2

ГЛАВА

Базовые возможности JavaScript

Язык подготовки сценариев используется для применения, настройки и автоматизации средств существующей системы. В случае JavaScript такой системой обычно является Web-браузер и связанные с ним технологии HTML, CSS и XML. JavaScript сам по себе является относительно простым языком, и во многом его мощь обусловлена встроенными объектами и объектами документа, предоставленными браузером.

Базовые возможности JavaScript, о которых говорится в этой главе, — это правила синтаксиса, которым должны подчиняться сценарии и основные конструкции, используемые для хранения данных и управления потоком выполнения программы. Поняв основы языка, более утонченные его возможности можно осваивать независимо, не погружаясь в пучину бесчисленных деталей. Программистам, использующим C/C++ и Java, синтаксис JavaScript покажется знакомым, и они смогут достаточно быстро перейти к освоению более сложных возможностей этого языка.

Данная глава является вступительной; в ней представлен краткий обзор всех базовых возможностей JavaScript. Большинство тем данной главы будут изучены более подробно в последующих главах. Опытным программистам будет знакомо многое из материала этой главы, поэтому они могут только просмотреть этот материал.

Основные определения

Большие группы людей, имеющих общий интерес или цель, стараются достичь взаимопонимания с наименьшими усилиями: они создают жаргон. Попробовав работать с компьютером, вы заметите, что инженеры-программисты с особым чувством относятся к языку, который они используют для обмена идеями программирования. С помощью терминов, применяемых при обсуждении языков программирования и составляющих особый технический словарь, соответствующие конкретные идеи могут передаваться ясно и лаконично.

Здесь мы вводим определенную терминологию языка программирования, которая будет использоваться в нашей книге. В табл. 2.1 приводятся точные определения по-

нятий, которые зачастую имеют несколько неопределенный контекст. Эти термины будут использоваться во всех следующих главах.

Таблица 2.1. Терминология языков программирования

Термин	Определение	Примеры
Лексема (token)	Наименьшая неделимая лексическая единица языка. Непрерывная последовательность символов, значение которой может измениться вследствие добавления одного пробела	Все идентификаторы и ключевые слова, а также литералы типа `3.14` и `"Это строка символов"`

Литерал (literal)	Значение, содержащееся непосредственно в сценарии	`3.14`
		`"Это строка символов"`
		`[2, 4, 6]`

Идентификатор (identifier)	Имя переменной, объекта, функции или метки	`X`
		`myValue`
		`username`

Знак операции (operator)	Лексемы, представляющие встроенные операции языка, такие как присваивание, сложение или вычитание	`=`
		`+`
		`-`
		`*`

Выражение (expression)	Группа лексем, как правило, литералов и идентификаторов, в комбинации со знаками операций, для которой можно вычислить конкретное значение	`2.0`
		`"Это строка символов"`
		`(x + 2) * 4`

Оператор (statement)	Императивная команда. Операторы обычно приводят к изменению состояния выполняемого окружения (переменной, определения или потока). Программа представляет собой просто список операторов	`x = x + 2;`
		`return(true);`
		`if (x) { alert("Это x");}`
		`function myFunc()`
		`{`
		`  alert("Всем привет");`
		`}`

Ключевое слово (keyword)	Слово, являющееся частью языка. Ключевые слова нельзя использовать в качестве идентификаторов	`while`
		`do`
		`function`
		`var`

Зарезервированное слово (reserved word)	Слово, которое может стать частью языка. Зарезервированные слова не следует использовать в качестве идентификаторов, хотя это правило иногда является не слишком строгим	`class` `public`

Характеристика языка

При изучении нового языка программирования важно выделить его главные особенности — то, как выполняется программный код, как интерпретируются символы пустого пространства, разделяются операторы и т.д. Именно такие вопросы будут рассмотрены в данном разделе, поскольку их следует выяснить до того, как приступить к обсуждению типов данных, операторов вообще и операторов JavaScript в частности.

Порядок выполнения сценариев

Программный код JavaScript, присутствующий в документах (X)HTML, интерпретируется строка за строкой в том порядке, в каком он находится на странице. Это означает, что определения функций и декларации переменных лучше размещать в разделе заголовка документа, окруженном дескрипторами `<head> ... </head>`, если эти

определения будут использоваться в других местах страницы. Некоторые фрагменты программного кода — например, тело функций и программы, связанные с обработкой ошибок, — не предназначены для немедленного выполнения.

Чувствительность к регистру символов

JavaScript чувствителен к регистру символов. Это означает, что в нем прописные буквы отличаются от строчных. Например, если вы используете в сценарии идентификаторы `result`, `Result` и `RESULT`, то каждый из этих идентификаторов будет определять отдельную переменную. Чувствительность к регистру касается всех аспектов языка: ключевых слов, операторов, имен переменных, программ обработки событий, объектных свойств и т.д. Все ключевые слова JavaScript состоят из символов нижнего регистра, поэтому при использовании структуры типа `if`, следует печатать именно `if`, а не `If` или `IF`. Поскольку в JavaScript применяется соглашение об использовании "криволинейного" стиля для имен, в именах многих методов и свойств содержатся символы как нижнего, так и верхнего регистров. Например, в имени свойства `lastModified` объекта `Document` буква `M` должна быть набрана в верхнем регистре, а при использовании буквы `m` нижнего регистра мы получим неопределенное значение.

Главным следствием чувствительности к регистру является необходимость уделять особое внимание прописным буквам при определении переменных и доступе к ним, при использовании конструкций типа `if` и `while` и доступе к свойствам объектов. Одна опечатка может изменить значение всего сценария и потребовать существенных усилий при отладке.

Замечание. *Одним исключением в смысле чувствительности JavaScript к регистру является Internet Explorer 3. В этой версии указанного браузера объекты и свойства клиента нечувствительны к регистру символов. Это исключение не создает проблем при разработке сценариев. Это лишь означает, что некоторые старые сценарии, полагающиеся на нечувствительность к регистру в Internet Explorer 3, в более новых браузерах могут не работать.*

Чувствительность к регистру и HTML

В HTML до версии 4 включительно имена элементов и атрибутов нечувствительны к регистру. Например, следующие два дескриптора эквивалентны:

```
<IMG SRC="plus.gif" ALT="Increment x" ONCLICK="x=x+1">
<img src="plus.gif" alt="Increment x" onClick="x=x+1">
```

Само по себе это не является проблемой. Проблема возникает тогда, когда новичок-программист замечает, что допускаются различные ссылки на обработчики событий HTML (как `ONCLICK` и `onClick` в предыдущем примере), и заключает, что точно так же на обработчики событий можно ссылаться и в JavaScript. Но это не так! Соответствующим обработчиком событий в JavaScript является `onclick`, и на него всегда следует ссылаться только так, а не иначе. Причина, по которой в HTML работают и `ONCLICK`, и `onClick`, заключается в том, что браузер автоматически переводит указанные имена в правильное имя `onclick` обработчика событий JavaScript.

Теперь рассмотрим следующие два варианта, которые не эквивалентны:

```
<img src="plus.gif" alt="Increment x" onclick="x=x+1">
<img src="plus.gif" alt="Increment x" onclick="X=X+1">
```

Они не эквивалентны потому, что в первом изменяется переменная x, а во втором — переменная X. Ввиду того что JavaScript чувствителен к регистру, эти переменные различаются. Это иллюстрирует важный аспект атрибутов HTML: хотя само имя атрибута нечувствительно к регистру, значение может быть чувствительным. Атрибут HTML onclick нечувствителен к регистру символов, а потому может быть записан как onClick, ONCLICK или даже oNcLiCk. Но так как присвоенное ему значение содержит программный код JavaScript, его *значение* чувствительно к регистру символов.

К счастью, с появлением XHTML, в котором требуется, чтобы имена элементов и атрибутов состояли из символов нижнего регистра, проблема чувствительности к регистру в пересечении двух технологий становится менее острой. Разработчики должны всегда предполагать чувствительность к регистру символов и во всей разметке предпочитать символы нижнего регистра.

Символы пустого пространства

Символы пустого пространства не имеют визуального представления, они просто создают на экране свободное место. К символам, о которых идет речь, относятся пробелы, символы табуляции и пустой строки. Любая лишняя последовательность таких символов в JavaScript игнорируется. Например,

```
x                                   =     x +        1;
```

означает то же самое, что и

```
x = x + 1;
```

Предполагается, что от этого больше пользы программисту, чем интерпретатору. Действительно, правильное использование отступов для выделения комментариев, содержимого циклов и деклараций делает программный код лучше для чтения и понимания.

Замечание. *Ввиду такой амбивалентности JavaScript и недовольства пользователей Web в связи с медленной загрузкой страниц некоторые программисты предпочитают "сжимать" сценарии, убирая лишние пробелы вручную или с помощью специальных программ.*

Промежутки между лексемами можно вообще удалить, если это не ведет к неоднозначности. Например, выражение

```
x=x+1;
```

не содержит пробелов, но вполне приемлемо, поскольку его значение понятно. Однако для большинства операторов, за исключением простейших арифметических функций, пробелы необходимы, поскольку от них может зависеть значение оператора. Рассмотрим следующий пример:

```
s = typeof x;
s = typeofx;
```

Первый оператор инициирует действие typeof на переменную x и помещает результат в переменную s. Второй оператор копирует значение переменной с именем *typeofx* в s. Один пробел полностью меняет значение оператора.

Правило игнорирования лишних пробелов в JavaScript имеет два исключения. Первое из них относится к строкам. Пробелы будут сохранены в любой строке, окруженной одиночными или двойными кавычками:

```
var s = "Такие     пробелы         с о х р а н я ю т с я.";
```

Опытные программисты могут поинтересоваться, что случится, если включить в строку символ перехода на новую строку. Ответ демонстрирует еще одну особенность JavaScript: использование *неявной точки с запятой* и ее связь с операторами.

Операторы

Операторы — это суть языка JavaScript. Они являются инструкциями интерпретатору по выполнению конкретных действий. Например, одним из наиболее часто используемых операторов является оператор присваивания. В операторе присваивания используется знак операции =, и в результате присваивания значение, стоящее справа от этого знака, помещается в переменную, расположенную слева. Например,

```
x = y + 10;
```

добавляет 10 к значению `y` и помещает полученное значение в `x`. Оператор присваивания не следует путать с оператором сравнения == (“равно”), который используется в условных выражениях (мы обсудим условные выражения немного позже в этой же главе). Ключевой особенностью операторов любого языка программирования является то, как они заканчиваются и группируются.

Разделители операторов: точка с запятой и переход на новую строку

Точка с запятой указывает конец оператора JavaScript. Например, можно разместить группу операторов в одной строке, разделяя операторы точками с запятой:

```
x = x + 1;  y = y + 1;  z = 0;
```

В одной строке можно указать и более сложные, и даже пустые операторы:

```
x = x + 1; ;; if (x > 10) { x = 0; }; y = y - 1;
```

В этом примере увеличивается значение `x`, пропускаются два пустых оператора, значение `x` устанавливается равным нулю, если `x` больше `10`, и, наконец, уменьшается значение `y`. Как видите, включение нескольких операторов в одну строку оказывается достаточно громоздким, поэтому такой практики следует избегать.

Хотя операторы обычно завершаются точкой с запятой, ее можно опустить, если операторы разделяются символом перехода на новую строку. Например,

```
x = x + 1
y = y - 1
```

интерпретируются как

```
x = x + 1;
y = y - 1;
```

Конечно, если вы желаете разместить два оператора в одной строке, то для их разделения придется использовать точку с запятой:

```
x = x + 1; y = y - 1
```

Формальные правила для вставки неявной точки с запятой намного сложнее, чем предыдущее описание. Теоретически лексемы одного оператора могут разделяться символами перехода на новую строку без генерирования ошибки. Однако если лексемы в строке без точки с запятой составляют полный оператор JavaScript, точка с запятой будет вставлена автоматически, даже если следующая строка очевидно ин-

терпретируется как продолжение первой. Классический пример — оператор `return`. Ввиду того что аргумент оператора `return` необязателен, размещение `return` и его аргумента в разных строках ведет к выполнению оператора `return` без аргумента. Например,

```
return
x
```

интерпретируется как

```
return;
x;
```

а не так, как, вероятно, предполагалось программистом:

```
return x;
```

Таким образом, надежда на неявную точку с запятой является не слишком хорошей идеей, а к тому же и плохим стилем программирования. На практике такого подхода следует избегать, если только вы не знаете в совершенстве все тонкости правил вставки точки с запятой в JavaScript.

Блоки

Фигурные скобки (`{ }`) используются для группирования операторов. Можно интерпретировать применение фигурных скобок как создание одного большого оператора (или блока программного кода). Например, в фигурные скобки заключаются операторы, формирующие тело функции:

```
function add(x, y)
{
   var result = x + y;
   return result;
}
```

Если в результате оценки условного выражения или в цикле необходимо выполнить более одного оператора, такие операторы группируются аналогично:

```
if (x > 10)
{
   x = 0;
   y = 10;
}
```

Но, сгруппированные или нет, операторы в общем случае необходимы для изменения данных, которые часто представлены в форме переменных.

Переменные

В переменных хранятся данные. Каждая переменная имеет имя, называемое *идентификатором*. В JavaScript переменные объявляются с помощью `var` — ключевого слова, которое выделяет память для новых данных и сообщает интерпретатору о том, что вводится новый идентификатор. Объявить переменную просто:

```
var x;
```

Этот оператор говорит интерпретатору о том, что новая переменная x готова к использованию. При объявлении переменной можно присвоить ей и начальное значение:

```
var x = 2;
```

Кроме того, одним оператором var можно объявить много переменных, если разделить переменные запятыми:

```
var x, y = 2, z;
```

Не следует использовать переменные без предварительного их объявления, хотя в определенных случаях это возможно. Использование переменной в правой стороне оператора присваивания без предварительного ее объявления приводит к ошибке.

Опытные программисты заметят, что, в отличие от C, C++ и Java, в JavaScript имеется только один способ объявления переменной. Это подчеркивает тот факт, что в JavaScript обращение с типами данных фундаментально отличается от многих других языков, включая C, C++ и Java.

Базовые типы данных

Каждая переменная имеет свой *тип данных*, указывающий, какой вид данных хранит эта переменная. Базовые типы данных в JavaScript — это строковый, числовой и логический (булев). Строка — это набор символов, и строковый литерал обозначается с помощью заключения такого набора символов в одиночные или двойные кавычки. Строки могут содержать любое число символов, включая пробелы и специальные символы (например, \n — переход на новую строку). Числовые данные — это целые значения или значения с плавающей десятичной точкой, и числовые литералы задаются естественным образом. Булевы данные могут принимать одно из двух значений: true (ИСТИНА) или false (ЛОЖЬ). Соответствующие литералы указывают с помощью значений true или false непосредственно в исходном программном коде. Вот пример использования трех указанных типов данных:

```
var stringData = "JavaScript использует строки.\n Это правда.";
var numericData = 3.14;
var booleanData = true;
```

JavaScript поддерживает еще два базовых типа — неопределенный и пустой. Все эти типы данных, а также специальные символы подробно обсуждаются в главе 3. Однако одна из особенностей реализации поддержки типов данных в JavaScript заслуживает специального упоминания здесь — речь идет о слабом контроле типов.

Динамический контроль типов

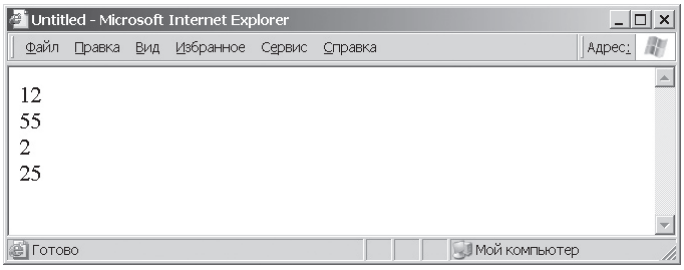
Главным отличием JavaScript от других языков программирования, известных читателю, является то, что в JavaScript осуществляется *динамический контроль типов* (или, согласно другим определениям, *слабый контроль типов*). Каждая переменная JavaScript имеет определенный тип данных, но этот тип данных логически зависит от содержимого переменной. Например, переменная, которой присвоено строковое значение, считается переменной строкового типа. Следствием автоматического определения типа данных в JavaScript является то, что тип переменной может меняться в ходе выполнения сценария. Например, переменная может сначала хранить строковое значение, а позже ей можно присвоить значение типа Boolean. Тип переменной

изменится в соответствии с хранимыми ею данными. Это и объясняет, почему в JavaScript допускается только один способ объявления переменных: нет необходимости указывать тип в декларации переменной.

Слабый контроль типов — это и преимущество, и недостаток JavaScript. Слабый контроль типов вроде бы освобождает программиста от необходимости объявлять типы переменных заранее, но при этом могут возникать коварные ошибки ввода. Например, рассмотрев сценарий, в котором обрабатываются различные строковые и числовые значения, мы увидим, что преобразования типов могут вести к потенциальной неоднозначности:

```
document.write(4*3);
document.write("<br />");
document.write("5" + 5);
document.write("<br />");
document.write("5" - 3);
document.write("<br />");
document.write(5 * "5");
```

Результат вычислений этого примера, если включить его в документ HTML, оказывается следующим:



Обратите внимание на то, что в большинстве случаев перед вычислением строка превращалась в число, поэтому получался правильный результат. Конечно, если мы попытаемся вычислить что-нибудь наподобие `"cat" - 3`, мы увидим в результате **NaN** (здесь строка `"cat"` конвертируется в **NaN**, поэтому результатом вычитания тоже будет **NaN**). Но и в случае сложения `"5" + 5` результатом оказалась строка `"55"`, а не число `10`. Причиной, по которой сложение здесь не работает, является двойное назначение знака плюс — он используется и для сложения, и для конкатенации строк.

Преобразование типов в совокупности с перегруженными операторами наподобие `+` могут быть источником недоразумений как для начинающего, так и для опытного программиста, поэтому мы собираемся уделить достаточно много внимания обсуждению этого вопроса в главе 3. К счастью, предлагаемые там правила вполне логичны, и в JavaScript есть много способов предсказуемого конвертирования данных с помощью методов, подобных `parseFloat()`, а кроме того, существует также возможность проверки значения переменной с помощью оператора `typeof`. Например,

```
var x = "5";
alert (typeof x);
```

правильно идентифицирует текст после `x`, как строковое значение, что демонстрирует следующая иллюстрация:

Составные типы данных

В отличие от примитивных типов данных, таких как числовые и строковые, *составные типы данных* складываются из разнородных данных. Составной тип данных может содержать не только строковые, числовые, логические, неопределенные и пустые значения, но и другие составные типы. JavaScript поддерживает три составных типа данных: объекты, массивы и функции. В главах 6 и 7 вы узнаете, что массивы и функции в действительности являются специальными видами объектов, но сейчас, не затрагивая тонкостей объектно-ориентированной природы JavaScript, мы обсудим только основы.

Массивы

Массив — это упорядоченное множество значений, сгруппированных вместе с посредством одного идентификатора. Существует много способов создания массивов, но проще всего определить массив как стандартный идентификатор, а затем просто указать группу значений в скобках. Следующий оператор определяет массив `myArray` с четырьмя числовыми значениями:

```
var myArray = [1,5,68,3];
```

Массивы могут содержать любые элементы данных, так что определение

```
var myArray = ["Thomas", true, 3, -47.6, "x"];
```

тоже оказывается допустимым.

Другим способом определения массивов, отражающим их объектную сущность, является использование ключевого слова `new` для вызова конструктора объекта `Array`, как показано здесь:

```
var myArray = new Array();
```

В результате `myArray` будет создан как массив неопределенной длины.

Можно без труда задать длину этого массива, указав конкретное числовое значение. Например,

```
var myArray = new Array(4);
```

определяет массив, длина которого равна 4.

В рамках синтаксиса конструктора можно даже заполнить массив значениями:

```
var myArray = new Array(1,5,"Thomas", true);
```

Независимо от способа определения массива, доступ к его элементам осуществляется одним способом: чтобы сослаться на конкретный элемент массива, следует указать соответствующий индекс в скобках. Так, в результате выполнения операторов

```
var myArray = new Array(1,5,"Thomas", true);
var x = myArray[2];
var y = myArray[0];
```

значением x будет строка "Thomas", а значением y — число 1. Причина в том, что массивы в JavaScript индексируются, начиная с 0. Вот пример определения массива и его заполнения с использованием индексов:

```
var myArray = new Array(4);
myArray[0] = 1;
myArray[1] = 5;
myArray[2] = "Thomas";
myArray[3] = true;
```

Как упоминалось выше, массивы — это на самом деле объекты, имеющие множество свойств и методов, которые можно использовать для того, чтобы управлять ими. Соответствующие возможности будут обсуждаться в главе 7. Но давайте сначала познакомимся с объектами в JavaScript.

Объекты

Объекты могут содержать любые типы данных и позволяют решать множество полезных задач. Браузер предлагает для использования большое число объектов. Например, вы можете взаимодействовать с пользователем с помощью объекта Window или изменить содержимое Web-страницы с помощью объекта Document.

Данные, содержащиеся в объекте, называют *свойствами* объекта. Доступ к свойствам обеспечивает операция "точка", которая представляется просто точкой, за которой следует имя свойства. Синтаксис таков:

имяОбъекта.имяСвойства

Например, можно получить доступ к свойству lastModified объекта Document с помощью document.lastModified.

Функции, содержащиеся в объекте, называют *методами* объекта. Методы тоже доступны с помощью указанной операции:

имяОбъекта.имяМетода()

Мы уже использовали методы в предыдущих примерах. Метод write() объекта Document использовался для вывода текста на экран:

```
document.write("Всем привет от JavaScript!");
```

Вы увидите, что при использовании объектов длина идентификатора, требующегося для доступа к конкретному свойству, может оказаться достаточно большой. Так, многократное печатание document.write становится утомительным занятием, не говоря уже о более глубоко вложенных подобъектах. С помощью ключевого слова with можно избежать необходимости ссылаться на полный путь к свойству или методу объекта:

```
with (document)
{
   write("это проще, ");
   write("чем выписывать ");
```

```
    write("полный путь");
}
```

Кроме использования встроенных объектов типа `Document` или `Window`, с помощью ключевого слова `new` можно создавать собственные объекты. Пример использования ключевого слова `new` уже предлагался при создании массива в предыдущем разделе. С помощью ключевого слова `delete` можно уничтожить свойство или элемент в массиве. Например, ниже мы определяем элемент массива, а затем сразу же уничтожаем его:

```
var myArray = new Array(4);
myArray[0]="Thomas";
delete myArray[0];
```

По своей сути JavaScript является объектно-ориентированным языком программирования, и в нем все получают от различных объектов, обеспечиваемых либо самим языком, либо браузером. Например, JavaScript предлагает объекты, соответствующие примитивным типам данных (`String`, `Number` и `Boolean`), вместе с методами, необходимыми для работы с этими типами данных. Предлагаются и более сложные объекты, связанные с данными (например, `Array`, `Math` и `Date`), а также объекты, ориентированные на браузер (такие, как `Navigator` и `History` или исключительно мощный по своим возможностям объект `Document`). Существует даже базовый объект `Object`, который можно использовать для построения своих собственных объектов. Детали процесса создания и использования объектов требуют отдельного объяснения, которое вы найдете в главе 6.

Замечание. *Экземпляры объектов обычно именуют с помощью символов нижнего регистра, в то время как соответствующий тип объекта пишется с прописной буквы. Пока что об этом не беспокойтесь — этот вопрос подробно обсуждается в главах 6 и 7.*

Выражения

Выражения — это важная часть JavaScript, поскольку они являются строительными блоками многих операторов JavaScript. Выражения представляют собой группы лексем, которые можно оценивать. Например,

```
var x = 3 + 3;
```

является оператором присваивания, который берет выражение `3 + 3` и помещает его значение в переменную `x`. Простейшими выражениями являются литералы и переменные; используя их и знаки операций, можно создавать более сложные выражения.

Знаки операций

К основным знакам операций относятся всем известные символы арифметических операций: = (присваивание), + (сложение), – (вычитание или унарное отрицание), * (умножение), / (деление) и % (модуль); все они используются в следующем примере:

```
var x=3, y=6;
x = -x;
x = y + 2;
x  = y - 1;
x = y * y;
```

```
x = y  / x;
x = y % 4;
```

В этом примере переменной x сначала присваивается значение -3, затем — 8, затем — 5, затем — 36, затем — 2 и, наконец, снова — 2. Возможно, незнаком вам только знак операции модуля (%), результатом которой является остаток целочисленного деления.

JavaScript предлагает и побитовые операции, среди которых & (И), | (ИЛИ), ^ (НЕ), ~ (исключающее ИЛИ), << (сдвиг влево) и >> (сдвиг вправо). Побитовые операции привычны для программистов, использующих язык C, но в данном случае, с учетом высокоуровневой природы JavaScript, при использовании этих операций в контексте Web-страниц они могут показаться несколько неуместными.

Для сравнения объектов JavaScript предлагает также операции отношения: == (равно), != (не равно), < (меньше чем), > (больше чем), <= (меньше или равно), >= (больше или равно). Использование операций отношения в выражениях заставляет оценивать выражение как `true` (ИСТИНА), если соответствующее условие выполняется, или `false` (ЛОЖЬ) в противном случае. Так что выражение

```
5 < 10
```

будет оценено как `true`, а выражению

```
11 < 10
```

будет приписана оценка `false`.

Программист должен с особой осторожностью использовать = и ==. Первое означает оператор присваивания, а второе — условный оператор сравнения. Неправомерная замена одного другим является одной из самых распространенных ошибок в программах JavaScript. Например,

```
x = 5;
```

присваивает значение переменной x, а

```
x == 5;
```

выполняет сравнение значения x с литералом 5. Если перепутать указанные знаки операций в выражении `if`, возникнет неприятная ошибка.

При сравнениях можно использовать логические операторы && (И), || (ИЛИ) и ! (НЕ), чтобы создавать более сложные условия. Например,

```
if ((x >= 10) && (y < 3))
{
   z = z + 1;
}
```

выполняет приращение z, если x больше или равно 10, а y меньше 3.

Ввиду важности операций приращения (а также отрицательного приращения), в JavaScript, как и в других языках, предлагается соответствующая нотация. Знак ++ означает добавление единицы к некоторому значению, а -- — вычитание единицы. Так, в результате выполнения

```
var x=4;
x++;
```

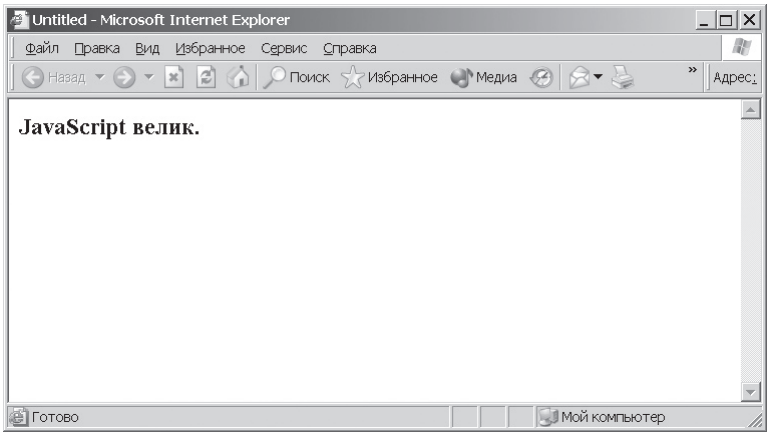
значением x станет 5.

Замечание. *Имеется большая разница при размещении ++ или -- до и после значения, о чем говорится в главе 4.*

Одним из очень полезных знаков операций является знак строковой операции +, используемый для соединения строк. Как показывает следующая иллюстрация, сценарий

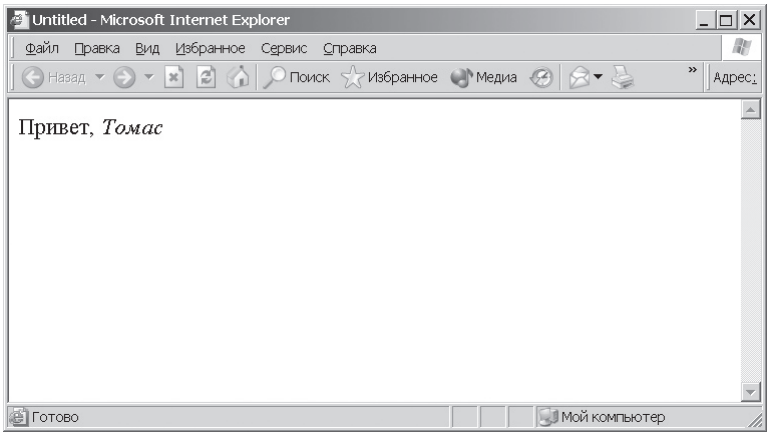
```
document.write("JavaScript " + "велик.");
```

выводит объединенную строку:



Комбинируя знаки операций с переменными и HTML, можно создать более сложный вывод:

```
var myName="Томас";
document.write("Привет, <i>"+myName+" </i>");
```



Приоритет операций

При использовании знаков операций приходится учитывать порядок их выполнения. С учетом того, что одни операции могут иметь преимущество над другими, порядок их выполнения по умолчанию может не соответствовать тому, что требуется. Рассмотрим следующий пример:

```
var x = 4 + 5 * 8;
```

Какое значение присваивается переменной x — 72 или 44? В данном случае это 44, потому что операция умножения имеет более высокий приоритет по сравнению со сложением. Можно использовать круглые скобки, чтобы сгруппировать выражения и заставить их выполняться в определенном порядке. Так, чтобы в предыдущем примере установить x равным 72, следует записать:

```
var x = (4+5)*8;
```

Этот пример очень прост, но иногда порядок выполнения операций не столь очевиден, поэтому при наличии сомнений всегда используйте скобки. Тонкости всех операций обсуждаются в первой части главы 4.

Операторы управления потоком

Операторы выполняются в порядке их появления в сценарии. Чтобы создавать полезные программы, обычно необходимо использовать *управление потоком*, т.е. код, управляющий "течением" выполнения программы. JavaScript поддерживает условные операторы типа if/else и switch/case, которые дают возможность выборочного выполнения кусков программного кода. Примером применения оператора if/else является

```
if (x > 10)
{
   x = 0;
}
else
{
   x = x + 1;
}
```

Сначала оценивается условие оператора if и, если сравнение дает true (т.е. x действительно больше 10), то x устанавливается равным нулю. Иначе для x выполняется приращение.

Отметьте, что можно использовать оператор if и без соответствующего else, а также использовать множество других операторов if в рамках else. Это делает оператор if исключительно запутанным, и в таких случаях более подходящим оказывается оператор switch. Например, вместо использования каскада операторов if можно использовать один оператор switch с множеством выражений case:

```
var x=3;
switch (x)
{
  case 1: alert('x равно 1');
    break;
  case 2: alert('x равно 2');
    break;
  case 3: alert('x равно 3');
    break;
  case 4: alert('x равно 4');
    break;
  default: alert('x не равно 1, 2, 3 или 4');
}
```

В этом примере значение x определяет, какое сообщение должно быть напечатано, путем сравнения значения переменной с разными выражениями case. Если совпадения не обнаруживается, выполняется оператор default. Для выхода из оператора switch после того, как обнаружено совпадение, обычно используют оператор break. Но оператор break не менее часто используется и в циклах, которые обсуждаются ниже.

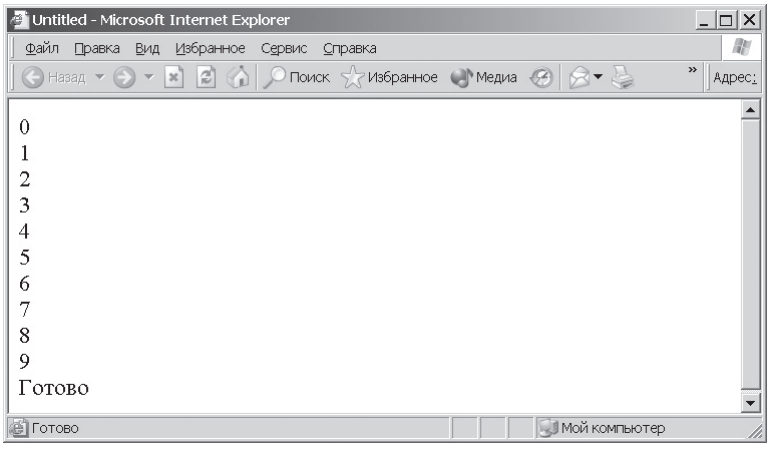
Замечание. *Оператор switch был введен в язык, начиная с версии JavaScript 1.2, поэтому в очень старых браузерах этот оператор должен использоваться с осторожностью.*

Циклы

Часто требуется повторить выполнение ряда операторов несколько раз, пока выполняется соответствующее условие. Например, может потребоваться выполнить одно и то же действие с каждым элементом массива, пока не будет достигнут конец этого массива. Как и в других языках программирования, в JavaScript это реализуется с помощью операторов *цикла*. Циклы продолжают выполнять операторы тела их программного кода до тех пор, пока не будет выполнено условие остановки. В JavaScript поддерживаются циклы while, do/while, for и for/in. Вот пример цикла while:

```
var x=0;
while (x < 10)
{
  document.write(x);
  document.write("<br />");
  x = x + 1;
}
document.write("Готово");
```

В этом цикле выполняется приращение x до тех пор, пока условие цикла — x меньше 10 — имеет значение true. Как только x принимает значение 10, условие получает значение false, поэтому выполнение цикла заканчивается, а выполнение сценария продолжается с первого оператора, следующего после тела цикла, как показано ниже:



Цикл do/while подобен циклу while, за исключением того, что проверка условия выполняется в конце цикла. Это значит, что указанный цикл всегда выполняется как минимум один раз, если только не будет обнаружен оператор break.

```
var x=0;
do
{
  document.write(x);
  document.write("<br />");
  x = x + 1;
} while (x < 10)
```

Тот же цикл, записанный с помощью оператора for, оказывается немного более компактным, поскольку он включает в себя установку переменной цикла, проверяемое условие и приращение, размещая все это в одной строке, как показано в следующем примере:

```
for (x=0; x < 10; x++)
{
  document.write(x);
  document.write("<br />");
}
```

Интересной вариацией цикла for является конструкция for/in. Эта конструкция позволяет выполнить цикл по различным свойствам объекта. Например, можно выполнить цикл по всем свойствам объекта окна браузера и напечатать их, используя оператор for/in следующего вида:

```
var aProperty
for (aProperty in window)
 {
   document.write(aProperty)
   document.write("<br />");
}
```

Опытным программистам этот оператор знаком, а для всех остальных он приобретет больше смысла после обсуждения объектов, предложенного в главе 6.

Управление циклами

JavaScript поддерживает и операторы, используемые для изменения управления потоком, в частности операторы break и continue. Действие этих операторов подобно соответствующим конструкциям в C, и они часто используются с циклами. Оператор break останавливает выполнение цикла до его логического завершения, а оператор continue возвращает выполнение к проверке условия цикла. В следующем примере, где выписываются значения x, начиная с 1, при x равном 3 оператор continue продолжит выполнение цикла без печати соответствующего значения. Когда x будет равно 5, произойдет выход из цикла с помощью оператора break.

```
var x=0;
while (x < 10)
{
   x = x + 1;
   if (x == 3)
     continue;
```

```
   document.write("x = "+x+"<br />");
   if (x == 5)
     break;
}
document.write("Loop done");
```

Все формы операторов, включая операторы управления потоком и операторы цикла, подробно обсуждаются в главе 4.

Функции

Функции используются для инкапсуляции программного кода, с помощью которого выполняется определенное задание. Иногда функции применяются для часто выполняемых задач, чтобы избежать необходимости печатать одни и те же операторы снова и снова. В общем, функции используются для того, чтобы хранить предназначенный для выполнения конкретной задачи программный код в одном месте, что улучшает перспективы его повторного использования и делает программу более понятной.

Функции JavaScript объявляются с помощью ключевого слова `function`, а содержащиеся в них операторы заключаются в фигурные скобки. Аргументы функции указываются в круглых скобках после имени функции и разделяются запятыми. Например:

```
function add(x, y)
{
   var sum = x + y;
   return sum;
}
```

Этот программный код объявляет функцию с именем *add*, которая вычисляет сумму аргументов и "возвращает" вычисленное значение. Оператор `return` сообщает интерпретатору, какое значение должна вычислить функция. Можно, например, установить значение функции равным значению некоторой переменной:

```
var result = add(2, 3);
```

Функции передаются аргументы `2` и `3`, выполняются операторы тела функции, а результат сложения аргументов — значение `5` — помещается в переменную *result*.

Кроме передачи функции литеральных значений, можно передавать функциям и переменные. Например:

```
var a = 3, b=5;
var result;
result = add(a,b);
```

Опытный программист поинтересуется: можно ли модифицировать значения переменных, передаваемых функции? Ответ, в котором содержится скорее совет, будет следующим: *нет*. Для примитивных типов данных в JavaScript используется передача по значению, поэтому значения переменных `a` и `b` будут оставаться неизмененными, независимо от того, что происходит в функции `add`. Однако, другие типы данных, такие как объекты, могут после передачи меняться (они передаются посредством ссылок), и это вызывает затруднения. Те, кто имеет опыт программирования на других языках, знают, что функции могут называться по-разному — процедурами, подпрограммами и методами. Вы увидите, что функции, которые будут обсуждаться подробно в главе 6, являются очень мощным средством программирования.

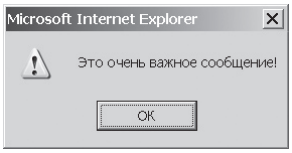
Ввод и вывод в JavaScript

Способность выполнять операции ввода-вывода — неотъемлемая составляющая большинства языков программирования. Поскольку JavaScript функционирует во внешнем окружении, подобном Web-браузеру, средства ввода-вывода могут отличаться от привычных вам. По очевидным причинам безопасности обычный JavaScript клиента не позволяет чтение или запись файлов в локальной файловой системе. Здесь имеются исключения, но ввиду их сложности нам придется обсудить их позже, в одной из следующих глав.

Ввод-вывод, как и другие наиболее важные задачи JavaScript, выполняется посредством объектов, предлагаемых браузером. Взаимодействие с пользователем обычно реализуется с помощью объекта `Window`, некоторые методы которого описываются здесь. Одним из наиболее часто применяемых методов ввода-вывода в JavaScript является использование метода `alert()` объекта `Window`, отображающего свой аргумент-сообщение в диалоговом окне, содержащем кнопку OK. Например, использование

```
alert("Это очень важное сообщение!");
```

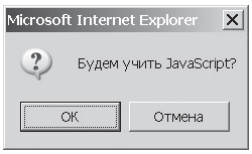
приведет к тому, что перед глазами пользователя появится следующее диалоговое окно:



Другую форму диалога с пользователем предлагает метод `confirm()`, который отображает аргумент-сообщение в диалоговом окне с двумя кнопками: OK и Отмена. Используя сценарий

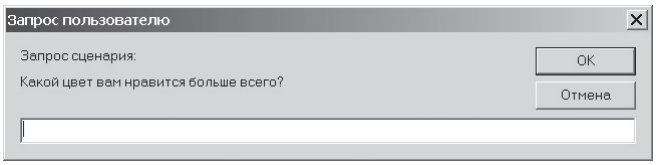
```
confirm("Будем учить JavaScript?");
```

вы увидите следующее окно:



Наконец, можно использовать метод `prompt()`, позволяющий получить данные от пользователя. Метод `prompt()` отображает сообщение своего аргумента в диалоговом окне и предоставляет пользователю возможность ввести данные в текстовое поле, как иллюстрируют следующий пример:

```
var answer = prompt("Какой цвет вам нравится больше всего?","");
```



Замечание. *Все указанные выше методы являются частью объекта* `Window`*, но вместо* `window.alert("привет")` *мы использовали* `alert("привет")`*. Допустимость такого сокращения вытекает из правил обзора объектов JavaScript, которые будут обсуждаться в главах 6 и 9.*

Как правило, вывод осуществляется посредством объекта `Document`. Этот объект предлагает множество способов управления Web-страницами. Для вывода проще всего использовать методы `write()` и `writeln()`. Метод `write()` записывает свои аргументы в текущий документ. Метод `writeln()` ему идентичен, за исключением того, что после записи аргумента вставляется переход на новую строку. Например:

```
document.write("Здесь нет символа перехода на новую строку. ");
document.writeln("Однако здесь используется writeln().");
document.write("Поэтому и происходит переход на новую строку.");
```

Причина, по которой вы можете не заметить никаких отличий при тестировании данного примера, заключается в том, что JavaScript, как правило, направляет вывод в документ (X)HTML. В главе 1 уже отмечалось, что пересечение двух языков может порождать проблемы для программиста. Браузеры, поддерживающие (X)HTML, заменяют символы перехода на новую строку пробелом, поэтому символы перехода на новую строку не оказывают никакого влияния на вывод. Такая особенность, вероятно, и объясняет, почему большинство программистов на JavaScript предпочитают использовать `document.write()` вместо `document.writeln()`. Чтобы увидеть разницу между `document.write` и `document.writeln`, можно использовать дескриптор `<pre>`, поместив пример в его окружение:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Write/Writeln Example</title>
<meta http-equiv="Content-Type"
 content="text/html; charset=windows-1251" />
</head>
<body>
<pre>
 <script type="text/javascript">
  document.write("Здесь нет символа перехода на новую строку. ");
  document.writeln("Но здесь используется writeln().");
  document.write("Поэтому и происходит переход на новую строку.");
 </script>
</pre>
</body>
</html>
```

Результат выполнения кода этого примера в браузере показан на рис. 2.1.

В дополнение к `write()` и `writeln()` объект `Document` обеспечивает мощные возможности манипуляции HTML и XML через объектную модель документа (DOM). Модель DOM, обсуждаемая в главе 10, может использоваться для замены и вставки текста, изменения особенностей форматирования, а также для записи или чтения форм HTML.

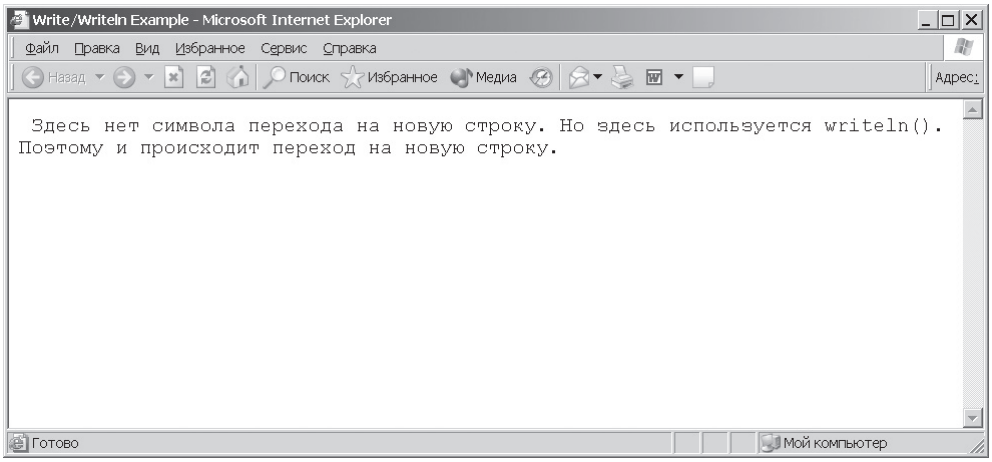


Рис. 2.1. Вывод методов `write()` и `writeln()`

Регулярные выражения

Последней из основных возможностей JavaScript являются регулярные выражения. Регулярное выражение, определяемое конструктором `RegExp`, используется для выполнения сравнений с образцом.

```
var country = new RegExp("England");
```

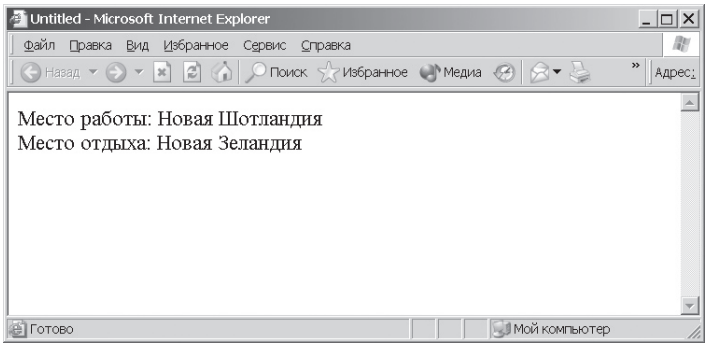
То же самое можно было бы сделать и с помощью прямого присваивания:

```
var country = /England/;
```

Определив регулярное выражение, мы можем его использовать в качестве шаблона для поиска и изменения строк. В следующем простом примере ищется соответствующий кусок строки в переменной `geographicLocation` и заменяется другой строкой.

```
var country = new RegExp("Шотландия");
var geographicLocation = "Новая Шотландия";

document.write("Место работы: "+geographicLocation+"<br />");
geographicLocation = geographicLocation.replace(country, "Зеландия");
document.write("Место отдыха: "+geographicLocation);
```



Результат выполнения этого сценария показан выше.

Реализация поддержки регулярных выражений в JavaScript является исключительно мощной и очень похожа на реализацию их поддержки в Perl, поэтому многие программисты сразу почувствуют себя вполне комфортно при работе с соответствующими возможностями JavaScript. Более подробная информация о регулярных выражениях содержится в главе 8.

Комментарии

В заключение рассмотрим еще один очень важный аспект хорошего стиля программирования — сопровождение программного кода комментариями. Комментарии позволяют поместить замечания, касающиеся программы, непосредственно в исходный код, что делает программу более удобной для чтения как для самого автора программы, так и для других. Все комментарии, которые вы включите в программный код, будут проигнорированы интерпретатором JavaScript. Комментарии в JavaScript подобны комментариям в C++ и Java. Имеются два вида комментариев: те, которые тянутся только до конца текущей строки, и те, которые занимают несколько последовательно идущих строк.

Однострочные комментарии начинаются с двойной косой черты (//), которая заставляет интерпретатор пренебречь всем, начиная с этого указателя, до конца строки. Например,

```
var count = 10;     // число заказанных единиц товара
```

Комментарии, занимающие несколько строк, окружаются парой косая-звездочка (/*) и звездочка-косая (*/) в стиле C. В следующем примере кода иллюстрируются оба вида комментариев:

```
/* Функция возведения в квадрат имеет аргумент числового типа и
   возвращает значение, равное его квадрату.
   Например, чтобы возвести в квадрат значение 10
   и поместить его в переменную y, используйте функцию так:
   var y = square(10);
   Эта функция должна вызываться только с данными числового типа!
*/
function square(x)
{
   return x*x;  // умножить x на x и вернуть значение
}
```

Все между /* и */ интерпретатором игнорируется. Заметьте, что вкладывать многострочные комментарии не допускается. Такое вложение генерирует ошибку:

```
/* Это -
/* вложенные комментарии, и это
*/
определенно приведет к ошибке! */
```

Еще раз подчеркнем, насколько важны комментарии для создания хорошего программного кода. Комментарии должны добавлять информацию, которая не является очевидно вытекающей непосредственно из программного кода. Например, хорошим стилем всегда считалось добавление комментария для каждой определяемой функ-

ции, в котором должны быть описаны значения, ожидаемые этой функцией, выполняемые ею действия, возможные побочные эффекты и тип возвращаемого значения. Сложные операторы или циклы тоже должны снабжаться комментариями, как и любые объекты, которые вы создаете для своих целей. Кроме того, каждый сценарий должен иметь вступительный комментарий, в котором указывается назначение сценария, а также все известные недостатки и оговорки, касающиеся содержащегося в нем программного кода.

Комментарии делают программный код более понятным для других. Наиболее страшный кошмар для программиста — необходимость исправлять или поддерживать большие куски некомментированного программного кода. Вы можете сэкономить вашему преемнику многие часы работы, включив информацию о вашей логике и рассуждениях в создаваемый программный код. Профессиональные программисты всегда комментируют свой программный код, особенно в таком неустойчивом окружении, как Web.

Комментарии делают программный код более понятным и для вас. Все, кто потратил достаточно много времени на создание программного обеспечения, могут рассказать вам о случаях, когда по прошествии некоторого времени созданные ими же куски старого программного кода ставили их в тупик. Вы же не собираетесь помнить детали и тонкости выполняемых вами заданий вечно. Поэтому, даже исключительно ради собственной выгоды, не забывайте добавлять комментарии в создаваемые сценарии.

Замечание. *С точки зрения безопасности и повышения производительности может возникнуть желание удалить комментарии из сценария перед тем, как предоставить его конечным пользователям в Web. Но на всякий случай всегда сохраняйте копию с комментариями в пределах досягаемости.*

Резюме

Эта глава содержит краткий обзор основных возможностей JavaScript — простого, но весьма мощного языка подготовки сценариев, обычно используемого в рамках Web-браузеров. Большинство возможностей этого языка аналогичны возможностям других языков, таких как C или Java. Язык содержит общие программные конструкции — операторы `if`, циклы `while` и функции. Однако JavaScript нельзя назвать упрощенным языком, он предлагает и более сложные механизмы, среди которых составные типы данных, объекты и регулярные выражения. Самая важная составляющая JavaScript — это использование объектов (как создаваемых пользователем, так и встроенных, таких как `Window`, `Navigator` и `Document`). Значительная часть материала книги будет посвящена обсуждению вопросов использования этих объектов. Опытные программисты могут пожелать бегло ознакомиться с содержанием следующих нескольких глав, сосредоточившись на тонкостях отличий JavaScript от других языков программирования. Однако новички должны внимательно прочитать следующие пять глав, чтобы получить твердую основу, на которую можно будет опереться в дальнейшем.