Telerik Academy

Classical Inheritance in JavaScript

The way of Object-oriented Ninja

JavaScript OOP

Telerik Software Academy

http://academy.telerik.com

Telerik Academy

Table of Contents

- Objects in JavaScript
  - Object-oriented Design
  - OOP in JavaScript
- Classical OOP
- Prototype-chain
- Object Properties
- Function Constructors
- The values of the `this` object
- Implementing Inheritance

Telerik Academy

Object-oriented Design

Telerik Academy

Object-oriented Programming

- OOP means that the application/program is constructed as a set of objects
  - Each object has its purpose
  - Each object can hold other objects
- JavaScript is prototype-oriented language
  - Uses prototypes to define hierarchies
    - Does not have definition for class or constructor
    - ECMAScript 1.6 introduces classes

Telerik Academy

OOP in JavaScript

Telerik Academy

OOP in JavaScript

- JavaScript is dynamic language
  - No such things as variable types and polymorphism
- JavaScript is also highly expressive language
  - Most things can be achieved in many ways
- That is why JavaScript has many ways to support OOP
  - Classical/Functional, Prototypal
  - Each has its advantages and drawbacks
  - Usage depends on the case

Telerik Academy

Classical OOP

Classical OOP

- JavaScript uses functions to create objects
  - It has no definition for class or constructor
- Functions play the role of object constructors
  - Create/initiate object by calling the function with the "new" keyword

```
function Person(){}
var gosho = new Person(); //instance of Person
var maria = new Person(); //another instance of Person
```

Telerik Academy

Creating Objects

- When using a function as an object constructor it is executed when called with new

```
function Person(){}
var personGosho = new Person(); //instance of Person
var personMaria = new Person(); //instance of Person
```

  - Each of the instances is independent
    - They have their own state and behavior
- Function constructors can take parameters to give instances different state

Creating Objects

- Function constructor with parameters
  - Just a regular function with parameters, invoked with new

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

var person1 = new Person("George", 23);
console.log(person1.name);
//logs: George

var person2 = new Person("Maria", 18);
console.log(person2.age);
//logs: 18
```

Telerik Academy

Function Constructors

Live Demo

Telerik Academy

Prototypes

Telerik Academy

The prototype Object

- JavaScript is prototype-oriented programming language
  - Every object has a prototype
  - Its kind of its parent object
- Prototypes have properties available to all instances
  - The `Object` type is the parent of all objects
    - Every object inherits object
    - Object provides common methods such as `toString` and `valueOf`

The prototype Object (2)

- When adding properties to a prototype, all instances will have these properties

```
//adding a repeat method to the String type
String.prototype.repeat = function (count) {
  var str,
      pattern,
      i;
  pattern = String(this);
  if (!count) {
    return pattern;
  }
  str = '';
  for (i = 0; i < count; i += 1) {
    str += pattern;
  }
  return str;
};
```

The prototype Object (2)

- When adding properties to a prototype, all instances will have these properties

```
//adding a repeat method to the String type
String.prototype.repeat = function (count) {
  var str,
      pattern,
      i;
  pattern = String(this);
  if (!count) {
    return pattern;
  }
  str = '';
  for (i = 0; i < count; i += 1) {
    str += pattern;
  }
  return str;
};
```

Add method to all strings

The prototype Object (2)

- When adding properties to a prototype, all instances will have these properties

```
//adding a repeat method to the String type
String.prototype.repeat = function (count) {
  var str,
      pattern,
      i;
  pattern = String(this);
  if (!count) {
    return pattern;
  }
  str = '';
  for (i = 0; i < count; i += 1) {
    str += pattern;
  }
  return str;
};
```

Add method to all strings

Here this means the string

The prototype Object (2)

- When adding properties to a prototype, all instances will have these properties

```
//adding a repeat method to the String type
String.prototype.repeat = function (count) {
  var str,
      pattern,
      i;
  pattern = String(this);
  if (!count) {
    return pattern;
  }
  str = '';
  for (i = 0; i < count; i += 1) {
    str += pattern;
  }
  return str;
};
```

Add method to all strings

Here this means the string

```
//use it with:
'-'.repeat(25);
```

Telerik Academy

Prototypes

Live Demo

Object Members

- Objects can also define custom state
  - Custom properties that only instances of this type have
- Use the keyword `this`
  - To attach properties to object

```
function Person(name,age){
  this.name = name;
  this.age = age;
}
var personMaria = new Person("Maria",18);
console.log(personMaria.name);
```

Object Members (2)

- Property values can be either variables or functions
  - Functions are called methods

```
function Person(name,age){
  this.name = name;
  this.age = age;
  this.sayHello = function(){
    console.log("My name is " + this.name +
                " and I am " + this.age + "-years old");
  }
}
var maria = new Person("Maria",18);
maria.sayHello();
```

Telerik Academy

Object Members

Live Demo

Attaching Methods

- Attaching methods inside the object constructor is a tricky operation
  - Its is slow
  - Every object has a function with the same functionality, yet different instance
  - Having the function constructor

```
function Constr(name, age){
  this.m = function(){
    // Very important code
  };
}
```

Attaching Methods

- Attaching methods inside the object constructor is a tricky operation
  - Its is slow
  - Every object has a function with the same functionality, yet different instance
  - Having the function constructor

```
function Constr(name, age){
  this.m = function(){
    // Very important code
  };
}
```

And the code

```
var x = new Constr();
var y = new Constr();
console.log (x.m === y.m);
```

Attaching Methods

- Attaching methods inside the object constructor is a tricky operation
  - Its is slow
  - Every object has a function with the same functionality, yet different instance
  - Having the function constructor

```
function Constr(name, age){
  this.m = function(){
    // Very important code
  };
}
```

And the code

```
var x = new Constr();
var y = new Constr();
console.log (x.m === y.m);
```

Logs 'false'

Telerik Academy

Different Method Instances

Live Demo

Telerik Academy

Better Method Attachment

- Instead of attaching the methods to this in the constructor

```
function Person(name,age){
  //…
  this.sayHello = function(){
    //…
  }
}
```

Telerik Academy

Better Method Attachment

- Instead of attaching the methods to this in the constructor

```
function Person(name,age){
  //…
  this.sayHello = function(){
    //…
  }
}
```

- Attach them to the prototype of the constructor

```
function Person(name,age){
}
Person.prototype.sayHello =
  function(){
    //…
  };
```

Telerik Academy

Better Method Attachment

- Instead of attaching the methods to this in the constructor

```
function Person(name,age){
  //…
  this.sayHello = function(){
    //…
  }
}
```

- Attach them to the prototype of the constructor

```
function Person(name,age){
}
Person.prototype.sayHello =
  function(){
    //…
  };
```

- And each method is created exactly once

Telerik Academy

Attaching Methods to the Prototype

Live Demo

Telerik Academy

Pros and Cons When Attaching Methods

- Attaching to this
- Attaching to prototype

Telerik Academy

Pros and Cons When Attaching Methods

- Attaching to this
  - Code closer to other languages 
- Attaching to prototype

Telerik Academy

Pros and Cons When Attaching Methods

- Attaching to this
  - Code closer to other languages ✓
- Attaching to prototype
  - Using JavaScript as it is meant ✓

Telerik Academy

Pros and Cons When Attaching Methods

- Attaching to this
  - Code closer to other languages ✓
- Attaching to prototype
  - Using JavaScript as it is meant ✓

JavaScript is NO other language

It should be treated like a first class language

Telerik Academy

Pros and Cons When Attaching Methods

- Attaching to this
  - Code closer to other languages ✓
  - Hidden data ✓
- Attaching to prototype
  - Using JavaScript as it is meant ✓

Telerik Academy

Pros and Cons When Attaching Methods

- Attaching to this
  - Code closer to other languages 
  - Hidden data 
- Attaching to prototype
  - Using JavaScript as it is meant 
  - No hidden data 

Telerik Academy

Pros and Cons When Attaching Methods

- Attaching to this
  - Code closer to other languages 
  - Hidden data 
- Attaching to prototype
  - Using JavaScript as it is meant 
  - No hidden data 

Hidden data is not such a big problem

Prefix "hidden" data with _(underscore) and be done with it

Telerik Academy

Pros and Cons When Attaching Methods

- Attaching to this
  - Code closer to other languages 
  - Hidden data 
  - Not good performance 
- Attaching to prototype
  - Using JavaScript as it is meant 
  - No hidden data 

Telerik Academy

Pros and Cons When Attaching Methods

- Attaching to this
  - Code closer to other languages 
  - Hidden data 
  - Not good performance 
- Attaching to prototype
  - Using JavaScript as it is meant 
  - No hidden data 
  - A way better performance 

Telerik Academy

Pros and Cons When Attaching Methods

- Attaching to this
  - Code closer to other languages 
  - Hidden data 
  - Not good performance 
- Attaching to prototype
  - Using JavaScript as it is meant 
  - No hidden data 
  - A way better performance 

Performance is a big deal!

It should be taken into serious consideration

Telerik Academy

Properties

Creating Getters and Setters

Telerik Academy

Properties in JavaScript

- JavaScript supports properties
  - i.e. a way to execute code when:
    - Getting a value
    - Setting a value
- They are two ways to create properties in JS:
  - At object declaration with

```
get propName(){ }
```

and

```
set propName(propValue){ }
```

  - Anytime with

```
Object.defineProperty(obj, propName, descriptor)
```

Telerik Academy

Object.defineProperty()

- `Object.defineProperty(obj, p, dscrptr)` defines property p on object obj
  - Example:

```
Object.defineProperty(Person.prototype, 'name', {
  get: function () {
    return this._name;
  },
  set: function (name) {
    if (!validateName(name)) {
      throw new Error('Name is invalid');
    }
    this._name = name;
  }
});
```

```
//calls the setter
p.name = 'Jane Doe';
//calls the getter
console.log(p.name);
```

Telerik Academy

Object.defineProperty()

Live Demo

Telerik Academy

ES 6 Get and Set

- ES6 introduces a new way to create properties and directly attach them to the prototype of the object:
  - Yet they can be defined only at the declaration of the object/function constructor

```
Person.prototype = {
    get name() {
      return this._name;
    },
    set name(name) {
      if (!validateName(name)) {
        throw new Error('Name is invalid');
      }
      this._name = name;
      return this;
    }
}
```

```
//calls the setter
p.name = 'Jane Doe';
//calls the getter
console.log(p.name);
```

Telerik Academy

ES6 Get and Set

Live Demo

Telerik Academy

The this Object

Telerik Academy

The this Object

- this is a special kind of object
  - It is available everywhere in JavaScript
    - Yet it has a different meaning
- The this object can have two different values
  - The parent scope
    - The value of this of the containing scope
    - If none of the parents is object, its value is window
  - A concrete object
    - When using the new operator

this in Function Scope

- When executed over a function, without the new operator
  - this refers to the parent scope

```
function Person(name) {
  this.name = name;
  this.getName = function getPersonName() {
    return this.name;
  }
}
var p = new Person("Gosho");
var getName = p.getName;
console.log(p.getName()); //Gosho
console.log(getName()); //undefined
```

this in Function Scope

- When executed over a function, without the new operator
  - this refers to the parent scope

```
function Person(name) {
  this.name = name;
  this.getName = function getPersonName() {
    return this.name;
  }
}
var p = new Person("Gosho");
var getName = p.getName;
console.log(p.getName()); //Gosho
console.log(getName()); //undefined
```

Here this means the Person object

this in Function Scope

- When executed over a function, without the new operator
  - this refers to the parent scope

```
function Person(name) {
  this.name = name;
  this.getName = function getPersonName() {
    return this.name;
  }
}
var p = new Person("Gosho");
var getName = p.getName;
console.log(p.getName()); //Gosho
console.log(getName()); //undefined
```

Here this means the Person object

Here this means its parent scope (window)

Telerik Academy

The this function object

Live Demo

Telerik Academy

Function Constructors

- JavaScript cannot limit function to be used only as constructors
  - JavaScript was meant for simple UI purposes

```
function Person(name) {
  var self = this;
  self.name = name;
  self.getName = function getPersonName() {
    return self.name;
  }
}
var p = Person("Peter");
```

Telerik Academy

Function Constructors

- JavaScript cannot limit function to be used only as constructors
  - JavaScript was meant for simple UI purposes

```
function Person(name) {
  var self = this;
  self.name = name;
  self.getName = function getPersonName() {
    return self.name;
  }
}
var p = Person("Peter");
```

What will be the value of `this`?

Telerik Academy

Function Constructors (2)

- The only way to mark something as contructor is to name it PascalCase
  - And hope that the user of you code will be so nice to call PascalCase-named functions with new

Telerik Academy

Invoking Function Constructors Without new

Live Demo

Telerik Academy

Function Constructors with Modules

Telerik Academy

Constructors with Modules

- Function constructors can be put inside a module
  - Introduces a better abstraction of the code
  - Allows to hide constants and functions
- JavaScript has first-class functions, so they can be easily returned by a module

```
var Person = (function () {
  function Person(name) {
    //…
  }
  Person.prototype.walk = function (distance){ /*...*/ };
  return Person;
}());
```

Telerik Academy

Function Constructors with Modules

Live Demo

Telerik Academy

Hidden functions

What to do when we want to hide something?

Hidden Funcitons

- When a function constructor is wrapped inside a module:
  - The module can contain hidden functions
  - The function constructor can use these hidden functions
- Yet, to use these functions as object methods, we should use `apply` or `call`

Telerik Academy

Hidden Functions: Example

- Using hidden functions

```
var Rect = (function () {
  function validatePosition() {
    //…
  }
  function Rect(x, y, width, height) {
    var isPositionValid = validatePosition.call(this);
    if (!isPositionValid) {
      throw new Error('Invalid Rect position');
    }
  }

  Rect.prototype = { /* … */};
  return Rect;
}());
```

Telerik Academy

Hidden Functions: Example

- Using hidden functions

```
var Rect = (function () {
  function validatePosition() {
    //…
  }
  function Rect(x, y, width, height) {
    var isPositionValid = validatePosition.call(this);
    if (!isPositionValid) {
      throw new Error('Invalid Rect position');
    }
  }

  Rect.prototype = { /* … */};
  return Rect;
}());
```

This is not exposed from the module

Telerik Academy

Hidden Functions: Example

- Using hidden functions

```
var Rect = (function () {
  function validatePosition() {
    //…
  }
  function Rect(x, y, width, height) {
    var isPositionValid = validatePosition.call(this);
    if (!isPositionValid) {
      throw new Error('Invalid Rec
    }
  }

  Rect.prototype = { /* … */};
  return Rect;
}());
```

This is not exposed from the module

Use call() to invoke the function over this

Telerik Academy

Hidden Functions

Live Demo

Telerik Academy

Inheritance in Classical OOP

Like in C#, Java or C++

Telerik Academy

Inheritance in Classical OOP

- Inheritance is a way to extend the functionality of an object, into another object
  - Like Student inherits Person
  - Person inherits Mammal, etc…
- In JavaScript inheritance is achieved by setting the prototype of the derived type to the prototype of the parent

```
function Person(fname, lname) {}
function Student(fname, lname, grade) {}
Student.prototype = Person.prototype;
```

  - Now all instances of type Student are also of type Person and have Person functionality

```
var student = new Student("Kiro", "Troikata", 7);
```

Telerik Academy

Inheritance in Classical OOP

Live Demo

Telerik Academy

The Prototype Chain

The way to search properties in JavaScript

The Prototype Chain

- Objects in JavaScript can have only a single prototype
  - Their prototype also has a prototype, etc…
  - This is called the prototype chain
- When a property is called on an object
  1. This object is searched for the property
  2. If the object does not contain such property, its prototype is checked for the property, etc…
  3. If a null prototype is reached, the result is undefined

Telerik Academy

Calling Parent Methods

Use some of the power of OOP

Calling Parent Methods

- JavaScript has no direct way of calling its parent methods
  - Function constructors actually does not know who or what is their parent
- Calling parent methods is done using call and apply

Telerik Academy

Calling Parent Methods: Example

- Having Shape:

```
var Shape = (function () {
  function Shape(x, y) {
    //initialize the shape
  }
  Shape.prototype = {
    serialize: function () {
      //serialize the shape
      //return the serialized
    }
  };
  return Shape;
}());
```

- Inheriting it with Rect

```
var Rect = (function () {
 function Rect(x, y,
               width, height) {
   Shape.call(this, x, y);
   //init the Rect
 }
 Rect.prototype = new Shape();
 Rect.prototype.serialize=function (){
   Shape.prototype
        .serialize
        .call(this);
   //add Rect specific serialization
   //return the serialized;
 };
 return Rect;
}());
```

Telerik Academy

Calling Parent Methods: Example

- Having Shape:

```
var Shape = (function () {
  function Shape(x, y) {
    //initialize the shape
  }

  Shape.prototype = {
    serialize: function () {
      //serialize the shape
      //return the serialized
    }
  };

  return Shape;
}());
```

- Inheriting it with Rect

```
var Rect = (function () {
 function Rect(x, y,
               width, height) {
   Shape.call(this, x, y);
   //init the
 }
 Rect.proto                    );
 Rect.prototype.serialize=function (){
   Shape.prototype
        .serialize
        .call(this);
   //add Rect specific serialization
   //return the serialized;
 };
 return Rect;
}());
```

Call parent constructor

Telerik Academy

Calling Parent Methods: Example

- Having Shape:

```
var Shape = (function () {
  function Shape(x, y) {
    //initialize the shape
  }

  Shape.prototype = {
    serialize: function () {
      //serialize the shape
      //return the serialized
    }
  };

  return Shape;
}());
```

- Inheriting it with Rect

```
var Rect = (function () {
 function Rect(x, y,
               width, height) {
   Shape.call(this, x, y);
   //init the
 }
 Rect.proto                    ;
 Rect.prototype.serialize=function (){
   Shape.prototype
        .serialize
        .call(this);
   //add Rect speci
   //return the ser
 };

 return Rect;
}());
```

Call parent constructor

Call parent method

Telerik Academy

Calling Parent Methods

Live Demo

Telerik Academy

Inheritance in Classical OOP

Questions?

http://academy.telerik.com