

Объекты и наследование в Javascript

Часть 1. Введение в Javascript. (И немного о замыканиях)

Блоки и комментарии

C-подобный синтаксис

В конце строки — необязательная точка с запятой

Блок задаётся фигурными скобками { }

Комментарии:

Однострочные //

Многострочные /* */

Удобный способ прятать большие блоки кода:

/* в начале блока и */ в конце:

```
cnv.Layer.prototype.getChildIdx = function( child ) {
  var childIdx = 0;
  /*
  while ( this.children[ childIdx ] != child
        childIdx < this.children.length ) {
    childIdx++;
  }
  */
  if ( childIdx < this.children.length ) {
```

```
cnv.Layer.prototype.getChildIdx = function( child ) {
  var childIdx = 0;
  /*
  while ( this.children[ childIdx ] != child &&
        childIdx < this.children.length ) {
    childIdx++;
  }
  */
  if ( childIdx < this.children.length ) {
```

Переменные и объекты

- Переменные можно объявлять где угодно
- Все переменные являются объектами (обладают свойствами и методами)
- Чтобы начать использовать переменную, достаточно присвоить значение:

```
a = 2;
```

- Можно изменять тип переменной:

```
a = "hello";
```

- От типа переменной зависят некоторые операции:

```
a = "hel" + "lo"; // строка "hello"
```

```
a = 1 + 2; // число 3
```

- Неявное преобразование типов к более общему:

```
a = 1 + "abc" // строка "1abc"
```

ФУНКЦИИ

- Функцию можно определить так:

```
function myMagicFunction( arguments ) { /* body */ }
```

- Возвращение значения через **return**:

```
function myMagicFunction() { return 0; }
```

- Функция может принимать в качестве аргументов любые объекты, не только переменные.

- Функция — тоже объект, поэтому:

- Функцию можно определить где угодно
- Функцию можно присвоить любой переменной
- Функцию можно определять ещё и так:

```
myMagicFunction = function() { return 0; }
```

- Функция может возвращать другую функцию:

```
myMagicFunction = function() {  
  return function() {  
    window.alert( "hello" );  
  }  
}
```

- Аргументы функции доступны в массива **arguments**
- Поддерживаются замыкания

Область видимости (scope)

- Область видимости — часть кода, где переменная доступна для использования
- Если создавать переменную через обычное присвоение — будет создана «глобальная переменная»
- Если создавать переменную с использованием слова **var**, тогда будет создана «локальная переменная»:

```
var a = 2;
```

- Локальная переменная перестаёт существовать после завершения работы функции
- Время жизни локальной переменной можно продлить, используя замыкание (об этом чуть позже).

Замыкание

```
▼ first = function() {  
    var a = 1;  
    ▼ second = function() {  
        print( "Внутри: " + a );  
    }  
}
```

```
first();
```

```
// это приведёт к ошибке, переменная не определена
```

```
// print( a );
```

```
second();
```

Локальная и глобальная переменные

```
a = 0;
```

```
first = function() {
```

```
    var a = 1;
```

```
    second = function() {
```

```
        print( "Внутри функции: " + a );
```

```
    }
```

```
};
```

```
print( "Снаружи функции: " + a );
```

```
second();
```

Операторы и конструкции

(такие же, как и в других Си-подобных языках)

- Операторы:
 - арифметические: + - * / %
 - побитовые: & | << >> ^
 - логические: && || !
 - сравнение: == != < > <= >= === !==
 - тернарный оператор: ()?():()
- Конструкции языка:
 - while() {} и do {} while ()
 - for(expr1; expr2; expr3) {}
 - break / continue
 - switch .. case

Ловушка при использовании замыкания

Автор этого скрипта хотел создать массив функций, каждая из которых выводит свой порядковый номер:

```
function createThem() {  
    var result = new Array();  
    var i;  
    for ( i = 0; i < 10; i++ ) {  
        result[ i ] = function() {  
            print( i );  
        }  
    }  
    return result;  
}  
  
list = createThem();  
  
list[5]();
```

Пример-подсказка

```
function makeShout() {  
    var phrase = "Привет!"  
  
    var shout = function() {  
        print(phrase)  
    }  
  
    phrase = "Готово!"  
  
    return shout  
}  
  
shout = makeShout();  
// что выдаст?  
shout();
```

Массивы

- Создание массива:

```
a = new Array( "a", "b", "c" );
```

- Другой способ:

```
a = [ "a", "b", "c" ];
```

- Или так:

```
a = new Array();
```

```
a[ 0 ] = "a";
```

```
a[ 1 ] = "b";
```

```
a[ 2 ] = "c";
```

Хэш

- Создаются так:

```
a = { one: 1, two: 2, three: 3 }
```

- Собственные элементы объекта будем называть "слотами"
- Получить элемент объекта можно двумя способами:

```
a.one    или так:    a["one"]
```

- Можно в любой момент добавить объекту новый слот:

```
a.four = 4;
```

- Любой объект может быть значением слота.
- Если в слоте функция - тогда получается метод.

Обращение к слотам объекта из метода

- Внутри метода **this** указывает на сам объект:

```
a.showThree = function() { print( this.three ) }
```

- С помощью **call()** или **apply()** можно вызывать метод в контексте другого объекта:

```
b = { three : "3 from b" }  
a.showThree.call( b );
```

- Другой пример:

```
showTwo = function() { print( this.two ) }  
showTwo.call( a );
```

- Следовательно, **this** не привязано жёстко к конкретному объекту, и каждый раз указывает на тот объект, в контексте которого вызван метод.

Анонимные объекты

Анонимные объекты могут создаваться на лету внутри выражений:

```
"hello".toUpperCase() // вернёт строку "HELLO"
```

```
[ "a", "b", "c" ][2] // вернёт "c"
```

```
{ one: "a", two: "b" }["one"] // вернёт "a"
```

```
(function(a,b){return a+b})(1,2) // вернёт 3
```

Исправленный пример на замыкание

```
function createThem() {  
    var result = new Array();  
    var newFunc;  
    for ( var i = 0; i < 10; i++ ) {  
        newFunc = function(x) {  
            return function() {  
                print( x );  
            }  
        }  
        result[ i ] = newFunc( i );  
    }  
    return result;  
}
```

```
list = createThem();
```

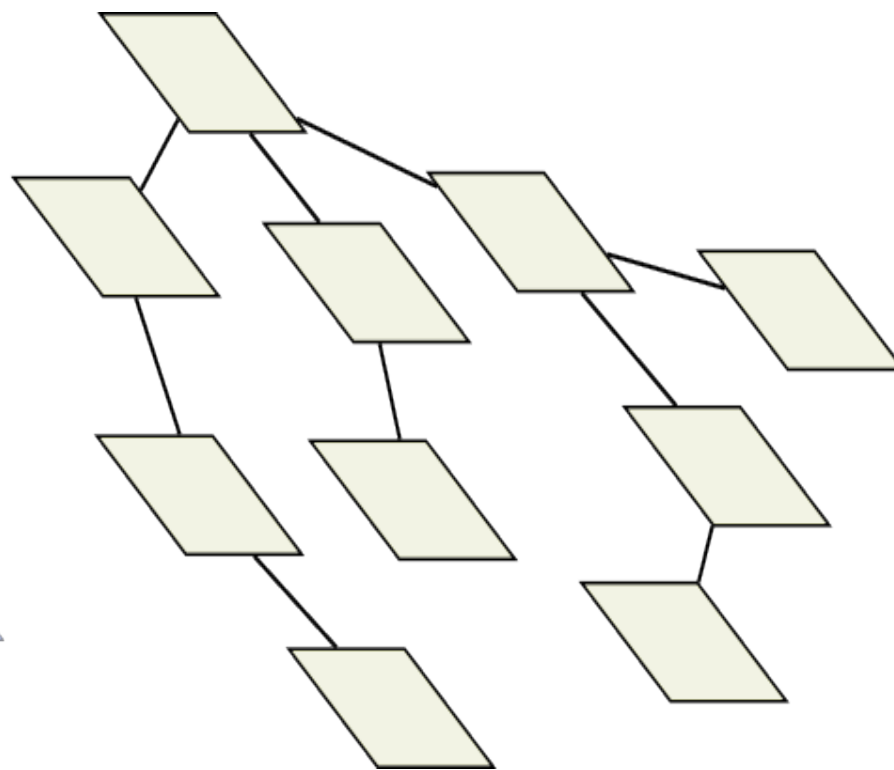
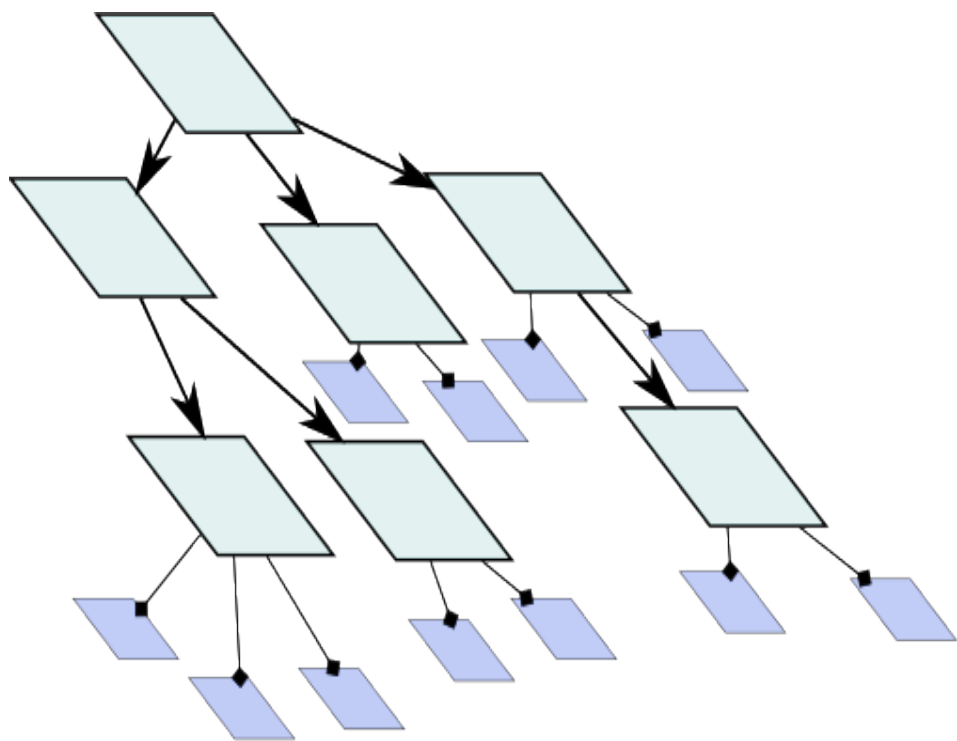
```
list[5]();
```

Исправленный пример на замыкание

```
function createThem() {  
    var result = new Array();  
    for ( var i = 0; i < 10; i++ ) {  
        result[ i ] = (function(x) {  
            return function() {  
                print( x );  
            }  
        })( i );  
    }  
    return result;  
}  
  
list = createThem();  
  
list[5]();
```

Часть 2. Наследование в Javascript. Прототипы.

Классовое и прототипное наследование



Прототипное наследование в JS

- С каждым объектом связан некоторый другой «родительский» объект, называемый *прототипом* объекта.
- Когда мы обращаемся к какому-то слоту объекта:
 - Сначала этот слот ищется у самого объекта
 - Если у объекта слота с таким именем нет — тогда он ищется у прототипа
 - Затем у прототипа прототипа
 - И так далее...
- Поэтому слот в прототипе является общим для всех дочерних объектов
- Слоты, определённые в прототипе могут быть переопределены в дочерних объектах.

Создание объекта с помощью конструктора.

- Существует ещё один способ создания объекта (помимо перечисления элементов)
- Любую функцию можно использовать в качестве конструктора, вызвав её через **new**, и тогда будет создан новый объект:

```
var A = function() {  
    // ...  
}
```

```
var myA = new A();
```

- В свойство **prototype** функции можно поместить другой объект, который будет выступать в качестве прототипа для создаваемых с помощью этой функции объектов.
- Это единственный способ организовать наследование (то есть, обозначить, какой объект будет прототипом)

Простейшая реализация наследования

```
▼ protoObj = {  
    x : "1"  
}
```

```
▼ MyType = function() {  
}
```

```
MyType.prototype = protoObj;
```

```
newObj = new MyType();
```

```
print( newObj.x ); // will print 1
```

Общий вид наследования

```
▼ ChildType = function() {  
    ParentType.call( this );  
    // инициализируем свойства объекта  
    this.a = 1;  
    ....  
}  
ChildType.prototype = new ParentType();  
// доопределяем методы в прототип  
ChildType.prototype.someMethod = function() {...}  
....
```

Тип Animal

```
▼ Animal = function( name, speed ) {  
    this.name = name;  
    this.position = 0;  
    this.speed = speed;  
}
```

```
▼ Animal.prototype.move = function( period ) {  
    this.position += period * this.speed;  
}
```

```
myAnimal = new Animal( "Petty", 5 );  
print( myAnimal.name );  
print( myAnimal.position );  
myAnimal.move( 2 );  
print( myAnimal.position );
```

УЧТЁМ КОЛИЧЕСТВО НОГ

```
▼ Animal = function( name, speed ) {  
    this.name = name;  
    this.position = 0;  
    this.speed = speed;  
    this.legNumber = 4;  
}  
▼ Animal.prototype.move = function( period ) {  
    this.position += period * this.speed * this.legNumber;  
}  
  
myAnimal = new Animal( "Petty", 1 );  
print( myAnimal.name );  
print( myAnimal.position );  
myAnimal.move( 2 );  
print( myAnimal.position );
```

Научим животное говорить

```
Animal = function( name, speed ) {
  this.name = name;
  this.position = 0;
  this.speed = speed;
  this.legNumber = 4;
}
Animal.prototype.move = function( period ) {
  this.position += period * this.speed * this.legNumber;
}

myAnimal = new Animal( "Petty", 1 );
print( myAnimal.name );
print( myAnimal.position );
myAnimal.move( 2 );
print( myAnimal.position );

myAnimal.say = function( times ) {
  for ( var i = 0; i < times; i++ ) {
    print( "кудахт" );
  }
}

myAnimal.say( 2 );
```

Опишем тип для Куриц

Метод `say()` перенесём ко всем животным

```
Animal = function( name, speed ) {  
    this.name = name;  
    this.position = 0;  
    this.speed = speed;  
    this.legNumber = 4;  
}  
Animal.prototype.move = function( period ) {  
    this.position += period * this.speed * this.legNumber;  
}  
Animal.prototype.say = function( times ) {}
```

Унаследованный тип Hen

```
▼ Hen = function( name, productivity ) {
  Animal.call( this, name, 1 );
  //исправляем ноги
  this.legNumber = 2;
  this.productivity = productivity;
}
Hen.prototype = new Animal( "", 0 );
// Исправляем голос
▼ Hen.prototype.say = function( times ) {
▼   for ( var i = 0; i < times; i++ ) {
      print( "кудахт" );
    }
}
// Курица может нестись
▼ Hen.prototype.layEggs = function( period ) {
  var eggNumber = this.productivity * period;
▼   for ( var i = 0; i < eggNumber; i++ ) {
      print( "о" );
    }
}
```

Создание конкретной курицы

```
myRyaba = new Hen( "Ряба", 2 );  
print( myRyaba.name );  
print( myRyaba.position );  
myRyaba.move( 2 );  
print( myRyaba.position );  
myRyaba.say( 2 );  
myRyaba.layEggs( 3 );
```

Опишем собаку

```
Dog = function( name, breed ) {  
    Animal.call( this, name, 4 );  
    this.breed = breed;  
}  
  
Dog.prototype = new Animal( "", 0 );  
Dog.prototype.say = function( times ) {  
    for ( var i = 0; i < times; i++ ) {  
        print( "гав" );  
    }  
}
```

Два подтипа собак, и Тузик

```
Assistant = function( name, breed ) {  
    Dog.call( this, name, breed );  
}
```

```
Assistant.prototype = new Dog( "", "" );
```

// про охотничьих нам также будет интересна выносливость

```
Hound = function( name, breed, stamina ) {  
    Dog.call( this, name, breed );  
    this.stamina = stamina;  
}
```

```
Hound.prototype = new Dog( "", "" );
```

```
myTuzik = new Hound( "Тузик", "Борзая", 10 );  
print( myTuzik.name );
```

Faith





125



```
myFaith = new Assistant( "Вера", "Лабрадор" );  
myFaith.legNumber = 2;  
print( myFaith.name );  
print( myFaith.position );  
myFaith.move( 3 );  
print( myFaith.position );
```

Басенджи — собака, которая не умеет лаять.



Опишем для неё новый тип

```
▼ Basenji = function( name ) {  
    Hound.call( this, name, "Басенджи", 4 );  
}  
Basenji.say = function() {}  
  
myBobik = new Basenji( "Бобик" );  
print( myBobik.name );  
myBobik.say();
```

Часть 3. Некоторые паттерны проектирования.

Синглетон

Используется для создания объектов, про которые мы хотим быть уверены, что он будет ровно один:

```
Singleton = function() {  
    if (!Singleton.instance) {  
        Singleton.instance = this;  
    } else {  
        return Singleton.instance;  
    }  
  
    // ...  
}  
  
var a = new Singleton();  
var b = new Singleton();
```

Обсервер

Используется для создания объектов, которые будут посылать сигнал и уведомлять других о событии:

```
Signal = function() {
  this.subscribers = [];
}
Signal.prototype.subscribe = function( method ) {
  this.subscribers.push( method );
}
Signal.prototype.unsubscribe = function( method ) {
  for ( var i = 0; ((i < this.subscribers.length) &&
    (this.subscribers[ subscriberNum ] != method));
  if ( subscriberNum < this.subscribers.length ) {
    this.subscribers.splice( subscriberNum, 1 );
  }
}
Signal.prototype.send = function() {
  for ( var i = 0; i < this.subscribers.length; i++ ) {
    this.subscribers[ i ].apply( null, arguments || [] );
  }
}
```

Стратегия

Используется для обеспечения взаимозаменяемости среди семейства алгоритмов:

```
context = {
  data : "Hello"
}

strategy1 = {
  display : function() {
    print( this.data );
  }
}

strategy2 = {
  display : function() {
    window.alert( this.data );
  }
}

strategy1.display.call( context );
strategy2.display.call( context );
```

Прокси

Используется для перехвата вызовов и контроля доступа к другому объекту:

```
subject = {  
    add : function( a, b ) { return a + b },  
    sub : function( a, b ) { return a - b }  
}  
  
proxy = {  
    add : function( a, b ) { return subject.add( a, b ) },  
    sub : function( a, b ) { return subject.sub( a, b ) }  
}  
  
print( proxy.add( 4, 2 ) );  
print( proxy.sub( 4, 2 ) );
```