JavaScript.
Работа с DOM

Консорциум W3C определил следующие «уровни» DOM:

DOM уровня 0. Грубо говоря, это эквивалент того, что было предложено в Netscape 3.0 и Internet Explorer 3.0. Мы называем этот стандарт DOM классической, или традиционной, объектной моделью JavaScript. Указанная форма DOM - поддерживает общие коллекции объектов документа (forms [ ], images [ ], anchors [ ], links [ ] и applets [ ]).

DOM уровня 1. Обеспечивает возможность работы со всеми элементами документа посредством стандартного набора функций. В модели DOM уровня 1 представлены все элементы, и все части страницы открыты для чтения и записи в любое время. Здесь обеспечиваются возможности, подобные тем, что предлагает коллекция document.all [ ] в Internet Explorer, но модель DOM уровня 1 является стандартизованной и обеспечивает совместимость (в отношении браузеров).

DOM уровня 2. Обеспечивает более совершенные возможности доступа к элементам страницы, путем объединения моделей DOM уровня 0 и уровня 1 и добавления возможностей доступа к таблицам стилей для работы с ними. Этот вариант DOM предлагает также усовершенствованную модель событий и менее известные расширения, такие как операции обхода древовидной структуры и работы с диапазонами. К сожалению, за исключением доступа к таблицам стилей, другие возможности DOM уровня 2 не полностью поддерживаются типичными Web-браузерами.

DOM уровня 3. До настоящего времени в разработке. Доступен в виде черновиков W3C.

Консорциум W3C разделяет концепцию DOM на следующие пять категорий:

DOM Core (ядро DOM). Задает типовую модель в виде древовидной структуры для просмотра и изменения документа, содержащего элементы разметки.

DOM HTML. Определяет расширение ядра DOM для работы с HTML. Расширение DOM HTML обеспечивает возможности работы с документами HTML, используя синтаксис, типичный для традиционных объектных моделей JavaScript. В основном, это DOM уровня 0 плюс средства работы с объектами, соответствующими элементам HTML.

DOM CSS. Определяет интерфейсы, необходимые для программного управления правилами CSS.

DOM Events (события DOM). Добавляет в DOM средства обработки событий. Такими событиями могут быть как привычные события интерфейса пользователя, например щелчки кнопки мыши, так и специфические события, связанные с самой моделью DOM, происходящие при модификации дерева документа.

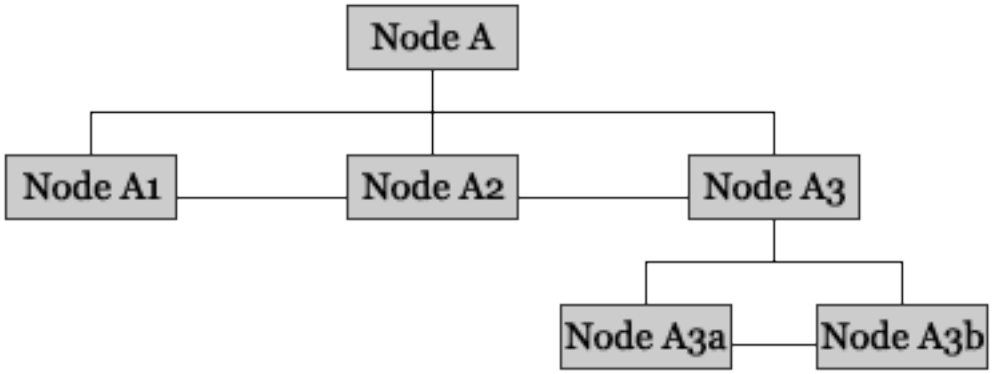
DOM XML. Определяет расширение ядра DOM для работы с XML. Расширение DOM XML призвано обеспечить решение специфических задач XML — для работы с разделами CDATA, инструкциями, пространством имен и т.д.

Дерево документа (Document Tree)

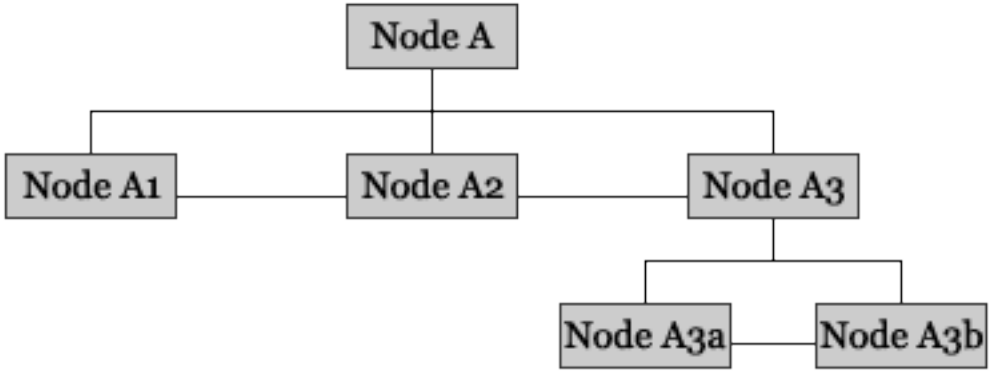
Когда браузер загружает страницу, он создает иерархическое отображение ее содержимого похожее на ее HTML структуру. Это результат древообразной организации узлов, каждый из которых содержит элемент, атрибут, текстовый или любой другой объект.

Узлы (Nodes)

Каждый их этих различных типов объектов имеет свои уникальные методы и свойства, а также обеспечивает взаимосвязь узлов (Nodes). Это общие установки методов и свойств, связанные с древообразной структурой документа. Чтобы лучше понять, как все это происходит, рассмотрим простое дерево узлов.



Каждый узел (Node) содержит свойства, отражающие эту структуру и позволяющие перемещаться по всем узлам дерева документа. Ниже показаны примеры взаимоотношений этих узлов:



NodeA.firstChild = NodeA1
NodeA.lastChild = NodeA3
NodeA.childNodes.length = 3
NodeA.childNodes[0] = NodeA1
NodeA.childNodes[1] = NodeA2
NodeA.childNodes[2] = NodeA3
NodeA1.parentNode = NodeA
NodeA1.nextSibling = NodeA2
NodeA3.previousSibling = NodeA2
NodeA3.nextSibling = null
NodeA.lastChild.firstChild = NodeA3a
NodeA3b.parentNode.parentNode = NodeA

Интерфейс узлов содержит методы для динамического добавления, обновления или удаления узлов, например:

insertBefore()
replaceChild()
removeChild()
appendChild()
cloneNode()

Корень Документа (Document Root)

Объект document является основанием дерева документа. Он обслуживается тем же интерфейсом, что и узел (Node). У него есть дочерние узлы (child nodes), но отсутствуют родительский узел (parent node) и узлы одного с ним уровня, так как он начальный узел. В добавление к существующему Node-интерфейсу, он содержит Document-интерфейс.

Этот интерфейс содержит методы доступа к другим узлам и создания новых узлов в дереве документа. Вот некоторые из них:

getElementById()
getElementsByTagName()
createElement()
createAttribute()
createTextNode()

Эти методы не такие, как у других узлов. Они могут быть только у объекта document. Все указанные выше методы (кроме getElementsByTagName()) могут использоваться только объектом document, т.е. их синтаксис должен быть: document.*methodName*().

Объект document может содержать и некоторые другие свойства устанавливаемые более ранними версиями DOM. Например многие браузеры все еще поддерживают массивы document.images и document.links или свойства document.bgColor и document.fgColor, соответствовавшие атрибутам bgcolor и text тэга <body>.

Эти свойства предназначены для обеспечения обратной совместимости.

Перемещение по Дереву Документа

Как уже было сказано, дерево документа отражает структуру HTML кода страницы. Каждый тэг или пара тэгов изображены как узел элемента, с узлами представляющими атрибуты или символьные данные (например, текст).

Формально, объект document имеет только один дочерний элемент (child element) устанавливаемый как **document.documentElement**. Для web-страниц он установлен тэгом <html>, который является корневым элементом дерева документа. У него есть дочерние элементы, установленные тэгами <head> и <body>, у которых в свою очередь, есть другие дочерние элементы.

Учитывая это и используя методы интерфейса Node можно перемещаться по дереву документа и обращаться к любому узлу этого дерева. Рассмотрим пример:

```
<html>
<head><title></title>
</head>
<body><p>Здесь какой-то текст.</p></body>
</html>
```

и этот код: **alert(document.documentElement.lastChild.firstChild.tagName);**

который покажет "p", название тэга, представленного этим узлом. В этом коде:

document.documentElement - возвращает тэг <html>. **Пример**

.lastChild - возвращает последний дочерний элемент для <html>, т.е. тэг <body>.

.firstChild - возвращает первый дочерний элемент внутри <body>.

.tagName - возвращает название тэга, в данном случае "p".

При этом всплывает очевидная проблема доступа к узлам. Достаточно изменить код документа, например, если добавить другие элементы, текст или изображения, то будет изменена структура дерева документа. И путь к этому узлу также может измениться.

Менее очевидна совместимость с доступом к узлам в некоторых браузерах. В приведенном выше примере между тэгами <body> и <p> нет ничего, даже пробелов, а теперь мы добавим между ними пару переводов строки до и после **<p>:**

```
<html>
<head>
<title></title>
</head>
<body>

<p>Здесь какой-то текст.</p>

</body>
</html>
```

В этом случае Netscape обнаружит в этих местах новые узлы, тогда как IE не станет этого делать. В Netscape вышенаписанный код JavaScript покажет "undefined", т.к. теперь в этом месте появился текстовый узел в виде пробела. Так как это не узел элемента, он не содержит теперь имени тэга. С другой стороны, IE не добавит узлов для таких пробелов, как здесь и по прежнему будет указывать на тэг "p".

Прямой доступ к элементам

Для этого существует удобный метод document.getElementById(). Для работы с ним необходимо добавить атрибут id тэгу "p" (да и любому другому тоже можно), тогда появляется возможность обращаться к элементу напрямую:

```
<p id="myP">Здесь какой-то текст.</p>

    ...

    alert(document.getElementById("myP").tagName);
```

Пример

В этом случае можно не опасаться некоторой несовместимости браузеров, а также того, где в дереве документа размещен узел для тэга "p". Важно лишь помнить о том, что атрибут id должен быть уникальным в пределах этого документа.

Более сложный доступ к узлу элемента представлен методом **document.getElementsByTagName().** Он возвращает массив узлов всех элементов документа, содержащих указанный HTML тэг. Для примера, сделать все ссылки на странице красными можно таким образом:

```
var nodeList = document.getElementsByTagName("a");
for (var i = 0; i < nodeList.length; i++)
    nodeList[i].style.color = "#f00";
```

Пример

Типы узлов (Node Types)

Более детально рассмотрим типы узлов. Как упоминалось ранее, в Объектной Модели Документа определены несколько типов узлов, но в разметке web-страницы в основном используются element, text и attribute.

Узел **Element** соответствует отдельному тэгу или паре тэгов в HTML коде. У него могут быть дочерние узлы (child node), которые могут быть элементами или текстовыми узлами.

Узел **Text** представляет обычный текст, или набор символов. У него предполагается наличие родительского узла (parent node), могут быть соседние узлы от того же родительского (sibling), но не может быть дочерних узлов (child node).

Узел **Attribute** отличается от вышеописанных. Он не участвует в построении дерева документа, у него нет ни родительских, ни дочерних, ни соседних узлов. Вместо этого он используется для доступа к атрибутам узла элемента. Он представляет атрибуты, определенные у HTML тэга элемента, например, атрибут href тэга <a> или src тэга <img />.

Значение атрибута всегда будет представлено текстовой строкой!

Атрибут, как узел Attribute

Существует несколько способов обращения к атрибутам элемента. Причина тому — стандарт DOM2, который был разработан для различных типов документов (т.е. XML), не только HTML. Таким образом он формально определяет тип узла для атрибута.

Но для всех документов он обеспечивает несколько схожих методов для доступа к атрибутам, добавления новых или изменения существующих. Как это происходит, смотрите дальше.

Метод document.createAttribute() дает возможность создать новый узел атрибута, которому можно присвоить значение, и применить его к узлу элемента:

```
var attr = document.createAttribute("myAttribute");
attr.value = "myValue";
var el = document.getElementById("myParagraph");
el.setAttributeNode(attr);
```

Однако, существует более простой способ доступа к атрибутам элемента, используя **getAttribute()** и **setAttribute()** методы элемента:

```
var el = document.getElementById("myParagraph");
el.setAttribute("myAttribute", "myValue");
```

Атрибуты элемента могут быть представлены и как свойства узла элемента. Другими словами, можно сделать и так:

```
var el = document.getElementById("myParagraph");
el.myAttribute = "myValue";
```

Следует обратить внимание, что Explorer 5.5 и более ранние версии не поддерживают тип узла Attribute и такой метод, как document.createAttribute() там не работает, в то время как *element*.getAttribute() уже как-то поддерживается. В этих браузерах получить доступ к атрибутам можно так: *element.attributeName*. А начиная с версии IE 6.0 уже поддерживаются узлы атрибутов, их методы и свойства.

Возможно определять ваши атрибуты в HTML тэгах, например:

```
<p id="myParagraph" myAttribute="myValue">Здесь какой-то текст.</p>
...
alert(document.getElementById("myParagraph").getAttribute("myAttribute"));
```

Атрибуты можно удалять из узла элемента, используя метод removeAttribute() или removeAttributeNode(), а также заменяя *element.attributeName* пустой строкой ("").

Изменение атрибутов — один из вариантов создания динамических эффектов. Например можно легко изменить выравнивание текста по правому или левому краю:

```
<p id="sample1" align="left">Здесь какой-то текст.</p>
<p><a href="" onclick="document.getElementById('sample1').setAttribute('align', 'left'); return
false;">Нале-во!</a>
::
<a href="" onclick="document.getElementById('sample1').setAttribute('align', 'right'); return
false;">Напра-во!</a></p>
```

Атрибуты стилей

Большинство атрибутов для HTML тэгов примитивны, они могут определять значение свойства только конкретному тэгу. Применение стилей более интересно. Для примера, можно изменить атрибут style или class HTML тэгу. Но этот метод добавит элементу все параметры стилей. Часто приходится только какой-нибудь один или несколько параметр стиля убрать или изменить, не трогая остальные.

Атрибут style узла элемента определен как объект со свойствами для всех возможных параметров стилей. Можно получить доступ и изменить каждый отдельный параметр стиля как угодно. Следующий пример похож на предыдущий:

Но в этом случае выравнивание текста определено параметрами стилей вместо атрибута align. Вот фрагмент кода:

```
<p id="sample2" style="text-align:left">Здесь какой-то текст.</p>
<p><a href="#" onclick="document.getElementById('sample2').style.textAlign = 'left';
">Нале-во!</a>
<a href="#" onclick="document.getElementById('sample2').style.textAlign = 'right';">Напра-
во!</a></p>
```

Как и в случае (X)HTML, главной заботой является вопрос соответствия имен свойств CSS и имен свойств DOM. В случае CSS имена свойств часто пишутся через дефис, например **background-color**, что в JavaScript превращается в **backgroundColor**. Вообще говоря, записанные через дефис свойства CSS изображаются в DOM одним словом с использованием "криволинейной" записи, когда начальные буквы слов (кроме первого) являются прописными. Это правило соблюдается для всех свойств CSS, за исключением float, которое превращается в cssFloat, поскольку "float" является зарезервированным словом JavaScript.

Динамический контент

Изменить текстовое содержимое страницы довольно просто. Каждая непрерывная строка символов в теле HTML страницы представляется текстовым узлом. Свойство nodeValue содержит этот текст. Изменяя его значение можно соответственно изменять текст на странице.

Текстовые узлы (Text Nodes)

Следующий пример использует простейший тэг параграфа <p> и текст внутри него.

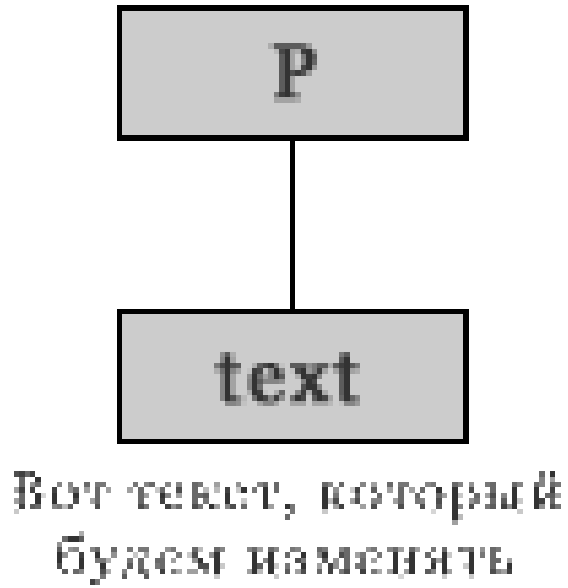
Рассмотрим код примера:

```
<p id="sample1">Вот текст, который будем изменять.</p>
<p>
<a href="" onclick="document.getElementById('sample1').firstChild.nodeValue = 'If something`s hard to do...'; return false;">меняем раз</a>
::
<a href="" onclick="document.getElementById('sample1').firstChild.nodeValue = '...then it`s not worth doing!'; return false;">меняем еще</a>
</p>
```

У текстового узла отсутствует атрибут id, который может быть у узла элемента. Таким образом напрямую, используя метод document.getElementById() или document.getElementsByTagName(), к нему обращаться нельзя.

Зато получить доступ к текстовому узлу можно зная его родительский узел (parent node), в данном случае элемент параграфа с id="sample1". Этот узел элемента содержит один дочерний узел (child node), тот самый текстовый узел, который мы хотим изменить.

Итак, document.getElementById('sample1').firstChild.nodeValue используется для доступа к этому текстовому узлу и чтения или изменения его значения, т.е. текста внутри параграфа.



DOM2 Events

Методы привязки событий DOM2

Для добавления в страницу обработчика событий можно использовать новый метод ***addEventListener()***, предлагаемый DOM2.
Имеются три причины для использования этой функции вместо непосредственной установки свойства обработчика событий для объекта:

1) возможность привязать к объекту сразу несколько обработчиков событий для одного и того же события. Когда обработчики событий привязаны таким образом, при наступлении соответствующего события происходит вызов каждого из обработчиков, но порядок их вызова произволен.
2) возможность обработать события в течение фазы захвата (когда событие "спускается" к цели). Обработчики событий, привязанные к атрибутам типа *onclick* или *onsubmit*, вызываются только в фазе возврата.
3) возможность привязать обработчик событий к текстовому узлу, что было невозможно до появления DOM2.

Синтаксис использования метода *addEventListener()* следующий:

объект.**addEventListener**(«событие», обработчик, фазаЗахвата);

• объект задает узел, к которому привязывается приемник события;
• "событие"соответствует событию, которое должно отслеживаться;
• обработчик задает функцию, которая должна вызываться, когда происходит соответствующее событие;
• фазаЗахвата является логическим значением, указывающим, когда должен вызываться обработчик событий — в фазе захвата (true) или в фазе возврата (false).

Например, чтобы назначить функцию changeColor() на роль обработчика событий mouseover в фазе захвата для абзаца с id, равным myText, необходимо выполнить вызов:

document.getElementByld('myText').addEventListener("mouseover", changeColor, true);

а чтобы добавить обработчик событий swaplmage() в фазе возврата:

document.getElementByld('myText').addEventListener("mouseover", swaplmage, false);

Обработчики событий удаляются с помощью removeEventListener() с теми же аргументами, что и при их добавлении. Чтобы исключить первый обработчик событий из предыдущего примера, нужно использовать вызов:

document.getElementByld('myText').removeEventListener("mouseover", changeColor, true);

Методы привязки событий Microsoft

Методы, предложенные Microsoft, работают только в браузерах Internet Explorer и Opera(она поддерживает метод Microsoft для лучшей совместимости).

Установка обработчика:

element.attachEvent("on"+имя события, обработчик)

Удаление обработчика:

element.detachEvent("on"+имя события, обработчик)

```
var myElement2 = document.getElementById("myElement2");
var handler = function() {
 alert('Спасибо!');
}

var handler2 = function() {
  alert('Еще раз спасибо!');
}

myElement2.attachEvent("onclick", handler);
myElement2.attachEvent("onclick", handler2);
```

Объект Event

Браузеры, обеспечивающие поддержку модели событий DOM2, передают обработчикам событий в виде первого аргумента **объект Event,** содержащий дополнительную информацию о произошедшем событии.

Свойства этого объекта зависят от произошедшего события, но все объекты Event обязательно имеют доступные только для чтения свойства, указанные в таблице.

Свойство только для чтения	**Описание**
bubbles	Логическое значение, указывающее, имеет ли событие фазу возврата
cancelable	Логическое значение, указывающее, допускает ли объект возможность отмены
currentTarget	Узел, обработчик события которого выполняется в данный момент (т.е. узел, к которому привязан данный обработчик событий)
eventPhase	Числовое значение, указывающее текущую фазу течения события (1 — для фазы захвата, 2—для нахождения в целевом объекте, 3—для фазы возврата) **Замечание**. Вместо числовых значений 1,2 и 3 для свойства eventPhase можно использовать константы Event. CAPTURING_PHASE, Event. AT_ TARGET и Event. BUBBLING_PHASE, соответственно.
type	Строка, соответствующая типу события (например, "click")
target	Узел, к которому изначально направлялось событие (т.е. узел, в котором событие произошло). В Internet Explorer у объекта window.event для этого есть свойство srcElement

В **Internet Explorer** существует глобальный объект window.event, который хранит в себе информацию о последнем событии. А первый аргумент обработчика отсутствует.

```
// обработчик без аргументов
function doSomething() {
    // window.event - объект события
}
 element.onclick = doSomething;
```

Можно кросс-браузерно получить объект события следующим образом:

```
function doSomething(event) {
    event = event || window.event

    // Теперь event - объект события во всех браузерах.
}

element.onclick = doSomething
```

Отмена действий, выполняемых по умолчанию

DOM уровня 2 позволяет отменить действие, по умолчанию связанное с событием, возвратив значение false из обработчика события. Но в DOM уровня 2 предлагается также метод *preventDefault()* для объектов Event. Если в какой-то момент существования объекта Event обработчик события вызывает метод *preventDefault()*, то действие, принятое для события по умолчанию, отменяется.

Здесь важно учитывать один важный момент: если для события вызывается метод *preventDefault()*, то типичное действие для события будет отменено — возвращение значения true другим обработчиком события не заставит снова выполняться действие, принятое по умолчанию.

Пример

Отмена распространения

Предотвратить дальнейшее распространение события по дереву документа возможно с помощью вызова *stopPropagation()*.

Пример

Перенаправление *событий*

Каждый узел имеет метод **dispatchEvent()**, который можно вызвать для перенаправления события к этому узлу. Указанный метод принимает Event в качестве аргумента и возвращает false, если хотя бы один обработчик, обслуживающий данное событие, вызывает *preventDefault()* или возвращает false.
При использовании такого подхода для перенаправления события узел, в котором вызывается *dispatchEvent()*, становится новым целевым объектом события.

События мыши

События мыши в DOM2, соответствуют событиям в (X)HTML. Они представлены в таблице. В соответствии со спецификациями DOM2 не все события допускают фазу возврата и не все действия по умолчанию могут быть отменены:

Событие	**Фаза возврата**	**Возможность отмены**
click	Да	Да
mousedown	Да	Да
mouseup	Да	Да
mouseover	Да	Да
mousemove	Да	Нет
mouseout	Да	Да

Когда происходит событие мыши, браузер наполняет объект Event дополнительной информацией:

Свойство	**Описание**
altKey	Логическое значение, указывающее, была ли нажата клавиша ALT
button	Числовое значение, соответствующее использовавшейся кнопке мыши (обычно 0 соответствует левой, 1 — средней, а 2 — правой кнопкам)
clientX	Горизонтальная координата точки, в которой произошло событие, относительно окна содержимого браузера
clientY	Вертикальная координата точки, в которой произошло событие, относительно окна содержимого браузера
ctrlKey	Логическое значение, указывающее, была ли нажата клавиша CTRL
detail	Указывает число щелчков мыши (если таковые были вообще)
metaKey	Логическое значение, указывающее, была ли нажата клавиша META
relatedTarget	Ссылка на узел, связанный с событием; например, для mouseove г она указывает на узел, по которому указатель мыши движется, а для mouseout — на узел, который указатель мыши покидает
screenX	Горизонтальная координата точки, в которой произошло событие, относительно всего экрана
screenY	Вертикальная координата точки, в которой произошло событие, относительно всего экрана
shiftKey	Логическое значение, указывающее, была ли нажата клавиша SHIFT

Пример

Несовместимости браузеров:

		Internet Explorer	Firefox, Safari Win и Opera	Konqueror
ЛЕВАЯ КНОПКА	event.which	undefined	1	1
	event.button	1	0	1
СРЕДНЯЯ КНОПКА	event.which	undefined	2	2
	event.button	4	1	4
ПРАВАЯ КНОПКА	event.which	undefined	3	3
	event.button	2	2	2

Кросс-браузерный код:

```
if (!e.which && e.button) {
  if (e.button & 1) e.which = 1
  else if (e.button & 4) e.which = 2
  else if (e.button & 2) e.which = 3
}
```

События клавиатуры

DOM уровня 2 не определяет событий клавиатуры. Они содержатся в в черновике DOM уровня 3.

Поскольку в (X)HTML для многих элементов предусмотрены события **keyup, keydown и keypress**, их поддерживают и браузеры.

В таблице ниже указаны связанные с клавиатурой события для браузеров с поддержкой DOM2.

Событие	Фаза возврата	Возможность отмены
keyup	Да	Да
keydown	Да	Да
keypress	Да	Да

Специфические для Mozilla связанные с клавиатурой свойства объекта Event приводятся в таблице ниже.

Свойство	**Описание**
altKey	Логическое значение, указывающее, была ли нажата клавиша ALT
charCode	Для печатаемых символов является числовым значением, указывающим значение Unicode нажатой клавиши
ctrlKey	Логическое значение, указывающее, была ли нажата клавиша CTRL
isChar	Логическое значение, указывающее, был ли при нажатии клавиши сгенерирован символ (полезное свойство, поскольку некоторые комбинации клавиш, например CTRL-ALT, символов не генерируют)
keyCode	Для непечатаемых символов является числовым значением, указывающим значение Unicode нажатой клавиши
metaKey	Логическое значение, указывающее, была ли нажата клавиша META
shiftKey	Логическое значение, указывающее, была ли нажата клавиша SHIFT

Пример

События браузера

Браузеры DOM2 поддерживают привычный набор событий браузера и форм, присутствующий во всех главных браузерах. Список этих событий приводится в таблице

Событие	**Фаза возврата**	**Возможность отмены**
load	Нет	Нет
unload	Нет	Нет
abort	Да	Нет
error	Да	Нет
select	Да	Нет
change	Да	Нет
submit	Да	Да
reset	Да	Нет
focus	Нет	Нет
blur	Нет	Нет
resize	Да	Нет
scroll	Да	Нет

События мутации

Из-за возможности динамической модификации иерархии объектов документа, в DOM2 включены события для выявления структурных и логических изменений документа.

Событие	"Восхождение"	Возможность отмены	Описание
DOMSubtreeModified	Да	Нет	Зависит от реализации; возникает, когда модифицируется часть поддерева узла
DOMNodeInserted	Да	Нет	Возникает для узла, добавляемого в дерево в качестве дочернего узла другого узла
DOMNodeRemoved	Да	Нет	Возникает для узла, удаляемого из дерева "родителя"
DOMNodeRemovedFromDocument	Нет	Нет	Возникает для узла, удаляемого из документа
DOMNodeInsertedIntoDocument	Нет	Нет	Возникает для узла, добавляемого в документ
DOMAttrModified	Да	Нет	Возникает для узла при модификации одного из его атрибутов
DOMCharacterDataModified	Да	Нет	Возникает для узла при модификации его данных

ООП в JavaScript

В большинстве объектно-ориентированных языков программирования существует возможность определять *классы* объектов и затем создавать отдельные объекты как *экземпляры* этих классов. Язык **JavaScript не обладает полноценной поддержкой классов**, как другие языки, например Java, C++ или C#. Тем не менее в JavaScript существует возможность определять псевдоклассы с помощью таких инструментальных средств, как функции конструкторы и прототипы объектов.

Объекты в JavaScript

Самый простой способ создания объектов заключается во включении в программу литерала объекта. Литерал объекта – это заключенный в фигурные скобки список свойств (пар «имя–значение»), разделенных запятыми. Имя каждого свойства может быть JavaScript-идентификатором или строкой, а значением любого свойства может быть константа или JavaScript-выражение.

```
var empty = {}; // Объект без свойств
 var point = { x:0, y:0 };
 var circle = { x:point.x, y:point.y+1, radius:2 };
  var homer = {
  “name": "Homer Simpson",
  "age": 34,
  "married": true,
  "occupation": "plant operator",
  }
```

Перечисление свойств. Цикл for/in предоставляет средство, позволяющее **перебрать, или перечислить**, свойства объекта. Это обстоятельство можно использовать при отладке сценариев или при работе с объектами, которые могут иметь произвольные свойства с заранее неизвестными именами. В следующем фрагменте демонстрируется функция, которая выводит список имен свойств объекта:

```
function DisplayPropertyNames(obj) {
 var names = "";
 for(var name in obj) names += name + "\n";
 alert(names);
}
```

Проверка существования свойств. Для проверки факта наличия того или иного свойства у объекта может использоваться оператор in. С левой стороны от оператора помещается имя свойства в виде строки, с правой стороны – проверяемый объект. Например:

```
// Если объект o имеет свойство с именем "x", установить его
if ("x" in o) o.x = 1;
```

При обращении к несуществующему свойству возвращается значение undefined. Таким образом, указанный фрагмент обычно записывается следующим образом:

```
// Если свойство x существует и его значение
// не равно undefined, установить его.
if (o.x !== undefined) o.x = 1;
```

Удаление свойств. Для удаления свойства объекта предназначен оператор delete:

delete book.chapter2;

При удалении свойства его значение не просто устанавливается в значение undefined; оператор delete действительно удаляет свойство из объекта.

Свойства и методы универсального класса Object

Все объекты в JavaScript наследуют свойства и методы класса Object. При этом специализированные классы объектов, как, например, те, что создаются с помощью конструкторов Date() или RegExp(), определяют собственные свойства и методы, но все объекты независимо от своего происхождения помимо всего прочего поддерживают свойства и методы, определенные классом Object.

Свойство constructor

В JavaScript любой объект имеет свойство constructor, которое ссылается на функцию-конструктор, используемую для инициализации объекта. Например:

```
var d = new Date( );
d.constructor == Date; // Равно true
```

Метод toString() не требует аргументов; он возвращает строку, каким-либо образом представляющую тип и/или значение объекта, для которого он вызывается. Интерпретатор JavaScript вызывает этот метод объекта во всех тех случаях, когда ему требуется преобразовать объект в строку.

```
var s = { x:1, y:1 }.toString( );//s=="[object Object]":
```

Метод valueOf() во многом похож на метод toString(), но вызывается, когда интерпретатору JavaScript требуется преобразовать объект в значение какого-либо элементарного типа, отличного от строки, – обычно в число.

Метод hasOwnProperty() возвращает true, если для объекта определено не унаследованное свойство с именем, указанным в единственном строковом аргументе метода. В противном случае он возвращает false. Например:

```
var o = {};
o.hasOwnProperty("undef"); // false: свойство не определено
o.hasOwnProperty("toString"); // false: toString – это унаследованное свойство
Math.hasOwnProperty("cos"); // true: объект Math имеет свойство cos
```

Метод hasOwnProperty() определяется стандартом ECMAScript v3 и реализован в JavaScript 1.5 и более поздних версиях.

Метод propertyIsEnumerable() возвращает true, если в объекте определено свойство с именем, указанным в единственном строковом аргументе метода, и это свойство может быть перечислено циклом for/in. Все свойства объекта, определяемые пользователем, являются перечислимыми. Неперечислимыми обычно являются унаследованные свойства:

```
var o = { x:1 };
o.propertyIsEnumerable("x"); // true: свойство существует и является перечислимым
o.propertyIsEnumerable("y"); // false: свойство не существует
o.propertyIsEnumerable("valueOf"); // false: свойство неперечислимое
```

Метод isPrototypeOf() возвращает true, если объект, которому принадлежит метод, является прототипом объекта, передаваемого методу в качестве аргумента.

В противном случае метод возвращает false. Например:

```
var o = {};
Object.prototype.isPrototypeOf(o); // true: o.constructor == Object
Object.isPrototypeOf(o); // false
o.isPrototypeOf(Object.prototype); // false
Function.prototype.isPrototypeOf(Object); // true: Object.constructor == Function
```

Конструкторы

Создание новых пустых объектов возможно как с помощью литерала {}, так и с помощью следующего выражения:

new Object()

Кроме того, была продемонстрирована возможность создания объектов других типов примерно следующим образом:

var array = new Array(10);
var today = new Date( );

За оператором new должно быть указано имя функции конструктора. Оператор new создает новый пустой объект без каких-либо свойств, а затем вызывает функцию, передавая ей только что созданный объект в виде значения ключевого слова **this**. Функция, применяемая совместно с оператором new, называется *функцией конструктором*, или просто *конструктором*. Главная задача конструктора заключается в инициализации вновь созданного объекта – установке всех его свойств, которые необходимо инициализировать до того, как объект сможет использоваться программой. Чтобы определить собственный конструктор, достаточно написать функцию, добавляющую новые свойства к объекту, на который ссылается ключевое слово **this**. В следующем фрагменте приводится определение конструктора, с помощью которого затем создаются два новых объекта:

```
// Определяем конструктор.
// Обратите внимание, как инициализируется объект с помощью "this".
function Rectangle(w, h) {
 this.width = w;
 this.height = h;
}
// Вызываем конструктор для создания двух объектов Rectangle. Мы передаем ширину и высоту
// конструктору, чтобы можно было правильно проинициализировать оба новых объекта.
var rect1 = new Rectangle(2, 4); // rect1 = { width:2, height:4 };
var rect2 = new Rectangle(8.5, 11); // rect2 = { width:8.5, height:11 };
```

Прототипы и наследование

Предположим, что необходимо рассчитать площадь прямоугольника, представленного объектом Rectangle. Вот один из возможных способов:

```
function computeAreaOfRectangle(r) { return r.width * r.height; }
```

Эта функция прекрасно справляется с возложенными на нее задачами, но она не является объектно-ориентированной. Работая с объектом, лучше всего вызывать методы этого объекта, а не передавать объекты посторонним функциям в качестве аргументов. Этот подход демонстрируется в следующем фрагменте:

```
// Создать объект Rectangle
var r = new Rectangle(8.5, 11);
// Добавить к объекту метод
r.area = function() { return this.width * this.height; }
// Теперь рассчитать площадь, вызвав метод объекта
var a = r.area();
```

Конечно же, не совсем удобно добавлять новый метод к объекту перед его использованием. Однако ситуацию можно улучшить, если инициализировать свойство area в функции конструкторе. Вот как выглядит улучшенная реализация конструктора Rectangle():

```
function Rectangle(w, h) {
 this.width = w;
 this.height = h;
 this.area = function( ) { return this.width * this.height; }
}
```

С новой версией конструктора тот же самый алгоритм можно реализовать по-другому:

```
// Найти площадь листа бумаги формата U.S. Letter в квадратных дюймах
var r = new Rectangle(8.5, 11);
var a = r.area();
```

Свойства area каждого отдельно взятого объекта Rectangle всегда будут ссылаться на **одинаковые экземпляры** (копии) функции area.

Для решения этой проблемы все объекты в JavaScript содержат внутреннюю ссылку на объект, известный как **прототип**. Любые свойства прототипа становятся свойствами другого объекта, для которого он является прототипом. То есть, говоря другими словами, любой объект в JavaScript *наследует* свойства своего прототипа.

```
// Функция конструктор инициализирует те свойства, которые могут
 // иметь уникальные значения для каждого отдельного экземпляра.
 function Rectangle(w, h) {
  this.width = w;
  this.height = h;
 }
 // Прототип объекта содержит методы и другие свойства, которые должны
 // совместно использоваться всеми экземплярами этого класса.
 Rectangle.prototype.area = function() { return this.width * this.height; }
```

Конструктор определяет «класс» объектов и инициализирует свойства, такие как width и height, которые могут отличаться для каждого экземпляра класса. Объект прототип связан с конструктором, и каждый объект, инициализируемый конструктором, наследует тот набор свойств, который имеется в прототипе. Это значит, что объект прототип – идеальное место для методов и других свойств констант.
Свойства *не* копируются из объекта-прототипа в новый объект; они просто присутствуют, как если бы были свойствами этих объектов.

Унаследованные свойства ничем не отличаются от обычных свойств объекта. Они поддаются перечислению в цикле for/in и могут проверяться с помощью оператора in. Отличить их можно только с помощью метода Object.hasOwnProperty():

```
var r = new Rectangle(2, 3);
r.hasOwnProperty("width"); // true: width – непосредственное свойство "r"
r.hasOwnProperty("area"); // false: area – унаследованное свойство "r"
"area" in r; // true: area – свойство объекта "r"
```

Чтение и запись унаследованных свойств

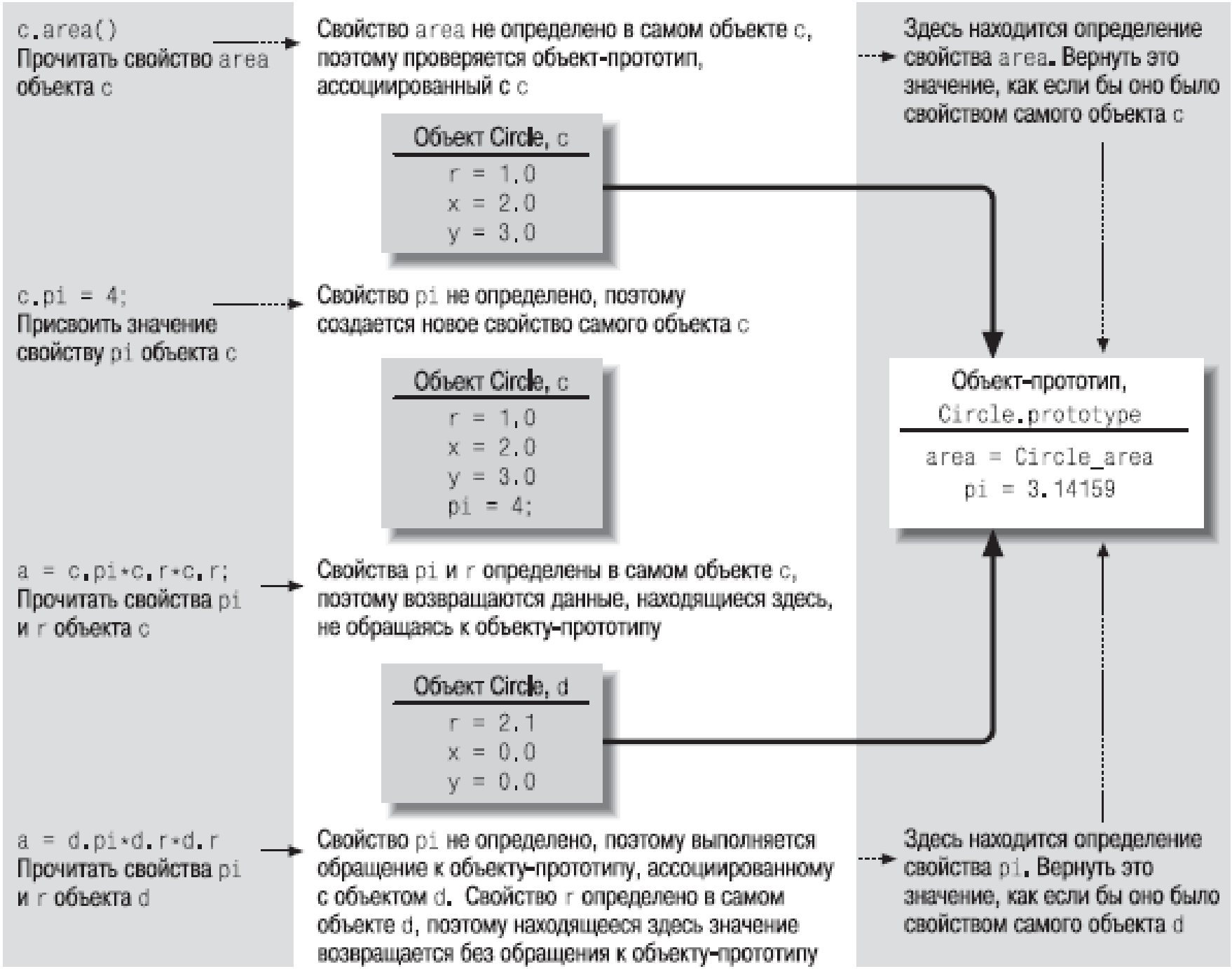
У каждого класса имеется один объект-прототип с одним наборов свойств, но потенциально может существовать множество экземпляров класса, каждый из которых наследует свойства прототипа. Свойство прототипа может наследоваться многими объектами, поэтому интерпретатор JavaScript должен обеспечить фундаментальную асимметричность между чтением и записью значений свойств.

При чтении свойства *prop* объекта *obj*, JavaScript сначала проверяет, есть ли у объекта *obj* свойство с именем *prop*. Если такого свойства нет, то проверяется, есть ли свойство с именем *prop* в объекте-прототипе.

Однако **когда свойству присваивается значение**, JavaScript **не использует объект-прототип**.

Предположим, необходимо установить значение свойства *obj.prop*, а у объекта *obj* нет свойства с именем *prop*. Предположим теперь, что JavaScript найдет свойство *prop* в объекте-прототипе объекта *obj* и изменит значение свойства прототипа. В результате изменится значение *prop* для всего класса объектов, а это не то, что требовалось.

Поэтому **наследование свойств прототипа происходит только при чтении значений свойств, но не при их записи**.


c.area()
Прочитать свойство area объекта c
Свойство area не определено в самом объекте c, поэтому проверяется объект-прототип, ассоциированный с c
Здесь находится определение свойства area. Вернуть это значение, как если бы оно было свойством самого объекта c
Объект Circle, c
r = 1.0
x = 2.0
y = 3.0
c.pi = 4;
Присвоить значение свойству pi объекта c
Свойство pi не определено, поэтому создается новое свойство самого объекта c
Объект Circle, c
r = 1.0
x = 2.0
y = 3.0
pi = 4;
Объект-прототип, Circle.prototype
area = Circle_area
pi = 3.14159
a = c.pi*c.r*c.r;
Прочитать свойства pi и r объекта c
Свойства pi и r определены в самом объекте c, поэтому возвращаются данные, находящиеся здесь, не обращаясь к объекту-прототипу
Объект Circle, d
r = 2.1
x = 0.0
y = 0.0
a = d.pi*d.r*d.r
Прочитать свойства pi и r объекта d
Свойство pi не определено, поэтому выполняется обращение к объекту-прототипу, ассоциированному с объектом d. Свойство r определено в самом объекте d, поэтому находящееся здесь значение возвращается без обращения к объекту-прототипу
Здесь находится определение свойства pi. Вернуть это значение, как если бы оно было свойством самого объекта d

Расширение встроенных типов

Не только классы, определенные пользователем, имеют объекты-прототипы. Встроенные классы, такие как String и Date, также имеют объекты-прототипы, и можно присваивать им значения. Например, следующий фрагмент определяет новый метод, доступный всем объектам String:

```
// Возвращает true, если последним символом является значение аргумента ch
String.prototype.endsWith = function(ch) {
 return (ch == this.charAt(this.length1))
}
```

Определив новый метод endsWith() в объекте прототипе String, мы сможем обратиться к нему следующим образом:

```
var message = "hello world";
message.endsWith('h') // Возвращает false
message.endsWith('d') // Возвращает true
```

Никогда не следует добавлять свойства к объекту Object.prototype.

Любые добавляемые свойства и методы становятся перечислимыми для цикла for/in, поэтому при их добавлении к объекту Object.prototype, они становятся доступными во всех JavaScript объектах. Пустой объект {}, как предполагается, не имеет перечислимых свойств. Любое расширение Object.prototype превратится в перечислимое свойство пустого объекта, что, скорее всего, приведет к нарушениям в функционировании программного кода, который работает с объектами как с ассоциативными массивами.

Свойства экземпляра, методы экземпляра, свойства класса и методы класса

Каждый объект имеет собственные копии ***свойств экземпляра***. Другими словами, если имеется 10 объектов данного класса, то имеется и 10 копий каждого свойства экземпляра. Например, в классе Rectangle любой объект Rectangle имеет свойство width, определяющее ширину прямоугольника.

По умолчанию любое свойство объекта в JavaScript является свойством экземпляра. Однако чтобы по-настоящему имитировать объектно-ориентированное программирование, мы будем говорить, что ***свойства экземпляра в JavaScript – это те свойства, которые создаются и/или инициализируются функцией-конструктором***.

Метод экземпляра во многом похож на свойство экземпляра, за исключением того, что это метод, а не значение. Методы экземпляра вызываются по отношению к определенному объекту, или экземпляру. Метод area() класса Rectangle представляет собой метод экземпляра. В методе экземпляра класса JavaScript приходится явно вставлять ключевое слово this перед именами свойств:

```
return this.width * this.height;
```

Методы экземпляра ссылаются на объект, или экземпляр, с которым они работают, при помощи ключевого слова this. Метод экземпляра может быть вызван для любого экземпляра класса, но это не значит, что каждый объект содержит собственную копию метода, как в случае свойства экземпляра. Вместо этого каждый метод экземпляра совместно используется всеми экземплярами класса. В JavaScript ***метод экземпляра класса определяется путем присваивания функции-свойству объекта-прототипа в конструкторе***. Так, все объекты, созданные данным конструктором, совместно используют унаследованную ссылку.

Свойство класса – это свойство, связанное с самим классом, а не с каждым экземпляром этого класса. Независимо от того, сколько создано экземпляров класса, есть только одна копия каждого свойства класса. Так же, как свойства экземпляра доступны через экземпляр класса, доступ к свойствам класса можно получить через сам класс.

Number.MAX_VALUE – это пример обращения к свойству класса Number в JavaScript, означающая, что свойство MAX_VALUE доступно через класс Number. Так как имеется только одна копия каждого свойства класса - свойства класса по существу являются глобальными.

Свойства класса имитируются в JavaScript простым **определением свойства самой функции-конструктора**.

Например, свойство класса Rectangle.UNIT для хранения единичного прямоугольника с размерами 1x1 можно создать так:

Rectangle.UNIT = new Rectangle(1,1);

Здесь Rectangle – это функция-конструктор, но поскольку функции в JavaScript представляют собой объекты, возможно создать свойство функции точно так же, как свойства любого другого объекта.

Метод класса – это метод, связанный с классом, а не с экземпляром класса; он вызывается через сам класс, а не через конкретный экземпляр класса.

Поскольку методы класса вызываются через функцию-конструктор, они *не* могут использовать ключевое слово this для ссылки на какой-либо конкретный экземпляр класса, поскольку в данном случае this ссылается на саму функцию-конструктор. Обычно ключевое слово this в методах классов вообще не используется.

Пример: метод parse() объекта объекты Date. Этот метод вызываются через сам конструктор Date(), а не через отдельные объекты Date:

Date.parse() Анализирует строковое представление даты и времени и возвращает внутреннее представление этой даты в миллисекундах.

Для того, ***чтобы определить метод класса*** *в JavaScript****, требуется сделать соответствующую функцию свойством конструктора***.

Пример

```
// Конструктор
function Circle(radius) {
 // r – свойство экземпляра, оно определяется и инициализируется конструктором.
 this.r = radius;
}
// Circle.PI – свойство класса, т. е. свойство функции-конструктора.
Circle.PI = 3.14159;
// Метод экземпляра, который рассчитывает площадь круга.
Circle.prototype.area = function( ) { return Circle.PI * this.r * this.r; }
// Метод класса – принимает два объекта Circle и возвращает объект с большим радиусом.
Circle.max = function(a,b) {
  if (a.r > b.r) return a;
  else return b;
}

// Примеры использования каждого из этих полей:
var c = new Circle(1.0); // Создание экземпляра класса Circle
c.r = 2.2; // Установка свойства экземпляра r
var a = c.area(); // Вызов метода экземпляра area()
var x = Math.exp(Circle.PI); // Обращение к свойству PI класса для выполнения расчетов
var d = new Circle(1.2); // Создание другого экземпляра класса Circle
var bigger = Circle.max(c,d); // Вызов метода класса max()
```

Приватные свойства

Одна из парадигм традиционных объектно-ориентированных языков программирования, называемая *инкапсуляцией данных*, заключается в возможности объявления частных (private) свойств класса, обращаться к которым можно только из методов этого класса и недоступных за пределами класса. JavaScript позволяет имитировать частные свойства посредством *замыканий.*

Для этого необходимо, чтобы методы доступа хранились в каждом экземпляре класса и по этой причине не могли наследоваться от объекта-прототипа.

Следующий фрагмент содержит реализацию объекта прямоугольника Rectangle, ширина и высота которого доступны, но не могут изменяться без обращения к специальным методам:

```
function ImmutableRectangle(w, h) {
  // Этот конструктор не создает свойства объекта, где может храниться
  // ширина и высота. Он просто определяет в объекте методы доступа
  // Эти методы являются замыканиями и хранят значения ширины и высоты
 // в своих цепочках областей видимости.
this.getWidth = function() { return w; }
this.getHeight = function() { return h; }
}
  // Класс может иметь  и обычные методы в объекте-прототипе.
ImmutableRectangle.prototype.area = function( ) {
   return this.getWidth( ) * this.getHeight( );
};
```

```
// Функция добавляет методы доступа к свойству объекта "o" с заданными именами. Методы получают
//имена get<name> и set<name>. Если дополнительно предоставляется функция(predicate) проверки,
//метод записи будет использовать ее для проверки значения перед сохранением. Если функция проверки
// возвращает false, метод записи генерирует исключение.
//Значение доступно только для этих двух методов
// и не может быть установлено или изменено иначе, как методом записи.
function makeProperty(o, name, predicate) {
  var value; // This is the property value
  // Метод чтения просто возвращает значение.
  o["get" + name] = function() { return value; };
  // Метод записи сохраняет значение или генерирует исключение,
  // если функция проверки отвергает это значение.
  o["set" + name] = function(v) {
    if (predicate && !predicate(v))   throw "set" + name + ": неверное значение " + v;
    else   value = v;
   };
}
var o = {}; // Пустой объект
// Добавить методы доступа к свойству с именами getName() и setName()
// Обеспечить допустимость только строковых значений
makeProperty(o, "Name", function(x) { return typeof x == "string"; });
o.setName("Frank"); // Установить значение свойства
print(o.getName( )); // Получить значение свойства
o.setName(0); // Попытаться установить значение ошибочного типа
```

Наследование в JavaScript

В Java, C++ и других объектно-ориентированных языках на базе классов имеется явная концепция *иерархии классов.* Каждый класс может иметь *надкласс*(родительский класс)*,* от которого он наследует свойства и методы. Любой класс может быть расширен, т. е. иметь *подкласс*(дочерний класс)*,* наследующий его поведение*.*

Как было показано ранее, *JavaScript* вместо наследования на базе классов поддерживает наследование прототипов .

Тем не менее в JavaScript могут быть проведены аналогии с иерархией классов. В JavaScript класс Object – это наиболее общий класс, и все другие классы являются его специализированными версиями, или подклассами. Можно сказать, что Object – это надкласс всех встроенных классов. Все классы наследуют несколько базовых методов класса Object.

Объекты JavaScript наследуют свойства от объекта-прототипа их конструктора. Но объект-прототип сам представляет собой объект; он создается с помощью конструктора Object(). Это значит, что объект-прототип наследует свойства от Object.prototype!

ПРИМЕР. Предположим, что мы хотим создать дочерний класс класса Rectangle, чтобы добавить в него свойства и методы, связанные с координатами прямоугольника.

Для этого необходимо, чтобы объект-прототип нового класса сам является экземпляром Rectangle и потому наследовал все свойства Rectangle.prototype.

```
// Определение простого класса прямоугольников. Этот класс имеет ширину и высоту и может вычислять свою площадь
function Rectangle(w, h) {
this.width = w; this.height = h;
}
Rectangle.prototype.area = function( ) { return this.width * this.height; }
// Далее идет определение подкласса
function PositionedRectangle(x, y, w, h) {
// В первую очередь необходимо вызвать конструктор надкласса для инициализации свойств width и height
//нового объекта.  Здесь используется метод call, чтобы конструктор был вызван как метод  инициализируемого объекта.
//(В ECMAScript есть два метода, определенные для всех функций, – call() и apply(). Эти методы позволяют вызывать
//функцию так, будто она является методом некоторого объекта. Первый аргумент методов call() и apply() – это объект, для
//которого выполняется функция; этот аргумент становится значением ключевого слова this в теле функции. Все оставшиеся
//аргументы call() – это значения, передаваемые вызываемой функции.)
  Rectangle.call(this, w, h);// вызов по цепочке - inherited
   // Далее сохраняются координаты верхнего левого угла прямоугольника
  this.x = x;  this.y = y;
}
// Если мы будем использовать объект-прототип по умолчанию, который создается при определении конструктора
//PositionedRectangle(), был бы создан подкласс класса Object.
// Чтобы создать подкласс класса Rectangle, необходимо явно создать объект-прототип.
PositionedRectangle.prototype = new Rectangle();
// Мы создали объект-прототип с целью наследования, но мы не собираемся  наследовать свойства width и height, которыми
// уже обладают все объекты класса Rectangle, поэтому удалим их из прототипа.
delete PositionedRectangle.prototype.width;
delete PositionedRectangle.prototype.height;
// Поскольку объект-прототип был создан с помощью конструктора Rectangle(), свойство constructor в нем ссылается на этот
// конструктор. Но необходимо, чтобы объекты PositionedRectangle ссылались на свой конструктор, поэтому далее
// выполняется присваивание нового значения свойству constructor
PositionedRectangle.prototype.constructor = PositionedRectangle;
// Теперь есть правильно настроенный прототип для подкласса, и можно приступать к добавлению методов экземпляров.
PositionedRectangle.prototype.contains = function(x,y) {
return (x > this.x && x < this.x + this.width &&
y > this.y && y < this.y + this.height);}
```

Наследование с использованием функции extend

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

```
// создание базового класса
function A(..) { ... }

// создание дочернего класса
function B(..)  { ... }
//наследование
extend(B, A)

// добавление в дочерний класс методов и свойств
B.prototype.BMethod = function(..) { ... }

// создание объекта класса-потомка и использование метода класса-родителя
b = new B(..)
b.AMethod()
```

Копирование свойств объекта

```
// копирует все свойства из src в dst,
// включая все те, что в цепочке прототипов src до Object
function mixin(dst, src){

for(var x in src){
        // копируем в dst свойства src, кроме тех, которые унаследованы от
Object
        if((typeof {}[x] == "undefined") || ({}[x] != src[x])){
            dst[x] = src[x];
        }
    }

}
```

```
/**
* Tooltip.js: простейшие всплывающие подсказки, отбрасывающие тень.
** Этот модуль определяет класс Tooltip. Объекты класса Tooltip создаются
* с помощью конструктора Tooltip(). После этого подсказку можно сделать
* видимой вызовом метода show(). Чтобы скрыть подсказку, следует
* вызвать метод hide().
** Обратите внимание: для корректного отображения подсказок с использованием
* этого модуля необходимо добавить соответствующие определения CSS_классов
* .tooltipShadow {
* background: url(shadow.png); /* полупрозрачная тень * /
* }
*
* .tooltipContent {
* left: _4px; top: _4px; /* смещение относительно тени * /
* background_color: #ff0; /* желтый фон * /
* border: solid black 1px; /* тонкая рамка черного цвета * /
* padding: 5px; /* отступы между текстом и рамкой * /
* font: bold 10pt sans_serif; /* небольшой жирный шрифт * /
* }
*
* В браузерах, поддерживающих возможность отображения полупрозрачных
* изображений формата PNG, можно организовать отображение полупрозрачной
* тени. В остальных браузерах придется использовать для тени сплошной цвет
* или эмулировать полупрозрачность с помощью изображения формата GIF.
*/
function Tooltip( ) { // Функция_конструктор класса Tooltip
this.tooltip = document.createElement("div"); // Создать div для тени
this.tooltip.style.position = "absolute"; // Абсолютное позиционирование
this.tooltip.style.visibility = "hidden"; // Изначально подсказка скрыта
this.tooltip.className = "tooltipShadow"; // Определить его стиль
this.content = document.createElement("div"); // Создать div с содержимым
this.content.style.position = "relative"; // Относительное позиционирование
this.content.className = "tooltipContent"; // Определить его стиль
this.tooltip.appendChild(this.content); // Добавить содержимое к тени
}
// Определить содержимое, установить позицию окна с подсказкой и отобразить ее
Tooltip.prototype.show = function(text, x, y) {
this.content.innerHTML = text; // Записать текст подсказки.
this.tooltip.style.left = x + "px"; // Определить положение.
this.tooltip.style.top = y + "px";
this.tooltip.style.visibility = "visible"; // Сделать видимой.
// Добавить подсказку в документ, если это еще не сделано
if (this.tooltip.parentNode != document.body)
document.body.appendChild(this.tooltip);
};
// Скрыть подсказку
Tooltip.prototype.hide = function() {
this.tooltip.style.visibility = "hidden"; // Сделать невидимой.
};
```