Telerik®
Develop experiences

Classical OOP in JavaScript

Classes and stuff

Telerik Software Academy
http://academy.telerik.com

Table of Contents

Telerik

OOP in JavaScript

OOP in JavaScript

- JavaScript is dynamic language
  - No such things as 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

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

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

Telerik

Creating Objects

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

```
function Person(name,age){
  console.log("Name: " + name + ", Age: " + age);
}
var personGosho = new Person("Georgi",23);
//logs:
//Name: Georgi, Age: 23
var personMaria = new Person("Maria",18);
//logs:
//Name: Maria, Age: 18
```

Function Constructors

Live Demo

Prototypes

The prototype Object

- JavaScript is prototype-oriented programming language
  - Every object has a hidden property 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
    - All objects has `toString()` method

Telerik

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;
};
```

Telerik

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);
```

Telerik

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

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 Person(name, age){
  this.introduce = function(){
    return 'Name: ' + name +
           ', Age: ' + age;
  };
}
```

Telerik

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 Person(name, age){
  this.introduce = function(){
    return 'Name: ' + name +
           ', Age: ' + age;
  };
}
```

```
var p1 = new Person();
var p2 = new Person();
console.log (p1 === p2);
```

Telerik

Different Method Instances

Live Demo

Better Method Attachment

- Instead of attaching the methods to this in the constructor

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

Telerik

Better Method Attachment

- Instead of attaching the methods to this in the constructor
  - Attach them to the prototype of the constructor

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

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

Telerik

Better Method Attachment

- Instead of attaching the methods to this in the constructor
  - Attach them to the prototype of the constructor

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

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

And each method is created exactly once

Telerik

Attaching Methods to the Prototype

Live Demo

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

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

The this Object

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

Telerik

`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
```

Telerik

The this function object

Live Demo

Function Constructors

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

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

Telerik

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

Invoking Function Constructors Without new

Live Demo

Function Constructor Fix

- John Resig (jQuery) designed a simple way to check if the function is not used as constructor:

```
function Person(name, age) {
  if (!(this instanceof arguments.callee)) {
    return new Person(name, age);
  }
  this._name = name;
  this._age = age;
}
```

Telerik

Function Constructor Fix

- John Resig (jQuery) designed a simple way to check if the function is not used as constructor:

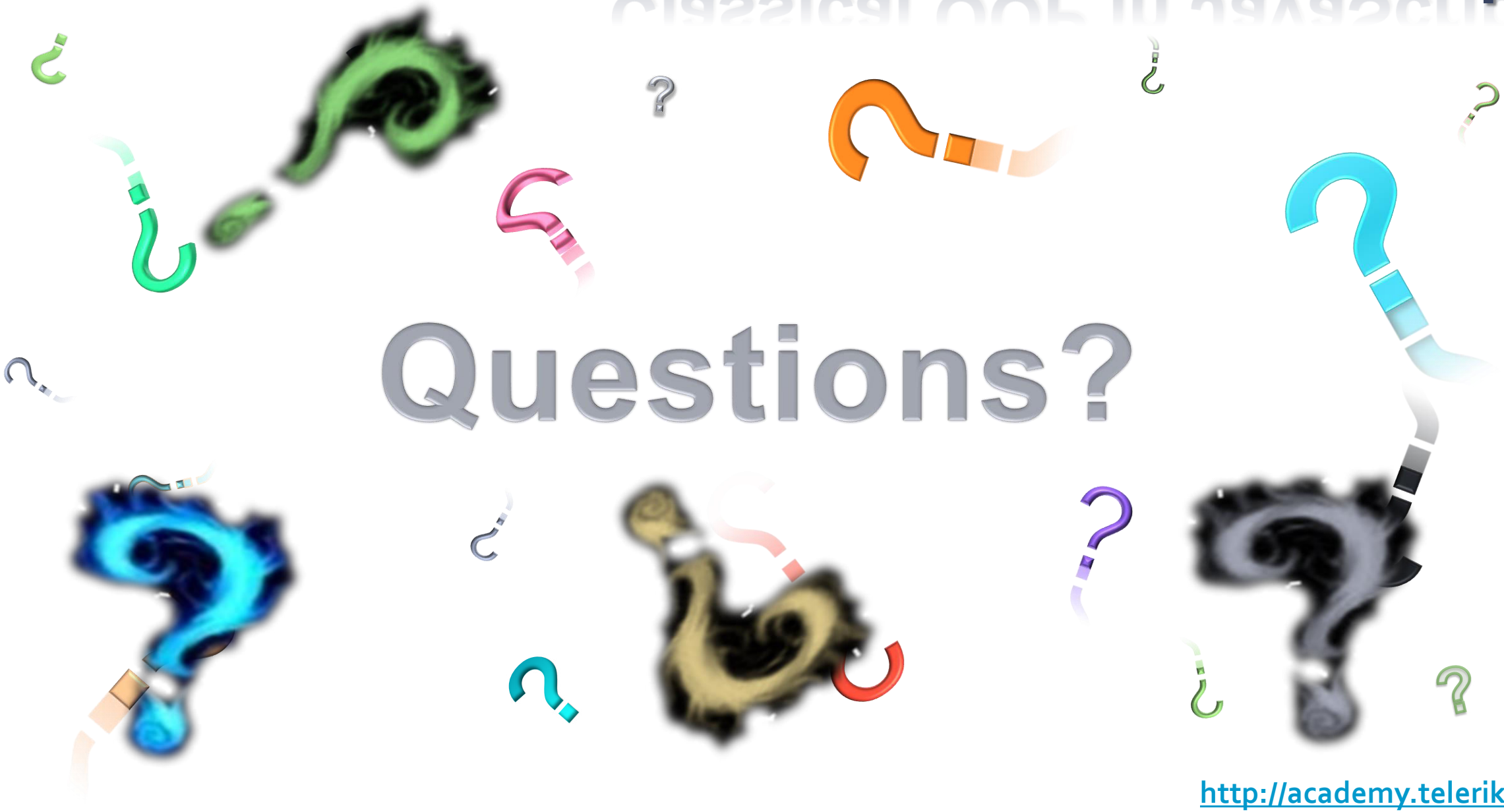
```
function Person(name, age) {
  if (!(this instanceof arguments.callee)) {
    return new Person(name, age);
  }
  this._name = name;
  this._age = age;
}
```

Telerik

John Resig Constructor Fix

Live Demo

Classical OOP in JavaScript



http://academy.telerik.com