Визитка

АНТОН ОКОЛЕЛОВ, занимается разработкой UI для исследовательских порталов глобальных компаний. Хочет научиться всегда выбирать только те решения, которые максимально быстро приближают к цели

JavaScript глазами PHP-программиста

Язык JavaScript может показаться очень странным и неудобным для программистов на «нормальных» языках, где существуют классы, где переменная this не меняет контекст как перчатки и где в конце концов функции — просто функции. Однако в существующей веб-реальности без этого необычного языка уже никуда, и, привыкнув, можно даже найти в нем свои плюсы и удобства

В этой статье я хотел бы описать некоторые особенности языка JavaScript с точки зрения программистов, имеющих опыт программирования на других языках, например, PHP. Почему это важно? Не буду говорить обо всех, но большая часть моих знакомых веб-программистов пошла примерно по такому пути: изучение HTML/CSS, после чего изучение PHP/MySQL, а потом жизнь заставила столкнуться с JavaScript, потому что, как выяснилось, сферические PHP-программисты в вакууме уже никому не нужны. И тут начинается масса проблем из-за того, что JavaScript совершенно ни на что не похож. Я постараюсь наметить несколько ключевых моментов, на которые можно будет опираться при чтении чужого кода. Ведь написать программу по-простому не так уж и сложно без глубоких знаний языка, а вот разобраться в том, что писали профи и расширить их код, гораздо сложнее.

Опять же нет смысла читать от корки до корки толстенные книги и мануалы, большая часть языка PHP-программисту должна быть понятна и так, просто глядя на примеры. Однако некоторые особенности все-таки требуют пристального внимания.

Итак, что тут такого, так сказать, не такого.

Функции

Например:

```
function Hello()
{
    alert('hello');
}
```

это то же самое, что написать:

```
var Hello = function()
{
    alert('hello');
}
```

После чего и в том и в другом случае можно вызвать Hello(), и получится один и тот же результат.

В принципе в PHP 5.3 тоже появились анонимные функции, но JavaScript гораздо более гибок. Я бы сказал, что он гибок до неприличия.

В приведенном выше примере в пространстве имен появляется переменная Hello, что может быть не желательно для универсальной библиотеки. Поэтому если нам нужна функция и ее вызов, но не нужно создание переменных, то можно написать так:

```
(function()
{
    alert('hello');
}) ();
```

То есть заключаем функцию в скобки, а потом запускаем ее с помощью (). Так, кстати, обычно сделаны плагины для библиотеки jQuery:

```
(function($) {
// Здесь могут создаваться переменные, но снаружи они будут
// не видны
})(jQuery);
```

То есть это по сути запуск анонимной функции, которой в качестве параметра передается переменная jQuery, и уже к этому jQuery лепится фунционал плагина. Кстати, $ – это просто переменная так называется, у них это можно.

Видимость переменных и замыкания

Видимость переменных достаточно необычна. Вся проблема в том, что в JavaScript функции могут быть созданы на лету, и они могут содержать внутри себя опять же создание новых функций. Поэтому существуют особые правила доступа к переменным.

Чтобы все это понять, рассмотрим пример:

```
function outer()
{
    var v = 5;
    function inner()
    {
        alert(v);
```

```
    }
    v = 6;
    return inner;
}
var innertest = outer();
innertest();
```

Этот пример выведет число 6, давайте рассмотрим, почему.

Внутренняя функция inner имеет доступ к переменным, определенным во внешней функции outer. Но что происходит в нашем примере, когда внешняя функция уже завершила свою работу,и после этого вызывается внутренняя? В этом случае JavaScript хранит и использует последние значения переменных, то есть то, чему они были равны перед самым завершением outer. Можно сказать другими словами: JavaScript хранит описание функции вместе с тем контекстом (со ссылками на переменные), в котором она создавалась. Это называется замыканием.

Но надо также понимать, что если бы мы сделали вызов inner() перед присвоением v=6 во время работы внешней функции, то такой вызов выдал бы нам число 5, потому что в данном случае интерпретатор еще «не знает» о том, что мы присвоим переменной другое значение в будущем.

Как это можно использовать. Допустим, нужно вам написать функцию, которая, выждав секунду, меняет цвет текста в DOM-элементе на зеленый.

Вот как это делается:

```
function changeColorInSecond(elem){
    setTimeout(function(){
        elem.style.color = "#00ff00";
    }, 1000);
}
```

Немного непривычно выглядит для PHP-программиста, но зато очень и очень удобно. Если писать этот код без использования замыканий, то надо вводить глобальную переменную, которая станет хранить ссылки на объекты используемых DOM-элементов, этим объектам надо будет придумать ID, чтобы потом отличать один от другого, добавить глобальную функцию timerFunc и в setTimeout придется использовать уродливую конструкцию типа setTimeout("timerfunc(" + elemId + ")"). В общем, тот факт, что JavaScript хранит переменные порождающей функции, очень сильно облегчает нам жизнь.

Глобальная область видимости

Если вы вводите новую переменную в глобальной области видимости, то есть не в функции (или же внутри функции, но не пишете ключевое слово var), то переменная является глобальной. А глобальные переменные на самом деле в браузерах хранятся в объекте window как обычные свойства. То есть:

```
x = 5;
```

это все равно, что написать:

```
window.x = 5;
```

Объектная модель

Классов в JavaScript нет. Совсем. Есть только объекты и взаимодействие между ними.

Объекты можно создать с помощью оператора new и конструктора, например, так:

```
function Hello()
{
    this.myprop = 5;
}
var obj = new Hello;
```

Или вот так:

```
var obj = {myprop: 5, mymethod: function(){alert('ok'}};
```

Практически любую функцию можно использовать как конструктор для объекта.

К объекту можно приделать все что угодно не только в конструкторе, но и уже после его создания:

```
obj.x = 5;
obj.mymethod = function() {}
```

Также можно создать пустой объект. Это можно сделать двумя эквивалентными способами:

```
var obj = {};
var obj = new Object;
```

В JavaScript все является объектом. Даже если вы напишете:

```
var x = 5;
```

то на самом деле вы создали объект Number. Проверить это можно, посмотрев, что выдается на x.constructor.name

Ну и, естественно, вышеупомянутая функция Hello тоже является объектом. Можно написать так:

```
Hello.myvar = 5;
```

Переменная this

При использовании переменной this есть некоторые тонкости. Дело в том, что в JavaScript одну и ту же функцию можно сделать методом или конструктором нескольких абсолютно разных объектов. Поэтому this внутри функции является ссылкой на объект, в контексте которого идет вызов.

Если, например, мы присвоим window.mymethod = myfunc и потом вызовем window.mymethod(), то во время этого вызова this внутри myfunc будет указывать на объект window. Естественно, когда функция используется как конструктор, this внутри нее будет указывать на создаваемый объект.

Кстати, в JavaScript можно вызвать функцию в контексте любого объекта без предварительного присвоения ее в качестве метода.

Это можно сделать так:

```
myfunc.call(obj, arg1, arg2);
```

или:

```
myfunc.apply(obj, [arg1, arg2]);
```

Apply – это то же самое, что и call, только аргументы передаются массивом. This в таких вызовах будет указывать на объект window.

Прототип

Функции – конструктору объекта можно указать объект-прототип. Это что-то вроде набора свойств по умолчанию.

```
function Person()
{
}

Person.prototype = {
    name: 'Default',
    sayName: function(){
        alert('My name is ' + this.name);
    }
}

//Теперь можно делать так
var john = new Person();
john.sayName(); // Выведется «My name is Default»
```

Как это работает. При обращении к какому-нибудь методу или свойству объекта JavaScript сначала смотрит, есть ли он в самом объекте, и, только если его нет, ищет в прототипе. Таким образом, объект в момент создания (в конструкторе) или после может переопределить эти свойства. Например:

```
john.name = 'John';
john.sayName(); // My name is John
```

Кстати, изменять свойства прототипа можно и после создания объекта:

```
Person.prototype.sayName = function(){
alert('Hello! My name is ' + this.name);
}
john.sayName(); // Hello! My name is John
```

По умолчанию прототип равен пустому объекту.

Наследование

Де-юре классов в JavaScript нет, но де-факто набор из конструктора и его свойств и методов, заданных через прототип, чертовски напоминает класс. И рано или поздно приходится задумываться о наследовании.

Допустим, у нас есть такой «класс» Person:

```
function Person(name){
    this.name = name;
}
Person.prototype.getName = function(){
    return this.name;
}
Person.prototype.averageSpeed = 5;
```

и мы хотим сделать класс-наследник Man. Первое, что приходит в голову, – это создать какой-нибудь объект на основе конструктора Person и сделать его прототипом класса Man:

```
function Man(name){
    this.name = name;
}
// Наследуем
Man.prototype =  new Person('');

// Переопределяем метод
Man.prototype.getName = function(){
    return "Mister " + this.name;
}
// Добавляем новый метод
Man.prototype.Hunt = function(){
    alert('Hunting....');
}
```

Проблемы видны невооруженным глазом. Мы создали лишний, совершенно ненужный объект с непонятным именем, да и вообще, мало ли что у него там происходит в конструкторе, а ведь new Person('') этот конструктор запускает!

Тогда, может быть, сделать так?

```
Man.prototype = Person.prototype;
```

Вроде бы все здорово, никаких лишних вызовов, никаких лишних объектов. но так делать нельзя. Если мы будем добавлять новые свойства к прототипу конструктора Man, то они чудесным образом появятся и в прототипе Person и в других его наследниках, например в Woman, а этого делать нельзя.

Для того чтобы сделать нормальное наследование, применяют несколько способов. Например, создается такая вспомогательная функция, которую можно наследовать:

```
function extend(Child, Parent) {
    var F = function() { }
    F.prototype = Parent.prototype
    Child.prototype = new F()
    Child.prototype.constructor = Child
    Child.superclass = Parent.prototype
}
```

т.е. наследование происходит через промежуточную функцию F.

Используется она примерно так:

```
function Woman(name)
{
    // Наследование конструктора, если нужно
    Person.call(this, name);
}

// Наследование всего остального
extend(Woman, Person);

Woman.prototype.getName = function(){ // Переопределение
    return "missis " + this.name;
}

// Добавляем новое свойство
Women.prototype.likeShopping = true;
```

Чтобы не писать все время «Women.prototype», вы можете воспользоваться функцией обьединения объектов, их легко можно найти в Интернете. Например, в составе популярной javascript-библиотеки jQuery есть функция jQuery.extend:

```
$.extend(Women.prototype,
{
    getName: function() { … },
    likeShopping: true,
    ...
});
```

Так иногда получается более наглядно.

Есть и другие способы наследования, не на прототипах. Есть специальные библиотеки, которые помогают конструировать «классы» и наследовать их друг от друга. Но это выходит за рамки обзорной статьи.

***

Теперь в общих чертах вы знаете, как писать программы на JavaScript, используя его особенности.

Разумеется, многое осталось за скобками, но на это есть Google и мануалы. EOF