Вадим Дунаев

Основы JavaScript

Здесь рассматриваются основные понятия, синтаксис и объекты JavaScript без претензий на исчерпывающую полноту. Тем не менее, приведенный материал можно использовать в качестве краткого справочника и элементарного учебника для начинающих. Нумерация разделов согласована с моей книгой "**HTML, скрипты и стили**", ***3-е издание, 2011г.***.

16.1. Ввод и вывод данных

Сценарии на JavaScript могут взаимодействовать с объектами браузера и загруженного в него документа. Для ввода и вывода данных можно воспользоваться методами этих объектов. Каких-то специфических собственных методов вывода данных у JavaScript, интерпретируемого Web-браузерами, не предусмотрено.

Объект, представляющий свойства браузера, называется `window` (окно), а три его метода — `alert()`, `prompt()` и `confirm()` — предназначены для ввода и вывода данных посредством диалоговых окон (панелей). Поскольку это методы объекта `window`, то для их вызова, согласно каноническим правилам, следует писать `window.alert()`, `window.prompt()` и `window.confirm()` соответственно. Здесь используется так называемый точечный синтаксис, согласно которому перед обращением к свойству (переменной или методу) указывают соответствующий объект, за которым следует точка. Вместе с тем, поскольку объект `window` корневой в объектной модели, при обращении к его свойствам имя можно опустить. В этом случае интерпретатор и так поймет, что вызываемый метод относится к объекту `window`. Иначе говоря, допускается сокращенная запись без префикса `window`. Например, если мы хотим вывести что-то, обозначенное через `x`, то можно написать в программе `alert(x)`.

Внешний вид указанных диалоговых окон зависит от браузера. На рис. 16.1 показано, как они выглядят в Internet Explorer.

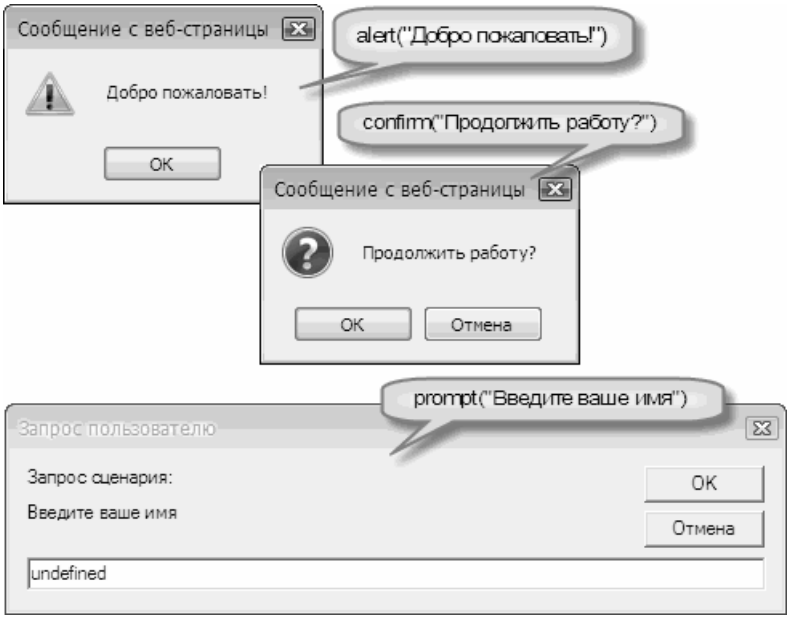

Рис. 16.1. Диалоговые окна, созданные методами `alert()`, `confirm()` и `prompt()`

Объект, представляющий свойства документа, называется `document`, а его методы `write()` и `writeln()` служат для вывода данных непосредственно в клиентскую область браузера, т. е. туда, где отображаются Web-страницы. Объект `document` очень важный, но не корневой в объектной модели, а потому при обращении к его свойствам требуется указывать имя объекта, например,

```
document.write("Привет!").
```

В следующих разделах мы рассмотрим перечисленные методы подробнее.

16.1.1. Метод *alert()*

Данный метод позволяет выводить диалоговое окно с заданным сообщением и кнопкой **OK**. В такое окно обычно выводят предупреждения. Его также удобно использовать для вывода промежуточных результатов при отладке скриптов. Синтаксис соответствующего выражения:

`alert(`*`сообщение`*`).`

Если ваше *`сообщение`* конкретно, т. е. представляет собой вполне определенный набор символов, то его необходимо заключить в двойные или одинарные кавычки: `alert("Привет!")`. Вообще говоря, *`сообщение`* представляет собой

данные любого типа: последовательность символов, заключенную в кавычки, число (в кавычках или без них), переменную или выражение.

Диалоговое окно, выведенное на экран методом `alert()`, можно закрыть, щелкнув на кнопке **OK**. До тех пор, пока вы не сделаете этого, переход к ранее открытым окнам невозможен. Окна, обладающие свойством останавливать все последующие действия пользователя и программ, называются *модальными*. Таким образом, метод `alert()` создает модальное окно.

Ранее уже отмечалось, что метод `alert()` пригоден для вывода промежуточных и окончательных результатов программ при их отладке. При этом можно отобразить результат вычисления какого-либо выражения и приостановить дальнейшее выполнение работы программы до тех пор, пока вы не нажмете кнопку **OK**.

16.1.2. Метод *confirm()*

Метод `confirm` позволяет вывести диалоговое окно с сообщением и двумя кнопками: **OK** и **Отмена** (Cancel). В отличие от `alert()`, этот метод возвращает логическую величину, значение которой зависит от того, какую из кнопок нажал пользователь. Если он щелкнул на кнопке **OK**, то возвращается значение `true` (истина, да); если же была нажата кнопка **Отмена** (Cancel), то возвращается значение `false` (ложь, нет). Возвращаемое значение можно обработать
в программе и, следовательно, создать эффект интерактивности, т. е. диалогового взаимодействия программы с пользователем. Синтаксис метода `confirm()`:

```
confirm(сообщение).
```

Если *`сообщение`* представляет собой вполне определенный набор символов, то его необходимо заключить в кавычки, двойные или одинарные:

```
confirm("Вы действительно хотите выйти из программы?").
```

Как и ранее *`сообщение`* может быть любым: последовательностью символов, заключенной в кавычки, числом (в кавычках или без них), переменной или выражением.

Диалоговое окно, выведенное на экран методом `confirm()`, можно убрать щелчком на любой из двух кнопок (**OK** или **Отмена**). До тех пор, пока вы не сделаете этого, переход к ранее открытым окнам невозможен. Следовательно, окно, создаваемое посредством `confirm()`, модальное. Если пользователь щелкнет на кнопке **OK**, то метод вернет логическое значение `true` (истина, да), а если он щелкнет на кнопке **Отмена** (Cancel), то — `false`. Возвращаемое значение доступно для дальнейшей обработки в программе и реализации эффекта интерактивности, т. е. диалогового взаимодействия программы с пользователем.

16.1.3. Метод *prompt()*

Метод `prompt()` позволяет вывести на экран диалоговое окно с сообщением, а также с текстовым полем, в которое пользователь может ввести данные. Кроме того, в окне предусмотрены две кнопки: **OK** и **Отмена** (Cancel). В отличие от `alert()` и `confirm()`, данный метод принимает два параметра: сообщение и значение, которое должно появиться в текстовом поле ввода данных по умолчанию. Если пользователь щелкнет на кнопке **OK,** метод вернет содержимое поля ввода данных, а если он щелкнет на кнопке **Отмена**, то возвращается логическое значение `false` (ложь, нет). По аналогии с предыдущими методами возвращаемое значение можно проанализировать и создать эффект интерактивности.

Синтаксис метода `prompt()`:

`prompt(`*сообщение, значение_поля_ввода_данных*`).`

Параметры метода `prompt()` необязательные. Если вы их не укажете, то будет выведено окно без сообщения, а в поле ввода данных Internet Explorer подставит значение по умолчанию — `undefined` (не определено), а другие браузеры — пустую строку. Если вы не хотите, чтобы в поле ввода данных появлялось значение по умолчанию, то в качестве значения второго параметра укажите пустую строку `""`:

```
prompt("Введите Ваше имя", "").
```

Диалоговое окно, выведенное на экран методом `prompt()`, можно закрыть щелчком на любой из двух кнопок, **OK** или **Отмена**. Пока данное окно открыто, переход к ранее открытым окнам невозможен, т. е. диалоговое окно модальное.

16.1.4. Метод *document.write()*

Метод `document.write(список_строк)` позволяет вывести в окно браузера список текстовых строк, разделенных запятыми. Если выводимая строка представляет собой (X)HTML-код, то браузер отобразит результат его интерпретации, а не последовательность тегов. Иначе говоря, теги HTML в выводимом сообщении браузер воспринимает как команды, а не просто как текстовые символы.

Метод `document.writeln(список_строк)` отличается от предыдущего только тем, что выводит указанную строку (или несколько строк), добавляя в конце невидимый управляющий символ перехода на новую строку. Однако этот и другие специальные символы браузер не воспринимает как команды, управляющие отображением, и заменяет их пробелом, т. е. игнорирует. Если требуется выполнить какое-либо форматирование выводимого текста, то следует указать соответствующие теги HTML, например, для перевода на новую стро-

ку пригодны теги `<br/>` и `<p>`. На рис. 16.2 показан пример вывода в окно браузера трех текстовых строк, первые две из которых содержат HTML-теги.

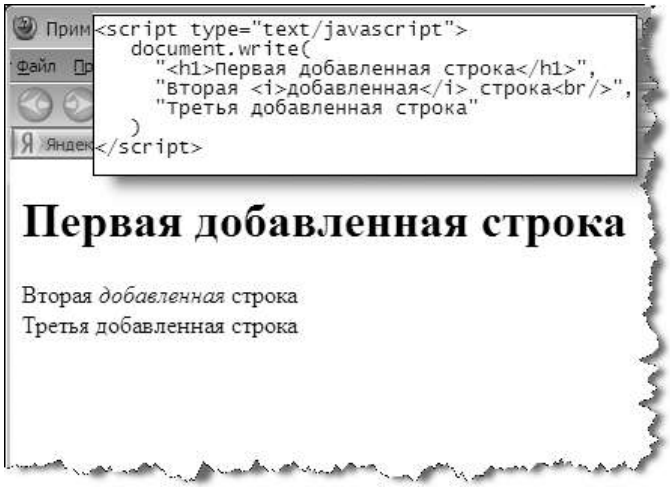

Рис. 16.2. Пример вывода текстовых строк методом `document.write()`

Как уже говорилось, строки, которые вы хотите вывести в окно браузера, указывают в качестве параметров метода `document.write()`, разделяя запятыми. Вместе с тем, эти строки можно представить как одну длинную строку или же объединить их в одну строку с помощью оператора `+`, означающего в JavaScript для строк операцию конкатенации или, другими словами, склейки.

С помощью метода `document.write()` можно сформировать часть или даже весь (X)HTML-код документа. При этом исходный html-файл может содержать только сценарий, формирующий (X)HTML-код, записываемый в документ посредством `document.write()`. Однако не следует думать, что с помощью этого метода можно частично изменять содержимое уже загруженного в браузер документа. Если документ уже полностью загружен, то вызов `document.write()` из его сценария приведет к отображению содержимого параметров данного метода в новом окне или новой вкладке браузера. Поэтому следует соблюдать осторожность при использовании `document.write()`. Изменить содержимое документа без его перезагрузки можно с помощью свойства `innerHTML`, которым обладает любой (X)HTML-элемент (см. *разд. 18.3.3*).

16.2. Типы данных

Данные в JavaScript, как и в любом другом языке программирования, относятся к тому или иному типу. Типы данных, в свою очередь, разделяют на примитивные и составные.

Примитивные типы содержат простые значения, с которыми мы встречаемся в повседневной жизни: числа, строки и логические значения вроде "Да" и "Нет". Так, с числами нам приходится оперировать не только при расчетах перед кассовым аппаратом в магазине, но и при решении более сложных математических задач. Семантику чисел определяет математика. Строки включают какие угодно символы и их семантика заранее не определяется.

Произвольные данные, без структуры и достаточно четко оформленных идей их дальнейшего употребления, программисту дозволено и рекомендуется оформлять в виде строк. Восприятие строк — дело пользователя, а не интерпретатора языка. Однако применительно к скриптовым (интерпретируемым, а не компилируемым) языкам последнее замечание не вполне точно. Интерпретируемые языки, такие как JavaScript и PHP, не "оставляют в покое" строки. Символьные последовательности, участвующие в выражениях с данными других типов, автоматически преобразуются к соответствующему, с точки зрения интерпретатора, типу (см. разд. 16.2.3). Интерпретаторы этих языков пытаются "извлечь" некий смысл, который, возможно, пользователь или программист в них вложил. Например, встречаясь со строкой "25", содержащей число 25, интерпретатор JavaScript попытается привести его к тому или иному типу в зависимости от контекста. Для новичков такая поддержка их интуиции со стороны интерпретатора языка более чем кстати. Но для искушенных программистов, привыкших все держать под своим контролем, это просто "беспредел".

Строка, пусть даже содержащая одни только цифры, не является числом. Поэтому арифметические операции над строками, содержащими числа, могут привести к результатам, существенно отличным от тех, которые мы ожидаем в случае действий над самими числами.

Чтобы как-то определиться, мы должны признать, что число — объект, достаточно ясно определенный в математике, — в языке программирования представляется символически так или иначе. Строка — это набор произвольных символов, которую нельзя интерпретировать, например, как число, дату или что-либо другое. Над строками мы производим одни операции, а над числами — другие. Логическое значение — одно из двух возможных значений, которые обычно обозначаются как true (ИСТИНА) и false (ЛОЖЬ). Для работы с логическими значениями существует своя алгебра,

отличная от арифметики. Мы можем сформулировать некое утверждение, которое в данном контексте может быть истинным или ложным. Для обозначения оценки истинности логического выражения в языках программирования предусмотрены специальные ключевые слова: `true` (истина) и `false` (ложь).

Составные типы данных, в отличие от примитивных, могут содержать разнородные данные (в том числе и других составных типов). Типичный пример составного типа данных — массив. Массив в JavaScript — это упорядоченный набор данных (называемых элементами), каждое из которых принадлежит к тому или иному типу. Место каждого элемента в массиве фиксировано, а потому доступ к ним возможен по порядковому номеру (индексу). Объект — еще один составной тип данных, причем настолько общий, что позволяет представить все другие составные типы данных, такие как массивы, даты, функции и даже объекты. В объекте данные не обязаны быть упорядоченными, как в массиве.

16.2.1. Примитивные типы данных

В JavaScript имеются пять примитивных типов данных (табл. 16.1).

Таблица 16.1. *Примитивные типы данных в JavaScript*

Тип данных	**Примеры**	**Описание значений**

Строковый или символьный (`String`)	`"Привет всем!"`
	`"т.123-4567"`
	`"Сегодня 14.01.2010г."`
		Последовательность символов, в том числе и пустая, заключенная в кавычки, двойные или одинарные

Числовой (`Number`)	`3.14`
	`-567`
	`+2.5`
	`5.7e16`
	`67e-28`
		Десятичное число, представленное в виде последовательности цифр, перед которой может быть указан знак числа (+ или −); перед положительными числами не обязательно ставить знак +; целая и дробная части чисел разделяются точкой. Возможно представление в экспоненциальной, шестнадцатеричной и восьмеричной формах. Числа записываются без кавычек.
		Диапазон чисел: от $\pm 2{,}2250 \times 10^{-308}$

		до ±1,7976×10^{308}. Диапазон целых чисел: от –2^{31} до 2^{31} – 1
Логический (булевый, `Boolean`)	`true` `false`	Два значения: `true` (истина, да) и `false` (ложь, нет)
Пустой `(Null)`	`null`	Одно значение — `null`, указывающее на отсутствие какого бы то ни было значения
Неопределенный `(Undefined)`	`undefined`	Одно значение — `undefined`, указывающее, что переменной не присвоено никакое значение

Из перечисленных типов только `String`, `Number` и `Boolean` могут содержать настоящие данные, а `Null` и `Undefined` служат специальным целям.

Если вы не хотите присваивать переменной какое-либо конкретное значение, то назначьте ей для определенности значение `null`. Это действие будет отвечать вашим чаяниям наилучшим образом. Переменная, имеющая значение `null`, определена. Она имеет пустое значение. Заметим, что пустая строка (не содержащая ни одного символа, даже пробела), а также число 0, — данные, отличающиеся от `null`.

Если какая-то переменная встречается в выражении впервые, и ей ранее не присвоено никакое значение, то это может вызвать ошибку выполнения, как в листинге 16.1 слева, справа показано, как избежать появления ошибки.

Листинг 16.1. Пример инициализации переменной

Код с ошибкой

```
<html>
<script type="text/javascript">
   alert(x); /* здесь возникнет
                ошибка */
</script>
</html>
```

Код без ошибки

```
<html>
<script type="text/javascript">
   x = null;
   alert(x) /* выведет сообщение
               "null", но ошибки не
                будет */
</script>
</html>
```

Если переменная объявлена с помощью оператора `var`, например, `var x`, то она не имеет пока никакого значения и относится к типу `undefined` (листинг 16.2).

Листинг 16.2. Пример переменной типа `undefined`

```
<html>
<script type="text/javascript">
   var x;   // объявление переменной без присвоения ей значения
   alert(x); // выведет сообщение "undefined", но ошибки не будет
</script>
</html>
```

16.2.2. Составные типы данных

В JavaScript поддерживаются три составных типа данных: объекты, массивы и функции. В действительности массивы и функции также являются объектами. Другими словами, такие понятия как массив и функция реализованы в JavaScript в виде объектов.

Объект — понятие, обладающее настолько большой общностью, что противопоставляется исторически более раннему понятию "тип данных". Действительно, объект есть контейнер, содержащий переменные и функции, или одну из этих категорий: только переменные или только определения функций. В свою очередь, переменные внутри объекта могут содержать данные как примитивных, так и составных типов (в том числе и данные типа объект). Последнее обстоятельство как раз и позволяет конструировать сколь угодно сложные типы данных. Если исходить из более традиционного понятия "тип данных", то объект — это сложный (составной) тип данных. С другой стороны, поддержка как примитивных, так и сложно устроенных данных может быть обеспечена соответствующими объектами.

Исторически сложилось, что сначала появилось понятие типа данных (в смысле множества допустимых значений). Более того, в первых языках программирования это было множество однородных значений, например, только чисел, строк или логических величин. Нельзя было сформировать тип данных, состоящий одновременно, например, из чисел и строк (числа и строки — разнородные данные). Далее появились структуры в виде упорядоченного множества (последовательности) данных различных типов. Структура — это составной тип данных, если угодно прибегнуть к такой интерпретации. Наконец, было введено еще более общее понятие объекта, который мог содержать не только разнородные значения (как и структура), но еще и функции (называемые методами). Другими словами, объект может содержать не только пассивные данные (переменные), но и активные (функции).

Обратите внимание, что при разъяснении того, что такое объект, мы использовали понятие "тип данных". Но если мы усвоили понятие "объект", то его, как будто, можно рассматривать в качестве первичного по отношению к понятию "составной тип данных". Однако это можно сделать лишь с некоторой натяжкой. Понятия "тип данных" и "объект" не вполне эквивалентны. Многие типы данных реализуются и представляются в конкретных языках, в том числе и в JavaScript, посредством соответствующих объектов — программных конструкций особого рода. Например, массив реализован и представлен в JavaScript как некий объект под названием `Array`. Этот объект имеет ряд свойств, содержащих не только элементы представляемого им массива, но и другие важные вещи: количество всех элементов массива и методы работы с ним (например, сортировка элементов, объединение двух массивов). Однако объект — неупорядоченное множество его свойств, а массив — упорядоченное множество его элементов. Кроме того, операции над массивом являются чем-то внешним по отношению к самому массиву — упорядоченному набору данных. По этим и некоторым другим причинам массивы выделяют в особый тип данных, отличающийся от объектов. Как бы то ни было, начинающие, и не только, могут относиться к массивам как к особому типу данных, т. е. как к специальной конструкции, идея и польза которой очевидны. Позже мы уточним это понятие настолько, чтобы его можно было легко применять в программировании.

Аналогично, все, что связано с понятием функции, можно реализовать и представить в виде некоторого объекта. В JavaScript каждая функция реализуется посредством объекта `Function`. Такое представление обладает многими достоинствами, которые оказываются востребованными при составлении более или менее сложных программ. Однако любой программист может ограничиться традиционным пониманием функции как блока программного кода, который можно вызвать для исполнения по имени этой функции. В этом смысле функцию можно отнести к особому типу данных. Интерпретаторы, прежде чем начать выполнение кода, анализируют его в целом на присутствие в нем определений функций, т. к. код функций необходимо "усвоить" прежде, чем он будет вызван где-то и когда-то (до или после определения). В отличие от функций, другие операторы выполняются интерпретатором друг за другом в порядке их упоминания в исходном коде.

Если все сказанное внесло некоторую сумятицу в сложившуюся у вас систему понятий, то не расстраивайтесь, со временем все станет на свои места. На первых порах вы можете пропустить как будто путаные общие рассуждения о типах и объектах, сосредоточившись пока на чем-то более понятном (например, на простом определении типа данных). Простейшие типы данных, как я уже говорил, — это примитивные данные, такие как числа, строки символов и двухэлементное множество, содержащее логические значения `true`

и `false`. Необходимость в других типах данных обусловлена более сложными задачами их обработки.

В JavaScript имеется множество встроенных объектов, обеспечивающих работу с числами, строками, массивами, датами, функциями и др. Некоторые из этих объектов соответствуют, как говорят, типам данных. Например, для работы с массивами имеется объект `Array`, для манипулирования датами и временем — `Date`, а для выполнения более или менее сложных математических вычислений — `Math`. Перечень встроенных объектов приведен в табл. 16.2, а их подробное описание будет приведено в последующих разделах данной главы.

Таблица 16.2. *Встроенные объекты в JavaScript*

Объект	**Описание**
`Array`	Обеспечивает массив данных
`Boolean`	Соответствует логическому типу данных `Boolean`
`Date`	Обеспечивает работу с датами и временем
`Error`	Обеспечивает хранение информации об ошибках и возможность создания исключительных ситуаций
`Function`	Обеспечивает возможности, связанные с функциями, например, проверку количества переданных параметров
`Global`	Обеспечивает встроенные функции для преобразования и сравнения данных
Объект	**Описание**
`Math`	Обеспечивает множество математических функций и констант. Применяется для выполнения более сложных вычислений, чем те, которые реализуются арифметическими операторами
`Number`	Соответствует числовому типу данных
`Object`	Базовый объект, из которого получаются все остальные объекты, в том числе и пользовательские
`RegExp`	Обеспечивает сложные способы обработки строк на основе так называемых регулярных выражений
`String`	Соответствует строковому типу данных

16.2.3. Автоматическое преобразование типов данных

При составлении выражений на языке программирования нередко бывает, что вовлекаемые в обработку данные принадлежат различным типам, в то время как операции над ними могут быть корректно выполнены, если данные однотипны. Например, пусть переменная *x* имеет числовое значение 2, а переменная *y* — строковое значение "3". С обывательской точки зрения и то, и другое — числа, но для интерпретатора языка JavaScript это разные вещи: переменная *x* содержит само число, а переменная *y* — строку, содержащую число. Числа обычно составляются из цифр, знаков положительного и отрицательного числа, разделителя целой и дробной частей, а также других специальных символов. Строка — набор произвольных символов, который заключают в кавычки. Что будет означать операция сложения этих разнотипных данных, суммирование чисел или склейку соответствующих строк? Попытка сложить значения этих переменных в JavaScript даст в результате не число 5, а строку "23" (рис. 16.3).

```
<html>
<head><title>Пример</title>
</head>
<script type="text/javascript">
    x = 2;
    y = "3";
    alert(x + y);
</script>
</html>
```

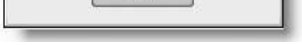

Рис. 16.3. Сложение числа и строки дает в результате строку

В данном случае интерпретатор JavaScript автоматически преобразует число 2 в строку "2", а операцию "+" выполнит как склейку строк "2" и "3".

JavaScript — язык со слабым или, как еще говорят, динамическим контролем типов данных. Это означает следующее:

- Одна и та же переменная может принимать значения различных типов. Переменная создается при первом ее упоминании в программе, а тип данных, которые ей можно присвоить, не задан раз и навсегда. В одном месте программы этой переменной можно присвоить значение одного типа, а в другом месте — иного типа. Например, вполне допустима такая последовательность операторов присваивания: `x=2; x="Вася"`.

> ***Примечание***
>
> Во многих других языках (например, C, Pascal) при объявлении переменной устанавливается и ее тип, который остается постоянным. Попытка присвоить такой переменной значение недопустимого типа всегда приводит к сообщению об ошибке. Но в JavaScript это не так.

- При выполнении операций над данными различных типов интерпретатор автоматически приводит их к некоторому общему требуемому типу. Например, при сложении числа и строки число приводится к строковому типу, в результате вторая строка приписывается ("приклеивается") к первой. Так, результат выражения 125 + "Вася" — строка "125Вася". Рассмотрим еще один пример. Иногда в выражении по смыслу требуется логический тип. Так, в операторе условия `if(x){…}` (читается "если истинно `x`, то выполнить `{…}`") подразумевается, что переменная `x` должна иметь логическое значение (`true` или `false`). Однако в действительности она может принимать значение любого типа, но интерпретатор приведет его к логическому типу автоматически сообразно с некоторыми правилами.

Как происходит автоматическое преобразование типов, очень важно знать, чтобы избежать возможных недоразумений. Однако существуют методы принудительного приведения к заданному типу, которые позволяют контролировать типы данных вручную (см. разд. 16.2.4).

Преобразование строк (*String*)

Правила автоматического преобразования строк (данных типа `String`) в данные других типов:

- **Преобразование в Boolean** — в результате получается `false` (ложь), если строка пуста (""), т. е. не содержит ни одного символа, даже пробела (строка, содержащая один или более пробелов не пуста); в противном случае — `true` (истина).
- **Преобразование в Number** — происходит анализ строки как числового литерала с целью получения подходящего значения. Если этот процесс заканчивается неудачей, то возвращается константа `NaN` (от Not a Namber — не число); в противном случае возвращается число. Например, строка "5.2" преобразуется в число 5.2, а в результате преобразования строки "100px" будет получено `NaN`.
- **Преобразование в Object** — возвращается объект `String`, свойство `value` которого равно значению данной строки.

Преобразование чисел (*Number*)

Правила автоматического преобразования чисел (данных типа `Number`) в данные других типов:

- **Преобразование в Boolean** — в результате получается `false` (ложь), если число равно нулю или NaN (Not a Number — не число); в противном случае — `true` (истина). `NaN` — константа, содержащаяся в объекте `Number`.

- **Преобразование в String** — в результате получается строка, содержащая (представляющая) данное число. Например, число 3.14 преобразуется в строку "3.14".
- **Преобразование в Object** — возвращается объект `Number`, свойство `value` которого равно значению данного числа.

Преобразование логических значений (*Boolean*)

Правила автоматического преобразования логических значений (данных типа `Boolean`) в данные других типов:

- **Преобразование в Number** — в результате получается 1, если `true` (истина); в противном случае — 0.
- **Преобразование в String** — в результате получается строка "true", если `true`; в противном случае — строка "false".
- **Преобразование в Object** — возвращается объект `Boolean`, свойство `value` которого равно данному логическому значению.

Преобразование пустого значения (*null*)

Правила автоматического преобразования пустого значения (данных типа `null`) в данные других типов:

- **Преобразование в Boolean —** в результате получается `false` (ложь).
- **Преобразование в Number** — в результате получается 0.
- **Преобразование в String —** в результате получится строка "null".
- **Преобразование в Object** — генерируется исключительная ситуация TypeError (ошибка преобразования типа).

Преобразование неопределенного значения (*undefined*)

Правила автоматического преобразования неопределенного значения (данных типа undefined) в данные других типов:

- **Преобразование в Boolean —** в результате получается `false` (ложь).
- **Преобразование в Number** — в результате получается `NaN` (не число).
- **Преобразование в String —** результате получится строка "undefined" (не определено).
- **Преобразование в Object** — генерируется исключительная ситуация TypeError (ошибка преобразования типа).

16.2.4. Принудительное преобразование типов данных

При попытке выполнить операции над разнотипными данными последние автоматически приводятся к определенному типу так, чтобы выражение могло быть выполнено (см. разд. 16.2.3). Во многих случаях результат оказывается ожидаемым, но иногда возникают недоразумения, устранить которые часто удается принудительным приведением данных к требуемому типу. Например, пусть переменная `width="100px"` хранит значение ширины окна в пикселах, причем единица измерения (px — пикселы) указана в самом этом значении строкового типа. Если мы захотим увеличить размер окна на 10 пикселов, используя для этого выражение `width = width + 10`, то значение переменной `width` станет равным `"100px10"`, а не ожидаемой величине 110. В данном случае интерпретатор автоматически приведет число 10 к строковому значению "10" и припишет его справа к строковому значению "100px". Чтобы добиться своего, нам потребуется принудительно преобразовать строку "100px" в соответствующее число, а именно в 100.

Для преобразования строк в числа в JavaScript предусмотрены встроенные функции `parseInt()` и `parseFloat()`. В действительности это методы объекта `global`, предназначенного для хранения общедоступных данных и методов. В силу глобальности объекта `global` для обращения к его методам не нужно указывать сам объект, т. е. для вызова `parseInt(x)` достаточно просто записать `parseInt(x)`. Возможно, по этой причине методы объекта `global` называют встроенными функциями JavaScript.

Функция

`parseInt(`*`строка`*`, `*`основание`*`)`

преобразует указанную в параметре строку в целое десятичное число. Второй параметр, *`основание`*, определяет систему счисления, в которой представлено содержащееся в строке число (8, 10 или 16); если основание не указано, то предполагается 10, т. е. десятичная система счисления. При этом число округляется простым отбрасыванием дробной части.

Примеры

```
parseInt("3.14")            // результат = 3
parseInt("-7.875")          // результат = -7
parseInt("123")             // результат = 123
parseInt("Саша")            // результат = NaN, т. е. не является числом
parseInt("25 руб. 50 коп.") // результат = 25
parseInt("25.5e3")          // результат = 25
parseInt("15",8)            // результат = 13
```

```
parseInt("0xFF",16)          // результат = 255
parseInt("ff",16)            // результат = 255
```

Функция

`parseFloat(`*строка*`)`

преобразует указанную строку в десятичное число с плавающей разделительной (десятичной) точкой.

Примеры

```
parseFloat("3.14")      // результат = 3.14
parseFloat("-7.875")    // результат = -7.875
parseFloat ("123")      // результат = 123
parseFloat ("Саша")     // результат = NaN, т. е. не является числом
parseFloat("25 руб. 50 коп.") // результат = 25
parseFloat ("25.5")           // результат = 25.5
parseFloat ("25.5 рублей")    // результат = 25.5
parseFloat("25.5e3")          // результат = 25500
```

Данные функции исследуют строку на содержание в ней числа. Если первый символ не цифра или знак числа, то сразу возвращается `NaN`. В противном случае анализируется следующий символ строки и если это не цифра, то анализ прекращается и возвращается число, выделенное на предыдущих этапах. Метод `parseInt()` закончит работу, как только встретит символ, не являющийся цифрой (например, точку, пробел или букву). Поэтому строка, представляющая собой число в экспоненциальной форме, будет преобразована в число, содержащее только целую часть мантиссы. Например, `parseInt("25.5e3")` вернет 25, а не 25500. Метод `parseFloat()` анализирует строку глубже: десятичная точка и даже буква "e", строчная или прописная, а также знак и цифры за ней будут вовлечены в обработку для получения числа.

Числа в строки можно преобразовать еще проще. Для этого достаточно к пустой строке прибавить это число, т. е. воспользоваться оператором сложения `+`. Например, вычисление выражения `"" + 3.14` даст в результате строку `"3.14"`. Понятно, что в данном случае преобразование выполняет сам интерпретатор, используя свой "интеллект".

Для определения того, является ли значение выражения числом, служит метод (или встроенная функция)

`isNaN(`*значение*`)`,

который возвращает результат логического типа. Если указанное значение не является числом, функция возвращает `true`, иначе — `false`. Однако здесь термин "число" не совпадает с понятием "значение числового типа". Функция

`isNaN()` считает числом и данные числового типа, и строку, содержащую только число. Логические значения также идентифицируются как числа. При этом значению `true` соответствует 1, а значению `false` — 0. Таким образом, если `isNaN()` возвращает `false`, то это указывает, что значение параметра имеет числовой тип, либо является числом, преобразованным в строковый тип, либо является логическим (`true` или `false`).

Примеры

```
isNaN(1234)       // результат false (т. е. это число)
isNaN("1234")     /* результат false (т. е. это число,
                     хотя и в виде строки) */
isNaN("25 рублей")// результат true (т. е. это не число)
isNaN("Саша")     // результат true (т. е. это не число)
isNaN(true)       // результат false
isNaN(false)      // результат false
```

Как видно, значение `isNaN(x)` для любого логического значения `x` возвращает `false`, что указывает на то, что `x` — число. Дело в том, что функция `isNaN` пытается привести значение `x` к числовому типу по правилам автоматического преобразования типов (см. разд. 16.2.3). Логические значения `true` и `false` преобразуются по этим правилам в числа 1 и 0 соответственно. Автоматическое приведение значений "Саша" и "25 рублей" к числовому типу заканчивается получением значения `NaN` и поэтому функция `isNaN()` возвращает для этих значений `true`.

Кроме рассмотренных, в JavaScript имеются и другие функции приведения данных к заданному типу:

- `Number(`*выражение*`)` — десятичное число или `NaN` (если не удалось преобразовать указанное *выражение* в число). Эта функция работает следующим образом:
  - если *выражение* есть число, то возвращается это же число;
  - если *выражение* есть логическая величина, то возвращается 1 или 0 в зависимости от ее значения (если `true`, то 1; если `false`, то 0);
  - если *выражение* есть строка, то функция пытается преобразовать ее в число как описанные ранее функции `parseInt()` и `parseFloat()`;
  - если *выражение* есть `udefined` (не определено), то результатом является `NaN` (не число).

- `String(`*выражение*`)` — приводит данные, получающиеся в результате вычисления выражения, которое указано в качестве параметра, к строковому типу. Преобразование происходит по следующим правилам:
  - если значение выражения имеет строковый тип, то возвращается это же значение;
  - если значение выражения имеет числовой тип, то возвращается строка, содержащая это число;
  - если значение выражения имеет логический тип, то возвращается строка `"true"` или `"false"` в зависимости от значения выражения;
  - если значение выражения не определено (`undefined`), то возвращается пустая строка `""`.
- `Boolean(`*выражение*`)` — приводит данные, получающиеся в результате вычисления выражения, которое указано в качестве параметра, к логическому типу. Преобразование происходит по следующим правилам:
  - если значение выражения имеет логический тип, то возвращается это же значение;
  - если значение выражения имеет числовой тип, то возвращается `true`, если число не равно нулю, и `false` — в противном случае;
  - если значение выражения имеет строковый тип, то возвращается `true`, если строка не пуста, и `false` — в противном случае; напомню, что строка, содержащая только пробелы, не пуста.
- `Array(`*элемент0* `,`*элемент1*`, ..., `*элементN*`)` — создает массив из элементов, указанных в качестве параметров.

В действительности перечисленные функции являются обращениями к соответствующим объектам: `Number`, `String`, `Boolean` и `Array`. Например, `String(`*выражение*`)` — обращение к так называемому статическому строковому объекту. Об этом и о других объектах будет рассказано далее.

16.3. Переменные и оператор присваивания

16.3.1. Имена переменных

Как уже упоминалось, переменная — это контейнер для хранения значений. Значения могут быть любых типов, предусмотренных в JavaScript, а присваивание происходит с помощью оператора, обозначаемого знаком равенства `=`. Например в выражении `x = 5` имя (идентификатор) переменной — `x`, а значение, которое ей присваивается (приписывается, назначается) — `5` (в данном

случае это числовое значение), следовательно, переменная `x` числовая. Если мы теперь запишем в своей программе выражение `x = "Вася"`, то переменная `x` приобретет новое значение строкового типа `"Вася"`.

Переменная имеет имя — последовательность букв, цифр, символа подчеркивания и даже символа `$`, которая не должна начинаться с цифры.

Примеры правильных имен переменных:

`myname`, `myName`, `_myname`, `my_Name` , `myName134`, `$myname`.

Примеры неправильных имен переменных:

`my name`, `310group`, `1000 $`.

Имя, соответствующее указанным правилам, может быть каким угодно. Однако желательно, чтобы оно отражало суть содержащихся в соответствующей переменной значений и/или цель ее использования. Если в имени переменной вы хотите указать несколько слов, то разделите их символом подчеркивания или напишите каждое из этих слов (начиная со второго) с прописной буквы, например, `my_first_name`, `myFirstName`. Нередко первый символ в имени переменной указывает на ее тип. Например, в имени `sMyName` первый символ обозначает, что данная переменная содержит данные строкового типа (`string`). Вы можете придумать свой стиль образования имен переменных. Важно лишь то, чтобы он был понятен хотя бы вам самому. Тем не менее, профессиональные программисты JavaScript не рекомендуют выбирать в качестве первого символа имени переменной подчеркивание или знак `$`.

JavaScript регистрозависимый язык. Это означает, что изменение регистра символов (с прописных на строчные и наоборот) в имени переменной приводит к другой переменной. Например, `myvar`, `Myvar` и `MYVAR` — различные переменные.

При выборе имен переменных (а также имен функций) недопустимы зарезервированные ключевые слова, которые используются или, планируются к применению в последующих версиях в качестве элементов синтаксиса. Так, не следует выбирать в качестве имен слова из списка:

`abstract`	`else`	`int`	`super`
`boolean`	`extends`	`interface`	`switch`
`break`	`false`	`long`	`synchronized`
`byte`	`final`	`native`	`this`
`case`	`finally`	`new`	`throw`
`catch`	`float`	`null`	`throws`
`char`	`for`	`package`	`transient`

class	function	private	true
const	goto	protected	try
continue	if	public	typeof
default	Implements	reset	var
delete	import	return	void
do	in	short	while
double	instanceof	static	with

16.3.2. Создание переменных

Здесь мы затронем чрезвычайно важную тему видимости переменных. Она актуальна не только в JavaScript, но и вообще во всех языках программирования. Интересно, что писать вполне работоспособные скрипты для сайтов удается в большинстве случаев и тем, кто данной темой совсем не владеет. Однако это "пиррова победа", обусловленная благополучным стечением обстоятельств или простотой задачи. Стоит лишь немного усложнить сценарий, потребовать взаимодействия нескольких программных блоков, как сразу же возникнут недоразумения и ошибки, обусловленные, скорее всего, недопониманием того факта, что переменная может быть, а может и не быть доступной в том или ином блоке кода или контексте выполнения сценария.

Переменную в JavaScript можно создать с помощью оператора присваивания:

```
имя_переменной = значение;
```

Например, в выражении `myname = "Вадим"` переменной с именем (идентификатором) `myname` присваивается значение строкового типа `"Вадим"`. Пока данная переменная не получит какого-нибудь значения другого типа, она останется строковой.

Кроме того, для объявления переменной предусмотрен специальный оператор `var`, например,

```
var myname;
myname = "Вадим";
```

или еще короче:

```
var myname = "Вадим";
```

В данном примере одновременно с объявлением переменной ей присваивается значение. Присваивать значения не обязательно всем объявляемым переменным. Например, в выражении

```
var x, y, z = "Привет!"
```

объявлены (созданы) переменные `x`, `y` и `z`, причем только переменной `z` присвоено конкретное значение. Переменные `x` и `y`, которым пока не присвоены

значения в программе, относятся к типу `undefined` и имеют такое же значение. На рис. 16.4 показан соответствующий пример.

Обратите внимание, что с помощью одного ключевого слова (оператора) `var` можно создать (инициализировать) сразу несколько переменных. При этом имена переменных и/или операторы присваивания разделяют запятыми.

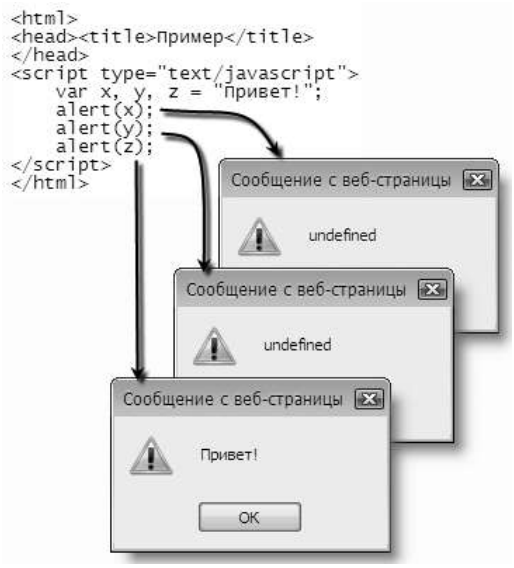

Рис. 16.4. Переменные `x` и `y` объявлены, но не определены, а переменная `z` определена

Хотя переменную в JavaScript можно создать как с указанием оператора `var`, так и без него, просто с помощью оператора присваивания, хороший стиль программирования состоит в применении `var`. Он хорош потому, что позволяет читателю программного кода достаточно четко различить, где переменная была объявлена впервые, а где она просто используется. Кроме того, оператор `var` позволяет задать так называемые локальные переменные, которые доступны только из некоторой части программного кода или, как еще говорят, в рамках некоторого контекста. И это, пожалуй, самое главное назначение данного оператора. Дело в том, что переменные могут быть глобальными и локальными. Начинающие программисты обычно не чувствуют разницу, пока не встретятся с недоразумениями на практике.

Переменные, введенные в обиход с помощью оператора присваивания (без оператора `var`) глобальные, т. е. видимы или, иначе говоря, доступны во всей программе, включая и программные блоки (например, тела функций), если

в них не приняты специальные меры, о которых будет рассказано далее. Вместе с тем, глобальные переменные можно создать и с помощью оператора `var`.

Итак, мы приступаем к подробному рассмотрению очень важной темы — области видимости и контекста переменной. Область видимости — это часть программы, в которой переменная доступна или, как еще говорят, видна. Видимая переменная доступна в программе, а невидимая — нет. Контекст — набор определенных данных, образующих среду выполнения программы. При запуске браузера (интерпретатора JavaScript) создается глобальный контекст, который помимо специфических объектов браузера и загруженного в него документа содержит объекты собственно JavaScript (например, объекты массива `Array`, даты `Date`, объект для сложных математических вычислений `Math` и др.). При активизации обработчика события (например, после щелчка кнопкой мыши на элементе Web-страницы) создается локальный контекст этого обработчика. То же самое происходит и при вызове функции: интерпретатор создает локальный контекст, существующий до тех пор, пока не завершится выполнение этой функции. Все переменные, объявленные в теле функции с помощью оператора `var`, существуют только в рамках контекста этой функции во время ее выполнения. По завершении работы функции этот контекст уничтожается. Контексты могут быть иерархически вложены друг в друга. Интерпретатор, встречая переменную в некотором контексте, анализирует, определена ли она в нем в нем. Если да, то это — локальная переменная, существующая в данном контексте. В противном случае интерпретатор ищет определение переменной в объемлющем контексте. Эти действия выполняются рекурсивно, пока не будет найден контекст, содержащий определение переменной. Не исключено, что при выходе на глобальный контекст определение переменной так и не будет найдено. Тогда переменная оказывается неопределенной.

В ранее рассмотренных примерах все переменные были глобальными. В следующем примере (рис. 16.5) переменная `x` глобальная и имеет значение `"Привет!"`, несмотря на то, что после первичного ее объявления и присваивания значения эта же переменная вновь объявляется, но без присваивания.

```
<script type="text/javascript">
   var x = "Привет!";
   var x;
   alert(x);
</script>
```

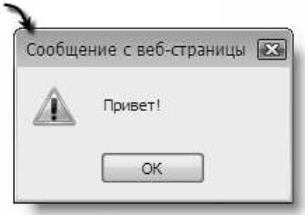

Рис. 16.5. Переменная `x` глобальная

Здесь интерпретатор, в контексте метода `alert()` не находит определения переменной `x`, а в объемлющем контексте он не останавливается на промежуточном определении `var x`, поскольку в данном пункте `x` остается `undefined`. Продолжение поиска в рас-

смотренном примере заканчивается успехом: выражение `var x = "Привет!"` полностью определяет переменную `x`. Это значение метод `alert()` выводит в диалоговом окне.

В теле функции можно использовать переменные, объявленные в программном коде более высокого уровня, т. е. в коде, объемлющем определение данной функции. Например, в основной программе, из которой данная функция вызывается. Будут ли такие переменные видны в теле функции? А если в теле функции изменить значения таких переменных, то станут ли они доступными во внешней программе?

На рис. 16.6 представлен исходный код сценария, иллюстрирующий видимость переменной в контексте функции `myfunc()` и в глобальном контексте, а также результат его выполнения в окне браузера. Переменная `x` определена в основной программе (глобальном контексте) вне тела функции и является глобальной. Она видна также и в теле функции (в контексте ее выполнения), а новое значение "До свидания!" этой переменной, присвоенное в теле функции, остается таким же и после завершения работы этой функции, т. е. в глобальном контексте.

Теперь немного изменим сценарий: переменную `x` в теле функции объявим с помощью оператора `var`. Код этого сценария в текстовом редакторе и результат его выполнения в браузере показаны на рис. 16.7. В данном случае при первом упоминании в теле функции `myfunc()` переменная `x` оказывается неопределенной (`undefined`). Действительно, при вызове функции интерпретатор, встречая в первый раз переменную `x`, ищет ее определение в локальном контексте (т. е. в теле этой функции) и находит его: `var x = "До свидания!"`.

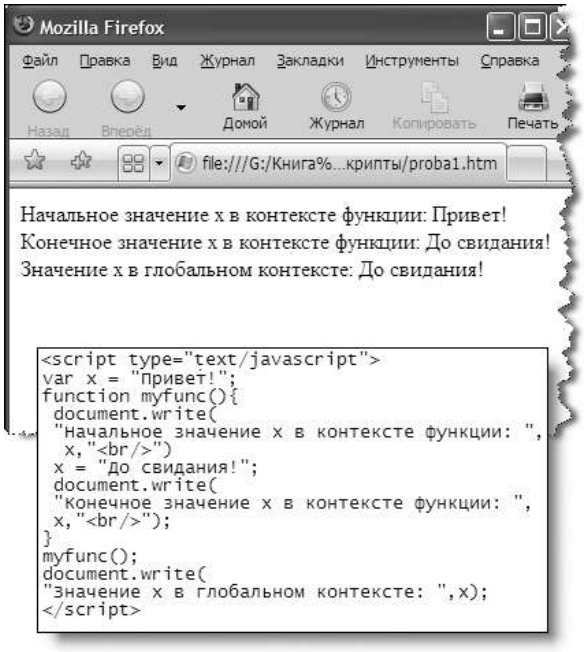

Рис. 16.6. Глобальная переменная `x`, видимая в теле функции

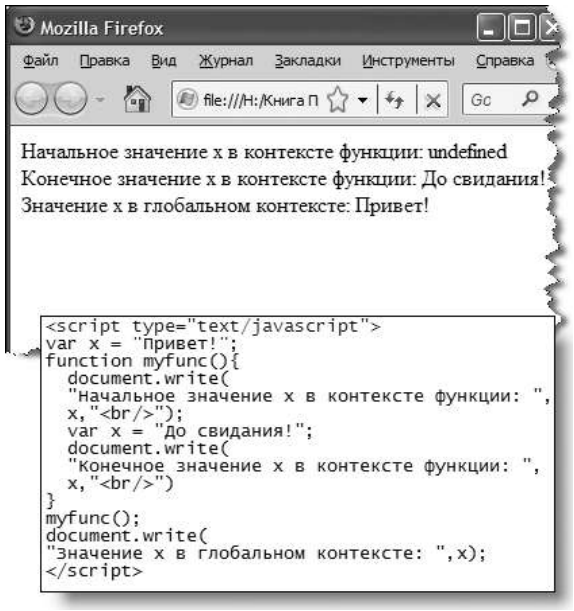

Рис. 16.7. Локальная переменная `x` в теле функции

Следовательно, это локальная переменная. Однако определение переменной `x` в теле функции следует после ее первого упоминания в пределах кода той же функции. Поэтому при выполнении выражения

```
document.write("Начальное значение x в контексте функции: ", x, "<br/>")
```

она еще не определена, и потому имеет значение `undefined`. Зато в выражении

```
document.write("Конечное значение x в контексте функции: ",x,"<br/>"),
```

следующем за определением `var x = "До свидания!"`, она имеет заданное в этом определении значение. Но значение `"До свидания!"` переменная `x` имеет только при выполнении функции `myfunc()`. Вне этого контекста (т. е. после завершения работы функции) переменная `x` видна как имеющая значение `"Привет!"`.

Таким образом, указание оператора `var` перед именем переменной в теле функции делает ее локальной, отличной от одноименной глобальной переменной. При этом говорят, что оператор `var` в программном блоке (например, в теле функции) маскирует эту переменную, т. е. делает невидимой в объемлющем контексте. На рис. 16.8 приведен еще один пример, показывающий маскирующий эффект оператора `var` в теле функции.

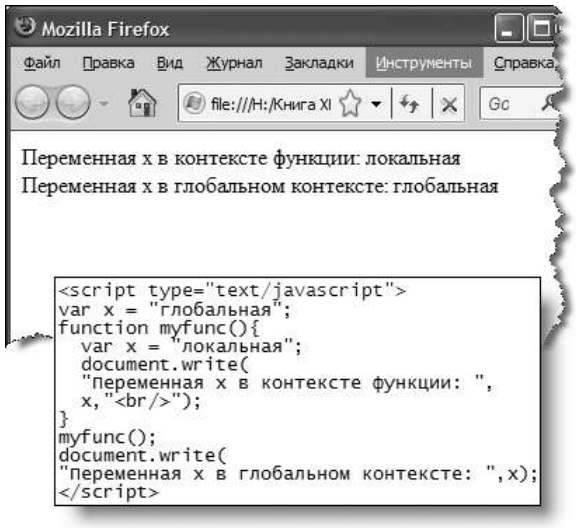

Рис. 16.8. Маскирующий эффект оператора `var`

Теперь рассмотрим очень важный для понимания областей видимости, но, возможно, не часто применяемый способ задания переменных. На рис. 16.9 показан пример, в котором переменная `x` объявлена в основной программе,

а также в функции `myfunc1()`, которая содержит вызов другой функции — `myfunc2()`. Последняя функция просто задействует, но не объявляет переменную `x`. Как переменная `x` видна в функции `myfunc2()`? Иначе говоря, какое значение, заданное в функции `myfunc1()`, или же в основной программе, будет видно в контексте выполнения функции `myfunc2()`? В данном случае объемлющим контекстом для `myfunc2()` будет не тело функции `myfunc1()`, а глобальный контекст. Оператор `var x` в функции `myfunc1()` делает переменную `x` локальной в полном смысле этого слова, т. е. не допускает ее видимости даже в функции `myfunc2()`, вызываемой из `myfunc1()`.

Рис. 16.9. Маскирующий эффект оператора `var` в случае вызова одной функции из другой

При задании определения функции в круглых скобках около ее имени указывают переменные, называемые формальными параметрами: `function(x, y) {…}`. Эти переменные-параметры в теле функции являются локальными переменными. На рис. 16.10 показан пример, в котором параметр функции `myfunc(x)` имеет то же имя `x`, что и глобальная переменная. Однако при вызове этой функции создается локальная переменная `x`, которая принимает значение одноименной глобальной переменной. Любое изменение значения переменной `x` в теле функции не повлияет на значение глобальной переменной `x`.

В JavaScript можно задавать так называемые обработчики событий — программные коды, реагирующие на события, возникающие при работе с документом, загруженным в браузер, такие как щелчок кнопкой мыши, наведение ее указателя на элемент Web-страницы, нажатие на клавишу клавиатуры

и т. п. В обработчиках событий нередко также требуется создавать переменные для сохранения данных. Программный код обработчика событий обладает собственным контекстом, как и тело функции. В листинге 16.3 приведен пример, иллюстрирующий особенности видимости переменных, определенных в обработчике событий.

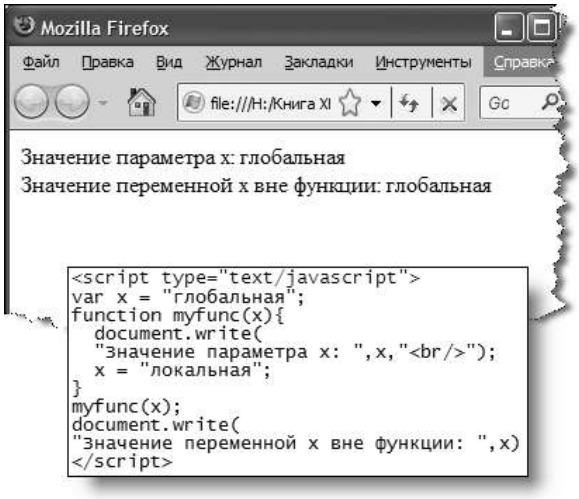

Рис. 16.10. Формальный параметр функции является локальной переменной в ее теле

Листинг 16.3. Пример определения переменных в обработчике событий

```
<html>
<head><title>Пример</title></head>
<script type="text/javascript">
var x = "глобальная";
</script>
<input type="button" value="Кнопка 1"
onmouseover = "var x = 'локальная'; alert('Переменная x '+ x)">
<input type="button" value="Кнопка 2"
onmouseover = "alert('Переменная x '+ x)"/>
</html>
```

Данный код отображает в браузере две кнопки, наведение указателя мыши на которые выводит диалоговое окно с сообщением о значении переменной `x`. Вывод этого окна обеспечивает обработчик события `onmouseover`. В основной программе (секции, заданной тегом `<script>`) определена глобальная переменная `x` со значением "глобальная". Наведем указатель мыши сначала

на первую кнопку, а затем на вторую. При наведении указателя мыши на первую кнопку обработчик этого события объявляет локальную переменную x со значением "локальная" и выводит соответствующее сообщение в диалоговом окне. При наведении указателя на вторую кнопку будет выведено сообщение "Переменная x глобальная". Таким образом, значение "локальная" существует только в контексте обработчика события для первой кнопки. В обработчике события для второй кнопки нет определения переменной x, поэтому интерпретатор ищет его в объемлющем контексте — в данном случае в глобальном контексте всего документа.

Аналогичная ситуация наблюдается и в случае нескольких обработчиков различных событий, привязанных к одному и тому же элементу документа (листинг 16.4).

Листинг 16.4. Пример привязки нескольких обработчиков к одному элементу

```
<html>
<head><title>Пример</title></head>
<script type="text/javascript">
var x = "глобальная";
</script>
<input type="button" value="Кнопка"
onmouseover="var x='локальная'; alert('Переменная x '+ x)"
 onclick="alert('Переменная x '+ x)"/>
</html>
```

Здесь кнопка — элемент документа, заданный тегом <input>, имеет два обработчика событий: onmouseover и onclick, которые запускаются при наведении указателя мыши на кнопку и при щелчке на ней соответственно. Наведение указателя мыши на кнопку выводит сообщение "Переменная x локальная", а щелчок на кнопке — сообщение "Переменная x глобальная". Таким образом, переменная x является локальной в обработчике события onmouseover и не видна в обработчике события onclick. При щелчке (событие onclick) в соответствующем обработчике остается видимой лишь глобальная переменная x со значением "глобальная".

В сценарии можно создать так называемую статическую переменную, областью видимости которой является тело некоторой функции. Такая функция может изменять значение статической переменной, которое сохраняется между вызовами функции. Однако во внешнем коде можно только прочитать, но не изменить значение статической переменной. Подробнее об этом рассказывается в *разд. 16.5.3*.

16.3.3. Операторы присваивания

Ранее мы уже неоднократно упоминали оператор присваивания значения переменной, обозначаемый знаком равенства. Этот оператор (выражение вида *имя_переменной = значение*) возвращает значение, совпадающее с тем, которое присваивается переменной (рис. 16.11).

Оператор присваивания можно использовать каскадно для записи одного и того же значения сразу в несколько переменных, как это показано на рис. 16.12.

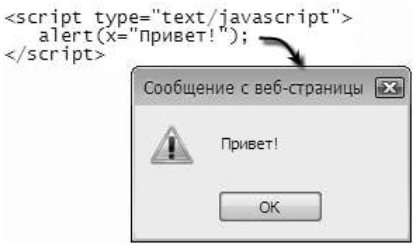

Рис. 16.11. Оператор присваивания `x="Привет!"` возвращает значение "Привет!"

Рис. 16.12. Каскадное использование оператора присваивания значения сразу нескольким переменным

Переменной можно присвоить значение другой переменной:

```
x = 2;
y =3;
x = y // значение x равно 3
```

Переменной можно присвоить значение некоторого выражения:

```
y = 2;
x = y + 3 // значение x равно 5
```

Кроме указанного и наиболее распространенного оператора присваивания в виде знака равенства в JavaScript имеются еще несколько его форм, представляющих собой комбинации оператора присваивания с арифметическими и поразрядными операторами, о которых будет рассказано позже. Это своего рода сокращения, обеспечивающие более экономную запись. Здесь мы рассмотрим лишь наиболее употребительные примеры. Так, в следующем при-

мере значение переменной x формируется как результат арифметического сложения текущего значения x со значением переменной y:

```
x = 2;
y = 3;
x += y // значение x равно 5.
```

Символ + в JavaScript обозначает не только арифметическое сложение чисел, но и склейку (конкатенацию) строк. В этом случае его также можно комбинировать с обычным символом присваивания:

```
x = "Здравствуй,";
y = "Вася";
x += y // значение x равно "Здравствуй,Вася".
```

Разумеется, в обоих случаях вместо выражения x += y можно написать равносильное по смыслу выражение

```
x = x + y.
```

Для числовых переменных помимо += возможны и такие конструкции: -=, *=, /=, %=. Эти операторы выполняют вычитание, умножение, деление и вычисление целого остатка от деления в комбинации с присвоением результата, например,

```
x = 10;
x*=2 // значение равно 20
x%=7 // значение равно 6.
```

Подробнее о дополнительных операторах присваивания см. *разд. 16.4.3*.

16.3.4. Проверка типа переменной

Иногда в сценариях требуется проверить, к какому типу принадлежит данная переменная (точнее, к какому типу относится ее текущее значение). Для этого служит оператор `typeof` *имя_переменной*, который также можно записать в виде вызова функции `typeof(`*имя_переменной*`)`. Пример на рис. 16.13 иллюстрирует использование оператора `typeof`. Данный оператор возвращает строковое значение, указывающее на один из следующих типов значений: `string` (строка), `number` (число), `boolean` (логический тип), `object` (объект), `function` (функция) и `undefined` (неопределенный).

Обратите внимание на значения, возвращаемые оператором `typeof`. Так, в случае массива возвращается `"object"`. И действительно, в JavaScript массив реализован как объект. Однако для функции оператор `typeof` возвращает `"function"`, а не `"object"`, хотя функция также реализована в виде объекта.

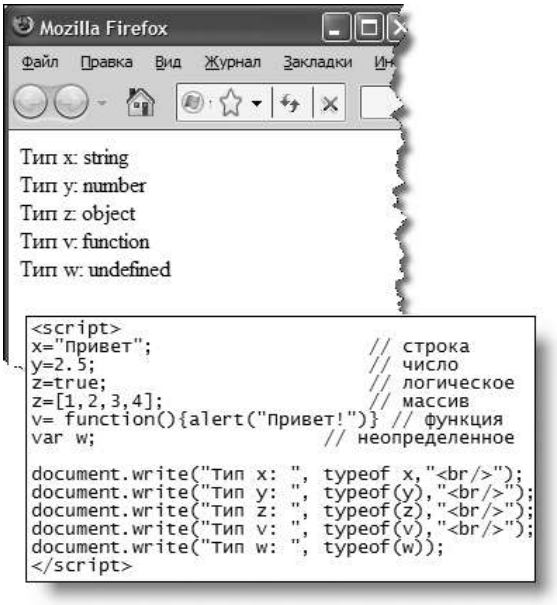

Рис. 16.13. Применение оператора `typeof` для выяснения типа переменной

16.4. Операторы

Из *операторов* составляют выражения. Оператор применяется к одному или двум данным, которые в этом случае называются *операндами*. Например, оператор сложения применяют к двум операндам, а оператор логического отрицания — к одному операнду. *Элементарное выражение*, состоящее из операндов и оператора, вычисляется интерпретатором и, следовательно, имеет некоторое значение. В этом случае говорят, что оператор возвращает значение. Например, оператор сложения, примененный к числам 2 и 3, возвращает значение 5. Типы оператора и возвращаемого им значения совпадают. Поскольку элементарное выражение с оператором и операндами возвращает значение, это выражение можно присвоить переменной.

16.4.1. Комментарии

Начнем с самых "безобидных", но важных в практическом программировании операторов комментария. Они позволяют выделить фрагменты кода, которые не выполняются (игнорируются) интерпретатором, а служат лишь для пояснений содержания программы.

В JavaScript, как и во многих других языках, допустимы два вида оператора комментария:

- `//` — одна строка символов, расположенная справа от этого оператора, считается комментарием;
- `/*...*/` — все, что заключено между `/*` и `*/`, считается комментарием; с помощью этого оператора можно выделить несколько строк в качестве комментария.

Примеры

```
x=y +10  // Это однострочный комментарий
// Это тоже однострочный комментарий
z="Привет!" /* Однострочный комментарий справа от выражения */
/* Это однострочный комментарий слева от выражения */ x=3 + 4
var xyz  /* Это двухстрочный
       комментарий */
```

Комментарии игнорируются браузером при выполнении программы, но передаются и загружаются в него вместе с основным кодом. Иногда они могут составлять даже более половины всего объема программы. Некоторые программисты избегают комментариев, чтобы уменьшить объем кода и сэкономить время передачи и загрузки. Однако даже при значительном объеме кода (несколько десятков килобайтов) польза от наличия комментариев все же больше, чем от их отсутствия, особенно в долгосрочной перспективе.

Не пренебрегайте комментариями в тексте программы. Они помогут при ее отладке и сопровождении. На этапе разработки лучше сначала превратить ненужный фрагмент программы в комментарий, чем просто удалить (а вдруг его придется восстанавливать). Даже опытные программисты, возвращаясь к работе над своей программой через месяц, с большим трудом вспоминают, что к чему. В примерах мы часто будем добавлять комментарии.

16.4.2. Арифметические операторы

Арифметические операторы, такие как сложение, умножение и т. д., в JavaScript могут применяться к данным любых типов. Что из этого получается, мы рассмотрим немного позже, а сейчас просто перечислим их (табл. 16.3).

Таблица 16.3. *Арифметические операторы*

Оператор	Название	Пример
+	Сложение	X+Y
-	Вычитание	X-Y
*	Умножение	X*Y
/	Деление	X/Y
%	Деление по модулю	X%Y
++	Увеличение на единицу	X++
--	Уменьшение на единицу	Y--

Действие арифметических операторов лучше всего проиллюстрировать на примерах с числами. Первые четыре оператора широко известны и применительно к числам не вызывают вопросов. Оператор деления по модулю возвращает целочисленный остаток от деления первого числа на второе. Например, `8%3` возвращает 2, поскольку 8 = 3×2 + 2.

Символ "–" служит не только бинарным (для двух операндов) оператором вычитания, но и указывает на то, что число отрицательное, например, –2.5.

Операторы `++` и `--` сочетают в себе действия операторов присваивания и соответственно сложения и вычитания. Выражения `X++` и `X--` почти эквивалентны выражениям `X=X+1` и `X=X-1` соответственно. Употребление слова "почти" вскоре прояснится.

Операторы `++` и `--` называют соответственно *инкрементом* и *декрементом*. Они имеют префиксную и постфиксную формы записи. Так, `x++` — постфиксная запись (`++` после операнда), а `++x` — префиксная (`++` перед операндом). В обоих случаях переменная `x` увеличит свое значение на единицу, но само выражение `x++` вернет предыдущее значение переменной `x`, а выражение `++x` — новое значение. Аналогично, выполнение выражения `x--` уменьшает текущее значение переменной `x` на единицу, но возвращает не новое, а прежнее значение переменной `x`. Выражение `x--` уменьшает `x` на единицу и возвращает новое значение. Таким образом, `++x` полностью эквивалентно `x=x+1`, а `--x` полностью эквивалентно `x=x-1`. Напомню, что выражения `x=x+1` и `x=x-1` изменяют текущее значение переменной и возвращают новое ее значение, т. е. их выполнение в точности совпадает с выполнением операторов соответственно `++x` и `--x`. Однако операторы `x++` и `x--` лишь изменяют значение переменной `x`, но возвращают не вновь полученное значение, чем, собственно, и отличаются от `x=x+1` и `x=x-1` соответственно. Сказанное иллюстрирует пример на рис. 16.14.

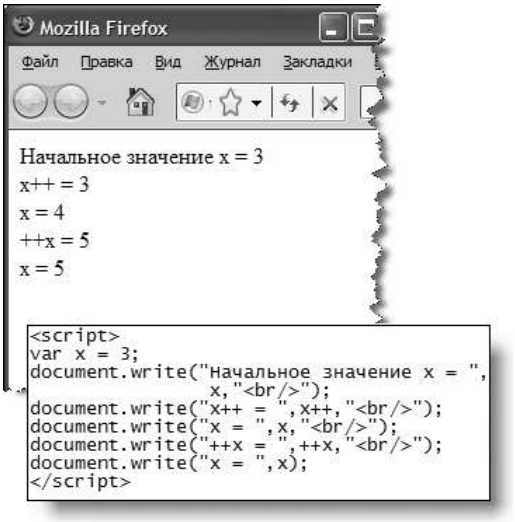

Рис. 16.14. Различия между префиксной и постфиксной записями оператора ++

Как уже отмечалось, по отношению к числам действие арифметических операторов обычно не вызывает недоразумений. Однако формально ничто не мешает нам применить эти операторы к данным других типов. В случае строковых данных оператор сложения дает в результате строку, полученную путем приписывания справа второй строки к первой. Например, выражение `"Саша"+"Маша"` возвращает в результате `"СашаМаша"`. Поэтому для строк оператор сложения называется оператором склейки или конкатенации. Остальные арифметические операторы для строк дают результат `NaN` (не число). Когда оператор сложения применяется к строке и числу, интерпретатор переводит число в соответствующую строку и выполняет склейку двух строк.

Примеры

```
x="Саша"    // значение переменной x равно "Саша"
y="Маша"    // значение переменной y равно "Маша"
z=x+" "+y   // значение переменной z равно "Саша Маша"
z=x+5       // значение переменной z равно "Саша5"
n="20"+5    // значение переменной n равно "205", а не 30 или "30"
```

В случае логических данных интерпретатор переводит логические значения операндов в числовые (`true` в 1, `false` в 0), выполняет вычисление и возвращает числовой результат. То же самое происходит, когда один операнд логический, а другой — числовой.

Примеры

```
true + true   // возвращает 2
true + false  // возвращает 1
true * true   // возвращает 1
true/false    /* возвращает Infinity (бесконечность, т. к. на
              нуль делить нельзя) */
true + 5      // возвращает 6
false + 5     // возвращает 5
true * 5      // возвращает 5
true/5        // возвращает 0.2
```

Если один операнд строкового типа, а другой — логического, то при сложении интерпретатор переводит оба операнда в строковый тип и возвращает строку символов, а для других арифметических операторов он переводит оба операнда в числовой тип.

Примеры

```
"Вася" + true   // возвращает "Васяtrue"
"5" + true      // возвращает "5true"

"Вася" * true   /* возвращает NaN (т. е. значение,
                не являющееся числом) */
"5" * true     // возвращает 5
"5" * false    // возвращает 0
"5"/true       // возвращает 5
```

16.4.3. Дополнительные операторы присваивания

В разд. 16.3.3 уже отмечалось, что кроме обычного оператора присваивания = имеются еще пять дополнительных, сочетающие присваивание и арифметические действия. Дополнительные операторы присваивания мы не будем здесь подробно рассматривать, а лишь перечислим их (табл. 16.4) и приведем примеры.

Таблица 16.4. *Дополнительные операторы присваивания*

Оператор	Пример	Эквивалентное выражение
+=	X+=Y	X=X+Y
-=	X-=Y	X=X-Y
*=	X*=Y	X=X*Y
/=	X/=Y	X=X/Y
%=	X%=Y	X=X%Y

Примеры

```
x=3;
x+=2;    // значение x равно 5 (т. е. 3+2)
x*=4     // значение x равно 20 (т. е. 5*4)
x/=2     // значение x равно 10 (т. е. 20/2)
x%=3     // значение x равно 1 (т. е. 10%3)
```

16.4.4. Операторы сравнения

В программах часто приходится сравнивать значения и проверять выполнение каких-либо условий. Например, в сценариях Web-страниц нередко проверяется, какой браузер установлен у пользователя: Microsoft Internet Explorer, Mozilla Firefox и т. д. В зависимости от результата процесс дальнейших вычислений может пойти по тому или другому пути. Проверяемые условия формируются на основе операторов сравнения, таких как "равно", "меньше", "больше" и т. п. (табл. 16.5). Результат вычисления элементарного выражения, содержащего оператор сравнения и операнды (сравниваемые данные), — логическое значение, т. е. `true` или `false`. Так, если выражение сравнения выполняется (верно, справедливо, истинно), то возвращается `true`, в противном случае — `false`.

Таблица 16.5. *Операторы сравнения*

Оператор	Название	Пример
==	Равно	X==Y
===	Равно и того же типа (строгое равенство)	X===Y
!=	Не равно	X!=Y
!==	Не равно или другого типа (строгое неравенство)	X!==Y

Таблица 16.5 *(окончание)*

Оператор	Название	Пример
>	Больше, чем	X*Y
>=	Больше или равно (не меньше)	X>=Y
<	Меньше, чем	X<Y
<=	Меньше или равно (не больше)	X<=Y

> ***ПРИМЕЧАНИЕ***
>
> Различные варианты равенства записывают с помощью двух или трех символов = без пробелов между ними. Одиночный символ = обозначает, как известно, оператор присваивания. Новички часто совершают трудновыявляемую ошибку, указывая в выражении сравнения одиночный символ равенства. Если такое выражение фигурирует в качестве условия в операторе управления (например, `if` или `while`), то может оказаться, что оно всегда будет возвращать одно и то же значение истинности. В этих операторах должно быть выражение логического типа, а если это не так, то интерпретатор автоматически приведет его к логическому типу. Например, выражение `Boolean(x=5)` всегда возвращает `true`, а `Boolean(x=null)` — `false`.

Числам присущ порядок: для любых двух различных вещественных чисел можно указать, какое из них больше другого. Операторы сравнения применительно к числам во всех языках программирования понимаются так же, как и в математике. Например, выражение `2<5` вернет `true`, а выражение `4==3.14` — `false`.

Сравнение строк несколько сложнее, оно происходит путем последовательного анализа символов слева направо с учетом их регистра, а также длин сравниваемых строк. Общие правила таковы:

- короткая строка меньше более длинной;
- символы верхнего регистра меньше символов нижнего;
- буквы, расположенные в алфавите раньше, меньше букв, расположенных позже;
- символы с меньшими ASCII- или Unicode-значениями меньше символов с бóльшими значениями этих кодов.

Мы не приводим здесь таблиц кодов ASCII и Unicode для всех символов. Отметим лишь, что некоторые из них упорядочены по возрастанию ASCII-кодов следующим образом: сначала идет пробел, затем в порядке возрастания цифры от 0 до 9, символ подчеркивания, латинские и кириллические буквы по алфавиту (сначала прописные, а затем строчные).

Примеры

```
"abcd" == "abc"     // возвращает false
"abc" == "abcd"     // возвращает false
"abcd" == "abcd"    // возвращает true
"abcd" !== "abcd"   // возвращает false
"abcd" == " abcd"   /* возвращает false
                    (1-й символ 2-го операнда — пробел) */
"" == " "           // возвращает false
"A" < "a"           // возвращает true
" " < "A"           // возвращает true
"" < " "            // возвращает true
"abcd" > " abcd"    // возвращает true
"abc" < "abcd"      // возвращает true
"235ab" < "abcdxyz" // возвращает true
"235xyz" < "abc"    // возвращает true
```

Логические значения сравниваются между собой так же, как и числа 1 и 0 (`true` соответствует 1, а `false` — 0). Например, выражение `true >false` вернет `true`, а выражение `true!==true` вернет `false`.

Операторы сравнения можно применить и к разнотипным данным. Например, допускается сравнивать числа со строками, строки с логическими значениями и логические значения с числами. В таких случаях нечисловые данные сначала автоматически приводятся к числовому типу, а затем сравниваются полученные результаты. При приведении строки к числовому типу не всегда получается число, иногда возвращается константа `NaN` (например, строка "25рублей" преобразуется в `NaN`, а не в число 25). Все операторы сравнения, кроме `!=` и `!==`, при сравнении числа с константой `NaN` возвращают `false`, а операторы `!=` и `!==` возвращают `true`. Пустая строка (`""`) или содержащая только пробелы (например, `" "`) преобразуются в число 0.

Примеры

```
57 == "57"       // возвращает true
57 ==="57"       // возвращает false
57 == "57руб."   // возвращает false
57 <= "57руб."   // возвращает false
57 >= "57руб."   // возвращает false
57 != "57руб."   // возвращает true
57 !== "57руб."  // возвращает true
57 > " "         // возвращает true
57 > ""          // возвращает true
```

```
false < "57"     // возвращает true
false < "-2.5"   // возвращает false
true >= "0.5"    // возвращает true
false < "57ab"   // возвращает false
false > "57ab"   // возвращает false
true == "true"   // возвращает false
true != "true"   // возвращает true
true <= "true"   // возвращает false
true >= "true"   // возвращает false

true == 1        // возвращает true
true == 0        // возвращает false
false == 0       // возвращает true
false == 1       // возвращает false
true === 1       // возвращает false
false === 0      // возвращает false
true === true    // возвращает true
false === false  // возвращает false
```

Между специальными типами `null` и `undefined` выполняется равенство (`null==undefined` возвращает `true`), но не выполняется строгое равенство (`null===undefined` возвращает `false`).

16.4.5. Логические операторы

Логические данные, обычно получаемые с помощью элементарных выражений, содержащих операторы сравнения, можно объединять в более сложные конструкции, используя логические (булевские) операторы И, ИЛИ, НЕ (табл. 16.6). Например, нам может потребоваться сформировать сложное логическое условие:

`"x<= 30 и y>10, или name=='Вася'".`

В этом примере есть и операторы сравнения, и логические операторы. Выражения с логическими операторами возвращают значение `true` или `false` в зависимости от значений операндов.

Таблица 16.6. *Логические операторы*

Оператор	Название	Пример
`!`	Отрицание (НЕ)	`!X`
`&&`	И	`X&&Y`
`||`	ИЛИ	`X||Y`

Оператор отрицания `!` изменяет значение единственного операнда на противоположное: если `X` имеет значение `true`, то `!X` возвращает `false`, и наоборот.

В табл. 16.7 указаны результаты операторов И, ИЛИ при различных логических значениях двух операндов. По существу данная таблица является определением логических операций И (`&&`) и ИЛИ (`||`). Кратко суть этих операций можно сформулировать так: `X&&Y` истинно (возвращает `true`) только тогда, когда оба операнда, `X` и `Y`, истинны (имеют значение `true`); `X||Y` истинно только тогда, когда истинен хотя бы один из операндов, `X` или `Y`.

Таблица 16.7. *Значения, возвращаемые операторами И, ИЛИ*

X	**Y**	**X&&Y**	**X||Y**
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

Операторы `&&` и `||` еще называют *логическим умножением* и *логическим сложением* соответственно (в математике логические операции И и ИЛИ известны под названиями *конъюнкции* и *дизъюнкции*).

Если вспомнить, что значению `true` можно сопоставить единицу, а `false` — нуль, то нетрудно понять, как вычисляются элементарные выражения с логическими операторами. Нужно только учесть, что в алгебре логики 1 + 1 = 1 (а не 2). Аналогично, оператору отрицания соответствует вычитание из единицы числового эквивалента логического значения операнда.

Примеры

```
x=true                  // значение переменной x равно true
x=false || 2*2==4       // значение переменной x равно true
x=5<2 || "abcd" <="xy" // значение переменной x равно true
y=!x                    // значение переменной y равно false
```

```
z=x && y              // значение переменной z равно false
z=x || y              // значение переменной z равно true
```

Сложные логические выражения, состоящие из нескольких более простых, соединенных операторами И и ИЛИ, выполняются по так называемому *принципу короткой обработки*. Дело в том, что значение всего выражения зачастую можно определить, вычислив лишь одно или несколько его составляющих, не затрагивая остальные. Например, выражение `x&&y` вычисляется слева направо; если значение `x` оказалось равным `false`, то значение `y` не вычисляется, поскольку и так известно, что все выражение равно `false` (см. табл. 16.7). Аналогично, если в выражении `x||y` значение `x` равно `true`, то `y` не вычисляется, поскольку уже ясно, что все выражение равно `true`. Данное обстоятельство требует особого внимания при тестировании сложных логических выражений. Так, если какая-нибудь логическая конструкция содержит ошибку, то она может остаться невыявленной, поскольку эта часть выражения просто не выполнялась при тестировании.

16.4.6. Операторы условия

Вычислительный процесс можно направить по тому или другому пути в зависимости от того, выполняется ли некоторое условие, или нет. Для управления вычислениями служат операторы условия или, как еще говорят, операторы условного перехода, важнейшие из них — `if` (если), `?` и `switch` (переключатель).

Оператор *if*

Оператор `if` (если), присутствующий во всех языках программирования, позволяет реализовать структуру условного выражения:

если ..., то ..., иначе ...

Синтаксис оператора `if` следующий:

```
if (условие)
 {код, который выполняется, если условие истинно}
else
 {код, который выполняется, если условие ложно}
```

В фигурных скобках располагают блок кода: одно или несколько выражений. Если в блоке не более одного выражения, то фигурные скобки можно опустить. Часть этой конструкции, определяемая ключевым словом `else` (иначе), необязательна. В этом случае остается только часть, определенная ключевым словом `if`:

```
if (условие)
```

```
 {код, который работает, если условие истинно}
```

Конструкция оператора условного перехода допускает вложение других операторов условного перехода внутри блока кода, расположенного в фигурных скобках. Условие обычно представляет собой логическое выражение, значение которого есть `true` или `false`. Чаще всего это элементарные выражения с операторами сравнения.

Пример

```
/* Вывод диалогового окна с тем или иным сообщением в зависимости от значения переменной age (возраст) */
var age=15;
if (age<18)
 {alert("Вы слишком молоды для просмотра этого сайта")}
else
 {alert("Подтвердите свое решение заглянуть на этот сайт")}
/* Вывод диалогового окна с сообщением, если только значение переменной age меньше 18 */
if (age<18)
 {alert("Вы слишком молоды для просмотра этого сайта")}
```

Делать ли отступы при написании операторов, где располагать фигурные скобки — дело вкуса. Следует руководствоваться наглядностью и ясностью структуры, при которой легко проверить правильность расстановки скобок. Кроме описанного ранее способа, часто встречается еще и такой:

```
if (условие) {
 код, который работает, если условие выполнено
} else {
  код, который работает, если условие не выполнено
}
```

Возможно, этот способ записи лучше отражает структуру оператора условного перехода.

Более сложная структура оператора условного перехода получается при вложении других операторов `if` (листинг 16.5).

Листинг 16.5. Пример вложенных операторов `if`

```
if (условие1) {
 код, который работает, если условие1 выполнено
} else {
         if (условие2){
```

```
        код, который работает, если условие2 выполнено
      }else{
        код, который работает, если условие2 не выполнено
      }
}
```

Ранее уже отмечалось, что условие в операторе `if` чаще всего является логическим выражением. Однако это может быть также строковое или числовое выражение, массив и даже объект. В общем случае интерпретатор приводит выражение условия к логическому типу автоматически (см. разд. 16.2.3). Для строкового выражения условие считается истинным, если его значением будет непустая строка. Напомним, что пустая строка `""` не содержит ни одного символа, в том числе и пробела (строка, содержащая хотя бы один пробел, не пуста). Числовое выражение истинно, если его значение — число, отличное от нуля. Для массива это условие выполняется, если длина массива больше нуля. Хотя новичка подобная многозначность условий может обескуражить, если не просто ввергнуть в отчаяние, во многих случаях она оказывается очень удобной. Виртуозы легко и свободно используют эту возможность. Нужно просто привыкнуть к ней. Ортодоксам могу посоветовать записывать в качестве условия выражения явно логического типа, либо принудительно приводить их к логическому типу (см. разд. 16.2.4).

На рис. 16.15 приведено несколько примеров использования значений различных типов в качестве условий оператора `if`.

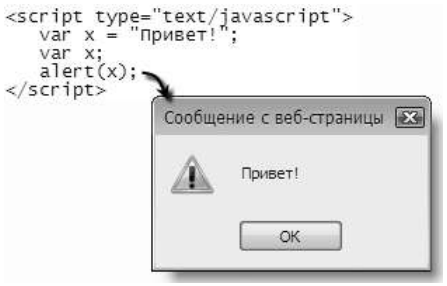

Рис. 16.15. Различные типы данных в качестве условия оператора `if`

Оператор условия *?:*

Этот оператор — сокращенная форма записи рассмотренного ранее `if...else...`. Обычно он применяется вместе с оператором присваивания одного из двух возможных значений, в зависимости от истинности или ложности условного выражения. Если это выражение само по себе возвращает значения нелогического типа, то интерпретатор автоматически попытается выполнить приведение к логическому типу. Синтаксис оператора условия следующий:

условие `?` *выражение1* `:` *выражение2*

Оператор условия возвращает значение выражения *выражение1*, если *условие* истинно, в противном случае — значение выражения *выражение2*.

Оператор условия можно указать в выражении присваивания:

переменная=условие ? *выражение1* : *выражение2*

Выражение, содержащее условие, автоматически приводится к логическому типу.

Примеры

```
var x=2;
alert(x==2 ? "x равно 2": "x не равно 2") // вывод сообщения "x равно 2"

var x=2;           // x имеет значение 2
y = x==2 ? "x равно 2": "x не равно 2";   /* переменная y имеет значение
                                             "x равно 2" */
alert(y)

var x="";            // x имеет значение "" (пустая строка)
y=x==2 ? "x равно 2": "x не равно 2"   // вывод сообщения "x не равно 2";
alert(y)             // вывод сообщения "x не равно 2"
```

Оператор *switch*

Для организации управления вычислениями с несколькими условиями вполне достаточно рассмотренного оператора `if`. Однако в случае однородных повторяющихся условий более удобным и наглядным оказывается оператор `switch` (переключатель). Он особенно целесообразен, если требуется проверить несколько условий, которые не являются взаимоисключающими.

Листинг 16.6 иллюстрирует синтаксис оператора `switch`.

Листинг 16.6. Синтаксис оператора `switch`

```
switch (выражение) {
 case вариант1:
  код
  [break]
 case вариант2:
  код
  [break]
 ...
 [default:
  код]
}
```

Здесь квадратные скобки лишь указывают на то, что заключенные в них выражения необязательные.

Параметр *выражение* оператора `switch` может принимать строковые, числовые и логические значения. Разумеется, в случае логического выражения возможны только два варианта. Ключевые слова (операторы) `break` и `default` необязательные (на это указывают прямоугольные скобки). Если оператор `break` указан, то остальные условия не проверяются. Если указан оператор `default`, то следующий за ним код выполняется тогда, когда значение выражения *выражение* не соответствует ни одному из вариантов. Если все варианты возможных значений предусмотрены в операторе `switch`, то оператор `default` можно опустить.

Оператор `switch` работает следующим образом. Сначала вычисляется *выражение*, указанное в круглых скобках сразу за ключевым словом `switch`. Полученное значение сравнивается с тем, которое указано в первом варианте. Если они не совпадают, то код этого варианта не выполняется и происходит переход к следующему варианту. Если же значения совпали, то выполняется код, соответствующий этому варианту. При этом если не указан оператор `break`, то выполняются коды и остальных вариантов, пока не встретится оператор `break`. Аналогичное правило действует и для остальных вариантов.

Пример 1

```
var x=2
switch (x) {
 case 1:
    alert(1)
 case 2:
    alert(2)
 case 3:
    alert(3)
}
```

В первом примере сработают 2-й и 3-й варианты. Если мы хотим, чтобы сработал только один какой-нибудь вариант (только тот, который соответствует значению выражения), то нужно добавить оператор `break`.

Пример 2

```
var x=2;
switch (x) {
 case 1:
    alert(1);
    break
 case 2:
    alert(2);
```

```
    break
 case 3:
    alert(3);
    break
}
```

Во втором примере сработает только 2-й вариант.

16.4.7. Операторы цикла

Оператор цикла обеспечивает многократное выполнение блока программного кода до тех пор, пока не выполнится некоторое условие. В JavaScript предусмотрены три оператора цикла: `for`, `while` и `do-while`. Вообще говоря, при создании программ вполне можно обойтись одним из них, `for` или `while`. Однако возникают ситуации, в которых один из операторов более удобен или естественен, чем другой.

Оператор *for*

Оператор `for` (для) также называют *оператором со счетчиком циклов*, хотя в нем совсем не обязательно использовать счетчик. Синтаксис этого оператора следующий:

```
for ([начальное_выражение]; [условие]; [выражение_обновления])
{
 код
}
```

Здесь квадратные скобки лишь указывают на то, что заключенные в них параметры необязательные. Как и в случае оператора условного перехода, возможна и такая запись:

```
for ([начальное_выражение]; [условие]; [выражение_обновления]) {
  код
}
```

Все, что находится в круглых скобках справа от ключевого слова `for`, называется *заголовком* оператора цикла, а содержимое фигурных скобок — его *телом*.

В заголовке оператора цикла начальное выражение выполняется только один раз в начале выполнения оператора. Второй параметр представляет собой условие продолжения цикла. Он аналогичен условию в операторе `if`. Третий параметр содержит выражение, которое выполняется после выполнения всех выражений кода, заключенного в фигурные скобки.

Оператор цикла работает следующим образом. Сначала выполняется *начальное_выражение*. Затем проверяется *условие*. Если оно истинно, то оператор цикла прекращает работу (при этом *код* не выполняется). В противном случае выполняется *код*, расположенный в теле оператора `for`, т. е. между фигурными скобками. После этого выполняется *выражение_обновления* (третий параметр оператора `for`). Так заканчивается первый цикл или, как еще говорят, первая *итерация* цикла. Далее, снова проверяется *условие*, и все действия повторяются.

Обычно начальным выражением служит оператор присваивания значения переменной, например, `i=0` или `var i=0`. Имя переменной и присваиваемое значение могут быть любыми. Эту переменную называют *счетчиком* цикла. В этом случае условие, как правило, представляет собой элементарное выражение сравнения переменной счетчика цикла с некоторым числом, например, `i<=nMax`. Выражение обновления в таком случае просто изменяет значение счетчика, например, `i=i+1` или более компактно `i++`.

В следующем примере оператор цикла просто изменяет значение своего счетчика, выполняя 15 итераций:

```
for (i=1; i <= 15; i++) {}
```

Немного модифицируем этот код, чтобы вычислить сумму всех целых положительных чисел от 1 до 15:

```
var s=1;
for (i=1; i <= 15; i++) {
  s=s + i
}
```

Заметим, что счетчик цикла может быть не только возрастающим, но и убывающим.

Рассмотрим еще пример. Допустим, требуется вычислить `x` в степени `y`, где `x`, `y` — целые положительные числа. Алгоритм решения этой задачи прост: нужно вычислить выражение `x*x*x*...*x`, в котором сомножитель `x` встречается `y` раз. Очевидно, что если `y` не превышает 10, то можно воспользоваться оператором умножения: выражение будет не очень длинным. А как быть, если `y` равно нескольким десяткам или сотням? Ясно, что в общем случае следует применить оператор цикла (листинг 16.7).

Листинг 16.7. Пример оператора цикла

```
/* Вычисляем x в степени y */
var z=x;   // z хранит результат
for (i=2; i <= y; i++) {
  z=z*x
```

```
}
```

В этом примере результат сохраняется в переменной z. Ее начальное значение равно x. Если y равно 1, то оператор цикла не будет выполняться, поскольку условие 2<=1 не справедливо (обратите внимание на начальное значение счетчика цикла) и, следовательно, мы получим верный результат: x в степени 1 равно x. Если y равно 2, то оператор цикла выполнит одну итерацию, вычислив выражение z=z*x при z, равном x (т. е. z=x*x). При y, равном 3, оператор цикла сделает две итерации. На второй, последней, итерации выражение z=z*x вычисляется при текущем значении z, равном x*x, и, следовательно, становится равным x*x*x (т. е. x в степени 3).

Рассмотрим довольно традиционный при изучении операторов цикла пример вычисления факториала числа. Факториал числа n в математике обозначают как n!. Для n, равного 0 и 1, n! равен 1. В остальных случаях n! равен 2*3*4*...*n. Поскольку возможны два варианта исходных данных, нам потребуется оператор условного перехода. Листинг 16.8 содержит код, решающий эту задачу. Результат сохраняется в переменной z.

Листинг 16.8. Вычисление факториала

```
/* Вычисляем n! */
var z=1;   // z хранит результат n!
if (n >1) {
 for (i=2; i <= n; i++) {
   z=z*i
 }
}
```

Для принудительного (т. е. не по условию) выхода из цикла используется оператор break (прерывание). Если вычислительный процесс встречает этот оператор в теле оператора цикла, то он сразу же завершается без выполнения последующих выражений кода в теле и даже выражения обновления. Обычно оператор break применяется при проверке некоторого дополнительного условия, выполнение которого требует завершения итераций, несмотря на то, что условие в заголовке цикла еще не выполнено. Типовая структура оператора цикла с break приведена в листинге 16.9.

Листинг 16.9. Оператор цикла с `break`

```
for ([начальное_выражение]; [условие1]; [выражение_обновления])
{
  код
```

```
  if (условие2){
   код
   break
  }
  код
}
```

Для управления вычислениями в операторе цикла допустим также оператор `continue` (продолжение), который, как и `break`, применяется в теле оператора цикла вместе с оператором условного перехода. Однако в отличие от `break`, оператор `continue` прекращает выполнение последующего кода, выполняет выражение обновления и возвращает вычислительный процесс в начало оператора цикла, где проверяется условие, указанное в заголовке. Листинг 16.10 иллюстрирует типовую структуру оператора цикла с `continue`.

Листинг 16.10. Оператор цикла с `continue`

```
for ([начальное_выражение]; [условие1]; [выражение_обновления] )
{
  код
  if (условие2){
    код
    continue
  }
  код
}
```

Оператор *while*

Оператор цикла `while` (до тех пор, пока) имеет структуру, более простую, чем `for`, и работает несколько иначе. Синтаксис этого оператора следующий:

```
while (условие)
{
  код
}
```

При выполнении этого оператора сначала проверяется условие, указанное в заголовке, т. е. в круглых скобках справа от ключевого слова `while`. Если оно истинно, то выполняется код в теле оператора цикла, заключенный в фигурные скобки. В противном случае код не выполняется. При выполнении кода (завершении первой итерации) вычислительный процесс возвращается к заголовку, где снова проверяется условие, и т. д.

Если сравнивать операторы `while` и `for`, то особенность первого заключается в том, что выражение обновления записывается в теле оператора, а не в заго-

ловке. Часто забывают указать это выражение, и в результате цикл не завершается (программа "зависает").

Приведем два примера, которые мы уже рассматривали при описании оператора цикла `for`.

Примеры с оператором `while`

```
/* Вычисляем x в степени y */
var z=x;   // z хранит результат
var i=2;
while (i=2) {
  z=z*x;
  i++
}

/* Вычисляем n! */
var z=1   // z хранит результат n!
if (n >1) {
  i=2;
  while ( i <= n ) {
   z=z*I;
   i++
  }
}
```

Во всех приведенных примерах есть счетчик цикла, который инициируется заранее, до оператора цикла и обновляется в теле цикла.

Для управления вычислительным процессом в операторе `while`, так же как и в операторе `for`, допустимы операторы прерывания `break` и продолжения `continue`.

> ***Внимание***
>
> Если в `while` вы задействуете счетчики циклов, то будьте осторожны, применяя `break` и `continue`.

Оператор *do-while*

Оператор `do-while` (делай до тех пор, пока) представляет собой конструкцию из двух операторов, работающих совместно. Синтаксис этой конструкции следующий:

```
do {
  код
}
while (условие)
```

В отличие от `while` в операторе `do-while` код выполняется хотя бы один раз, независимо от условия, которое проверяется после выполнения кода. Если оно истинно, то снова выполняется код в теле оператора `do`. В противном случае работа оператора `do-while` завершается. Напомню, что в операторе `while` условие проверяется в первую очередь, до выполнения кода в теле цикла. Если при первом обращении к оператору `while` условие ложно, то код не будет выполнен никогда.

Рассмотрим те же две задачи, которые мы уже решали ранее с помощью операторов цикла `for` и `while`.

Примеры с оператором `do-while`

```
/* Вычисляем x в степени y */
var z=x;   // z хранит результат
var i=2;
do{
  z=z*x;
  i++
}
while (i <= y)

/* Вычисляем n! */
var z=1;   // z хранит результат n!
if (n >1) {
  i=2;
  do {
   z=z*I;
   i++
  }
  while ( i <= n )
}
```

16.4.8. Об условиях в операторах условия и цикла

В условных операторах (см. разд. 16.4.6) и циклах (см. разд. 16.4.7) используются выражения условия, которые должны иметь логический тип, т. е. возвращать либо `true`, либо `false`. Если же эти выражения возвращают значения других типов (например, числа, строки, массивы и т. д.), то при наличии их в операторах условия или цикла интерпретатор JavaScript сделает попытку автоматически привести их значения к логическому типу. Так, число 0, пустую строку "" и массив, не содержащий элементов, он преобразует в логическое значение `false`.

Пример

```
var x=0; // число 0
var y=""; // пустая строка
var z=["Саша", "Маша", "Петя"]; // массив

if (x) alert("Переменная x >0")
else alert("Переменная x =0");

if (y) alert("Переменная y что-то содержит")
else alert("Переменная y пуста");

if (z) alert("Массив z не пуст")
else alert("Массив z пуст");
```

16.4.9. Побитовые операторы

Побитовые (поразрядные) операторы применяются к целочисленным значениям и возвращают результат такого же типа. При их выполнении операнды предварительно приводятся к двоичной форме, в которой число представляет собой 32-разрядную последовательность из нулей и единиц, называемых *двоичными разрядами* или *битами*. Далее производится некоторое действие над битами, в результате которого получается новая последовательность битов. В конце концов, эта последовательность битов преобразуется к обычному целому числу — результату побитового оператора. В табл. 16.8 приведен список побитовых операторов.

Таблица 16.8. *Побитовые операторы*

Оператор	Название	Левый операнд	Правый операнд
&	Побитовое И	Целое число	Целое число
|	Побитовое ИЛИ	То же	То же
^	Побитовое исключающее ИЛИ	То же	То же
~	Побитовое НЕ	—	То же
<<	Смещение влево	Целое число	Число битов, на которое производится смещение
>>	Смещение вправо	То же	То же
>>>	Заполнение нулями при смещении вправо	То же	То же

Операторы `&`, `|`, `^` и `~` напоминают логические операторы, но их область действия — биты, а не логические значения. Оператор `~` изменяет значение бита на противоположное: 0 на 1, а 1 — на 0. Действие операторов `&`, `|`, `^` ясно из табл. 16.9.

Таблица 16.9. *Работа операторов* `&`, `|`, `^`

X	Y	X&Y	X|Y	X^Y
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Например, `2&3` равно 2, а `2|3` равно 3. Действительно, в двоичном представлении 2 есть 10, а 3 — 11. Применение побитового оператора `&` даст двоичное число 10 (десятичное число 2), а оператора `|` — двоичное число 11 (десятичное 3).

У операторов смещения один операнд и один параметр, указывающий, на сколько битов выполняется смещение. Например, `3<<2` равно 12, потому что смещение влево на два бита двоичного числа 11 (десятичное 3) дает 1100, что в десятичной форме есть 12. Результат вычисления выражения `6>>2` равен 1. Действительно, число 6 в двоичной форме — это 110; смещение его вправо на два бита дает 1 как в двоичной, так и в десятичной форме.

16.4.10. Другие операторы

Кроме перечисленных, имеется и ряд других операторов. Так, `with`, `for…in`, `in`, `delete` и `instanceof`, предназначенные для работы с объектами, рассмотрены в *разд. 16.10.3*, а операторы для обработки ошибок (исключительных ситуаций) — в *разд. 16.11*.

16.4.11. Приоритет операторов

В одном и том же выражении могут присутствовать несколько операторов, которые выполняются в соответствии с их приоритетами. Операторы с одинаковыми приоритетами выполняются слева направо. Для изменения данного порядка служат круглые скобки. Круглые скобки, позволяющие изменить порядок выполнения операций, можно тоже считать операторами, обладающими наивысшим приоритетом. Скобки могут образовывать вложенные структуры. Выражения, находящиеся во внутренних круглых скобках, вы-

полняются раньше тех, которые заключены во внешние круглые скобки. Интерпретатор JavaScript начинает анализ выражения именно с выяснения структуры вложенности пар круглых скобок.

На втором месте по приоритету находится вычисление индексов и определение элементов массивов. Индексы массивов заключают в квадратные скобки.

Третье место по приоритету занимает вызов функции.

На последнем месте находится запятая, как разделитель параметров или элементов массива, перечисленных в квадратных скобках.

Распределение операторов по приоритетам приведено в табл. 16.10. Некоторые из указанных в ней операторов мы уже рассмотрели, о других будет рассказано в следующих разделах.

Таблица 16.10. *Распределение операторов по приоритетам*

Приоритет	Оператор	Комментарий
1	`( )`	От внутренних к внешним
	`[ ]`	Значение индекса массива
	`function()`	Вызов функции
2	`!`	Логическое НЕ
	`~`	Побитовое НЕ
	`-`	Отрицание
	`++`	Инкремент (приращение)
	`--`	Декремент
	`new`	–
	`typeof`	–
	`void`	–
	`delete`	Удаление объектного элемента
3	`*`	Умножение
	`/`	Деление
	`%`	Деление по модулю (остаток от деления)
4	`+`	Сложение (конкатенация)
	`-`	Вычитание
5	`<<`	Побитовые сдвиги

	>>	–
	>>>	–
6	<	Меньше
	<=	Не больше (меньше или равно)
	>	Больше
	>=	Не меньше (больше или равно)
7	==	Равенство
	===	Строгое равенство (с учетом типов)
	!=	Неравенство
	!==	Строгое неравенство (с учетом типов)
8	&	Побитовое И
9	^	Побитовое исключающее ИЛИ
10	|	Побитовое ИЛИ (дизъюнкция)
11	&&	Логическое И (конъюнкция)
12	||	Логическое ИЛИ
13	?	Условное выражение (оператор условия)
14	=	Операторы присваивания
	+=	–
	-=	–
	*=	–
	/=	–
	%=	–
	<<=	–
	>=	–
	>>=	–
	&=	–
	^=	–
	|=	–
15	,	Запятая

16.5. Функции

Функция представляет собой подпрограмму, которую можно вызывать для выполнения неоднократно, обратившись к ней по имени. Взаимодействие функции с внешней программой, из которой она была вызвана, происходит путем передачи функции параметров (аргументов) и приема от нее результата вычислений. Впрочем, функция в JavaScript может и не требовать параметров, а также ничего не возвращать.

Программный код функции заключают в фигурные скобки, а перед ним пишут ключевое слово `function`, за которым следуют круглые скобки, обрамляющие список параметров. Иначе говоря, определение функции выглядит так:

`function` *имя_функции*`(`*параметры*`) {`*код*`}`.

Формат вызова функции в программе следующий:

имя_функции`(`*параметры*`)`.

Параметры функции могут присутствовать или отсутствовать. Если параметры требуются, то их указывают в круглых скобках через запятую. Если параметры не предусмотрены, то в круглых скобках ничего не записывают.

16.5.1. Встроенные функции

В JavaScript имеются встроенные функции, которые в действительности являются методами встроенного объекта `global`. Поскольку при вызове этих методов не нужно указывать объект `global` (т. е. пишут `метод()`, а не `global.`*`метод`*`()`), то их называют функциями. Некоторые из них уже рассматривались ранее, например, `parseInt()`, `parseFloat()`, `isNaN()` (см. разд. 16.2.4). Далее в этом разделе рассмотрим другие встроенные функции.

- `eval(`*строка*`)` — вычисляет выражение в указанной строке. Выражение должно быть написано на языке JavaScript (не содержит тегов HTML).

Примеры

```
var y=3;                    // значение y равно 3
var x="if(y<10) {y=y+2}";   // значение x равно строке символов
eval(x);                    // значение y равно 5
```

Другой, более интересный, пример применения функции `eval()` — создание Web-страницы, содержащей текстовые области для ввода и отображения результатов выполнения выражений JavaScript. По существу это простой редактор скриптов, позволяющий их выполнять. Текст соответствующего

(X)HTML-кода со сценарием, содержащим функцию `eval()`, приведен в листинге 16.11.

Листинг 16.11. Редактор JavaScript

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head><title>Редактор JavaScript</title></head>
  <body>
  Введите скрипт:<br/>
  <textarea id="mycode" rows=10 cols=40></textarea>
  <br/>Результат:<br/>
  <textarea id="myrezult" rows=3 cols=40></textarea>
  <br/>
  <button onclick=
     "document.getElementById('myrezult').value =
      eval(document.getElementById('mycode').value)">
    Выполнить
  </button>
  <button onclick=
     "document.getElementById('mycode').value='';
      document.getElementById('myrezult').value=''">
    Очистить
  </ button >
  </body>
</html>
```

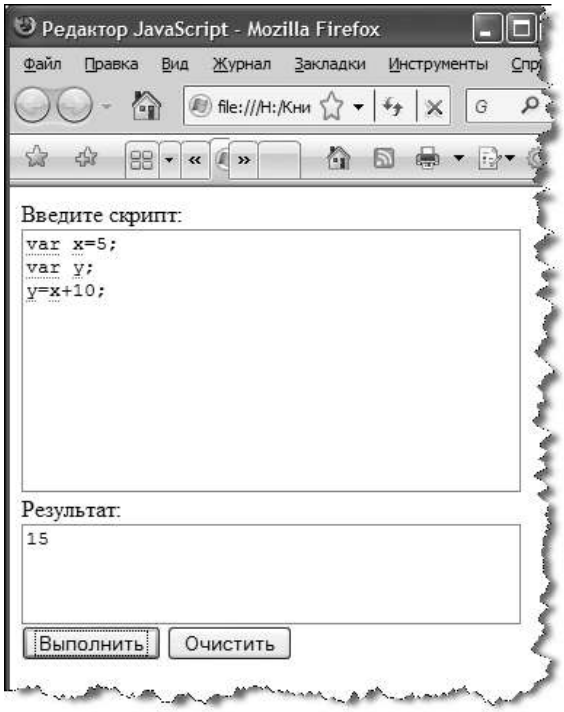

Рис. 16.16. Редактор кодов JavaScript, созданный с помощью функции `eval()` (листинг 16.11)

Здесь сценарии записаны в виде символьных строк в качестве значений атрибутов `onclik`, определяющих событие "Щелчок кнопкой мыши" на кнопках, которые заданы тегами `<button>`. На рис. 16.16 показан вид приведенного XHTML-документа в браузере.

- `escape(`*строка*`)` — представляет строку в формате Unicode, заменяя пробелы, знаки пунктуации, специальные символы и другие, выходящие за рамки базовой ASCII-кодировки, кодом вида `%`*XX*, где *XX* — шестнадцатеричное число; символы, ASCII-код которых превышает 255, представляются в виде `%u`*XXXX*; такую строку еще называют escape-последовательностью. Данный метод не кодирует символы `@`, `*`, `/` и `+`. Возвращает кодированную строку. Для кодирования компонентов URL вместо `escape()` предпочтительней использовать более современный метод (встроенную функцию) `encodeURIComponent(строка)`.

Пример

```
escape("How do you do")  /* значение равно "How%20do%20you%20do" */
escape("Привет")      /* значение равно
                            %u041F%u0440%u0438%u0432%u0435%u0442 */
```

- `unescape(`*строка*`)` — осуществляет преобразование, обратное преобразованию, выполняемому функцией `escape()`. Возвращает декодированную строку. Вместо `unescape()` предпочтительней использовать более современный метод `decodeURIComponent(`*строка*`)`.
- `encodeURI(`*строка*`)` — выполняет так называемое URI-кодирование строки, указанной в качестве параметра, возвращая кодированную строку; обычно используется для кодирования URL целиком, который в исходном виде содержит недопустимые символы (например, кириллические буквы, пробелы и др.). Данный метод не кодирует символы `@`, `/`, `?`, `#`, `+`, `'`, `!`, `$`, `*`, `(`, `)`, `=`, `;`. Примеры были приведены в *разд. 6.5.3*.
- `encodeURIComponent(строка)` — выполняет так называемое URI-кодирование строки, указанной в качестве параметра, возвращая кодированную строку. Данный метод не кодирует символы `'`, `!`, `*`, `(`, `)`. Применяется для кодирования отдельных компонентов URL (см. *разд. 6.5.3*).
- `decodeURI(`*строка*`)` — осуществляет преобразование, обратное преобразованию, выполняемому функцией `encodeURI()`. Возвращает декодированную строку.
- `decodeURIComponent` — осуществляет преобразование, обратное преобразованию, выполняемому функцией `encodeURIComponent()`. Возвращает декодированную строку.
- `typeof(`*объект*`)` — возвращает тип указанного объекта в виде символьной строки; например, "boolean", "function" и т. п. В разд. 16.3.4 рассматривался оператор `typeof`, который возвращает то же, что и встроенная функция `typeof()`.

Пример

```
var x = "Привет!", y = 5, z= function(){alert("Как дела?")};
document.write(typeof(x),"<br/>");
document.write(typeof(y),"<br/>");
document.write(typeof(z),"<br/>");
document.write(typeof x,"<br/>");
document.write(typeof y,"<br/>");
document.write(typeof z,"<br/>");
```

- `isFinite(`*значение*`)` — логическое значение, указывающее, является ли числовой параметр конечным, т. е. находящимся в допустимом диапазоне (см. табл. 16.1); если параметр есть `NaN`, то функция возвращает `false`.

Пример

```
var x = 25/(5-5);
if (isFinity(x) alert("x бесконечно велико");
```

16.5.2. Пользовательские функции

Пользовательские функции вы можете создать сами, по своему усмотрению, для решения собственных задач. Функция задается своим определением (описанием), которое начинается ключевым словом `function`. Описание функции имеет следующий синтаксис:

```
function имя_функции(параметры) {
  код
}
```

Имя функции выбирают так же, как и имя переменной. За именем функции обязательно следует пара круглых скобок. Программный код (тело) функции заключается в фигурные скобки. Если функция принимает параметры, то список их имен (идентификаторов) указывают в круглых скобках около имени функции. Имена параметров выбирают согласно тем же требованиям, что и имена обычных переменных. Если параметров несколько, то в списке их разделяют запятыми. Если параметры для данной функции не предусмотрены, то в круглых скобках около имени функции ничего не пишут.

Когда создается определение функции, список ее параметров (если он необходим) содержит просто формальные идентификаторы (имена) этих параметров, понимаемые как переменные. В определении функции в списке параметров, заключенном в круглые скобки сразу же за именем функции после ключевого слова `function`, недопустимы конкретные значения и выражения. В этом смысле определение функции задает код, оперирующий формальными параметрами, которые конкретизируются лишь при вызове функции из внешней программы.

Если требуется, чтобы функция возвращала некоторое значение, то в ее теле размещают оператор возврата `return` с указанием справа от него того, что следует возвратить. В качестве возвращаемой величины может выступать любое выражение: простое значение, имя переменной или вычисляемое выражение. Оператор `return` может встречаться в коде функции несколько раз. Впрочем, возвращаемую величину, а также сам оператор `return` можно и не указывать. В этом случае функция ничего не будет возвращать.

Определение функции, будучи помещенным в программу, усваивается интерпретатором, но сама функция (ее код или тело) не выполняется. Чтобы выполнить функцию, определение которой задано, необходимо написать в программе выражение вызова этой функции:

```
имя_функции(параметры)
```

Имя функции в выражении ее вызова должно полностью (вплоть до регистра) совпадать с ее именем в определении. Параметры, если они заданы в определении функции, в вызове функции представляются конкретными значениями, переменными или выражениями.

В JavaScript возможно несоответствие числа параметров в определении функции и в ее вызове. Если в функции определены, например, три параметра, а в вызове указаны только два, то последнему параметру будет автоматически присвоено значение `null`. Наоборот, лишние параметры в вызове функции будут просто проигнорированы.

Внутри тела функции можно создавать переменные либо оператором присваивания, либо с помощью ключевого слова `var`. При этом возникает вопрос об области действия переменных, который мы рассматривали в разд. 16.3.2.

Если вы используете просто оператор присваивания, то возможны две ситуации:

- Переменная в операторе присваивания встречается первый раз в вашей программе именно в теле функции. В этом случае она действует в пределах кода (тела) данной функции, но после выполнения функции эта переменная продолжает существовать и во внешней программе. Таким образом, эта переменная *глобальная*.
- Переменная в операторе присваивания уже определена во внешней программе. Тогда она действует также внутри функции и продолжает существовать после завершения ее выполнения, поскольку она была создана во внешней программе как глобальная переменная.

Если в теле функции вы инициализируете переменную с помощью ключевого слова `var`, то эта переменная будет действовать только в пределах кода функции, независимо от того, была ли одноименная переменная определена во внешней программе, или нет. Если переменная создана с ключевым словом `var` впервые в теле функции, то она будет *локальной*, т. е. недоступной из внешней программы. Таким образом, инициализация переменной с помощью выражения с ключевым словом `var` создает локальную переменную.

Если в теле функции вы используете переменную, определенную только во внешней программе, то изменение ее значения в теле функции сохранится даже после завершения выполнения этой функции, поскольку это глобальная переменная.

Если во внешней программе вы определили некоторые переменные, имена которых совпадают с формальными параметрами в определении функции, а затем указываете их в качестве параметров вызова этой функции, то произойдет следующее. Функция воспримет значения параметров, определенных во внешней программе, однако любые изменения их значений в теле функции останутся локальными, т. е. после завершения работы кода функции значения указанных переменных останутся прежними, как до ее вызова.

Программа может содержать и определение функции, и выражения ее вызова. При этом порядок их следования не важен: вы можете сначала написать определение функции, а затем где-нибудь в программе привести ее вызов, или, наоборот, написать сначала вызов функции, а ее определение разместить где-нибудь в конце программы или даже в отдельном файле с расширением js. Вместе с тем, если ваш сценарий расположен в нескольких разделах (контейнерах `<script>`), то определение функции должно находиться в ранее загружаемом разделе, либо в том же разделе, что и вызов данной функции.

В *разд. 16.4.7*, посвященном операторам цикла, мы уже рассматривали программный код вычисления факториала числа `n`. Здесь программу вычисления `n!` мы оформим в виде функции.

Пример функции вычисления факториала `n!`

```
function factorial(n){
  if(n <= 1) {return 1};
  rezult=2;   // result — переменная для результата
  for (i=3; i <=n; i++) {
   rezult=rezult*i
  }
  return result
}
```

Для вычисления факториала конкретного числа, например 12, следует записать выражение `factorial(12)`.

Если в нашей программе определена описанная функция `factorial()`, то вызов этой функции может выглядеть так:

```
var m=10;
x=factorial(m)  // значение x равно 3628800
```

Определение функции может содержать вызов этой же функции — это так называемое *рекурсивное определение функции*. В качестве примера рекурсивного определения функции в учебной литературе обычно приводят функцию вычисления факториала.

Пример функции вычисления факториала `n!` с помощью рекурсии

```
function factorial(n){
  if(n <= 1){ return 1};
  return n*factorial(n-1) // вызов функции factorial()
}
```

Чтобы избежать ошибок, старайтесь не использовать в коде функций имена переменных, которые были инициализированы во внешней программе. По возможности стремитесь к тому, чтобы все, что делается внутри вашей функции, было локальным, формально не зависимым от внешнего окружения.

Определение функции может содержать в себе определения других функций, однако такие вложенные функции доступны только из кода функции, содержащей их определения.

16.5.3. Объект *Function*

Определение функции, описанное в разд. 16.5.2, создает так называемый экземпляр объекта `Function`, обладающий полезными свойствами и методами. Однако функцию можно определить и непосредственно через объект `Function`:

имя_функции = new Function("*пар1*",… ,"*парN*", "*оператор1;… ; операторN*")

Здесь ключевое слово `new` — оператор вызова конструктора объекта `Function`, создающего экземпляр этого объекта. Названия всех параметров являются строковыми значениями и разделяются запятыми. Заключительная строка в списке параметров содержит операторы кода тела функции, разделенные точкой с запятой. Все элементы в круглых скобках необязательные. Имя функции совпадает с именем переменной в операторе присваивания.

Функцию, определенную как экземпляр объекта `Function`, можно вызвать обычным способом:

имя_функции(*параметры*)

Примеры

```
Srectangle = new Function("width", "height", "var s=width*height;
             return s");
Srectangle(2, 3);   // возвращает 6

var expr = "var s = width*height; return s";
Srectangle = new Function("width", "height", expr);
```

```
Srectangle(2, 3);   // возвращает 6

a = "width";
b="height";
expr = "var s = width*height; return s";
Srectangle = new Function(a, b, expr);
Srectangle(2, 3);   // возвращает 6
```

При любом задании функции (стандартном или с помощью ключевого слова `new`), автоматически создается экземпляр объекта `Function`, который обладает своими свойствами и методами.

Перечислим свойства и метод объекта `Function`.

- `arguments` — массив значений параметров, переданных функции. Элементы массива индексируются, начиная с нуля. Поскольку это массив, он имеет свойства и методы объекта `Array` (в частности, свойство `length` — длина массива). Массивы будут подробно рассмотрены в разд. 16.7.

Свойство `arguments` применяется в теле определения функции, когда требуется проанализировать параметры, переданные ей при вызове. Например, можно узнать, сколько в действительности было передано параметров, не являются ли их значения пустыми (`""`, `0`, `null`) и т. п. Это свойство особенно полезно при разработке универсальных библиотечных функций.

Синтаксис выражения следующий:

имя_функции.arguments

Пример

```
/* Функция, приводимая в этом примере, возвращает строку, содержащую значения параметров и их общее количество, которые были указаны в вызове функции (а не в ее определении!) */
   function myfunc(a, b,c){
    var arglenth=myfunc.arguments.length;  /* количество переданных
                                              параметров */
    var x="";
    for (i=0; i< myfunc.arguments.length;i++) {
     x += myfunc.arguments[i] + ","
    }
    return x+"Всего: "+ myfunc.arguments.length
   }

   myfunc(5, "Вася");    // "5, Вася, Всего: 2"
   myfunc();             // "Всего: 0"
   myfunc(null,"",0,25); // "null,,0,25,Всего: 4"
```

Пример

```
function sum(){
var arglenth=sum.arguments.length;  /* количество переданных
                                       параметров */
 var i, s=0;
 for (i=0; i< sum.arguments.length;i++) {
  s += sum.arguments[i];
 }
 return s
}
sum(2,5,10);    // возвращает 17
```

- `length` — количество параметров, указанных в определении функции. Синтаксис:

  имя_функции.length

Пример

```
function myfunc(a, b, c, d){
 return myfunc.arguments.length
}
myfunc(a,b);   // 2
myfunc.length;   // 4
```

- `caller` — содержит ссылку на функцию, из которой была вызвана данная функция. Если функция не вызывалась из другой функции, то значение этого свойства равно `null`. Синтаксис:

  имя_функции.caller

  В свойстве *имя_функции*.caller содержится все определение функции, из которой была вызвана функция *имя_функции*. В примере, показанном на рис. 16.17, вызов функции `f1()` приведет к вызову функции `f2()`, которая отобразит на экране окно с сообщением, содержащим код определения функции `f1()`.

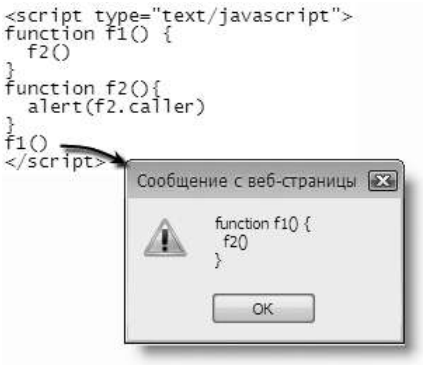

Рис. 16.16. Результат выполнения функции `f1()` — окно со значением `f2.caller`

Вы можете определить функцию без параметров, но обрабатывающую параметры, переданные ей при вызове. В листинге 16.12 приведено определение функции `sum()`, которая возвращает сумму любого количества значений, переданных ей в качестве параметров. Поскольку мы не знаем заранее, сколько параметров ей будет передано, то в определении функции последние и не указываются.

Листинг 16.12. Функция вычисления суммы значений произвольного числа параметров

```
function sum() {
/* Параметры должны быть числовыми */
 var s = 0;
 for (var i = 0; i<sum.arguments.length; i++){
   s+= sum.arguments[i];
 }
 return s // сумма значений параметров
}
```

Примеры вызовов функции `sum()` из листинга 16.12

```
sum(1,2,3)      // 6
sum(5,4)      // 9
sum()         // 0
```

- `toString()` — метод, возвращающий определение функции в виде строки. Синтаксис:

  имя_функции`.toString()`

  Иногда этот метод используют в процессе отладки программ с помощью диалоговых окон. В примере на рис. 16.18 в диалоговое окно выводится код определения функции.

Кроме рассмотренных ранее, возможны так называемые анонимные определения функций:

имя_переменной = function(*параметры*){*код*}.

Здесь имя функции задается именем переменной. Функция вызывается через выражение *имя_переменной*(*параметры*). Поскольку одной переменной можно присвоить другую, то к одной и той же функции можно обращаться под разными именами (рис. 16.19). В этом примере переменные `myfunc` и `Srectangle` имеют одно и то же значение — строку

`"function(width, height){var s= width*height; return s}"`.

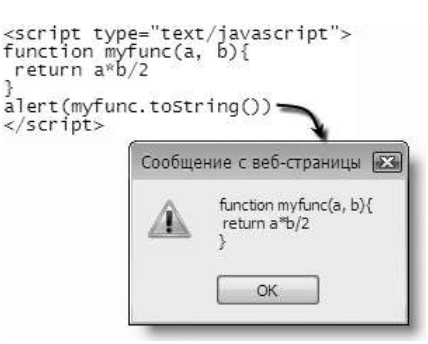

Рис. 16.18. Результат выполнения `alert(myfunc.toString())`

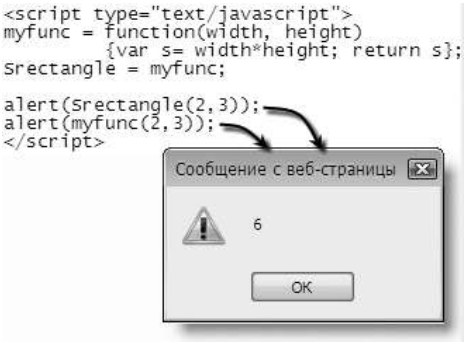

Рис. 16.19. Пример использования анонимного определения функции

Анонимные определения функций удобны при создании методов объектов (см. *разд. 16.10.1*).

Объектная природа функций позволяет создать так называемые статические переменные. Статическая переменная является локальной, область видимости которой — тело функции, где она была определена. Вместе с тем, ее значение сохраняется между вызовами данной функции и может быть изменено только в ней.

Статическая переменная создается как свойство функции. В следующем примере определяется функция `sum(x)`, которая суммирует переданный ей параметр с текущим значением статической переменной.

Пример создания статической переменной

```
function sum(x){
  sum.summa=sum.summa+x;// прибавляем к текущему значению
  return sum.summa
}
sum.summa=0; // определяем статическую переменную
/* Вызовы функции sum() */
sum(5);    // возвращает 5
sum(5);    // возвращает 10
sum(10);   // возвращает 20
```

Обратите внимание, что значение статической переменной нельзя изменить вне вызова функции `sum()`, например, с помощью выражения `sum.summa=3`. Это можно сделать только путем вызова функции. Однако в сценарии можно прочитать текущее значение статической переменной `sum.summa`.

16.6. Строки

При разработке сценариев для Web-сайтов работа с текстовыми строками занимает ведущее место. Так, нередко возникает задача проанализировать, что именно ввел пользователь в поля формы, является ли введенный текст адресом электронной формы или URL и т. п. Возможности работы со строками предоставляют встроенные объекты `String` и `RegExp`. Наиболее мощные (но и более сложные для понимания) средства обработки так называемых регулярных выражений (шаблонов) дает объект `RegExp`, который мы не будем рассматривать в данной книге подробно (пример см. в разд. 16.6.3). В большинстве случаев достаточно свойств и методов объекта `String`.

16.6.1. Кавычки и специальные символы

Строковое значение — последовательность символов, заключенная в кавычки, двойные или одинарные:

```
x = "Это — текстовая строка";
y = 'Это — тоже текстовая строка';
```

Внутри строки может находиться подстрока, заключенная в кавычки другого вида:

```
str1 = "ОАО 'Рога и копыта' предлагает хвосты";
str2 = 'ОАО "Рога и копыта" предлагает хвосты';
```

Внешние кавычки — это элементы синтаксиса: они ограничивают набор символов, относящихся к одному и тому же значению строкового типа. При выводе строки на экран эти кавычки не отображаются.

Внутренние кавычки интерпретатор рассматривает как символы самой строки, если только они отличаются от внешних. Например, если внешние кавычки двойные, то внутренние должны быть одинарными, и наоборот. В противном случае возникнет ошибка. Вместе с тем, внутри строки допустимы кавычки того же вида, что и внешние, если перед ними написать обратную косую черту (\):

```
str3 = "ОАО \"Рога и копыта\" предлагает хвосты";
str4 = 'ОАО \'Рога и копыта\' предлагает хвосты';
```

При написании простого, но длинного строкового значения в текстовом редакторе нельзя делать переход на новую строку с помощью клавиши <Enter>, как в следующем примере:

```
var x = "Дорогой друг!
Спешу сообщить тебе важную новость";
```

Если строка слишком велика, лучше написать так:

```
var x = "Дорогой друг!";
x+= "Спешу сообщить тебе важную новость";
```

В строках можно использовать специальные символы, называемые еще управляющими кодами. Два из них, `\"` и `\'`, мы уже рассмотрели. Полный список специальных символов приведен в табл. 16.11.

На рис. 16.20 показаны несколько примеров использования управляющих кодов. Обратите внимание, что символ `\n` перехода на другую строку работает при выводе в диалоговое окно, а при выводе в окно браузера заменяется пробелом.

Таблица 16.11. *Управляющие коды*

Управляющий код	**Значение**
`\b`	Возврат на одну позицию
`\t`	Табуляция горизонтальная
`\n`	Переход на новую строку
`\v`	Табуляция вертикальная
`\f`	Подача страницы
`\r`	Возврат каретки
`\"`	Двойная кавычка
`\'`	Одинарная кавычка
`\\`	Обратная косая черта
`\`*ooo*	Символ в кодировке Latin-1, представленный 8-ричными цифрами `ooo`. Допустимы значения от 000 до 377
`\x`*hh*	Символ в кодировке Latin-1, представленный 16-ричными цифрами `hh`. Допустимы значения от 00 до FF
`\u`*hhhh*	Символ в кодировке Unicode, представленный 16-ричными цифрами `hhhh`

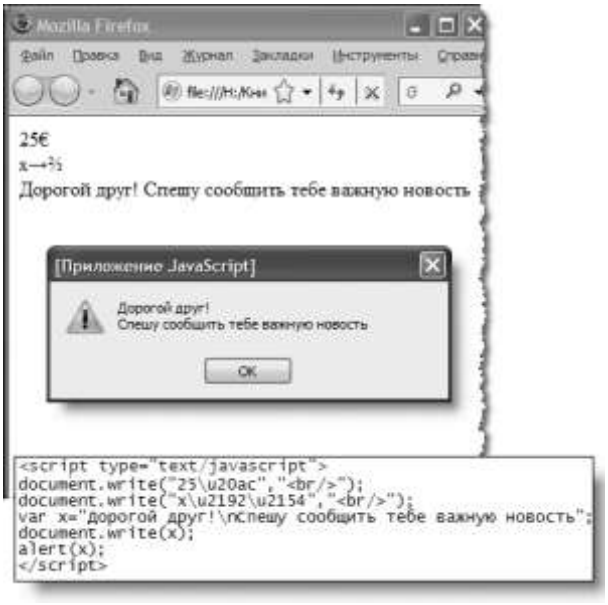

Рис. 16.20. Пример использования управляющих символов

16.6.2. Объект *String*

При использовании оператора присваивания переменной некоторого строкового значения автоматически создается экземпляр объекта `String`. Свойства и методы этого объекта применяются при обработке строк. Например, мы можем узнать длину строки, выделить из нее какую-нибудь часть (подстроку) и т. п.

Пусть `xstring` — переменная с некоторым строковым значением. Тогда синтаксис обращения к свойствам и методам объекта `String`, экземпляр которого ассоциирован с данной строкой, такой:

```
xstring.свойство;
xstring.метод(параметры);
```

Рассмотрим свойства и методы объекта `String` подробнее.

- `length` — длина или, иными словами, количество символов (включая пробелы) в строке; целое число.

Пример

```
"Иван".length      // значение равно 4
"Привет\nвсем".length  /* значение равно 11 (\n — один символ
              перевода строки) */
```

```
x=""        // пустая строка
x.length      // значение равно 0 (пустая строка имеет длину 0)
```

- `prototype` — свойство (прототип), позволяющее добавить новые свойства и методы ко всем создаваемым экземплярам строкового объекта, если уже существующих вам окажется недостаточно.

Пример.

```
/* Создадим новый метод для всех экземпляров строковых объектов. Со-
держание этого метода определяется пользовательской функцией. */

function myFunc() {   // функция для нового метода
 return "Вадим"
}
// Добавление нового метода myName к прототипу:
String.prototype.myName()=myFunc
mystring="Автор этой книги — " + "Дунаев ".myName()
// значение mystring равно "Автор этой книги — Дунаев Вадим"
```

Подробнее о свойстве `prototype` см. разд. 16.10.2.

- `charAt(`*индекс*`)` — возвращает символ, занимающий в строке указанную позицию. Синтаксис:

  строка`.charAt(`*индекс*`)`

  Возвращает односимвольную или пустую строку.

  Параметр *индекс* является числом, индекс первого символа равен нулю.

Примеры

```
"Привет".charAt(2)    // значение равно "и"
"Привет".charAt(15)   // значение равно ""
mystring="Привет"
mystring.charAt(mystring.length-1) /* значение последнего символа
                                      равно "т" */
```

- `charCodeAt(`*индекс*`)` — преобразует символ в указанной позиции строки в его числовой эквивалент (код). Синтаксис:

  строка`.charCodeAt(`*индекс*`)`

Примеры

```
"abc".charCodeAt()    // значение равно 97
"abc".charCodeAt(1)   // значение равно 98
"abc".charCodeAt(25)  // значение равно NaN
```

```
"".charCodeAt(25)     // значение равно NaN
"я".charCodeAt(0)     // значение равно 1103
```

- `fromCharCode(`*`номер1 , номер2 , ... , номерN`*`)` — возвращает строку символов, числовые коды которой указаны в качестве параметров. Синтаксис:

  `String.fromCharCode(`*`номер1 , номер2 , ... , номерN`*`)`

  Возвращает строку.

  Пример

  ```
  String. fromcharCode(97,98,1102) // значение равно "abю"
  ```

- `concat(`*`строка`*`)` — конкатенация (склейка) строк. Синтаксис:

  `строка1``.concat(`*`строка2`*`)`

  Возвращает строку. Этот метод действует так же, как и оператор сложения `+` для строк: к строке *`строка1`* приписывается справа *`строка2`*.

  Примеры

  ```
  "Иван".concat("Иванов");  // значение равно "ИванИванов"
  x="Федор" + " ";          // значение x равно "Федор "
  x.concat("Иванов");       // значение равно "Федор Иванов"
  ```

- `indexOf(`*`строка_поиска, индекс`*`)` — ищет строку, указанную параметром, и возвращает индекс ее первого вхождения. Синтаксис:

  `строка``.indexOf(`*`строка_поиска, индекс`*`)`

  Метод ищет позицию первого вхождения *`строка_поиска`* в строку *`строка`*. Возвращаемое число (индекс вхождения) отсчитывается от нуля. При неудаче возвращается –1. Поиск в пустой строке всегда возвращает –1. Поиск пустой строки всегда возвращает нуль.

  Второй (необязательный) параметр указывает индекс, с которого следует начать поиск.

  Этот метод хорош в комбинации с методом выделения подстроки `substr()` (см. далее), когда сперва требуется определить позиции начала и конца выделяемой подстроки. Рассматриваемый здесь метод подходит для определения начальной позиции.

  Примеры

  ```
  x ="В первых строках своего письма";
  x.indexOf("первых");          //значение равно 2
  ```

```
x.indexOf("первых строках");  //значение равно 2
x.indexOf("вторых строках");  //значение равно -1
x.indexOf("В");               //значение равно 0
x.indexOf("в");               //значение равно 5
x.indexOf("в", 7);            //значение равно 18
x.indexOf(" ");               //значение равно 1
x.indexOf(" ", 5);            //значение равно 8
x.indexOf("");                //значение равно 0
```

- `lastIndexOf(`*`строка_поиска`*`,`*`индекс`*`)` — ищет строку, указанную параметром, и возвращает индекс ее первого вхождения; при этом поиск начинается с конца исходной строки, но возвращаемый индекс отсчитывается от ее начала, т. е. от нуля. Синтаксис:

  `строка``.lastIndexOf(`*`строка_поиска`*`,`*`индекс`*`)`

  Возвращает число.

  Метод аналогичен рассмотренному ранее `indexOf()` и отличается лишь направлением поиска.

Примеры

```
x ="В первых строках своего письма"
x.lastIndexOf("первых"); // значение равно 2
x.lastIndexOf("а");      // значение равно 29
x.indexOf("а");          // значение равно 14
```

- `localeCompare(`*`строка`*`)` — позволяет сравнивать строки в кодировке Unicode, т. е. с учетом используемого браузером языка общения с пользователем. Синтаксис:

  `строка1``.localeCompare(`*`строка2`*`)`

  Возвращает число.

  Если сравниваемые строки одинаковы, метод возвращает нуль. Если *`строка1`* меньше, чем *`строка2`*, то возвращается отрицательное число, в противном случае — положительное. Сравнение строк происходит путем сопоставления сумм кодов их символов. Абсолютное значение возвращаемого числа зависит от браузера. Так, Internet Explorer, Firefox и Safari возвращают 1 или –1, а Chrome — разность сумм кодов в кодировке Unicode.

- `slice(`*`индекс1, индекс2`*`)` — возвращает подстроку исходной строки, начальный и конечный индексы которой указываются параметрами, за исключением последнего символа. Синтаксис:

  `строка``.slice(`*`индекс1, индекс2`*`)`

Данный метод не изменяет исходную строку.

Если второй параметр не указан, то возвращается подстрока с начальной позицией *индекс1* и до конца строки. Отсчет позиций начинается с начала строки. Первый символ строки имеет нулевой индекс. Если второй параметр указан, то возвращается подстрока исходной строки, начиная с позиции *индекс1* и до позиции *индекс2*, исключая последний символ. Если второй параметр отрицательный, то конечный индекс отсчитывается от конца строки. В этом основное различие методов `slice()` и `substr()`. Сравните также этот метод с методом `substring()`.

Примеры

```
x="В первых строках своего письма";
x.slice(3,8);  // значение равно "ервых"
x.slice(3,-2)  // значение равно "ервых строках своего пись"

/* Анализ адреса электронной почты */
x="mumu@gerasim.ru";
i=x.indexOf("@");            // значение равно 4
_name=x.slice(0, i);         // значение равно "mumu"
_domen=x.slice(i+1, x.indexOf(".")); // значение равно "gerasim"
_domen=x.slice(i+1, -3);             // значение равно "gerasim"
```

- `split(`*разделитель*`, `*ограничитель*`)` — возвращает массив элементов, полученных из исходной строки. Синтаксис:

  строка`.split(`*разделитель*`, `*ограничитель*`)`

  Первый параметр — строка символов в качестве разделителя строки на элементы. Второй необязательный параметр — число, указывающее, сколько элементов строки, полученной при разделении, следует включить в возвращаемый массив. Если разделитель — пустая строка, то возвращается массив символов строки.

Примеры

```
x="Привет всем";
x.split(" ");   // значение – массив из элементов: "Привет", "всем"
x.split("е");   // значение – массив из элементов: "Прив", "т вс","м"
x.split("е",2); // значение – массив из элементов: "Прив", "т вс"
```

- `substr(`*индекс*`, `*длина*`)` — возвращает подстроку исходной строки, начальный индекс и длина которой указываются параметрами. Синтаксис:

  строка`.substr(`*индекс*`, `*длина*`)`

  Данный метод не изменяет исходную строку.

Если второй параметр не указан, то возвращается подстрока с начальной позицией *индекс1* и до конца строки. Отсчет позиций выполняется с начала строки. Первый символ строки имеет нулевой индекс. Если второй параметр указан, то возвращается подстрока исходной строки, начиная с позиции *индекс1* и с общим количеством символов, равным значению *длина*. Сравните этот метод с методами `slice()` и `substring()`.

Примеры

```
x="Привет всем";
x.substr(7,4);     // значение равно "всем"

/* Анализ адреса электронной почты */
x = mumu@gerasim.ru;
i = x.indexOf("@");     // значение равно 4
name = x.substr(0, i);  // значение равно "mumu"
domen = x.substr(i+1)   // значение равно "gerasim.ru"
```

- `substring(`*индекс1*`, `*индекс2*`)` — возвращает подстроку исходной строки, начальный и конечный индексы которой указываются параметрами. Синтаксис:

  строка`.substring(`*индекс1*`, `*индекс2*`)`

  Данный метод не изменяет исходную строку.

  Порядок индексов не важен: наименьший из них считается начальным. Отсчет позиций выполняется с начала строки. Первый символ строки имеет нулевой индекс. Символ, соответствующий конечному индексу, не включается в возвращаемую строку. Сравните этот метод с методами `substr()` и `slice()`.

Примеры

```
x="Привет всем";
x.substring(0,6);         // значение равно "Привет"
x.substring(7, x.length); // значение равно "всем"
x.substring(7, 250);      // значение равно "всем"
x.substring(250, 7);      // значение равно "всем"
```

- `toLocaleLowerCase()`, `toLowerCase()` — переводят строку в нижний регистр. Синтаксис:

  строка`.toLocaleLowerCase()`
  строка`.toLowerCase()`

Возвращают строку.

Приведение строк к одному и тому же регистру требуется, например, при сравнении содержимого строк без учета регистра. Кроме того, многие серверы чувствительны к регистру, в котором определены имена файлов и папок (обычно требуется, чтобы они были записаны в нижнем регистре).

Примеры

```
x="ЗдраВствуйТе";
x.toLocaleLowerCase();    // значение равно "здравствуйте"
x.toLowerCase();          // значение равно "здравствуйте"
y="Здравствуйте";
x == y;                   // значение равно false
x.toLowerCase() == y.toLowerCase(); // значение равно true
```

- `toLocaleUpperCase()`, `toUpperCase()` — переводят строку в верхний регистр. Синтаксис:

  строка`.toLocaleUpperCase()`
  строка`.toUpperCase()`

  Возвращают строку.

  Назначение аналогично двум предыдущим методам.

Примеры

```
x="ЗдраВствуйТе";
x.toLocaleUpperCase();     // значение равно "ЗДРАВСТВУЙТЕ"
x.toUpperCase();           // значение равно "ЗДРАВСТВУЙТЕ"
y="Здравствуйте";
x == y;                    // значение равно false
x.toUpperCase() == y.toUpperCase(); // значение равно true
```

Следующая группа методов позволяет отформатировать строки при их выводе в окно браузера так же, как и с помощью тегов HTML:

- `anchor("`*anchor_имя*`")`
- `blink()`
- `bold()`
- `fixed()`
- `fontcolor(`*значение_цвета*`)`
- `fontsize(`*число от 1 до 7*`)`
- `link(`*расположение или URL*`)`
- `big()`
- `small()`
- `strike()`
- `sub()`
- `sup()`

- `italics()`

Примеры

```
document.write("Глава 2".anchor("volume2"))  /* эквивалентно HTML-коду:
                 <a name="volume2"> Глава 2 </a> */

document.write("Страница автора".link("http:/dunaevv1.narod.ru"))
   /* эквивалентно HTML-коду:
    <a href="http:/dunaevv1.narod.ru")> Страница автора </a> */
```

16.6.3. Функции вставки и замены подстрок

При обработке строк часто требуется вставить или заменить подстроки. Удаление подстроки из данной строки можно рассматривать как частный случай замены (на пустую строку). С помощью методов объекта `String` можно написать программу для решения этой задачи. Поскольку она часто встречается, то целесообразно оформить программу в виде нескольких функций.

Рассмотрим сначала функцию вставки одной строки в другую. Назовем ее, например, `insstr`. Данная функция должна принимать три параметра: исходную строку `s1`, вставляемую строку `s2` и индекс позиции вставки `n`. Листинг 16.13 содержит ее определение и примеры вызова.

Листинг 16.13. Функция вставки строки

```
function insstr(s1,s2,n) {
 return s1.slice(0,n) + s2 + s1.slice(n)
}

insstr("Привет, друзья", " мои", 7);   // "Привет, мои друзья"
insstr("Привет, друзья", " мои", 100);  // "Привет, друзья мои"
```

Теперь займемся функцией, которая заменяет в исходной строке все вхождения заданной подстроки на подстроку замены. Назовем эту функцию `replacestr`. Она должна принимать три параметра: исходную строку `s1`, заменяемую подстроку `s2` и подстроку `s3`, которой следует заменить все вхождения `s2` в `s1` (`s2` может встречаться в `s1` несколько раз).

Очевидно, прежде всего, необходимо найти все вхождения `s2` в `s1`. Если исходная строка не содержит в себе подстрок `s2`, то функция должна вернуть исходную подстроку без всяких изменений. В противном случае требуется изъять из `s1` все вхождения `s2`, а на их место вставить подстроку `s3`. Полу-

ченная таким образом строка должна возвращаться функцией в качестве результата. Листинг 16.14 содержит определение функции для замены подстрок и два примера ее вызова.

Листинг 16.14. Функция замены подстроки

```
function replacestr(s1, s2, s3) {
 var s="";       // обработанная часть строки
 var i;
 while (true) {
  i=s1.indexOf(s2)  // индекс вхождения s2 в s1
  if (i >= 0) {
   s=s + s1.substr(0, i) + s3;   /* обработанная часть строки */
   s1=s1.substr(i + s2.length);  /* оставшаяся часть строки */
  }else break;
 }
 return s + s1
}

replacestr("bacdae","a","X");  // "bXcdXe"
var x="Иван Иванов";
replacestr(x,"Иван","Федор");  // "Федор Федоров"
```

Примечание

Если заменяемая подстрока `s2` пустая, то цикл будет продолжаться бесконечно, поскольку пустая строка входит в состав любой строки.

Чтобы избежать зацикливания, следует немного изменить код тела функции. Но что должна возвращать наша функция, если `s2` — пустая строка? Вы можете решить, что в этом случае следует просто вернуть исходную строку `s1`, или же, например, считать пустую строку строкой, содержащей пробел. Последний вариант представляется мне более интересным. Листинг 16.15а содержит код, реализующий эту идею.

Листинг 16.15а Функция замены подстроки (вариант1)

```
function replacestr(s1, s2, s3) {
 if (s2 == "") s2=" "; // заменяем пустую строку на строку с пробелом
 var s="";             // обработанная часть строки
 while (true) {
  i=s1.indexOf(s2);     // индекс вхождения s2 в s1
  if (i >= 0) {
   s = s + s1.substr(0, i) + s3;   /* обработанная часть строки */
   s1 = s1.substr(i + s2.length);  /* оставшаяся часть строки */
```

```
  }else break;            // выход из цикла
 }
 return s + s1
}
```

Рассмотренные функции вставки и замены подстрок (`insstr()` и `replacestr()`) без всяких доработок пригодны к практическому применению в программах, где требуется обработка строк.

Более эффективные коды для решения задач поиска и замены подстрок можно создать с помощью встроенного объекта `RegExp`, описание которого довольно объемное и выходит за рамки книги. В листинге 16.15б приведен простой пример использования данного объекта для замены подстрок.

Листинг 16.15б Функция замены подстроки (вариант 2)

```
function  replacestr(s1, s2, s3) {
 if (s2 == "") s2=" "; // заменяем пустую строку на строку с пробелом
 var s=RegExp(s2, "g");
 return s1.replace(s, s3)
}
```

16.6.4. Функции удаления ведущих и заключительных пробелов

При обработке строк (например, введенных пользователем в поля формы) нередко возникает задача удаления лишних ведущих и завершающих пробелов. Во многих языках программирования есть соответствующие встроенные функции, но в JavaScript их нет. Поэтому не помешает создать их самостоятельно с помощью имеющихся средств.

Решить поставленную задачу можно несколькими способами. Вот один из них. Преобразуем исходную строку в массив слов, используя в качестве разделителя один пробел `" "`. Затем необходимо проанализировать первые или последние элементы массива в зависимости от того, что нам требуется: удалить ведущие или завершающие пробелы в исходной строке. Допустим, необходимо избавиться от ведущих пробелов. Тогда, если в исходной строке было `N` пробелов, то первые `N` элементов массива будут содержать пустые строки `""`. Мы проверяем в цикле значения первых элементов, пока не найдем непустой или пока не исчерпаем весь массив. Если первый непустой элемент массива имеет индекс `i`, то создадим новый массив, содержащий элементы исходного, начиная с этого индекса. Наконец, склеим элементы этого массива в одну строку, используя в качестве разделителя строку

с единственным пробелом " ". Все, что нам понадобится, — методы объекта String, операторы цикла и условного перехода.

Сначала рассмотрим функцию ltrim(), удаляющую ведущие пробелы из исходной строки (листинг 16.16).

Листинг 16.16. Функция удаления ведущих пробелов

```
function ltrim(xstr){
 if (!(xstr.indexOf(" ") == 0))
 return xstr;       /* вернуть исходную строку, если в ней нет
                       ведущих пробелов */
 var astr = xstr.split(" "); // создаем массив из слов строки
 var i=0;
 while (i < astr.length){
  if (!(astr[i] == (""))) break; /* выходим из цикла,
                                    если элемент не пуст */
  i++;
 }
 astr = astr.slice(i);   // создаем массив
 return astr.join(" ");     // склеиваем элементы массива в строку
}
```

Функция rtrim() для удаления заключительных пробелов в строке устроена аналогично (листинг 16.17). Ее отличие в том, что поиск пробелов происходит с конца строки. Обратите внимание, что счетчик цикла в этом случае убывающий.

Листинг 16.16. Функция удаления заключительных пробелов

```
function rtrim(xstr){
 if (!(xstr.lastIndexOf(" ") == xstr.length - 1)) return xstr;
 var astr=xstr.split(" ");
 var i=astr.length -1;
 while (i>0){
  if (!(astr[i] == (""))) break;
  i--;
 }
 astr = astr.slice(0, i+1);
 return astr.join(" ")
}
```

Чтобы удалить из строки и ведущие, и концевые пробелы, достаточно выполнить следующее выражение:

```
ystr=rtrim(ltrim(xstr));
```

Впрочем, можно создать специальную функцию `trim()`, которая тримингует строку:

```
function trim(xstr) {
 return rtrim(ltrim(xstr))
}
```

16.7. Массивы

Как уже упоминалось, *массив* представляет собой упорядоченный набор данных. В качестве элементов массива могут выступать как примитивные данные (например, строки, числа), так и другие массивы. В последнем случае можно говорить о многомерных массивах (т. е. о массивах массивов). Число элементов в массиве называется *длиной* массива. К элементам массива можно обращаться в программе по их порядковому номеру (*индексу*). Нумерация элементов массива начинается с нуля, так что первый элемент имеет индекс 0, а последний — на единицу меньший, чем длина массива.

Массивы применяются во многих более или менее сложных программах обработки данных, а в некоторых случаях без них просто не обойтись. Если среди используемых данных есть группы таких, которые обрабатываются одинаково, то, возможно, лучше организовать их в виде массива.

16.7.1. Создание массива

Существует несколько способов создания массива. Самый простой, пригодный в случае небольших массивов, заключается в том, чтобы задать список элементов, заключенный в квадратные скобки:

имя_массива = [*элемент1*, *элемент2*, … ,*элементN*];

Здесь *имя_массива* — переменная, значением которой является массив. Доступ к элементам массива обеспечивается с помощью выражения *имя_массива*[*индекс*], где *индекс* — целое число от 0 до *N*–1 (нумерация элементов массива начинается с нуля). Например, `элемент1` — первый элемент массива, но его индекс равен нулю. Это так называемая литеральная форма записи массива. При этом автоматически создается экземпляр встроенного объекта `Array`. То же самое можно сделать и с помощью явного выражения создания экземпляра объекта `Array`:

имя_массива = new Array(*длина_массива*);

Здесь *длина_массива* — необязательный числовой параметр. Если длина массива не указана, то создается пустой массив, не содержащий ни одного элемен-

та. В противном случае формируется массив с указанным количеством элементов, однако все они имеют значение `undefined` (т. е. не определены).

У объекта `Array` имеется свойство `length`, значением которого является длина массива. Получить значение этого свойства можно, указав выражение

имя_массива`.length`.

Создав массив, можно присвоить значения его элементам с помощью оператора присваивания. В его левой части указывают имя массива, а рядом с ним в квадратных скобках — индекс элемента. Мы уже говорили, что к элементам массива обращаются по индексу:

имя_массива`[`*индекс*`]`.

Рассмотрим создание массива `земля`, содержащего в качестве элементов некоторые характеристики нашей планеты (листинг 16.18). Обратите внимание, что элементы в этом массиве различных типов (строковые и числовые).

Листинг 16.18. Пример создания массива

```
земля = new Array(4);   // массив из 4-х элементов
земля[0] = "Планета";
земля[1] = "24 часа";
земля[2] = 6378;
земля[3] = 365.25;

земля.length       // значение равно 4
```

Если нам потребуется значение, например, третьего элемента массива, то достаточно использовать выражение `земля[2]`.

При создании массива его элементы можно указать в круглых скобках за ключевым словом `Array`:

```
земля = new Array("Планета", "24 часа", 6378, 365.25);
```

Индексы для элементов такого массива JavaScript создает автоматически.

Элементам массива можно задать индивидуальные имена — присвоить имя каждому элементу, подобно имени свойства объекта (листинг 16.19).

Листинг 16.19. Присвоение имен элементам массива

```
земля = new Array();   // пустой массив
земля.xtype = "Планета";
земля.xday = "24 часа";
земля.radius = 6378;
земля.period = 365.25;
```

В этом случае обращение к элементу происходит как к свойству объекта, например, `земля.radius`. Возможен и такой вариант: `земля["radius"]`. По числовому индексу к элементам в таком массиве обращаться нельзя. Например, значение элемента `земля[2]` равно `undefined`, а не `"radius"`. Массивы с символьными индексами (с именами значений) называют еще ассоциативными.

Для удаления элемента массива служит оператор `delete`, при этом длина массива остается прежней, а удаленный элемент приобретает значение `undefined` (листинг 16.20).

Листинг 16.20. Пример удаления элемента массива

```
myarray = new Array(1,2,3,4);
delete myarray[2];
alert(myarray);     // массив 1,2,,4
alert(myarray.length);   // 4
alert(myarray[2]);     // undefined
```

Чтобы удалить массив целиком, необходимо присвоить ему значение `null`.

16.7.2. Многомерные массивы

Рассмотренные ранее массивы одномерные. Их можно представить себе в виде таблицы из одного столбца. Однако элементы массива могут содержать данные различных типов, в том числе и объекты, а значит, и массивы. Если в качестве элементов некоторого одномерного массива создать массивы, то получится двумерный массив. В свою очередь, элементы-массивы в качестве своих элементов могут содержать некие массивы. В результате получается трехмерный массив. Аналогичным образом создается N-мерный массив. Обращение к элементам такого массива происходит в соответствии со следующим синтаксисом:

имя_массива[*индекс_уровня1*] [*индекс_уровня2*]… [*индекс_уровняN*].

Типичный пример — массив опций меню. У такого меню есть горизонтальная панель с опциями, называемая главным меню. Некоторым опциям главного меню соответствуют (возможно, раскрывающиеся) вертикальные подменю со своими опциями. Мы создаем массив, длина которого равна количеству опций главного меню. Элементы этого массива определяем как массивы названий опций соответствующих подменю. Чтобы была ясна структура нашей кон-струкции, мы выбрали названия опций надлежащим образом. Например, "Меню 2.1" — название первой опции подменю, соответствующего второй опции главного меню (листинг 16.21).

Листинг 16.21. Массив опций меню (вариант 1)

```
menu = new Array();
menu[0] = new array("Меню 1.1", "Меню 1.2", "Меню 1.3");
menu[1] = new array("Меню 2.1", "Меню 2.2");
menu[2] = new array("Меню 3.1", "Меню 3.2" , "Меню 3.3", "Меню 3.4");
```

Чтобы обратиться ко второй опции третьего подменю, следует написать:

```
menu[2][1]    // значение равно "Меню 3.2".
```

Усложним нашу конструкцию, чтобы она содержала не только названия опций подменю, но и названия опций главного меню (листинг 16.22).

Листинг 16.22. Массив опций меню (вариант 2)

```
menu = new Array();
/* Массив опций главного меню: */
menu[0] = new Array("Меню1", "Меню2", "Меню3");
menu[1] = new Array();
menu[1][0] = new Array("Меню 1.1", "Меню 1.2", "Меню 1.3");
menu[1][1] = new Array("Меню 2.1", "Меню 2.2");
menu[1][2] = new Array("Меню 3.1", "Меню 3.2" , "Меню3.3", "Меню 3.4");

menu[0][1]     // значение равно "Меню 2"
menu[0][2]     // значение равно "Меню 3"
menu[1][1][0]  // значение равно "Меню 2.1"
menu[1][2][3]  // значение равно "Меню 3.2"
```

Наиболее часто двумерными массивами представляются данные с табличной структурой. При этом возникает необходимость в таких средствах их обработки, как поиск и фильтрация данных. В листинге 16.23 приведены коды функции `asort()` для сортировки (упорядочивания) строк таблицы (двумерного массива) по значениям указанного столбца и функции `afilter()` для фильтрации (поиска) строк, удовлетворяющих заданному критерию отбора.

Листинг 16.23. Функции сортировки и фильтрации двумерных массивов

```
function asort(arr,fld,direct){
/* Сортировка двумерного массива:
   arr - двумерный массив
   fld - номер поля (столбца), по значениям которого требуется
         отсортировать строки (нумерация столбцов начинается с 0);
         если не указано, то 0
   direct - порядок сортировки (если пусто, то по возрастанию,
                                иначе - по убыванию)
```

```
   Возвращает отсортированный массив
*/
  if (!fld) fld=0;
  if (!direct) {direct=1;} else direct=-1;
  return arr.sort(comp)
  function comp(x,y){
    if (x[fld]<y[fld]) return -direct;
    if (x[fld]>y[fld]) return direct;
    if (x[fld]===y[fld]) return 0
  }
}
/*********************************************************/

function afilter(arr,criterion,count,field){
/* Фильтрация двумерного массива:
   arr – двумерный массив
   criterion - строка с выражением условия выбора строк,
               в которой поля обозначаются в виде fld[0], fld[1] и т.п.;
               например, "fld[0]=='2'||fld[1]=='Маша'"
   count - количество строк, не более которого следует отфильтровать;
           если пусто, то принимается число всех строк
   field - значение для обозначения поля в criterion;
           если не указано, то имеется в виду "fld" (на всякий случай)
   Возвращает массив отобранных строк
*/
  var fltr=[], curcount=0;
  if(!count) count=arr.length;
  if(!field) field="fld";
  var re=new RegExp(field,"g");
  criterion=criterion.replace(re,"arr[i]");
  for(var i=0;i<arr.length;i++){
     if (eval(criterion)){
       fltr=fltr.concat(arr[i]);
       curcount++;
       if(curcount>=count) break
     }
  }
  return fltr
}
```

16.7.3. Копирование массива

Иногда требуется создать копию массива, чтобы сохранить исходные данные и предохранить их от последующих модификаций. Например, рассмотренный далее метод сортировки элементов изменяет исходный массив и необходима предварительная операция копирования. Однако для копирования массива недостаточно просто присвоить его другой переменной. Используя новую переменную и оператор присваивания, мы создаем лишь еще одну ссылку на прежний массив, а не новый массив.

Пример

```
a = new Array(5, 2, 4, 3);
x = a;      // ссылка на массив a
a[2] = 25;  // изменение значение элемента с индексом 2
x[2]        // значение равно 25, т. е. новому значению a[2]
```

В этом примере массивы `a` и `x` совпадают.

Чтобы скопировать массив, т. е. создать новый массив, элементы которого равны соответствующим элементам исходного, следует воспользоваться оператором цикла, в котором элементам нового массива присваиваются значения элементов исходного.

Пример

```
a = new Array(5, 2, 4, 3);
x = new Array();              // ссылка на массив a
for(i = 0; i<a.length; i++)   /* копирование значений массива a
                                 в элементы массива x */
 { x[i] = a[i]}
```

16.7.4. Объект *Array*

Для работы с массивами предназначены свойства и методы объекта `Array`.

Пусть `myarray` — переменная с некоторым значением типа массив (имя некоторого массива). Тогда синтаксис обращения к свойствам и методам объекта `Array` такой:

`myarray.`*`свойство`*`;`

`myarray.`*`метод`*`(`*`параметры`*`);`

Рассмотрим свойства и методы объекта `Array` подробнее.

- `length` — длина или, иными словами, количество элементов в массиве; целое число. Синтаксис:

  имя_массива`.length`

  Поскольку индексация элементов массива начинается с нуля, индекс последнего элемента на единицу меньше длины массива. Это обстоятельство удобно использовать при добавлении к массиву нового элемента:

  `myarray[mayarray.length]=`*значение;*

  Чтобы удалить из массива `n` последних элементов, достаточно просто уменьшить его длину на `n`:

  имя_массива`.length = `*имя_массива*`.length – n;`

  Аналогично можно увеличить длину массива. При этом добавленные элементы будут иметь значение `undefined`.

- `prototype` — свойство (прототип), позволяющее добавить новые свойства и методы ко всем существующим и создаваемым массивам.

  Например, следующее выражение добавляет свойство `author` ко всем уже созданным массивам:

  ```
  Array.prototype.author="Иванов"
  ```

  Если теперь вместо `Array.prototype` написать имя существующего массива, то можно изменить значение свойства `author` только для этого массива (листинг 16.24).

Листинг 16.24. Использование свойства `prototype`

```
myarray = new Array();         // создание массива myarray
xarray = new Array();         // создание массива xarray
Array.prototype.author = "Иванов";  /* добавление свойства author
                   ко всем массивам */
myarray.author = "Иванов младший";  /* изменение свойства author
                   для myarray */
xarray.author = "Сидоров";       /* изменение свойства author
                   для xarray */
```

Прототипу можно присвоить функции, т. е. методы для специфической обработки массивов, которые пополнят множество уже существующих методов объекта `Array`.

Рассмотрим вычисление суммы элементов массива (листинг 16.25). Мы определяем функцию `aSum()`, которая возвращает сумму элементов числового массива. В качестве параметра эта функция принимает массив. Затем создаем конкретный массив чисел. Наконец, присоединяем к прототипу массива новый метод `Sum` — определенную ранее функцию `aSum()`.

Листинг 16.25. Вычисление суммы элементов массива

```
function aSum(xarray){
 var s = 0;
 for(i = 0; i < xarray.length; i++){
  s = s + xarray[i]
 }
 return s
}
myarray = new Array(2, 3, 4); // создаем массив myarray из 3-х чисел
Array.prototype.Sum = aSum;  /* присоединяем метод (около имени
                                функции скобки указывать не нужно) */
s = myarray.Sum(myarray);    /* применяем метод Sum к массиву
                                myarray, передавая его в качестве
                                параметра */
```

Этот пример призван просто проиллюстрировать действие свойства `prototype`. На самом деле вычислить сумму элементов массива можно гораздо проще:

```
s = aSum(myarray)
```

Подробнее о свойстве `prototype` см. разд. 16.10.2.

- `concat(`*массив*`)` — конкатенация массивов, объединяет два массива в третий. Синтаксис:

  имя_массива1`.concat(`*массив2*`)`

  Возвращает массив. Данный метод не изменяет исходные массивы.

Пример

```
a1 = new array(1, 2, "Звезда");
a2 = new array("а", "б", "в", "г");
a3 = a1.concat(a3);  /* результат — массив с элементами:
            1, 2, "Звезда", "а", "б", "в", "г" */
```

- `join(`*разделитель*`)` — создает строку из элементов массива с указанным разделителем между ними; является строкой символов (возможно, пустой). Синтаксис:

  имя_массива`.join(`*строка*`)`

Примеры

```
a = new array(1, 2, "Звезда");
a.join(",")   // значение — строка "1,2,Звезда"

a = new array(1, 2, "Звезда");
```

```
a.join(" ");   // значение – строка "1 2 Звезда"
```

- `pop()` — удаляет последний элемент массива и возвращает его значение. Синтаксис:

  имя_массива`.pop()`

  Возвращает значение удаленного элемента массива. Данный метод изменяет исходный массив.

- `push(`*значение или объект*`)` — добавляет к массиву указанное значение в качестве последнего элемента и возвращает новую длину массива. Синтаксис:

  имя_массива1`.push(`*значение или объект*`)`

  Данный метод изменяет исходный массив.

- `shift()` — удаляет первый элемент массива и возвращает его значение. Синтаксис:

  имя_массива`.shift()`

  Данный метод изменяет исходный массив.

- `unshift(`*значение или объект*`)` — добавляет к массиву указанное значение в качестве первого элемента. Синтаксис:

  имя_массива`.unshift(`*значение или объект*`)`

  Метод изменяет исходный массив.

- `reverse()` — изменяет порядок следования элементов массива на противоположный. Синтаксис:

  имя_массива`.reverse()`

  Возвращает массив. Данный метод изменяет исходный массив.

Пример

```
a = new array(1, 2, "Вася")
a.reverse() // массив с элементами в следующем порядке: "Вася", 2, 1
```

- `slice(`*индекс1*`, `*индексN*`)` — создает массив из элементов исходного массива с индексами указанного диапазона. Синтаксис:

  имя_массива`. slise(`*индекс1*`, `*индексN*`)`

  Возвращает массив. Данный метод не изменяет исходный массив.

  Второй параметр (конечный индекс) не является обязательным. Если он не указан, то создаваемый массив содержит элементы исходного массива, начиная с индекса *индекс1* и до конца. В противном случае создаваемый массив содержит элементы исходного массива, начиная с индекса *индекс1*

и до индекса *индекс2*, за исключением последнего элемента. При этом исходный массив остается без изменений.

Пример

```
a = new array(1, 2, "Звезда","a", "b");
a.slice(1,3);  // массив с элементами: 2, "Звезда"
a.slice(2)   // массив с элементами: "Звезда", "a", "b"
```

- `sort(`*функция_сортировки*`)` — сортирует (упорядочивает) элементы массива с помощью функции сравнения. Синтаксис:

  имя_массива`.sort(`*функция_сравнения*`)`

  Возвращает массив. Данный метод изменяет исходный массив.

  Параметр можно опускать, если он не указан, то при сортировке сравниваются ASCII-коды символов значений. Это удобно для сравнения символьных строк, но не совсем подходит для чисел. Так, число 357 при сортировке считается меньшим, чем 75, поскольку сначала сравниваются первые символы и только в случае их равенства сравниваются следующие, и т. д. Таким образом, метод `sort()` без параметра подходит для простой сортировки массива со строковыми элементами.

  Можно создать собственную функцию для сравнения элементов массива, с помощью которой метод `sort()` упорядочит весь массив. Имя этой функции (без кавычек и круглых скобок) передается методу в качестве параметра. При работе метода функции передаются два элемента массива, а ее код возвращает методу значение, указывающее, порядок следования элементов. Допустим, сравниваются два элемента, `x` и `y`. Тогда в зависимости от числового значения (отрицательного, 0 или положительного), возвращаемого функцией сравнения, методом `sort()` принимается одно из трех возможных решений (табл. 16.12).

Таблица 16.12. *Результаты сравнения двух элементов массива*

Значение, возвращаемое функцией сравнения	**Результат сравнения `x` и `y`**
Меньше 0	`y` следует за `x`
0	Порядок следования `x` и `y` не изменяется
Больше 0	`x` следует за `y`

Итак, критерий сортировки элементов массива определяется кодом функции сравнения. Если элемент массива имеет значение `null`, то в Internet Explorer он размещается в начале массива.

Пример

```
myarray = new Array(4, 2, 15, 3, 30 ); // числовой массив
function comp(x, y) {                  // функция сравнения
 return x - y
}
myarray.sort(comp)     /* массив с элементами в порядке:
                          2, 3, 4, 15, 30 */
```

- `splice(`*индекс*`, `*количество*`, `*элем1*`, `*элем2*` , ...`*элемN*`)` — удаляет из массива несколько элементов и возвращает массив из удаленных элементов, или заменяет значения элементов. Синтаксис:

  имя_массива `splice(`*индекс*`, `*количество*`, `*элем1*`, `*элем2*`, ...`*элемN*`)`

  Данный метод изменяет исходный массив.

  Первые два параметра обязательны, а следующие — нет. Первый параметр является индексом первого удаляемого элемента, а второй — количеством удаляемых элементов.

Пример

```
a = new Array("Вася", "Иаван", "Марья", 12, 5);
x = a.splice(1,3);  /* x — массив элементов: "Иван", "Марья", 12
                       a — массив элементов: "Вася", 5 */
```

Метод `splice()` позволяет также заменить значения элементов исходного массива, если указаны третий и, возможно, последующие параметры. Эти параметры представляют значения, которыми следует заменить исходные значения элементов массива. При таком использовании метода `splice()` важен первый параметр (индекс), а второй (количество) может быть равным нулю. В любом случае, если количество элементов замены больше значения второго параметра, то одна часть элементов исходного массива будет заменена, а другая — просто вставлена в него. При этом метод `splice()` возвращает другой массив, состоящий из элементов исходного, индексы которых соответствуют первому и второму параметрам. Но это справедливо, если второй параметр не равен нулю.

Пример

```
a = new Array("Вася", "Иван", "Марья", 12, 5);
x = a.splice(1,3,"Петр", "Кузьма", "Анна");
/* x — массив элементов: "Иван", "Марья", 12
  a — массив элементов: "Вася","Петр", Кузьма", "Анна",5 */

a = new Array("Вася", "Иван", "Марья", 12, 5);
x = a.splice(1,0,"Петр", "Кузьма", "Анна", "Федор", "Ханс")
/* x — пустой массив
  a — массив элементов: "Вася","Петр", "Кузьма","Анна", "Федор",
                        "Ханс" */
```

- `toLocaleString()`, `toString()` — преобразуют содержимое массива в символьную строку.

Алгоритм преобразования по методу `toLocaleString()` зависит от браузера. Для преобразования содержимого массива в строку я рекомендую метод `join()`.

16.7.5. Функции обработки числовых массивов

Во многих приложениях требуется получить статистические характеристики числовых данных, хранящихся в виде массива: сумму всех чисел, среднее, максимальное и минимальное значения. В листинге 16.26 приведены коды функций, вычисляющих эти величины.

Листинг 16.26. Примеры вычисления статистических характеристик

```
/* Функция, возвращающая сумму значений всех элементов непустого массива.
   Для вычисления среднего значения следует просто воспользоваться
   выражением: S(aN)/aN.length
*/
   function S(aN){
    var S=aN[0];
    for(var i=1; i< aN.length; i++){
     S += aN[i]
    }
    return S
   }

/* Функция, возвращающая минимальное значение среди элементов массива */
   function Nmin(aN){
    var Nmin=aN[0];
    for(var i=1; i < aN.length; i++){
```

```
     if (aN[i] < Nmin)
      Nmin=aN[i]
    }
    return Nmin
   }

/* Функция, возвращающая максимальное значение среди элементов массива */
   function Nmax(aN){
    var Nmax=aN[0];
    for(var i=1; i < aN.length; i++){
     if (aN[i] > Nmax)
      Nmax=aN[i]
    }
    return Nmax
   }
```

Мы можем создать одну функцию, которая вычисляет все перечисленные статистические характеристики и возвращает их как значения массива (листинг 16.27).

Листинг 16.27. Функция для вычисления статистических характеристик

```
function statistic(aN){
 if (aN == 0 || aN == null || aN == "") return new Array(0,0,0,0);
 var S=aN[0];
 var Nmin=aN[0];
 var Nmax=aN[0];
 for(var i=1; i<=aN.length-1; i++){
  S += aN[i];
  if (aN[i] < Nmin) Nmin=aN[i];
  if (aN[i] > Nmax) Nmax=aN[i];
 }
 return new Array(S, S/aN.length, Nmin, Nmax)
}
```

В начале кода функции `statistic()` мы проверяем, не является ли параметр пустым. Если это так, то все статистические характеристики считаются равными нулю.

16.8. Числа

При разработке Web-страниц математические операции над числами обычно используются для изменения координат элементов страницы (свойства `top`,

`left`, `width`, `height` каскадной таблицы стилей), а также для вычисления статистических характеристик данных, содержащихся, например, в некоторой таблице. Так или иначе, в этих задачах необходимо иметь дело с числами. О числах уже говорилось в *разд. 16.2*, посвященном типам данных. Теперь рассмотрим их более подробно.

16.8.1. Числа целые и с плавающей точкой

В JavaScript числа могут быть только двух типов: целые и с плавающей точкой. Целые числа не имеют дробной части и не содержат разделительной точки. Числа с плавающей точкой имеют целую и дробную части, разделенные точкой.

Операции с целыми числами процессор компьютера выполняет значительно быстрее, чем операции с числами, имеющими точку. Это обстоятельство имеет смысл учитывать, когда расчетов много. Например, индексы, длины строк являются целочисленными. Число π, многие числа, полученные с помощью оператора деления, денежные суммы и т. п. — числа с плавающей точкой.

В JavaScript можно выполнять операции с числами различных типов, что очень удобно. Однако при этом следует знать, какого числового типа будет результат. Если результат операции является числом с дробной частью, то он представляется как число с плавающей точкой. Если результат оказался без дробной части, то он приводится к целочисленному типу, а не представляется числом, у которого в дробной части одни нули.

Примеры

```
2 + 3     // 5 — целое число
2 + 3.6   // 5.6 — число с плавающей точкой
2.4 + 3.6 // 6 — целое число
6.00      // число с плавающей точкой
```

Числа можно представлять и в так называемой экспоненциальной форме, т. е. в формате: *числоlе число2* или *числоlЕ число2*. Такая запись числа означает $число1 \times 10^{число2}$.

Примеры

```
1e5     // 100000
2e6     // 2000000
1.5e3   // 1500
+1.5e3  // 1500
-1.5e3  // -1500
```

```
3e-4    // 0.0003
```

Числа в JavaScript можно представлять в различных системах счисления, т. е. в системах с различными основаниями: 10 (десятичной), 16 (шестнадцатеричной) и 8 (восьмеричной). К десятичной форме представления чисел мы привыкли, однако следует помнить, что числа в этой форме не должны начинаться с нуля, потому что так записываются числа в восьмеричной системе.

Запись числа в шестнадцатеричной форме начинается с префикса `0x` (или `0X`), где первый символ нуль, а не буква "О", затем следуют символы шестнадцатеричных цифр: `0`, `1`, `2`, ..., `9`, `a`, `b`, `c`, `d`, `e`, `f` (буквы могут быть в любом регистре). Например, шестнадцатеричное число `0x4af` в десятичном представлении есть `1199`.

Запись числа в восьмеричной форме начинается с нуля, за которым следуют цифры от `0` до `7`. Например, `027` (в десятичном представлении — `23`).

В арифметических выражениях числа могут быть представлены в любой из перечисленных систем счисления, однако результат всегда приводится к десятичной форме.

Рассмотрим пример функции преобразования из десятичной формы в шестнадцатеричную (листинг 16.28). Функция `to16()` здесь принимает в качестве параметра десятичное число и преобразует его в строку, содержащую это же число, но в шестнадцатеричной форме. При этом мы ограничиваемся числами, не превышающими 255 (тогда для представления числа потребуется не более двух шестнадцатеричных цифр).

Листинг 16.28. Функция пребразования десятичного числа в шестнадцатеричное (вариант 1)

```
function to16(n10) {
 var hchars="0123456789abcdef";   /* строка, содержащая все
                                     шестнадцатеричные цифры */
 if (n10 > 255) return null;
 var i=n10%16;
 var j=(n10 — i)/16;
 rezult="0x";
 rezult += hchars.charAt(j);
 rezult += hchars.charAt(i);
 return rezult
}

to16(250);          // "0xfa"
to16(30);           // "0x1e"
```

```
to16(30)+10;       // "0x1e10" — склейка, а не сложение
parseInt(to16(30)) + 10;  // 40
```

Функцию `to16()` можно также создать и на основе массивов (листинг 16.29).

Листинг 16.29 Функция преобразования десятичного числа в шестнадцатеричное (вариант 2)

```
function to16(n10) {
 var hchars=new Array("0", "1", "2", "3", "4", "5", "6", "7", "8",
"9", "a", "b", "c", "d", "e", "f"};
 if (n10 > 255) return null;
 var i=n10%16;
 var j=(n10 — i)/16;
 rezult="0x";
 rezult += hchars[j];
 rezult += hchars[i];
 return rezult
}
```

Теперь рассмотрим пример функции преобразования из десятичной формы в двоичную (листинг 16.30). Здесь функция `to2()` принимает в качестве параметра десятичное число и преобразует его в строку, содержащую это же число, но в двоичной форме. Результирующая строка состоит из нулей и единиц. Здесь реализован рекурсивный вызов функции `to2()` самой себя. Обратите внимание, что указание ключевого слова `var` в данном случае принципиально важно из-за рекурсии, иначе функция будет работать неправильно.

Листинг 16.30. Функция преобразования десятичного числа в двоичное

```
function to2(n10) {
 if (n10 < 2)return "" + n10; // чтобы результат был строковым
 var i=n10%2;
 var j=(n10-i)/2;
 return to2(j)+i
}
```

Для преобразования строк, содержащих числа, в данные числового типа служат встроенные функции `parseInt()` и `parseFloat()` соответственно для представления в целочисленном виде и в виде числа с плавающей точкой. Их мы уже рассматривали ранее (см. *разд. 16.2.4*). Здесь следует отметить, что параметрами таких функций могут быть строки, содержащие числа не только в десятичной, но и в шестнадцатеричной, и в восьмеричной формах. Для указания основания системы счисления служит второй параметр этих функций.

Примеры

```
parseInt("ff",16)   // значение равно 255
parseInt("ff")      // значение равно NaN
parseInt("010")     // значение равно 8
parseInt("010", 8)  // значение равно 8
parseInt("010", 10) // значение равно 10
parseInt("010", 2)  // значение равно 2
parseInt("010", 16) // значение равно 17
```

Для преобразования числа в строку, содержащую это число, достаточно использовать выражение сложения пустой строки с числом.

Примеры

```
"" + 25.78 // значение равно "25.78"
"" + 2.5e3 // значение равно "2500"
```

16.8.2. Объект *Number*

Обычно числа создают с помощью переменных и оператора присваивания. Однако с каждым числом ассоциируется экземпляр встроенного объекта `Number`, обладающий некоторыми полезными свойствами и методами, которые могут пригодиться.

Экземпляр объекта `Number` можно создать явно:

имя_переменной = new Number(*число*)

Доступ к свойствам и методам объекта `Number` обеспечивается такими выражениями:

число.свойство

Number.свойство

число.метод(*параметры*)

Number.метод(*параметры*)

Здесь экземпляр *Number* — объект `Number`. Например, следующий код представляет число 520 в экспоненциальной форме:

```
var value=new Number(520);
 var x=value.toExponential(2)
```

То же самое можно сделать и таким способом:

```
var value=520;
var x=value.toExponential(2)
```

Далее мы рассмотрим свойства и методы объекта `Number`.

- `MAX_VALUE` — константа, значение которой равно наибольшему допустимому в JavaScript значению числа (`1.7976931348623157e+308`);
- `MIN_VALUE` — константа, значение которой равно наименьшему допустимому в JavaScript значению числа (`5e-324`);
- `NEGATIVE_INFINITY` — число, меньшее, чем `Number.MIN_VALUE`;
- `POSITIVE_INFINITY` — число, большее, чем `Number.MAX_VALUE`;
- `NaN` — константа, имеющая значение `NaN`, посредством которой JavaScript сообщает, что данные (параметр, возвращаемое значение) не является числом (Not a Number);
- `prototype` — свойство (прототип), играющее такую же роль, что и в случае других объектов (`String, Array` и т. д.).

  Объект `Number` имеет несколько методов, из которых мы рассмотрим только четыре, предназначенные для представления чисел в виде строки в том или ином формате.

- `toExponential(`*`количество`*`)` — представляет число в экспоненциальной форме. Синтаксис:

  `число``.toExponential(`*`количество`*`)`

  Возвращает строку.

  Параметр представляет собой целое число, определяющее, сколько цифр после точки следует указывать.

  Примеры

```
x = 456;
x.toExponential(3);  // 4.560e+2
x.toExponential(2);  // 4.56e+2
x.toExponential(1);  // 4.6e+2
x.toExponential(0);  // 5e+2
```

- `toFixed(`*`количество`*`)` — представляет число в форме с фиксированным количеством цифр после точки. Синтаксис:

  `число``.toFixed(`*`количество`*`)`

  Возвращает строку.

  Параметр — целое число, определяющее, сколько цифр после точки следует указывать.

  Примеры

```
x = 25.65;
x.toFixed(3);  // 25.650
x.toFixed(2);  // 25.65
x.toFixed(1);  // 25.7
x.toFixed(0);  // 26
```

- `toPrecision(`*точность*`)` — представляет число с заданным общим количеством значащих цифр. Синтаксис:

  число`.toPrecision(`*точность*`)`

  Возвращает строку.

  Параметр — целое число, определяющее, сколько всего цифр, до и после точки, следует указывать.

Примеры

```
x = 135.45;
x.toPrecision(6); // 135.450
x.toPrecision(5); // 135.45
x.toPrecision(4); // 135.5
x.toPrecision(3); // 135
x.toPrecision(2); // 1.4e2
x.toPrecision(1); // 1e2
x.toPrecision(0); // Сообщение об ошибке
```

- `toString([`*основание*`])` — возвращает строковое представление числа в системе счисления с указанным основанием. Синтаксис:

  число`.toString([`*основание*`])`

  Если параметр не указан, имеется в виду десятичная система счисления.

  Вы можете указать 2 для двоичной системы, или 16 — для шестнадцатеричной. Заметим, что этот метод имеют все объекты.

Примеры

```
x = 127.18;
x.toString();    // "127.18"
x.toString(10);  // "127.18"
x.toString(16);  // "7f.2e147ae147b"
x.toString(8);   // "177.134121727024366"

x = 5;
x.toString(2);   // "101"
```

16.8.3. Объект *Math*

Объект `Math` предназначен для хранения некоторых математических констант (например, числа π) и выполнения преобразований чисел с помощью типичных математических функций. Доступ к свойствам и методам объекта `Math` обеспечивается следующими выражениями:

```
Math.свойство;
Math.метод(параметры);
```

Значения свойств объекта `Math` — математические константы (табл. 16.13).

Таблица 16.13. *Константы объекта* `Math`

Константа	Значение
`E`	Постоянная Эйлера 2,71828…
`LN10`	Натуральный логарифм числа 10
`LN2`	Натуральный логарифм числа 2
`LOG10E`	Десятичный логарифм экспоненты (числа e)
`LOG2E`	Двоичный логарифм экспоненты
`PI`	Число π
`SQRT1_2`	Квадратный корень из 1/2
`SQRT`	Квадратный корень из 2

Например, для вычисления длины окружности при известном радиусе требуется число π, которое можно взять как свойство объекта `Math`:

```
var R=10            // радиус окружности
circus=2*R*Math.PI  // длина окружности
```

Перечислим методы объекта `Math`.

- `abs(число)` — модуль (абсолютное значение) числа;
- `acos(число)` — арккосинус числа;
- `asin(число)` — арксинус числа;
- `atan(число)` — арктангенс числа;
- `atan2(x, y)` — угол в полярных координатах;
- `ceil(число)` — округление числа вверх до ближайшего целого;
- `cos(число)` — косинус числа;
- `exp(число)` — число `e` в степени число;

- `floor(`*число*`)` — округление числа вниз до ближайшего целого;
- `log(`*число*`)` — натуральный логарифм числа;
- `max(`*число1*`, `*число2*`)` — большее из чисел *число1*, *число2*;
- `min(`*число1*`,`*число2*`)` — меньшее из чисел *число1*, *число2*;
- `pow(`*число1*`, `*число2*`)` — *число1* в степени *число2*;
- `random()` — случайное число между 0 и 1;
- `round(`*число*`)` — округление числа до ближайшего целого;
- `sin(`*число*`)` — синус числа;
- `sqrt(`*число*`)` — квадратный корень из числа;
- `tan(`*число*`)` — тангенс числа.

Приведем конкретные примеры.

- Метод `random()` возвращает случайное число, лежащее в интервале от 0 до 1:
  - получить случайное число в пределах от 0 до `Nmax` можно, написав выражение
    ```
    x = Nmax*Math.random()
    ```
  - получить случайное число в интервале от `Nmin` до `Nmax` можно из элементарного отношения пропорций:
    ```
    x = Nmin + (Nmax − Nmin)*Math.random()
    ```
  - создать функцию для вычисления случайного числа в заданном интервале можно так:
    ```
    function rand(a, b) {
     return a + (b - a)*Math.random()
    }
    ```
    Эта функция может потребоваться, например, для внесения некоторой непредсказуемости (нерегулярности) перемещения элементов на Web-странице, выбора цветов для мигающей надписи и т. п.
- Вычислить значение тригонометрической функции `sin(x)`, у которой аргумент `x` выражен в градусах, можно так:
  ```
  Math.sin(Math.PI*x/180)
  ```

16.9. Дата и время

Во многих приложениях приходится отображать дату и время, подсчитывать количество дней, оставшихся до заданной даты и т. п. Некоторые программы

даже управляются посредством значений дат и времени. В основе всех операций, связанных с датами и временем, лежат текущие системные дата и время, установленные на вашем компьютере.

Со временем дела обстоят не так просто, как кажется на первый взгляд, поскольку существуют временны́е зоны (часовые пояса), а также сезонные поправки. Чтобы иметь возможность координировать деятельность во времени организаций и физических лиц в различных точках нашей планеты, была введена единая система отсчета, связанная с меридианом, проходящим через астрономическую обсерваторию в городе Гринвич в Великобритании. Эту временную зону называют средним временем по Гринвичу (Greenwich Mean Time, GMT). Кроме аббревиатуры GMT используется еще одна — UTC (Coordinated Universal Time, всеобщее скоординированное время).

Если системные часы вашего компьютера установлены правильно, то время отсчитывается в системе GMT. Однако в Панели управления системы Windows обычно устанавливается локальное время, соответствующее конкретному часовому поясу. Именно оно фиксируется при создании и изменении файлов на вашем компьютере. Вместе с тем операционная система знает разницу между локальным временем и GMT. При перемещении компьютера из одного часового пояса в другой, необходимо изменить установки именно часового пояса, а не текущего системного времени (показания системных часов). Дата и время, генерируемые в сценариях, сохраняются в памяти в системе GMT, но пользователю выводятся, как правило, в локальном виде.

В программе на JavaScript дата и время не являются литералами. Иначе говоря, нельзя написать, например, выражение `xdate = 10.03.2008`, чтобы получить значение даты, с которым в дальнейшем можно производить некие операции. Значения даты и времени создаются как экземпляры специального объекта `Date`. При этом объект сам будет "знать", что не бывает 31 июня и 30 февраля, а в високосных годах должно быть 366 дней.

16.9.1. Создание объекта *Date*

Итак, для работы с датами и временем необходимо, прежде всего, создать экземпляр встроенного объекта `Date`:

```
имяОбъектаДаты = new Date(параметры)
```

Здесь `имяОбъектаДаты` — переменная. Параметры могут отсутствовать.

Для манипуляций с объектом `Date` применяется множество методов. При этом используется такой синтаксис:

```
имяОбъектаДаты.метод()
```

Если, например, объекту `Date` требуется присвоить новое значение, то для этого применяется соответствующий метод:

имяОбъектаДаты.метод(*новое_значение*)

Пример изменения значения года текущей системной даты представлен в листинге 16.31.

Листинг 16.31. Изменение года текущей системной даты

```
xdate = new Date();       /* создание объекта, содержащего текущую дату
                             и время */
Year = xdate.getYear();   /* в переменной Year содержится
                             значение текущего года */
Year = Year + 3;          /* в переменной Year содержится значение,
                             большее, чем текущий год на 3 */
xdate.setYear(Year);      /* в объекте устанавливается новое
                             значение года */
```

При создании объекта даты с помощью выражения `new Date()` можно указать в качестве параметров, какую дату и время следует установить в этом объекте. Для этого существуют пять способов:

```
new Date("Месяц дд, гггг чч:мм:сс")
new Date("Месяц дд, гггг")
new Date(гг, мм, дд, чч, мм, сс)
new Date(гг, мм, дд)
new Date(милисекунды)
```

В первых двух вариантах параметры задаются в виде строки, в которой указаны компоненты даты и времени. Буквенные обозначения определяют шаблон параметров. Обратите внимание на разделители — запятые и двоеточия. Время указывать не обязательно. Если компоненты времени опущены, то устанавливается значение 0 (полночь). Компоненты даты обязательно должны быть указаны. Месяц задается своим полным английским названием (аббревиатуры не допускаются). Остальные компоненты указывают в виде чисел. Если число меньше 10, то можно писать одну цифру, не записывая ведущий нуль (например, 3:05:32).

В третьем и четвертом способах компоненты даты и времени представляют целыми числами, разделенными запятыми.

В последнем варианте задают целое число, которое представляет собой интервал в миллисекундах, начиная с 1 января 1970 года (с момента 00:00:00). Количество миллисекунд, отсчитанное от указанной стартовой даты, позволяет вычислить все компоненты и даты, и времени.

16.9.2. Методы объекта *Date*

Все многочисленные методы объекта `Date` (табл. 16.14) можно разделить на две категории: получения значений (их названия имеют префикс `get`) и установки новых значений (их названия имеют префикс `set`). В каждой категории выделяют две группы методов — для локального формата и формата UTC. Методы позволяют работать с отдельными компонентами даты и времени (годом, месяцем, числом, днем недели, часами, минутами, секундами и миллисекундами).

Таблица 16.14. *Методы объекта* `Date`

Метод	Диапазон значений	Описание
`getFullYear()`	1970 и далее	Год
`getYear()`	70 и далее	Год
`getMonth()`	0—11	Месяц (январь соответствует нулю)
`getDate()`	1—31	Число
`getDay()`	0—6	День недели (воскресенье соответствует нулю)
`getHours()`	0—23	Часы в 24-часовом формате
`getMinutes()`	0—59	Минуты
`getSeconds()`	0—59	Секунды
`getTime()`	0 и далее	Миллисекунды с 1.1.70 00:00:00 GMT
`getMilliseconds()`	0 и далее	Миллисекунды с 1.1.70 00:00:00 GMT
`getUTCFullYear()`	1970 и далее	Год UTC
`getUTCMonth()`	0—11	Месяц UTC (январь соответствует нулю)
`getUTCDate()`	1—31	Число UTC
`getUTCDay()`	0—6	День недели UTC (воскресенье соответствует нулю)
`getUTCHours()`	0—23	Часы UTC в 24-часовом формате
`getUTCMinutes()`	0—59	Минуты UTC

`getUTCSeconds()`	0—59	Секунды UTC
`getUTCMilliseconds()`	0 и далее	Миллисекунды UTC с 1.1.70 00:00:00 GMT
`setYear(`*знач*`)`	1970 и далее	Установка года (четырехзначного)
`SetFullYear(`*знач*`)`	1970 и далее	Установка года
`setMonth(`*знач*`)`	0—11	Установка месяца (январь соответствует нулю)
`setDate(`*знач*`)`	1—31	Установка числа
`setDay(`*знач*`)`	0—6	Установка дня недели (воскресенье соответствует нулю)
`setHours(`*знач*`)`	0—23	Установка часов в 24-часовом формате
`setMinutes(`*знач*`)`	0—59	Установка минут
`setSeconds(`*знач*`)`	0—59	Установка секунд
`setMilliseconds(`*знач*`)`	0 и далее	Установка миллисекунд с 1.1.70 00:00:00 GMT
`setTime(`*знач*`)`	0 и далее	Установка миллисекунд с 1.1.70 00:00:00 GMT
`setUTCFullYear(`*знач*`)`	1970 и далее	Установка года UTC
`setUTCMonth(`*знач*`)`	0–11	Установка месяца UTC (январь соответствует нулю)
`setUTCDate(`*знач*`)`	1—31	Установка числа UTC
`setUTCDay(`*знач*`)`	0—6	Установка дня недели UTC (воскресенье соответствует нулю)
`setUTCHours(`*знач*`)`	0—23	Установка часов UTC в 24-часовом формате
`setUTCMinutes(`*знач*`)`	0—59	Установка минут UTC
`setUTCSeconds(`*знач*`)`	0—59	Установка секунд UTC
`setUTCMilliseconds(`*знач*`)`	0 и далее	Установка миллисекунд UTC с 1.1.70 00:00:00 GMT
`getTimezoneOffset()`	0 и далее	Разница в минутах по отношению к GMT/UTC
`toDateString()`	–	Строка с датой (без времени) в формате браузера (IE5.5)
`toGMTString()`	–	Строка с датой и временем

		в глобальном формате
`toLocaleDateString()`	–	Строка с датой без времени в локализованном формате системы (NN6, IE5.5)
`toLocaleString()`	–	Строка с датой и временем в локализованном формате системы
`toLocaleTimeString()`	–	Строка со временем без даты и в локализованном формате системы (NN6, IE5.5)
`toString()`	–	Строка с датой и временем в формате браузера
`toTimeString()`	–	Строка со временем без даты в формате браузера (IE5.5)
`toUTCString()`	–	Строка с датой и временем в глобальном формате
`Date.parse("дата")`	–	Преобразование строки с датой в число миллисекунд
`Date.UTC(`*`знач`*`)`	–	Преобразование строки с датой GMT в число

Вычислять разность двух дат или создавать счетчик времени, оставшегося до некоторого заданного срока, можно с помощью методов как для локального формата, так и UTC. Не следует применять выражения, сочетающие различные форматы времени, поскольку результаты могут оказаться неправильными. Формат UTC обычно применяется в расчетах, учитывающих часовой пояс.

> ***Примечание***
>
> Следует также принять во внимание, что нумерация месяцев, дней недели, часов, минут и секунд начинается с нуля. Для компонентов времени это естественно. Однако при такой нумерации декабрь оказывается одиннадцатым месяцем в году, а не двенадцатым, как это принято повсеместно. Воскресенье (Sunday) — нулевой день недели, а не седьмой.

Значение года XX века представляется в двухзначном формате как разность между этим годом и 1900. Например, 1996 год представляется как 96. Годы до 1900 и после 1999 обозначаются в четырехзначном формате. Например, 2010 год нужно так и писать — 2010, поскольку 10 это 1910 год. Метод `getFullYear()` возвращает четырехзначное значение года.

Допустим, что сейчас 2010 год.

Примеры

```
today = new Date();      // текущие дата и время
Year = today.getYear();   // 2010
today = new Date(98,11,6); /* объект даты today содержит
                              информацию о дате 6 ноября 1998 года,
                              00:00:00 */
Year = today.getYear();   // 98
today.getFullYear();   // 1998
```

Любой компонент даты изменяется с помощью соответствующего метода, название которого начинается с приставки `set`. При этом значения других компонентов пересчитываются автоматически.

В следующем примере создается объект `Date`, содержащий некоторую конкретную дату. Затем мы устанавливаем новое значение года. При этом в объекте даты изменяется и число.

Пример

```
mydate = new Date(1952, 10,6);  /* "Thu Nov 6 00:00:00 UTC+0300 1952" */
myday = mydate.getDay();        // 6
mydate.setYear(2010);           // установка 2010 года
myday = mydate.getDay()         // 6
```

При попытке отобразить значение объекта `Date` оно автоматически преобразуется в строку методом `toString()`. Формат этой строки зависит от операционной системы и браузера. Например, для Windows XP и Internet Explorer строка, содержащая информацию объекта даты, имеет следующий вид:

```
Sat Jan 16 17:52:45 UTC+0300 2010
```

т. е. суббота, январь, 16, 17 часов 52 минуты 45 секунд Всеобщего времени, скорректированного на 3 часа, 2010 год.

Firefox ту же информацию отобразит в немного отличном виде:

```
Sat Jan 16 2010 17:52:45 GMT+0300
```

Если коррекция времени с учетом часового пояса не требуется, то для строкового представления даты и времени можно воспользоваться методом `toLocaleString()`:

```
mydate = new Date();
mydate.toLocaleString();    // "16 января 2010 г. 17:52:45"
```

Заметим, что формат представления даты и времени, обеспечиваемый методами `toLocalDateString()` и `toLocalTimeString()`, зависит от настроек операционной системы и браузера. Если вы с помощью этих методов выведете

дату и время на Web-страницу, то они будут выглядеть так, как пользователь привык их видеть на своем компьютере. Так, в браузерах Internet Explorer и Firefox метод `toLocalDateString()` вернет строку "16 января 2010 г. 17:52:45", в Opera — "16.01.2010 17:52:45", а в Safari и Chrome — "Satuday, January 16,.2010 17:52:45" и "Sat Jan 16 2010 17:52:45 GMT+0300 (Russia Standard Time)" соответственно.

Для представления в виде строк отдельно даты и времени имеются еще два метода:

```
mydate = new Date();
mydate.toLocaleDateString();  // "16 января 2010 г."
mydate.toLocaleTimeString();  // "17:52:45"
```

С датой иногда приходится выполнять различные вычисления, такие как определение даты через заданное количество дней от текущей или число дней между двумя датами. Вот здесь как раз и требуется предварительный подсчет миллисекунд, содержащихся в минуте, часе, сутках и т. д.

Пример

```
// Определение даты, которая наступит через неделю относительно текущей
   week =1000*60*60*24*7       /* количество миллисекунд в неделе —
                                  604800000 */
   mydate=new Date();          // объект Date с текущей датой
   mydate_ms=mydate.getTime(); /* текущая дата, представленная
                                  количеством миллисекунд
                                  от 1.01.1970 00:00:00 */
   mydate_ms += week;          // maydate_ms=mydate_ms + week
   mydate.setTime(mydate_ms);  /* установка новой даты в объекте
                                  mydate */
   newdate=mydate.toLocaleString() // новая дата в виде строки
```

Теперь определим количество дней между двумя датами, например, 16 января 2010 г. и 5 марта 2010 г. Для этого создадим сначала два соответствующих объекта даты:

```
date1=new Date(2010,1,16)
date2=new Date(2010,3,5)
```

Переменные `date1` и `date2` хранят длинные строки, содержащие даты (например, Sat `Jan 16 00:00:00 UTC+0300 2010`). Чтобы перевести их в миллисекунды, воспользуемся методом `parse()` объекта `Date`. Затем вычислим разность `Date.parse(date2)` и `Date.parse(date1)` и разделим ее на число миллисекунд в одних сутках:

```
days=(Date.parse(date2) - Date.parse(date1))/1000/60/60/24
```

```
    // результат: 48
```

Часто приходится иметь дело с датами, представленными в виде строк в формате "*дд.мм.гггг*", причем месяцы нумеруются с 1 до 12, а не с 0 до 11. В этом случае, чтобы воспользоваться методами объекта `Date`, необходимо предварительно выполнить соответствующие преобразования.

Допустим, исходная дата `strdate` представляется в виде строки в формате "*дд.мм.гггг*", а объект `Date` создан в формате "*гг, мм, дд*". Тогда необходимые преобразования исходных данных выглядят следующим образом:

```
astrdate=strdate.split(".");  /* массив подстрок, полученных из строки
                                 strdate с использованием точки
                                 как разделителя */
/* создаем объект Date */
mydate=new Date(astrdate[2], astrdate[1]-1, astrdate[0])
```

Развивая идею, рассмотренную в предыдущем примере, мы могли бы создать функцию, принимающую в качестве параметра строку, содержащую дату и время в привычном для нас формате "*дд.мм.гггг чч:мм:сс*" и возвращающую объект даты (листинг 16.32). Будем считать, что в строке параметра нашей функции дата отделена от времени одним пробелом, однако допускается, что информации о времени вообще может не быть. Далее, если время присутствует, секунды не обязательно должны указываться. Если параметр является пустой строкой или вообще отсутствует, то функция возвращает объект с текущей датой.

Листинг 16.32. Функция преобразования строки с датой в объект даты

```
function mydate(strdate){
 if (strdate == "" || strdate == null) return new Date()
 var astrdate=strdate.split(" ");
 if (astrdate.length == 1) astrdate[1] = "00:00:00";
 astrdate[0] = astrdate[0].split(".");
 astrdate[1] = astrdate[1].split(":");
 if (astrdate[1].length == 1)
  astrdate[1] = new Array(astrdate[1][0], "00","00")
 else {
  if (astrdate[1].length == 2)
   astrdate[1] = new Array(astrdate[1][0], astrdate[1][1] , "00")
 }
 return new Date(
    astrdate[0][2], astrdate[0][1]-1, astrdate[0][0],
    astrdate[1][0], astrdate[1][1], astrdate[1][2])
}
```

С использованием функции `mydate()` вычислить интервал (в днях) между двумя заданными датами можно следующим образом:

```
date1=mydate("16.1.2010");
date2=mydate("5.3.2010");
days=(Date.parse(date2) — Date.parse(date1))/1000/60/60/24;
```

16.10. Объекты

Хотя язык JavaScript объектно-ориентированный, данная технология программирования на этом языке редко применяется на практике. Во-первых, потому, что в JavaScript есть много обычных и более простых средств, вполне достаточных для решения типичных задач создания Web-приложений. Во-вторых, сценарии для Web-сайтов, как правило, настолько малы по объему, что трудно достичь явных преимуществ за счет применения объектно-ориентированной технологии. Вместе с тем, сценарии на JavaScript работают в основном с объектами — ассоциированными с типами данных (`String`, `Number`, `Array`, `Date` и т. п.), а также объектами документа и браузера (`document`, `window`). Поэтому знакомство с устройством объектов окажется полезным, даже если вы не будете создавать собственные объекты.

16.10.1. Создание объекта

Рассмотренные ранее встроенные объекты (`String`, `Number`, `Array` и т. п.), происходят от одного и того же корневого объекта `Object`. Вы можете создавать свои собственные (пользовательские) объекты, которые также будут базироваться на объекте `Object`.

Создать объект можно различными способами:

- *имяОбъекта* = new Object();

  имяОбъекта.свойство = *значение*;

- *имяОбъекта* = {*свойство1*: *значение1*, *свойство2*: *значение2*,...};

- function *имя_конструктора*(*пар1*,..., *парN*){

      код

  }

  имяОбъекта = new *имя_конструктора*(*пар1*,..., *парN*);

Для обращения к свойствам и методам объекта используется следующий синтаксис:

имяОбъекта.свойство
имяОбъекта.метод(*параметры*)

Рассмотрим перечисленные способы создания объектов подробнее. Допустим, например, что нам требуется создать объект `Сотрудник`, который содержал бы сведения о сотруднике некоторой фирмы, такие как `Имя`, `Отдел` и `Телефон`. Эту задачу можно решить так:

```
Сотрудник = new Object();
Сотрудник.имя = "Джеймс Бонд";
Сотрудник.отдел = "9";
Сотрудник.телефон = "123-456";
```

На рис. 16.21 показан пример обращения к свойствам объекта `Сотрудник`.

Рис. 16.21. Пример обращения к свойствам объекта

Тот же объект можно создать и посредством так называемой литеральной формы записи:

```
Сотрудник = {
  имя:"Джеймс Бонд",
  отдел:"9",
  телефон:"123-456"
}
```

Рассмотренные способы задания объекта являются по существу определением экземпляра объекта `Object`. Если в фирме несколько сотрудников, то для каждого из них потребуется создать свой объект, что не всегда удобно. Хотелось бы создать один объект `Сотрудник`, задающий структуру данных о множестве сотрудников, а каждого из них представлять соответствующим экземпляром объекта `Сотрудник`. Такой подход обеспечивается посредством определения конструктора объекта.

Конструктор объекта — это функция, в теле которой задаются свойства и методы объекта — переменные и функции с префиксом `this`. Ключевое слово `this` (этот) представляет ссылку на текущий, т. е. определяемый конструктором объект. Конструктору, как и обычной функции, можно передать параметры. Листинг 16.33 содержит конструктор уже рассматривавшегося объекта `Сотрудник`.

Листинг 16.33. Пример конструктора объекта

```
function Сотрудник(имя, отдел, телефон) {
  this.имя = имя;
  this.отдел = отдел;
  this.телефон = телефон;
 }
```

Здесь в коде конструктора задаются свойства объекта, которым присваивают значения передаваемых ему параметров.

Экземпляры объекта создаются вызовом конструктора с ключевым словом `new`:

```
agent007 = new Сотрудник("Джеймс Бонд", 9, "123-456");
Shtirlitz = new Сотрудник("Максимов", 4, "765-4321");
```

Здесь переменные `agent007` и `Shtirlitz` — имена объектов, являющихся различными экземплярами одного и того же объекта `Сотрудник`.

Доступ к свойствам этого объекта производится обычным способом:

```
agent007.имя       // "Джеймс Бонд"
Shtirlitz.отдел    // 4
```

В рассмотренном примере конструктор принимает параметры. Если при вызове конструктора параметры не передавать, то свойства будут иметь значения `undefined`. Для свойств можно задать значения по умолчанию, принимаемые в том случае, если параметры в вызове конструктора не указаны или имеют следующие значения: пустая строка, `0`, `false`, `null`, `undefined`. Значения по умолчанию указывают с помощью символа `||`, как в листинге 16.34.

Листинг 16.34. Указание значений по умолчанию в конструкторе

```
function Сотрудник(имя, отдел, телефон) {
 this.имя = имя || "неизвестно";
 this.отдел = отдел || 0;
 this.телефон = телефон || "xxx-xxxx";
}
```

Теперь модифицируем код конструктора объекта `Сотрудник`, добавив к нему метод `resume()`, который возвращает строку со значениями всех свойств объекта (листинг 16.35). Метод создается с помощью анонимного определения функции (см. *разд. 16.5.3*).

Листинг 16.35. Пример создания метода в конструкторе

```
function Сотрудник(имя, отдел, телефон){
  this.имя = имя;
  this.отдел = отдел;
  this.телефон = телефон;
  this.resume = function() {
    return this.имя+", отдел "+this.отдел+", тел."+this.телефон
  }
}
```

Чтобы изменить значение свойства экземпляра объекта, достаточно воспользоваться выражением:

```
имяОбъекта.свойство = значение;
```

Например,

```
Shtirlitz.имя = "Штирлиц";
```

Чтобы добавить новое свойство к экземпляру объекта, достаточно воспользоваться аналогичным выражением:

```
имяОбъекта.новое_свойство = значение;
```

Например, выражение `agent007.зарплата=10000` добавляет свойство `зарплата` со значением 10000 только экземпляру `agent007` объекта `Сотрудник`, а экземпляр `Shtirlitz` этого свойства не имеет.

Кроме так называемого точечного синтаксиса обращения к свойствам объекта (*имяОбъекта.свойство*) существует синтаксис обращения к элементам ассоциативного массива: `имяОбъекта["свойство"]`. Например, `Shtirlitz["имя"] = "Максимов"`.

Добавить новое свойство сразу всем существующим экземплярам объекта можно через так называемый прототип. Каждый объект имеет свойство

prototype, являющееся ссылкой на базовый объект Object. Используя prototype, можно добавить в прототип конструктора данные и программный код, которые должны быть общими для всех экземпляров объекта.

Пусть, например, требуется добавить свойство пол со значением муж. в объект Сотрудник (а значит, и во все его экземпляры). Это можно сделать так:

```
Сотрудник.prototype.пол="муж.";
```

Аналогично добавляются и изменяются методы. Так, все объекты имеют метод toString(), который для пользовательского объекта возвращает малоинформативную строку "[object Object]". Если вы хотите, чтобы для объекта Сотрудник данный метод возвращал строку "[object Сотрудник]", то следует выполнить такой код:

```
Сотрудник.prototype.toString = function() {
                    return "[object Сотрудник]"};
```

На рис. 16.22 в качестве иллюстрации изложенного приведены код и результат его выполнения.

Два объекта можно связать таким образом, что один из них (дочерний) будет наследовать свойства другого (родительского). Наследование свойств объектов осуществляется через прототип. Допустим, мы хотим создать объект Суперсотрудник, отличающийся от рассмотренного ранее объекта Сотрудник только тем, что имеет дополнительное свойство оружие. Иначе говоря, Суперсотрудник должен унаследовать все свойства объекта Сотрудник, сохранив при этом свои индивидуальные свойства. Для этого достаточно создать дочерний объект Суперсотрудник, содержащий только специфические (собственные) свойства, а затем через прототип связать его с родительским объектом Сотрудник (листинг 16.36).

Рис. 16.22. Создание объекта и изменение его свойств через прототип

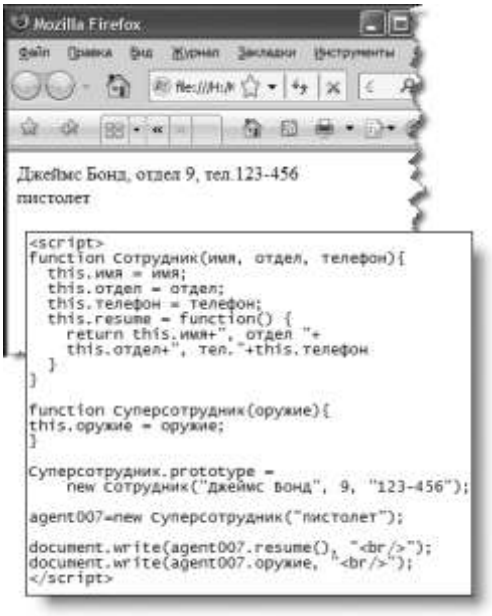

Рис. 16.23. Пример наследования свойств объекта через прототип

Листинг 16.36. Пример связи объектов через прототип

```
function Суперсотрудник(оружие){ /* содержит только специфические
                                    свойства */
  this.оружие=оружие;
}
/* Присоединяем свойства родительского объекта */
Суперсотрудник.prototype = new Сотрудник("Пронин", 13, "777-7777");
```

Аналогично через цепочку прототипов можно связать несколько объектов.

На рис. 16.23 приведен пример, показывающий, что экземпляр объекта `Суперсотрудник` не только обладает пистолетом, но и свойствами родительского объекта `Сотрудник`.

Объекты являются наборами данных и программного кода и поэтому занимают некоторый объем памяти. Если объект или его экземпляр перестал быть нужным, то его можно удалить. Для этого достаточно просто присвоить ему значение `null`: `agent007 = null; Суперсотрудник = null`.

16.10.2. Свойства и методы объекта *Object*

Как уже отмечалось ранее, `Object` — базовый объект, на основе которого строятся все остальные. Свойствами и методами объекта `Object` обладают также и другие объекты, такие как `Array`, `String`, `Number`, `Function` и т. п., а также пользовательские объекты. С некоторыми из них (например, `prototype` и `toString()`) мы уже встречались. Так, единственное свойство `prototype` подробно рассматривалось в разд. 16.10.1. Здесь будут перечислены методы объекта `Object`.

- `hasOwnProperty("свойство")` — возвращает логическое значение, указывающее, имеет ли экземпляр объекта собственное *`свойство`* (не унаследованное через `prototype`).
- `isPrototypeOf(объект)` — возвращает логическое значение, указывающее, находится ли *`объект`* в цепочке прототипов данного объекта.
- `toString()` — возвращает строку; по умолчанию — "[object Object]".
- `valueOf()` — возвращает примитивное значение, ассоциированное с объектом; часто это строка "[object Object]".

Рис. 16.24. Пример использования свойств объекта `Object`

На рис. 16.24 приведен пример использования указанных свойств. Так, объект `agent008` имеет собственное свойство `оружие`, но свойство `имя` он унаследовал от объекта `agent007`, а не приобрел как собственное при создании.

16.10.3. Объектные операторы

При работе с объектами иногда удобны специальные операторы:

- `width (`*`объект`*`) {`*`код`*`}` — позволяет выполнить блок кода применительно к указанному объекту.

  Пример

  ```
  width(document){
    write("<h1>Моя домашняя страница</h1>");
    write("Здесь приводятся примеры скриптов на JavaScript");
    write("<br/>");
    write("Они могут пригодиться вам.")
  }
  ```

  Без данного оператора нам пришлось бы писать `document.write()`. При использовании оператора `width` важно, чтобы упоминаемые в блоке кода свойства и методы относились к объекту, указанному в качестве параметра.

- `for(`*`переменная`* `in` *`объект`*`) {код}` — оператор циклического перебора свойств объекта. Переменная может иметь любое имя.

  На рис. 16.25 в качестве примера выводятся имена и значения объекта `agent007`, являющегося экземпляром объекта `Сотрудник`. Обратите внимание, что имя свойства читается в переменную `prop`, а значение данного свойства получается путем обращения к объекту как массиву: `agent007[prop]`.

  Чтобы просмотреть свойства объекта `document`, представляющего загруженный в браузер документ, достаточно выполнить следующий код:

  ```
  for (prop in document){
    document.write(prop,": " + document[prop],"<br/>")
  }
  ```

- `свойство in объект` — возвращает логическое значение, указывающее, обладает ли объект данным свойством.

  Пример

  ```
  function Сотрудник(имя, отдел, телефон){
   this.имя = имя;
  ```

```
 this.отдел = отдел;
 this.телефон = телефон;
 this.resume = function() {
    return this.имя+", отдел "+
           this.отдел+", тел."+this.телефон
 }
}

function Суперсотрудник(оружие){
  this.оружие=оружие
}

agent007=new Сотрудник("Штирлиц", 13, "123-456");
Суперсотрудник.prototype = agent007;
agent008 = new Суперсотрудник("пистолет");

alert("оружие" in agent008);     // true
alert("оружие" in agent007);     // false
```

- `delete объект.свойство` — позволяет удалить свойство объекта. Обычно служит для удаления элементов массива (см. разд. 16.7.1).

Пример

```
function Сотрудник(имя, отдел, телефон){
 this.имя = имя;
 this.отдел = отдел;
 this.телефон = телефон;
 this.resume = function() {
  return this.имя+", отдел "+
           this.отдел+", тел."+this.телефон
 }
}

agent007 = new Сотрудник("Штирлиц", 13, "123-456");
alert("имя" in agent007);   // true
delete agent007.имя;
alert("имя" in agent007);   // false
```

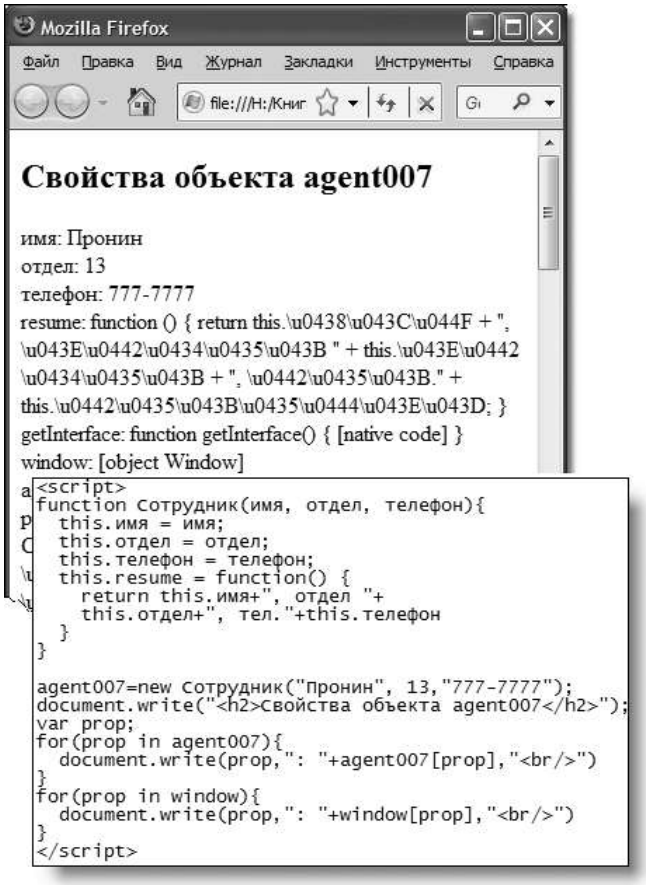

Рис. 16.25. Пример использования оператора `for(… in…)`

- *объект1* `instanceof` *объект2* — возвращает логическое значение, указывающее, принадлежит ли *объект1* объектной модели *объект2*.

Пример

```
function Сотрудник(имя, отдел, телефон){
 this.имя = имя;
 this.отдел = отдел;
 this.телефон = телефон;
 this.resume = function() {
  return this.имя+", отдел "+
    this.отдел+", тел."+this.телефон
 }
```

```
}

function Суперсотрудник(оружие){
this.оружие=оружие
}

agent007 = new Сотрудник("Штирлиц", 13, "123-456");
Суперсотрудник.prototype = agent007;
agent008 = new Суперсотрудник("пистолет");

alert(agent008 instanceof Object);         // true
alert(agent008 instanceof Сотрудник);      // true
alert(agent008 instanceof Суперсотрудник); // true

alert(agent007 instanceof Object);         // true
alert(agent007 instanceof Сотрудник);      // true
alert(agent007 instanceof Суперсотрудник); // false
```

16.10.4. JSON

JSON (JavaScript Object Notation) — представление объектов, основанное на синтаксисе JavaScript и литеральной форме записи объектов и массивов. Это текстовый формат представления данных, предложенный Д. Крокфордом для более компактного представления наборов данных, рассматриваемый как альтернатива XML и (X)HTML при обмене между сервером и клиентами.

Представление массивов и объектов в виде литералов уже рассматривалось в *разд. 16.7.1* и *16.10.1* соответственно. JSON допускает создание комбинированных литералов: массива объектов и, наоборот, объекта массивов. На рис. 16.26 показан пример, в котором создан массив из четырех объектов, представляющих информацию о сотрудниках, а в диалоговое окно выведены фамилия и должность третьего из них.

Те же данные можно представить и несколько иначе, а именно в виде объекта, свойства которого являются массивами (рис. 16.27).

Однако в формате JSON хранятся только терминальные данные, переменных и операторов присваивания быть не может. Так, на стороне сервера может быть сформирована символьная строка, содержащая описание объектов, массивов и/или их комбинацию. Чтобы клиент мог воспользоваться информацией в данной строке, ее необходимо преобразовать в данные соответствующего типа, т. е. в собственно объекты, массивы и/или их комбинацию. Для этой цели можно применить встроенную функцию `eval()` (см. *разд. 16.5.1*).

```
<script type="text/javascript">
var Сотрудники = [
  {
    "Фамилия":"Иванов",
    "Должность":"Директор",
    "Зарплата":100000
  },
  {
    "Фамилия":"Петрова",
    "Должность":"Секретарь",
    "Зарплата":50000
  },
  {
    "Фамилия":"Сидоров",
    "Должность":"Ведущий специалист",
    "Зарплата":10000
  },
  {
    "Фамилия":"Федоров",
    "Должность":"Специалист",
    "Зарплата":9000
  }
]
alert(
  Сотрудники[2].Фамилия+
  " - "+
  Сотрудники[2].Должность)
</script>
```

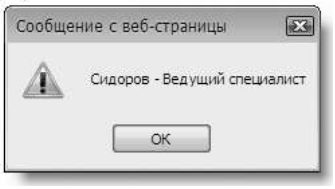

Рис. 16.26. Представление массива объектов в литеральной форме

```
<script type="text/javascript">
var Сотрудники = {
    "Фамилия":[
                "Иванов",
                "Петрова",
                "Сидоров",
                "Федоров"
               ],
    "Должность":[
                 "Директор",
                 "Секретарь",
                 "Ведущий специалист",
                 "Специалист"
                ],
    "Зарплата":[100000,50000,10000,9000]
    }

alert(
  Сотрудники.Фамилия[2]+
  " - "+
  Сотрудники.Должность[2])
</script>
```

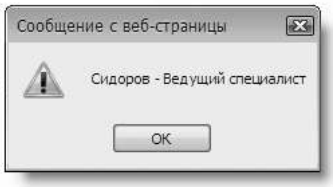

Рис. 16.27. Представление объекта массивов в литеральной форме

Однако функция `eval()` может выполнить любой JavaScript-код, что потенциально опасно для компьютера пользователя. Поэтому лучше использовать методы из специальной библиотеки, созданной Д. Крокфордом, исходный код которой можно получить по адресу **www.json.org/json2.htm**. На указанной странице находится JavaScript-код с обширным комментарием. Вы можете скопировать его в текстовый файл с произвольным именем (например,

json.js), удалив для сокращения объема комментарий. В данной библиотеке для преобразования JSON-строки в объекты служит метод `JSON.parse(`*строка*`)`, а для обратного преобразования — метод `JSON.stringify(`*объект*`)`. Оба метода имеют дополнительные параметры, значения которых даны в комментариях. Чтобы воспользоваться данными методами, следует включить файл с библиотекой в ваш (X)HTML-документ. Впрочем, для браузеров Internet Explorer 8.0, Firefox , Safari и Chrome это делать не обязательно.

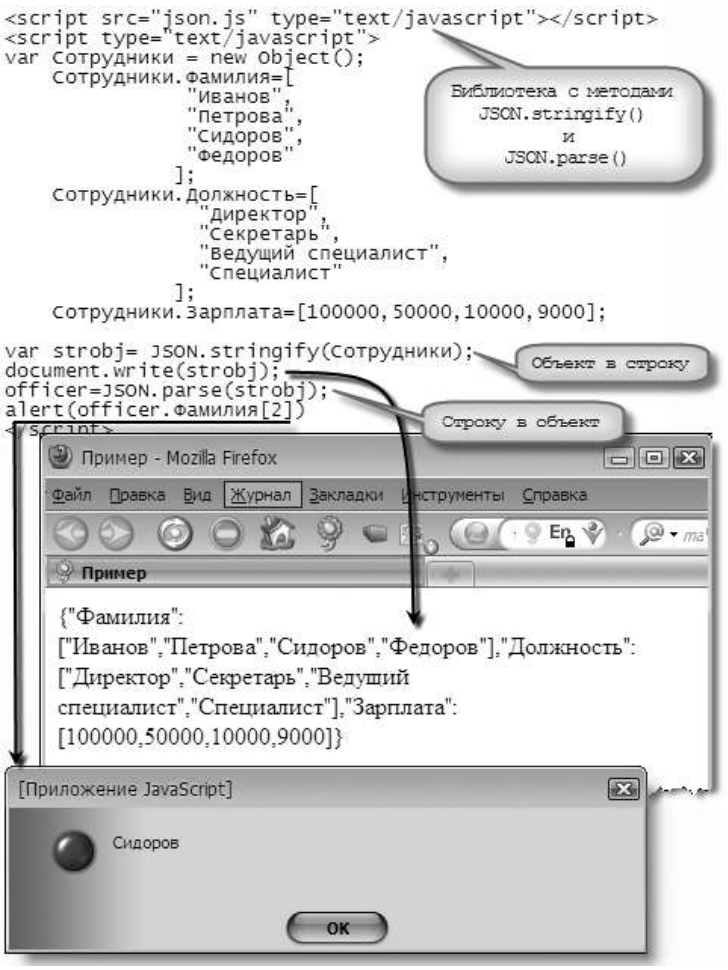

Рис. 16.28. Действие методов `JSON.stringify()` и `JSON.parse()`

На рис. 16.28 показан пример, в котором сначала объект `Сотрудники` определяется не посредством литеральной записи, а как экземпляр объекта `Object`.

Затем этот объект преобразуется методом `JSON.stringify()` в строку, которая содержит литеральную запись данного объекта и выводится в окно браузера. Далее методом `JSON.parse()` создается объект `officer` и с помощью метода `alert()` выводится фамилия третьего сотрудника.

Для работы с JSON на стороне сервера имеется специальное программное обеспечение на различных языках, таких как PHP, Perl, Pyton, Java и др. Подробности по JSON можно найти по адресу **www.json.org/json-ru.html**.

16.11. Операторы обработки исключительных ситуаций

При выполнении некоторых операций могут возникнуть ошибки, даже если они не связанны с нарушением синтаксиса выражений языка. Это так называемые ошибки времени выполнения сценария. Например, сценарий может обратиться к объекту, который не поддерживается данным типом или версией браузера, пользователь мог ввести в поле вместо числа последовательность букв и т. п. В результате возникает исключительная ситуация, которую можно, как говорят, перехватить и обработать. Однако следует подчеркнуть, что ошибки, обусловленные нарушением синтаксиса, нельзя обрабатывать как исключительные ситуации. Это дело интерпретатора JavaScript, т. е. браузера, который в лучшем случае выведет диагностическое сообщение.

Перехват и обработку исключительных ситуаций можно осуществить с помощью оператора, схема которого представлена в листинге 16.37.

Листинг 16.37. Перехват и обработка исключительных ситуаций

```
try{
  код, ошибки при выполнении которого перехватываются
} catch(e){
  код, выполняемый при перехвате ошибки
}
finally{
  код, выполняемый в любом случае
}
```

В блоке `try` (попытка) размещают ту часть программы, в которой могут возникнуть ошибки, которые вы желаете перехватить и каким-либо образом обработать. Собственно обработка ошибок выполняется в блоке `catch` (захват), которому в качестве параметра передаются сведения об ошибке (по умолчанию — объект ошибки). Здесь вы пишете программный код, который должен, по вашему мнению, осуществлять реакцию на возникшую исключительную

ситуацию. В простейшем варианте это может быть вывод сообщения об ошибке. Блок `finally` не обязателен и содержит код, который будет выполнен после кода блоков `try` или `catch`.

Теперь рассмотрим пример. Известно, что попытка выполнить выражение `alert(x)`, где `x` — переменная, которая не объявлена с помощью оператора `var` или которой не присвоено значение, вызовет ошибку. Кстати, такого рода ошибки трудно выявить путем простого наблюдения за выполнением сценария в браузере. Следующий сценарий (листинг 16.38) перехватит эту ошибку и в результате ее обработки выведет диалоговое окно с описанием ошибки.

Листинг 16.38. Пример сценария обработки ошибки

```
try{
  alert(x);
} catch(e){
  alert("Произошла ошибка:"+e.message);
}
```

Здесь `e.massage` — описание ошибки, возвращаемое объектом ошибки `Error`. Вместо `e` можно использовать любое имя переменной. В данной книге мы не будем рассматривать объект ошибки, а отметим только одно его важное свойство `message`, содержащее в качестве строкового значения описание ошибки. Это значение может быть тем или иным в зависимости от типа браузера. Вы можете программно проанализировать строку сообщения об ошибке, чтобы принять решение о способе ее обработки.

Блок `try` может включать в себя другие блоки `try` и `catch`. Если таких конструкций несколько, то они могут быть только вложенными друг в друга. Вложенные блоки `try` и `catch` обычно применяют для обработки нескольких видов ошибок.

Исключительную ситуацию можно вызвать принудительно, чтобы передать управление блоку обработки `catch`. Для этого существует оператор

`throw` *значение*

Указанное *значение* может быть любого типа, но рекомендуется использовать значение объекта ошибки или строковое сообщение об ошибке. Если это строковое значение, то оно будет передано в качестве параметра оператора `catch`. Листинг 16.39 иллюстрирует принцип применения операторов `try…catch` и `trow`.

Листинг 16.39. Пример принудительного вызова исключительной ситуации

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head><title>try</title></head>
<script type="text/javascript">
try{
  var x;
  x=prompt("Введите число","");
  if (isNaN(x)){ // если не число
     throw "Вы ввели не число" // перейти к блоку catch
 }
} catch(e){
     alert(e); // сообщение, переданное throw
 x=0;
}
alert("x="+x);
</script>
</html>
```

В рассматриваемом здесь примере, если пользователь введет в поле диалогового окна символы, не означающие в совокупности какое-нибудь число, то произойдет переход к блоку `catch` с передачей ему строки сообщения. Блок `catch` в этом случае выведет диалоговое окно с принятым сообщением и присвоит переменной `x` нулевое значение. Если же пользователь ввел число, то код в блоке `catch` не будет выполняться. Далее все идет своим чередом: выполняются выражения, расположенные за пределами оператора `try…catch`. В данном примере выводится диалоговое окно со значением переменной `x`. Разумеется, в своих проектах вы должны найти более разумное и, в конечном счете, эффективное применение конструкции `try…catch`.

В заключение замечу, что рассмотренная здесь в общих чертах языковая конструкция (т. е. оператор) `try…catch` довольно широко распространена в различных языках программирования, таких как PHP, C++, Pascal и др. Ее применяют в тех случаях, когда необходимо прежде проверить доступность какого-нибудь объекта, выбрать вариант доступа к нему (если это возможно в принципе), дождаться завершения соединения с удаленным компьютером или выполнить какую-нибудь многоэтапную операцию, успех которой зависит больше от конъюнктуры и времени, чем от соблюдения элементарного синтаксиса языковых выражений. Допустим, вы хотите получить данные от сервера. Но для их передачи в браузер требуется некоторое время, в течение

которого может произойти что угодно. Вы, очевидно, должны дождаться завершения передачи данных прежде, чем начать заниматься манипуляциями с ними. Обычно программные средства обмена данными предоставляют программисту сообщения об этапах передачи информации (например, объект загружен частично, загрузка завершена с ошибками, загрузка завершена без ошибок и т. п.). Ваша программа может контролировать или нет все этапы выполнения многошаговой операции. Если контроль отсутствует, то вероятность "зависания" программы высока, что не свидетельствует о вашем профессионализме. Ведь "зависание" программы — это самое худшее, что может с ней произойти. Именно предусмотрительность во всем — один из важнейших признаков высокой квалификации программиста. Здесь и приходит на помощь конструкция `try…catch`.