit); // true
```



→ Naturally, the constructor property is available to all rabbits through [[Prototype]]:

```
let rabbit = new Rabbit(); // inherits from {constructor: Rabbit}
alert(rabbit.constructor === Rabbit); // true (from prototype)
```



[Native Prototypes]

→ The "prototype" property is widely used by the core of JavaScript itself

→ All built-in constructor functions use it

→ We'll see how it is for plain objects first, and then for more complex ones

→ Let's say we output an empty object:

```
let obj = {};
alert(obj); // "[object Object]"
```

→ Where's the code that generates the string "[object Object]"?

→ The short notation obj = {} is the same as obj = new Object(), where **Object** is a built-in object

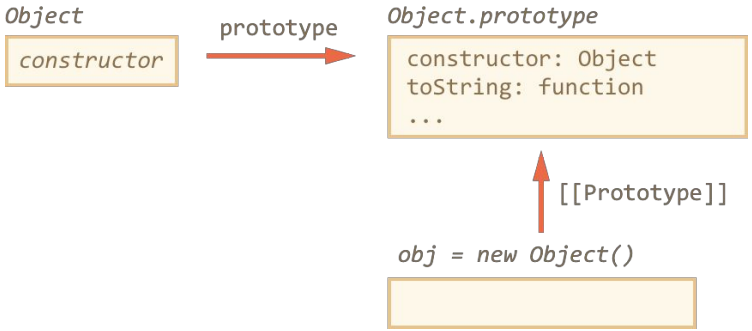
constructor function

→ That function has Object.prototype that references a large object with toString() and other functions

[DAN.IT] EDUCATION

[Native Prototypes]

→ When new Object() is called (or a literal object {...} is created), the [[Prototype]] of it is set to Object.prototype



→ Afterwards when obj.toString() is called – the method is taken from Object.prototype

```
let obj = {};
alert(obj); // "[object Object]"

alert(obj.toString === obj._proto_.toString); // true
alert(obj.toString === Object.prototype.toString); // true
```

[DAN.IT EDUCATION]

[Native Prototypes]

→ Other built-in objects such as Array, Date, Function and others also keep methods in prototypes

→ For instance, when we create an array [1, 2, 3], the default new Array() constructor is used internally, which writes the array data into the new object, and assigns Array.prototype to its prototype

→ The Array.prototype provides the methods for the new array

→ All built-in prototypes have Object.prototype on the top

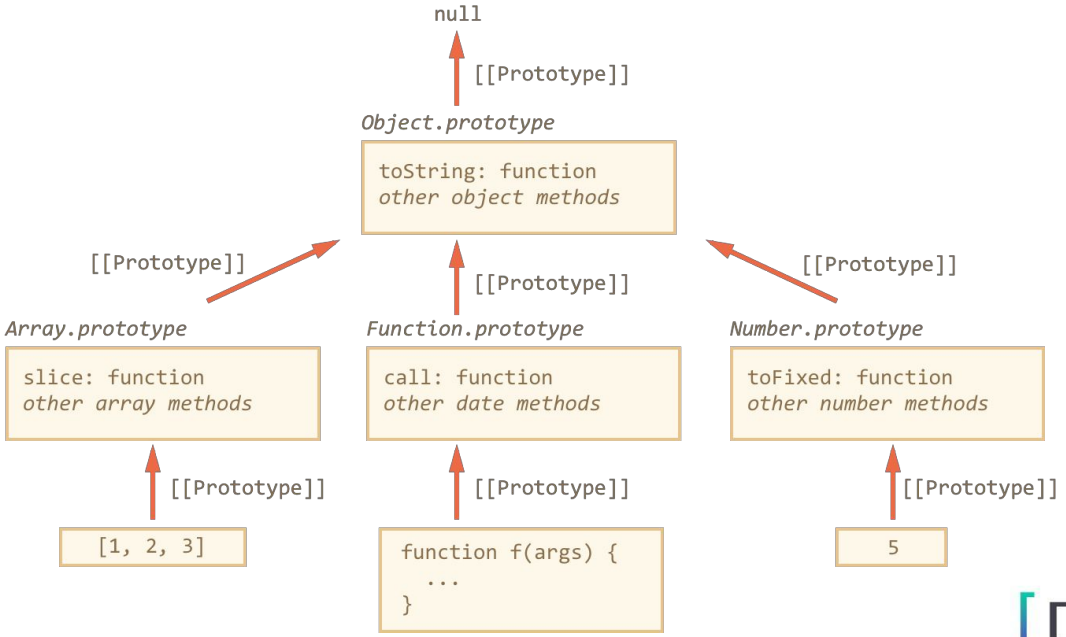
→ “everything inherits from objects”

[DAN.IT] EDUCATION

[Native Prototypes]

→ All built-in prototypes have Object.prototype on the top

→ “everything inherits from objects”



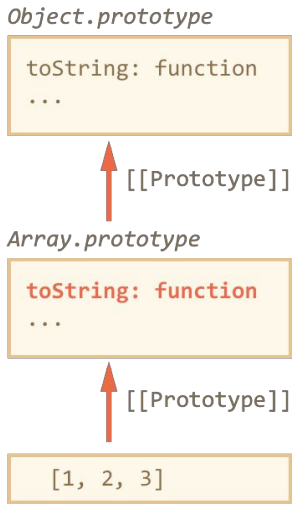
[Native Prototypes]

→ Some methods in prototypes may overlap

→ For example, Array.prototype has its own toString that lists comma-delimited elements:

```
let arr = [1, 2, 3]
alert(arr); // 1,2,3 <-- the result of Array.prototype.toString
```

→ Object.prototype has toString as well, but Array.prototype is closer in the chain, so the array variant is used



[DAN.IT] EDUCATION

Inspecting the Prototype Chain

→ In-browser tools like Chrome developer console shows the prototype inheritance when using console.log() (may need to use console.dir() for built-in objects):

```
let date = new Date();
console.dir(date);
```

[DAN.IT] EDUCATION

[Primitives]

→ As we remember, primitives such as strings and numbers are not objects

→ But if we try to access their properties, then temporary wrapper objects are created using built-in constructors **String**, **Number**, **Boolean,** which provide the methods and disappear

→ Methods of these objects also reside in prototypes, available as String.prototype, Number.prototype and Boolean.prototype

```
let str = "hello";
alert(str.__proto__=== String.prototype); // true

let num = 5;
alert(num.__proto__=== Number.prototype); // true
```

[DAN.IT] EDUCATION

[Exercise (2)]

→ What is the output of the following script?

```
let arr = [1, 2, 3];

alert(arr.__proto  ===  Array.prototype);  // ?
alert(arr.___proto__.__proto    === Object.prototype); // ?

alert(arr._proto_._proto_._proto_); // ?

alert(arr.constructor === Array.prototype.constructor); // ?

alert(arr.__proto  === new Array()._proto_); // ?

alert(arr.toString === Object.prototype.toString); // ?
```

[DAN.IT] EDUCATION

[Changing Native Prototypes]

→ Native prototypes can be modified

→ For instance, if we add a method to String.prototype, it becomes available to all strings:

```
String.prototype.show = function () {  alert(this);
};
"Hello!".show(); // Hello!
```

→ That is generally a bad idea, since prototypes are global, so it's easy to get a conflict

→ Modifying native prototypes is normally used for polyfills

→ i.e., if there's a method in JavaScript specification that is not yet supported by our JavaScript engine, then we may implement it manually

```
// if there's no such method add it to the prototype
if (!String.prototype.repeat) {
    String.prototype.repeat = function (n) {
        // repeat the string n times
        return new Array(n + 1).join(this);
    };
}
alert("La".repeat(3)); // LaLaLa
```

[DAN.IT EDUCATION]

[Control questions]

1. What is prototype inheritance?
2. What is prototype chain?
3. How does **this** keyword work with methods called from the prototype?
4. How does prototype inheritance work with constructor functions?
5. Do primitives have a prototype?

[DAN.IT] EDUCATION