JavaScript Inheritance

Inheritance

- JavaScript is object oriented but without classical inheritance
- Inheritance is achieved via the *prototype chain*.
  - Objects get access to properties and methods of their prototype object.
- `Object` is the end of the prototype chain.

```
// a.__proto__ is Object
var a = {};

// b.__proto__ is function
// b.__proto__.__proto__ is Object
var b = function(){};

// c.__proto__ is array
// c.__proto__.__proto__ is Object
var c = [];
```

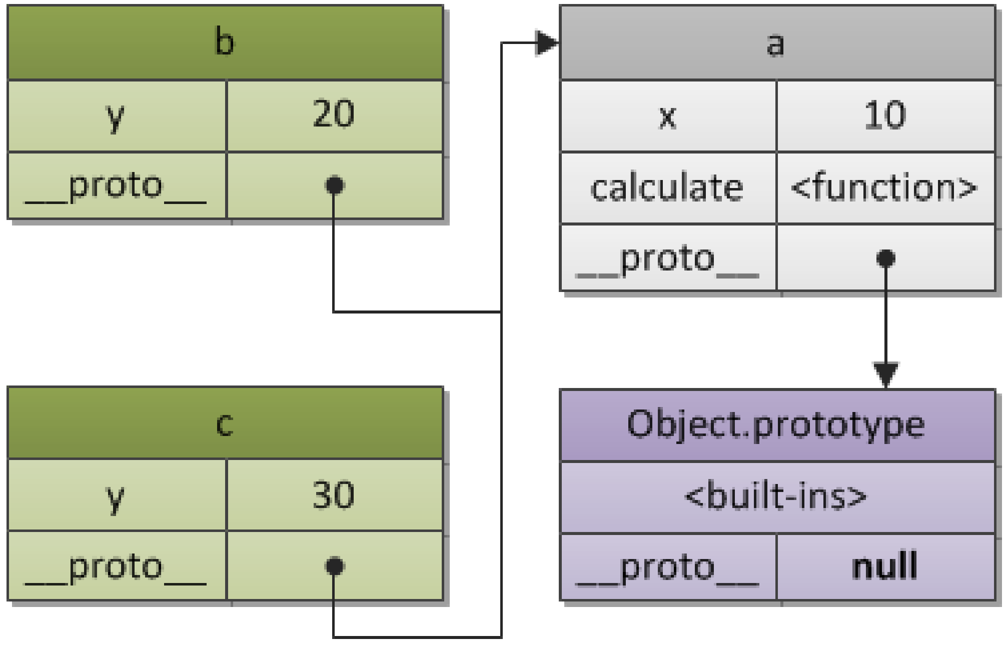
Objects & Prototypical inheritance

```
var a = {
    x: 10,
    calculate: function (z) {
        return this.x + this.y + z;
    }
};

var b = {
    y: 20,
    __proto__: a
};

var c = {
    y: 30,
    __proto__: a
};

// call the inherited method
b.calculate(30); // 60
c.calculate(40); // 80
```



Object.create()

- ES5 standardized an alternative way of prototype-based inheritance using Object.create() method.
- It sets `__proto__` property to original object for inheritance.

```
var person = {
        first: 'Default',
        last: 'Default',
        greet: function() { return 'Hi' + this.first; } //use this in functions
}

var jim = Object.create(person);
console.log(jim['first']); // Default - Inheritance
console.log(jim.hasOwnProperty('first')); // false
jim.first = 'Jim';
console.log(jim.hasOwnProperty('first')); // true

console.log(jim); // {first: 'Jim'} – No last & greet()
jim.greet(); // Hi Jim
```

JavaScript Object properties

- A JavaScript object is a collection of unordered properties.
  - Properties can usually be changed, added, and deleted, but some are read only.
    - The delete operator is designed to be used on object properties.
    - It has no effect on variables or functions.
    - The `delete` operator should not be used on predefined JavaScript object properties. It can crash your application.
- JavaScript object inherit the properties of their prototype
  - The `delete` keyword does not delete inherited properties, but if you delete a prototype property, it will affect all objects inherited from a prototype.

JavaScript Object properties

```
var student = {
       name : "Jim Carrey",
       course : "WAP",
       no : 12
};
for (var key in student) {
       console.log(key); // name, course, no
}
Object.keys(student) // [name, course, no]
stu = Object.create(student);
console.log(stu.name); // Jim Carrey
for (var key in stu) {
       console.log(key); // name, course, no
}
Object.keys(stu) // []
```

```
delete student.no;
console.log(student);

// Output:
Object{ name : "Jim Carrey",
       course : "WAP" };
```

Constructor functions

- **It’s a Function** used to create/construct other Objects and doesn’t return a value.
  - By convention Function Constructors start with a Capital letter.
  - To create new object from a Function Constructor we use the `new` keyword.

```
function Person(name, age){
    this.name = name;
    this.age = age;
    this.income = 0;
}

const person1 = new Person("Sally", 23);
console.log(person1);
person1.income = 1000;
console.log(person1);
```

The `prototype` property

- A property called `prototype` in the constructor is used to extend/add new functionalities to all objects created by the constructor using `new` keyword.
- When using `new` the `__proto__` of newly created object is set to the `prototype` property to the function constructor.

Sharing methods using prototype property

```
function Employee(){
        this.company = 'MUM';
        year = '2016';
}

var emp = new Employee(); // {company: "MUM"} – no year!
```

```
Employee.prototype.intro = function(){
        return 'Hi I work for ' + this.company;
}

emp.intro(); // "Hi I work for MUM"
```

We can create objects from the original function constructor with less memory space, as methods and common properties are shared. And we can extend the functionality of all objects by **adding methods and properties** to the `prototype` property at **runtime**. *(not to mix it up with `__proto__` which is used for inheritance)*

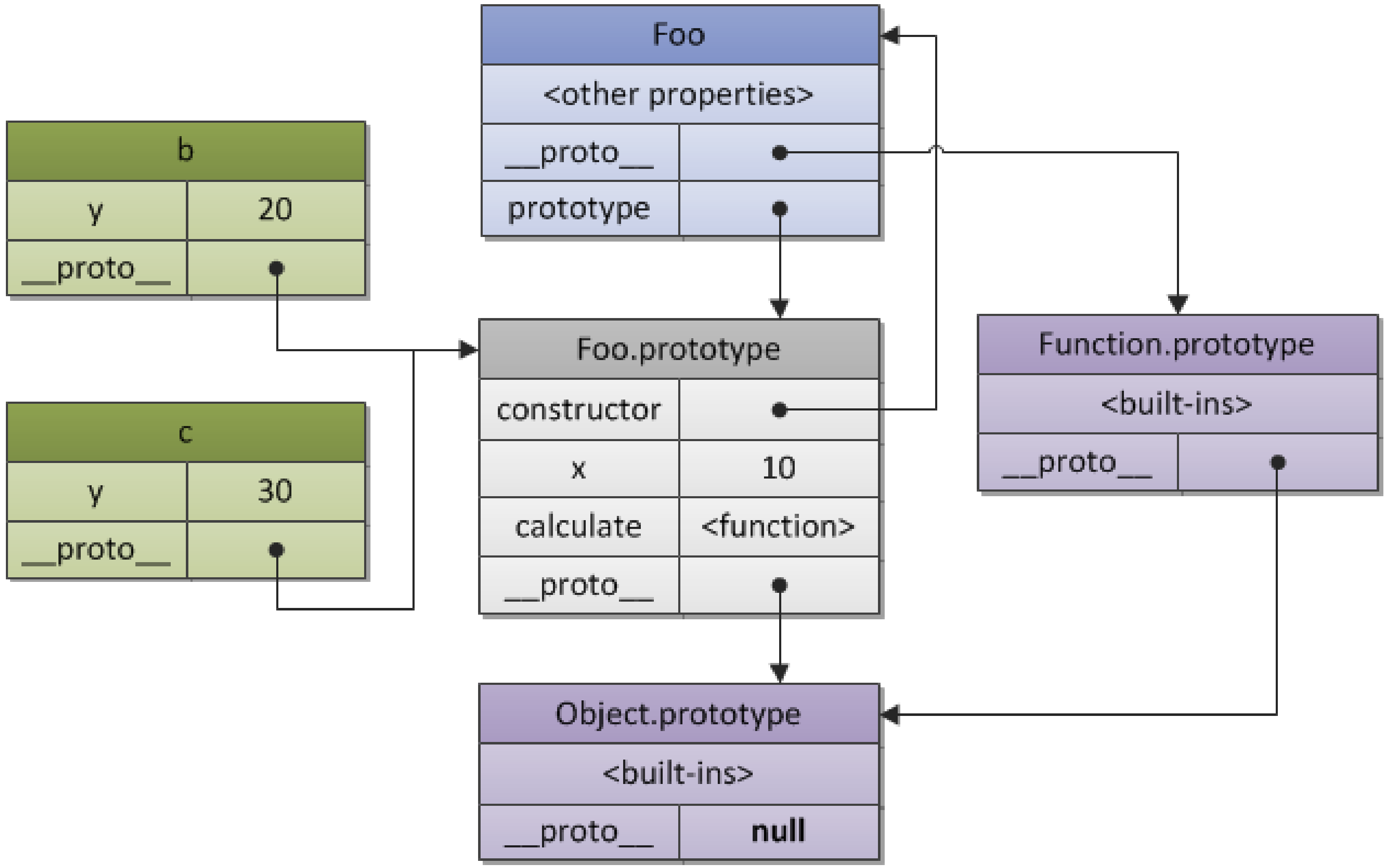
- We may rewrite previous example using a constructor function.

```
// a constructor function
function Foo(y) {
    this.y = y;
}
Foo.prototype.x = 10;
Foo.prototype.calculate = function (z) {
    return this.x + this.y + z;
};

var b = new Foo(20);
var c = new Foo(30);
// call the inherited method
b.calculate(30); // 60
c.calculate(40); // 80

console.log(
    b.__proto__ === Foo.prototype, // true
    c.__proto__ === Foo.prototype, // true
);
```

Constructor, `prototype` and `__proto__`



Example with Analysis

```
// By convention we use capital first letter for function constructor
function Course (coursename){
       this.coursename = coursename;
       console.log('Function Constructor Invoked!');
}
Course.prototype.register = function(){
       return 'Register ' + this.coursename;
}
var wap = new Course('WAP'); // Function Constructor Invoked!
```

```
console.log(wap); // Course {coursename: "WAP"}
console.log(wap.__proto__ === Course.prototype); // true
console.log(wap instanceof Course); // true
console.log(Course.prototype.register); // function(){ ... }
console.log(wap.register()); // Register WAP
```

Built-in Constructors

```
var x1 = new Object();    // A new Object object
var x2 = new String();    // A new String object
var x3 = new Number();    // A new Number object
var x4 = new Boolean();   // A new Boolean object
var x5 = new Array();     // A new Array object
var x6 = new RegExp();    // A new RegExp object
var x7 = new Function();  // A new Function object
var x8 = new Date();      // A new Date object
```

```
// Number.prototype, String.prototype, Date.prototype ... all have helper methods
available to the newly created objects.
x3.toString();
x8.getMonth();
```

Review – How to create Objects in JS

- From Object using: `Object.create()`;
  - The prototype chain (`__proto__`) will refer to original object
  - If we add any additional functionality to original object at runtime, it will be available to all derived objects
- From Function Constructors: `new FunctionConstructor();`
  - Only properties and methods with `this` will be copied from original function constructor (we prefer not to add any methods – only properties)
  - The prototype chain `(__proto__)` will refer to the `prototype` property of the constructor function.
  - If we add anything additional functionality to the original object's `prototype` property at runtime, it will be available to all derived objects.

Classes

- ES6 standardize the concept of class, and is implemented exactly as a constructor function.
- It provides syntactic sugar on top of the constructor function.

```
class Foo {
    constructor(name) {
        this._name = name;
    }

    getName() {
        return this._name;
    }
}

class Bar extends Foo {
    getName() {
        return super.getName() + ' Doe';
    }
}

var bar = new Bar('John');
console.log(bar.getName()); // John Doe
```

`this` one more time

- In JavaScript, the thing called `this`, is the **object that "owns" the JavaScript code**.
  - The value of `this`, when used in function, is the **object that "owns" the function**.
  - The value of `this`, when used in an object, is the **object itself**.
  - The `this` keyword in an object constructor (constructor function) does not have a value.
    - It is only a substitute for the new object.
    - The value of `this` will become the new object when the constructor is used to create the object.

Main Point
Inheritance

- JavaScript supports prototype based inheritance so that objects can inherit common functionality from a single ‘prototype’ object.

Science of Consciousness:

- Pure consciousness is a level of awareness that is a common experience shared by everyone.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Life Is Structured in Layers

1. JavaScript is a functional OO language with objects but no classes.

2. Closures and objects are fundamental to JavaScript best coding practices, particularly for promoting encapsulation, layering, and abstractions in code.

---

3. **Transcendental consciousness** is the experience of the most fundamental layer of all existence, pure consciousness, the experience of one’s own Self.

4. **Impulses within the transcendental field:** The many layers of abstraction required for sophisticated JavaScript implementations will be most successful if they arise from a solid basis of thought that is supported by all the laws of nature.

5. **Wholeness moving within itself:** In unity consciousness, one appreciates that all complex systems are ultimately compositions of pure consciousness, one’s own Self.