В. В. ЯНЦЕВ

JAVASCRIPT ОБРАБОТКА СОБЫТИЙ НА ПРИМЕРАХ

Учебное пособие

ЛАНЬ

· САНКТ-ПЕТЕРБУРГ · МОСКВА · КРАСНОДАР ·
· 2021 ·

УДК 004
ББК 32.973я73

Я 65 **Янцев В. В.** JavaScript. Обработка событий на примерах : учебное пособие для вузов / В. В. Янцев. — Санкт-Петербург : Лань, 2021. — 176 с. : ил. — Текст : непосредственный.

ISBN 978-5-8114-7559-9

Книг по программированию на JavaScript написано много. Однако подавляющее большинство из них рассказывают об основах и синтаксисе языка. И непропорционально мало книг, которые бы позволили читателю перейти от теории к практике. А такой переход бывает очень непростым. Восполнить этот пробел призвана книга «JavaScript. Обработка событий на примерах». В ней рассказывается о различных событиях, происходящих на страницах сайтов, об обработчиках этих событий, о многообразии вариантов их применения, о технологии создания сценариев на JavaScript.

Рекомендовано в качестве дополнительной литературы для студентов вузов, обучающихся по направлению «Информатика и вычислительная техника».

УДК 004
ББК 32.973я73

Обложка
П. И. ПОЛЯКОВА

Оглавление

1. Введение

1.1. Почему была написана эта книга

Я занимаюсь web-разработками с 2003 года. По моим представлениям – срок весьма солидный. Естественно, что через мои руки прошло множество книг по программированию, которые я с удовольствием прочитал, пополняя свой багаж новыми знаниями или освежая в памяти различные нюансы создания сценариев. Около половины таких книг были посвящены языку программирования JavaScript.

Мне этот язык очень нравится. И не только своим изяществом, синтаксисом и мощью. Нравится, что для программирования на JavaScript не нужно устанавливать дополнительное ПО, что написанную программу без лишних манипуляций можно тут же проверить в браузере. Нравится, что JavaScript позволяет создавать потрясающие сценарии, которые делают web-страницы отзывчивыми к действиям посетителей. Нравится, что язык предоставляет разработчику поистине неограниченные возможности для реализации его идей и задумок.

К сожалению, читая книги по JavaScript, я заметил одну, на мой взгляд, не совсем правильную тенденцию. Дело в том, что самоучителей, рассказывающих об основах и синтаксисе языка, написаны, образно выражаясь, горы. И непропорционально мало книг, которые бы позволили читателю перейти от теории к практике. А такой переход бывает очень непростым. И приходится начинающему разработчику преодолевать множество трудностей, сталкиваться с ошибками, напряженно искать ответы на свои (подчас не самые сложные) вопросы, набивать шишки прежде, чем он, наконец, освоится и начнет писать работающие программы.

Сделаем вывод: в перечне специальной литературы по JavaScript имеется серьезный пробел. Желание хотя бы частично ликвидировать его и послужило стимулом к написанию этой книги.

1.2. Что вы найдете в этой книге

Об обработчиках событий можно прочитать в каждом самоучителе по программированию на JavaScript. Однако мне еще не приходилось видеть изданий, в которых упор делался бы на особенностях их применения. А ведь разнообразие способов и случаев использования обработчиков настолько велико, что об этом можно написать отдельную книгу. Например ту, что вы держите сейчас в руках.

Итак, что вы найдете в книге? Вводную часть, содержащую ряд полезных советов и подсказок. Информацию о том, какие бывают события и как они обрабатываются. Примеры кода сценариев различного назначения. Подробный рассказ о создании локального хостинга. И, конечно, заключение.

Начнем с введения. Уже здесь вы обнаружите немало полезной информации, которую стоит учитывать в дальнейшем при изучении примеров и в процессе создания собственных программ. Поэтому советую не пропускать эту часть.

Второй раздел – рассказ о событиях и обработчиках. Я постарался написать данную часть простым и понятным языком. А для этого отказался от сложных наукоемких рассуждений в пользу легкости и ясности изложения материала.

Примеры. Их 32. Все они написаны в соответствии со стандартами HTML5, CSS3 и ECMAScript. Примеры демонстрируют не только обработку событий, но и то, какие разнообразные, порой неожиданные результаты может получить web-разработчик. Почти все сценарии достаточно простые и не вызовут затруднений в понимании принципов их работы.

Дополнительный, четвертый раздел посвящен созданию локального хостинга на собственном компьютере. О том, нужно ли вам создавать такой хостинг или нет, вы прочтете в параграфе «Необходимо ли дополнительное программное обеспечение?» (пункт 1.5 «Введения»).

1.3. Особенности изложения материала

Каждый сценарий – это действующая программа, которая имеет свое наглядное воплощение на сайте поддержки книги **https://testjs.ru/**. На страницах этого ресурса вы увидите примеры в действии, а также можете посмотреть и скопировать их код. Согласитесь, гораздо интереснее изучать и анализировать не сухие книжные строчки, а живые сценарии, наглядно продемонстрированные на сайте.

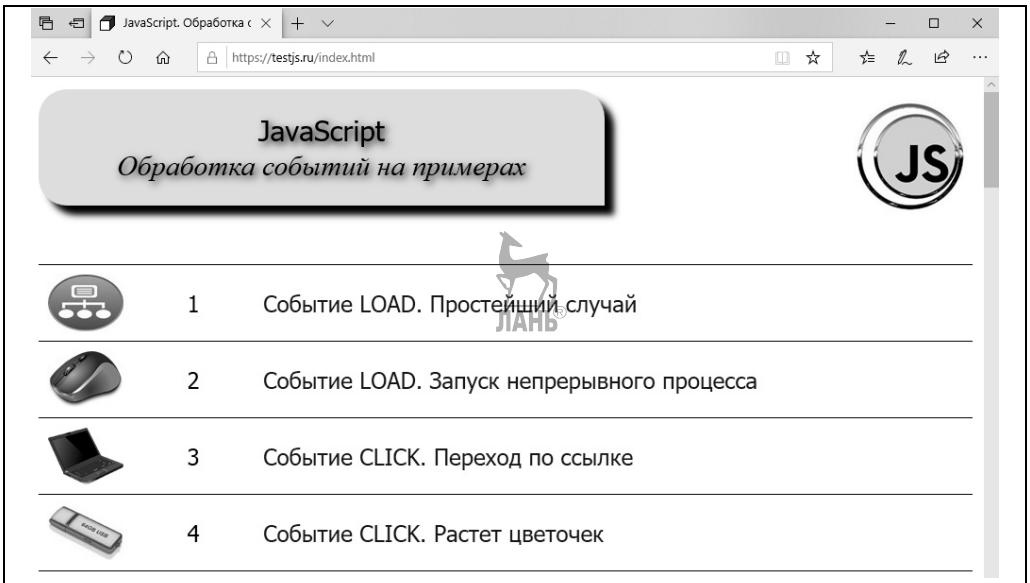

Рис. 1.3.1. Сайт с примерами для книги

Я старался придерживаться методики, при которой сначала рассматриваются все примеры для одного типа событий, потом для второго, потом для третьего и так далее. Но в этом правиле есть исключения. Последовательность нарушается, например, в тех случаях, когда в сценарии использовано 2 или 3 равноценных по степени важности, но разных по типу обработчика. Кроме того, сначала идут сценарии, связанные с событиями мыши, а потом остальные. Но и тут есть одно исключение – это предпоследняя программа. Она оставлена «на десерт» из-за своей сложности.

Количество примеров неодинаково – одним событиям посвящено несколько сценариев, другим 1-2. Это связано с различной долей событий, происходящих на страницах web-документов. Например, щелчки мышью на разных элементах разметки, по моим представлениям, самые частые события. Им посвящено больше всего примеров. А событие отправки формы происходит всегда только с формой. Так что это событие представлено всего одним примером.

Еще один важный момент. Некоторые сценарии в книге производят на начальном этапе вычисления, используя при этом такие параметры, как ширина и высота экрана компьютера в пикселях. Для корректной работы данных примеров необходимо, чтобы окно вашего браузера было развернуто на весь экран.

1.4. Особенности примеров

В этом разделе я познакомлю вас с принципами, по которым эти примеры создавались, отбирались и оформлялись.

Главных принципа три.

Первый. Все примеры написаны по самым современным стандартам, существующим в web-программировании.

Второй. Чтобы сделать примеры кода в этой книге максимально простыми и понятными, из них удалены все лишние элементы разметки и не играющие важной роли стили. Оставлено только самое необходимое.

Третий. Стили и сценарии полностью отделены от разметки и вынесены в заголовочный блок страницы. Хотя такой подход не является обязательным, однако среди программистов он считается хорошим тоном (соответственно, появление стилей или вызовов функций в теле документа – это «моветон»). В идеале, стили и скрипты надо располагать в отдельных файлах. Однако в нашем случае такой подход усложнит читателю жизнь. Согласитесь: изучая примеры на сайте, постоянно переключаться между несколькими документами – утомительное занятие.

Большая часть книги – это примеры кода. Все они проверены в 2 валидаторах Консорциума Всемирной паутины:

– в валидаторе HTML5 – **https://validator.w3.org/**.

– в валидаторе CSS3 – **http://jigsaw.w3.org/css-validator/**.

JavaScript-сценарии проверялись на соответствие стандартам ECMAScript при помощи валидатора **http://beautifytools.com/javascript-validator.php**.

На момент написания книги отклонений от стандартов HTML5, CSS3 и JavaScript в примерах не было.

Также все примеры были проверены в наиболее популярных браузерах – Microsoft Edge, Google Chrome, Mozilla Firefox, Opera и Яндекс.Браузер (а последний, тридцать второй сценарий – в их мобильных версиях). В перечисленных браузерах все примеры работали корректно, за одним исключением, о котором будет сказано ниже.

Несколько подробнее хочется сказать про браузеры. В прежние времена выделялся Internet Explorer. С одной стороны, он предоставлял программисту процентов на 30 больше возможностей, чем его основные конкуренты. С другой стороны, Internet Explorer часто отставал в вопросах поддержки тех возможностей, которые уже были внедрены у конкурентов и признавались стандартными. Даже последняя версия IE – одиннадцатая – не смогла избавиться от некоторых прежних недостатков. Поэтому замена Internet Explorer на Microsoft Edge в операционной системе Windows выглядит очень правильным ходом.

Увы, есть ощущение, что пальму первенства у Internet Explorer теперь старается перехватить Firefox. Представляю, как сейчас негодуют его поклонники, но объективности ради надо об этом сказать.

В книге есть программа перемещения изображения бабочки по странице. Ее первоначальная версия работала во всех браузерах. За одним исключением. И этим исключением стал Firefox. Поэтому пришлось модернизировать код, чтобы добиться работоспособности сценария в этом браузере.

Другой пример. В одном из параграфов я рассказываю, как создать визуальный редактор. Забегая вперед, скажу, что для этого в HTML-страницу необходимо добавить небольшой фрагмент кода. Сделать это можно двумя способами. Все основные браузеры поддерживают оба способа. Firefox – только один.

Еще пример – некорректная работа браузера с событием orientationchange, которому посвящен последний, тридцать второй сценарий. На момент написания книги мобильная версия Firefox не «воспринимала» это событие.

Были и другие трудности, с которыми я сталкивался, проверяя в Firefox примеры, которые без проблем работали в остальных браузерах.

К чему я все это веду? Создавая программы на JavaScript, всегда помните, что их обязательно надо тестировать в «особенном» Firefox.

Отдельно коснусь оформления программ. Часто в технической литературе можно наблюдать довольно странный, на мой взгляд, подход, при котором обычные и фигурные скобки размещаются так, что читать программу довольно затруднительно. Вот пример такого оформления:

```
function nz() {
    document.getElementById("vp").style.visibility="visible";
    let posi=document.getElementById("photo").src;
    for(let i=0; i<mas.length; i++) {
        if(posi.indexOf("city/"+mas[i]+".jpg")>=0) {
            i--;

document.getElementById("photo").src="city/"+mas[i]+".jpg";
            if(i==0) {
```

```
document.getElementById("nz").style.visibility="hidden";
            }
            break;
        }
    }
}
```

Глядя на фрагмент, понимаешь, что начинающему программисту разобраться будет непросто. А представьте, как выглядит код, в котором гораздо больше операторов и уровней их вложенности? Сплошная каша!

Мы в этой книге станем оформлять код, придерживаясь в первую очередь принципа удобочитаемости. Поэтому наш вариант форматирования приведенного выше фрагмента будет выглядеть так:

```
function nz()
{
document.getElementById("vp").style.visibility="visible";
let posi=document.getElementById("photo").src;

for(let i=0; i<mas.length; i++)
   {
   if(posi.indexOf("city/"+mas[i]+".jpg")>=0)
      {
      i--;
      document.getElementById("photo").src="city/"+mas[i]+".jpg";

      if(i==0)
         {
         document.getElementById("nz").style.visibility="hidden";
         }
      break;
      }
   }
}
```

При таком форматировании структура программы сразу видна как на ладони.

Не все программисты внимательно следят за изменениями в стандартах HTML5. Особенно это касается написания тега **<script>**. Обратите внимание! В теге тип **type** не указываем! Этому правилу уже несколько лет, но многие разработчики до сих пор не в курсе. Валидатор Консорциума Всемирной паутины воспринимает указание типа в теге **<script>** как ошибку и выдает предупреждение!

И последнее в этом параграфе – **var** или **let**? Как все-таки правильно объявлять переменную? Дело в том, что заметить разницу между этими объявлениями в простых программах чаще всего невозможно. И в одном, и в другом случае сценарии, приведенные в этой книге, будут работать одинаково.

Но в чем же принципиальное отличие? Вот что на эту тему сообщает Mozilla Foundation на сайте developer.mozilla.org. «Область видимости переменной, объявленной через **var**, это её текущий контекст выполнения. Который может ограничиваться функцией или быть глобальным для переменных, объяв-

ленных за пределами функции.» И еще. «Директива **let** позволяет объявить локальную переменную с областью видимости, ограниченной текущим блоком кода.»

Разница все равно не ясна? Тогда вот вам два коротких фрагмента, которые наглядно покажут это различие.

Создайте 2 html-файла. В один запишите такой код:

```
<script>
window.addEventListener("load", function()
{
var i=1;
if(true)
   {
   var i=2;
   }
alert(i);
});
</script>
```

Во второй:

```
<script>
window.addEventListener("load", function()
{
let i=1;
if(true)
   {
   let i=2;
   }
alert(i);
});
</script>
```

По очереди запустите эти сценарии в браузере. Что вы видите? В первом случае в диалоговом окне появится цифра **2**, а во втором – цифра **1**. А все потому, что вторая переменная **let** объявлена внутри блока **if** и ее значение – **2** – видно только внутри этого блока. Надеюсь, теперь разница ясна.

Опытные разработчики советуют использовать объявления **let**. Мы тоже будем придерживаться такого подхода. Хотя, если вы замените в примерах из этой книги **let** на **var** – в работе сценариев ничего не измениться. Вообще.

1.5. Необходимо ли дополнительное программное обеспечение?

Ответ на данный вопрос вам придется дать самостоятельно. Все зависит от того, как вы хотите изучать примеры.

Если вам достаточно сайта поддержки книги, если все сценарии вы будете оценивать на основе примеров с сайта, никакого дополнительного программного обеспечения не нужно.

Если хотите копировать примеры кода на свой компьютер и экспериментировать с ними, нужно будет установить http-сервер Apache. Почему? Дело в том, что часть программ запустится на вашем ПК только с использованием сервера. Установка Apache – это, по сути, создание простейшего локального хостинга.

Среди примеров сценариев есть такие, которые должны взаимодействовать с серверными программами. Однако, для упрощения, эти взаимодействия только имитируются, поэтому серверные скрипты нам не нужны. А значит, чтобы запустить все примеры на вашем компьютере, достаточно только сервера, где будут храниться html-страницы.

Хотите развиваться, расти дальше, создавать и тестировать программы «по-взрослому»? Установите весь предложенный комплект дополнительного программного обеспечения для создания на своем компьютере полноценного локального хостинга. То есть добавьте к серверу интерпретатор PHP. Так у вас появится возможность создавать и проверять серверные приложения, во многих случаях необходимые для взаимодействия со страницами, загруженными на ПК пользователя.

Кстати, обратите внимание: локальный хостинг работает без подключения к Интернету!

1.6. Благодарности

Написание книги – дело не самое простое. Сначала должна появиться идея, затем нужно сформировать концепцию, потом написать и протестировать несколько десятков программ и лишь после этого можно садиться за письменный стол. Но и это еще не все. Рукопись отправляется в издательство, там с ней работают редакторы и корректоры, печатники в типографии, а также курьеры, которые доставляют тираж в редакцию и на прилавки книжных магазинов. Автор выражает им искреннюю благодарность! Кроме того, неоценимую помощь мне оказали многие люди, имен которых вы не увидите на обложке. Их я хотел бы особо отметить на страницах этой книги.

Большое-пребольшое спасибо:

- заведующей редакцией литературы по информационным технологиям и системам связи издательства «Лань» Ольге Евгеньевне Гайнутдиновой – за оперативную, профессиональную работу и доброжелательное отношение к автору;
- моему давнему другу Василию Регентову за многолетние предоставления «площадки» для моих компьютерных экспериментов и терпеливое решение возникающих иногда проблем;
- программистам с различных форумов, добровольно помогавшим мне тестировать сценарии в самых разнообразных режимах;
- и конечно, моей любимой жене Светлане, которая мужественно и безропотно годами взирала на спину мужа, ежедневно сидящего за ноутбуком.

2. События и их обработка

2.1. События

События на страницах сайта происходят почти непрерывно. Загружаете ли вы документ, нажимаете кнопку мыши, прокручиваете страницу вниз или просто перемещаете указатель – все это для браузера события. Некоторые из них можно наблюдать «в прямом эфире», другие происходят «за кадром». Например, при достижении указателя определенной области на странице может запускаться процесс, о котором посетитель даже не догадывается.

Все события идентифицируются по типу (иногда говорят не тип события, а имя события). Мы на страницах этой книги рассмотрим следующие из них:
load
click
mouseover
mouseout
mousemove
mousedown
mouseup
dragstart
dragover
drop
resize
focus
blur
submit
change
scroll
paste
keyup
select
mouseenter
mouseleave
orientationchange.

Все события происходят с объектами или элементами документа. Например, события **load**, **scroll** и **orientationchange** чаще всего происходят в объекте window, а остальные из перечисленных связаны с элементами страницы (реже с объектами). Расскажем о событиях подробнее.

2.2. Типы событий

Событие **load** происходит в большинстве случаев при загрузке страницы в браузер. Оно генерируется сразу после того, как html-разметка и рисунки (картинки, изображения, фото) полностью отображаются на экране компьютера.

Однако такое же событие происходит и при получении ответа от сервера, на котором стоит программа, вызываемая с применением технологии Ajax.

Событие **click** возникает при щелчке кнопкой мыши на элементе разметки страницы и даже на визуально пустом месте документа. Это могут быть ссылки, кнопки, изображения, текстовые блоки, слои или тело документа.

Событие **mouseover** происходит, когда указатель мыши перемещается внутрь границ элемента.

Событие **mouseout** противоположно предыдущему и возникает, когда указатель мыши покидает границы элемента.

Событие **mousemove** генерируется в продолжение всего времени, пока указатель мыши перемещается над элементом.

Событие **mousedown** происходит при нажатии кнопки мыши, когда ее указатель находится над элементом.

Событие **mouseup** возникает при отпускании кнопки мыши (при этом указатель должен находиться над элементом, для которого генерируется событие).

Событие **dragstart** происходит, когда посетитель начинает перетаскивать элемент по странице.

Событие **dragover** возникает в процессе перетаскивания элемента по странице.

Событие **drop** происходит в момент отпускания перетаскиваемого элемента в точке назначения.

Событие **resize** возникает при изменении размеров окна браузера.

Событие **focus** случается при получении элементом фокуса (например, когда курсор будет установлен в текстовом поле).

Событие **blur** происходит, когда элемент теряет фокус.

Событие **submit** сопровождает отправку формы, например, нажатием кнопки.

Событие **change** происходит на элементах **<select>**, **<input>** и **<textarea>**, когда меняются их значения.

Событие **scroll** возникает при прокручивании страницы в окне браузера или во фрейме.

Событие **paste** происходит в момент вставки содержимого буфера обмена в элемент документа в позицию курсора (например, в однострочное или многострочное текстовые поля).

Событие **keyup** возникает в момент отпускания после нажатия любой кнопки на клавиатуре. Кроме того, с клавиатурой связаны события **keydown** (в момент нажатия клавиши) и **keypress** (клавиша нажата и отпущена). Однако из-за особенностей обработки двух последних событий браузерами мы не будем использовать их в сценариях (например, событие **keypress** обрабатывается не в момент его происхождения, а с отставанием на один шаг).

Событие **select** происходит в текстовых полях или в текстах на странице документа при выделении в них какого-либо фрагмента или всего текста в целом.

Событие **mouseenter** очень похоже на **mouseover**. Отличие в том, что событие **mouseenter** не всплывающее и не отменяемое (из этого понятно, что **mouseover** всплывающее и отменяемое).

Событие **mouseleave**, соответственно, похоже на **mouseout**. Разница такая же, как и в случае, описанном в предыдущем определении. Событие **mouseleave** не всплывающее и не отменяемое.

Событие **orientationchange** характерно только для браузеров мобильных устройств. Оно происходит при смене ориентации гаджета с вертикальной на горизонтальную и обратно.

2.3. Обработка событий

Обработчик события – это функция, которая запускается в ответ на то или иное действие посетителя (загрузка страницы – это тоже результат действий клиента, который ввел в браузере адрес страницы или перешел на нее по ссылке). Естественно, что разработчик сайта заранее выбирает, какие события на странице должны вызвать программный отклик, а какие нет.

Надо сказать, что в самом простом случае обработка может производиться не функцией, а, например, математическим выражением. Скажем, вот так:

```
let x=1;
window.onload=x++;
```

Однако вряд ли такой способ применим в реальных сценариях. Практически во всех случаях реакцию на событие прописывают в одной или нескольких функциях. Они могут выполняться параллельно или последовательно друг за другом.

2.4. Регистрация обработчиков

Чтобы вызвать обработчик в ответ на какое-либо событие, надо добавить к имени события префикс (приставку) **on**, например, **onload**, **onclick**, **onfocus**, **onsubmit** и так далее (так формируется имя свойства обработчика). А затем указать анонимную или именованную функцию, которая будет «откликаться» на то или иное событие.

Обработчики событий окна **load**, **scroll** и **orientationchange** зарегистрировать проще всего. Если, например, для события **load** предусмотрен только один обработчик, то достаточно написать

```
window.onload=function()
{
...
};
```

или

```
window.onload=func;

function func()
{
```

```
...
}
```

и соответствующая функция будет запущена автоматически сразу после загрузки страницы. Аналогично ситуация обстоит со **scroll** и **orientationchange** – функции для их обработки регистрируются на уровне событий окна и запускаются одновременно с началом прокрутки страницы в окне браузера или после смены ориентации мобильного устройства с вертикальной на горизонтальную и обратно.

Если после наступления событий **load**, **scroll** или **orientationchange** необходимо запустить сразу несколько функций, то это можно сделать, например, так:


```
window.onload=function()
{
func();
doc();
res();
};
```

В случае остальных событий (происходящих с отдельными элементами или объектами) тоже необходима предварительная регистрация обработчиков, чтобы браузер знал: в ответ на какие события он должен запускать ту или иную функцию.

(Мы не рассматриваем ситуации, когда вызов обработчика события помещается непосредственно в элемент разметки, например так: **<button onclick="func()">**. Считается хорошим тоном отделять программный код от разметки, а размещение вызова функции внутри тега элемента не соответствует этому принципу.)

Способы регистрации обработчиков.

1. Регистрация обработчика как ответ на событие **load**. Например, так:


```
window.onload=function()
{
document.getElementById("im").onclick=function()
   {
   ...
   };
};
```

или так:

```
window.onload=function()
{
document.getElementById("im").onclick=func;
};
```

2. Регистрация сразу нескольких обработчиков. Пример:

```
window.onload=function()
```

```
{
document.getElementById("im").onclick=func;
document.getElementById("im").onmouseout=doc;
document.getElementById("im").onmouseover=res;
};
```

3. Комбинированная регистрация – когда одни функции регистрируются, а другие запускаются без предварительной регистрации из уже зарегистрированных. Может выглядеть так:

```
window.onload=function()
{
document.getElementById("im").onclick=func;
};

function func()
{
...
sum();
}
```

4. Еще один комбинированный случай – регистрация обработчика и одновременный запуск внешней функции:

```
window.onload=function()
{
document.getElementById("im").onclick=func;
sum();
};
```

5. Регистрация обработчиков методом **addEventListener**. Метод принимает три аргумента. Два из них обязательные. Первый – тип события. Второй – функция, «реагирующая» на событие, или имя внешней функции. Третий аргумент – булево значение, которое допустимо не указывать. Методом **addEventListener** можно регистрировать обработчики так:

```
window.addEventListener("load", start);
window.addEventListener("load", func);
```

или так:

```
window.addEventListener("load", start);
window.addEventListener("load", function()
{
...
});
```

и даже так:

```
window.addEventListener("load", start);
window.addEventListener("load", function()
{
```

```
document.getElementById("im").addEventListener("click", func);
document.getElementById("im").addEventListener("mouseout", doc);
document.getElementById("im").addEventListener("mouseover", res);
});
```

Мы разобрали варианты применения **addEventListener** на примере регистрации обработчика для события **load**. Но этим методом можно регистрировать обработчики любых других событий. Например, так:

```
window.addEventListener("load", function()
{
document.getElementById("im").addEventListener("click", func);
});
```

или так:

```
window.addEventListener("load", function()
{
document.getElementById("im").addEventListener("click", function()
   {
   ...
   });
});
```

и даже вот так:

```
window.addEventListener("load", function()
{
document.getElementById("im").addEventListener("click", func);
});

function func()
{
...
document.getElementById("te").addEventListener("click", doc);
}

function doc()
{
...
}
```

Наконец, есть еще один способ регистрации обработчика методом **addEventListener**. Он в определенном смысле проще. Чтобы понять его, посмотрите пример вот такой страницы:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Регистрация обрабочика</title>
</head>

<body>
...
```

```
</body>
</html>

<script>
document.getElementById("im").addEventListener("click", func);

function func()
{
...
}
</script>
```

Здесь сценарий расположен ниже тела документа. В этом случае для регистрации обработчика методом **addEventListener** не нужно заключать его в блок

```
window.addEventListener("load", function()
{
...
});
```

Однако расположение сценария после тела документа лично я считаю как минимум спорной практикой. На мой взгляд, сценарии или вызовы внешних файлов со сценариями лучше всего располагать в заголовочной части документа. Во всяком случае, во всех примерах этой книги код JavaScript будет размещаться внутри тегов **<head>...</head>**. Однако я не настаиваю на такой модели наполнения страниц. Разрабатывая собственные программы, вы можете воспользоваться технологией размещения скриптов, показанной в последнем примере.

Думаю, вы уже поняли, что комбинаций регистрации обработчиков может быть много.

Зарегистрированные обработчики вызываются автоматически в момент наступления события, к которому они «приписаны». Ну а дальше на странице сайта происходит то, о чем будет рассказано в следующем разделе.

3. Обработка событий на примерах

3.1. Событие load. Простейший случай

Любая книга по программированию излагает основную тему, подкрепляя пройденный материал сначала наиболее простыми примерами. Будем следовать этой традиции и мы. Поэтому наш первый сценарий окажется максимально легким для понимания.

Прежде чем на странице случится какое-либо событие, необходимо, чтобы произошло самое главное – web-документ должен быть загружен браузером. Это наиболее важный акт программы, которой предстоит обрабатывать другие события. Как мы убедились в п. 2.4, свойство **onload** окна **window** часто применяют в документе для регистрации обработчиков. Однако событие **load** ценно не только этим. В данном параграфе вам предстоит убедиться, что загрузка документа может сразу (без предварительной регистрации обработчика) вызывать важные манипуляции с содержимым страницы (то есть запускать обработчик автоматически непосредственно после загрузки страницы).

В Интернете есть много сервисов, с помощью которых легко определить разрешение экрана компьютера, если оно вам не известно. Вообще, сервис – это громко сказано. Услышав такой солидный термин, многие представляют себе набор сложных программ, которые выполняют целую череду важных операций с данными. На самом деле все гораздо проще. Чтобы узнать разрешение монитора, достаточно написать очень короткий сценарий на JavaScript. Как – мы сейчас узнаем.

В первую очередь нам необходимо создать web-документ. Делать это мы будем, помня о том, что все наши примеры строго следуют стандартам HTML5.

Сначала напишем заголовочную часть страницы и тело документа. Они будут выглядеть так:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие load</title>
</head>

<body>

</body>
</html>
```

Добавим в тело документа элементы разметки, благодаря которым данные о мониторе будут выводиться на страницу:

```
Размер экрана - <span id="si"></span>
```

Пришло время написать первый сценарий на JavaScript.

Начнем с того, что поместим в заголовочную часть документа теги для размещения кода:

```
<script>

</script>
```

Теперь настала очередь получить данные о разрешении монитора. Для этого воспользуемся «встроенным» в JavaScript объектом **screen**. Он содержит информацию о размерах экрана компьютера или мобильного устройства. Из **screen.width** разработчик получает ширину экрана, из **screen.height** – высоту. Объявляем две переменные и присваиваем им значения, полученные из объекта **screen** (в пикселях):

```
let w=screen.width;
let h=screen.height;
```

Выводим результат на экран:

```
document.getElementById("si").innerHTML=w+" x "+h;
```

Все.

Если собрать разрозненные фрагменты воедино, получится такая страница:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие load</title>

<script>
window.addEventListener("load", function()
{
let w=screen.width;
let h=screen.height;
document.getElementById("si").innerHTML=w+" x "+h;
});
</script>
</head>

<body>

Размер экрана - <span id="si"></span>

</body>
</html>
```

Итак, как только страница загрузилась полностью, произошло событие **load**, которому мы приписали один обработчик – анонимную функцию. Внутри функции выполнено определение размеров монитора, после чего эти параметры были выведены на экран. Кстати, физически данный процесс настолько кратко-

временен, что человеческий глаз не в состоянии заметить интервала между загрузкой страницы и появлением чисел в ее теле. Для нас с вами оба действия выглядят одновременными – настолько молниеносно срабатывает функция (впрочем, JavaScript вообще работает быстро).

Чтобы убедиться в этом, вы можете посмотреть данный пример на сайте поддержки книги на странице **https://testjs.ru/p1.html**.

Нажав на ссылку «Код примера», вы увидите тот же самый сценарий, который мы разобрали только что.

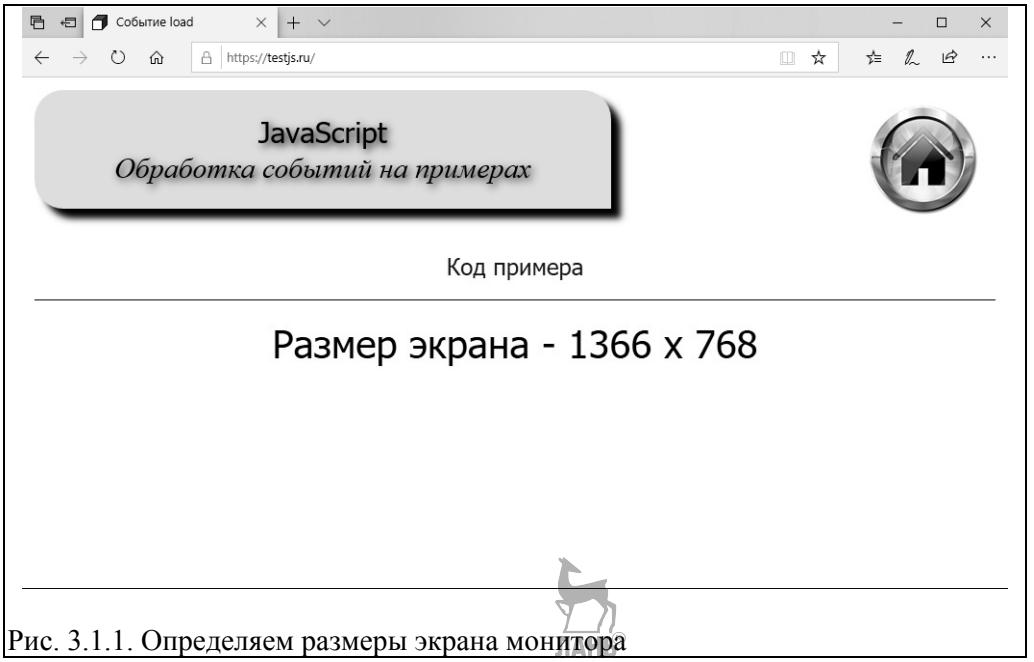

Рис. 3.1.1. Определяем размеры экрана монитора

Напомню, что все демонстрационные варианты в книге «облагорожены» – к ним добавлено оформление. В коде примера, который мы разбирали выше или который вы откроете, пройдя по ссылке на сайте, все оформление исключено – оставлено только самое необходимое.

3.2. Событие load. Запуск непрерывного процесса

В предыдущем параграфе мы с вами выполняли запуск «одноразовой» функции. Однако событие **load** может использоваться также для запуска непрерывного процесса, который будет продолжаться до тех пор, пока открыта страница с данной программой.

В нашем новом примере, как и в предыдущем случае, загрузка документа приведет к автоматическому запуску обработчика без его предварительной регистрации. Работу этого сценария вы можете видеть на сайте книги на странице

https://testjs.ru/p2.html или на рисунке 3.2.1. Программа эта – таймер, показывающий текущие дату и время с точностью до секунды.

Как и в предыдущем случае начнем с создания заголовочного блока и тела документа:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие load</title>
</head>

<body>

</body>
</html>
```

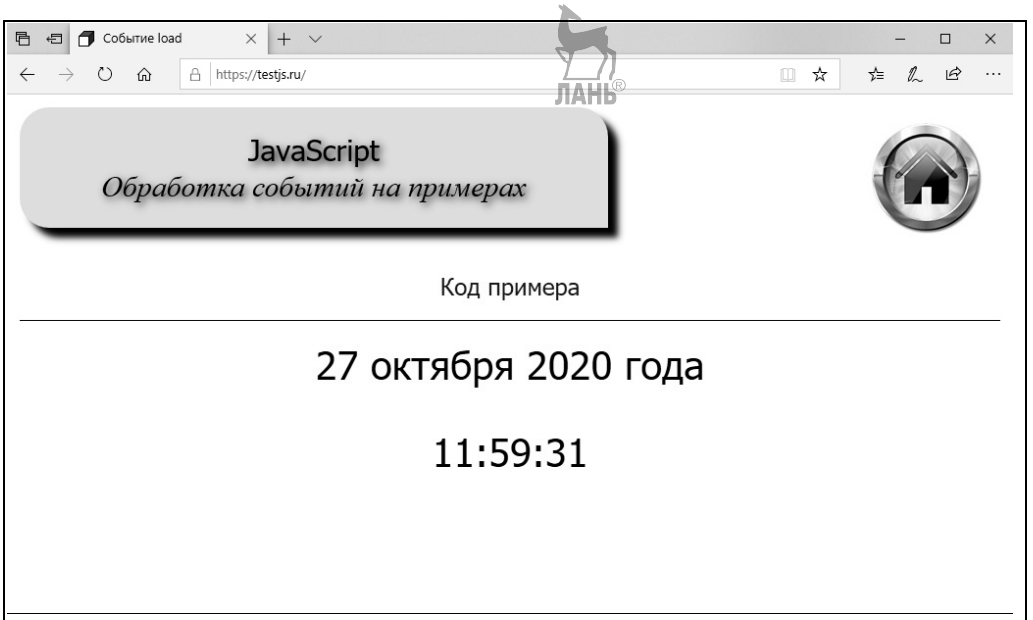

Рис. 3.2.1. Таймер на странице сайта

Условимся, что данные о дате и времени мы станем выводить в две строки: в верхней – дата, в нижней – время. Для этого в тело документа впишем два тега **p**:

```
<p id="god"></p>
<p id="min"></p>
```

Строка с **id="god"** нужна для показа даты, строка с **id="min"** – для индикации времени.

Теперь сам таймер. Его сценарий не так прост, как пример с определением разрешения экрана.

Добавляем в головную часть документа теги для нашего скрипта:

```
<script>

</script>
```

Вписываем внутри тегов строку, вызывающую обработчик события **load**:

```
<script>
window.addEventListener("load", tim);
</script>
```

Итак, функция-обработчик получила название **tim** (от *англ.* time – время). Начнем разбираться, что она делает.

Для определения даты и времени в JavaScript существует объект **Date**. Он содержит информацию о количестве миллисекунд, прошедших с 1 января 1970 года.

Создадим три новых экземпляра этого объекта:


```
let den=new Date();
let mes=new Date();
let god=new Date();
```

Если разбирать их по порядку, сверху вниз, то эти экземпляры нужны для вычисления дня, месяца и года. Поскольку эстетичнее отображать месяц не цифрами, а привычными нам названиями, добавим новую переменную


```
let mon="";
```

и новый массив

```
let arr=["января", "февраля", "марта", "апреля", "мая", "июня",
"июля",
                "августа", "сентября", "октября", "ноября", "де-
кабря"];
```

при помощи которых мы определим соответствие номера месяца его названию. Для этого напишем такой цикл:

```
for(let i=0; i<12; i++)
   {
   if(mes.getMonth()==i)
      {
      mon=arr[i];
      break;
      }
   }
```

В нем мы делаем 12 проходов – по количеству месяцев в году. При каждом проходе проверяем, равен ли номер месяца номеру прохода:

```
if(mes.getMonth()==i)
```

Метод **getMonth()** объекта **Date** вычисляет номер текущего месяца (январь имеет номер «0»). Таким образом, применив этот метод к новому объекту **mes** мы получим номер месяца.

Как только номер месяца совпадет с номером прохода цикла, переменной **mon** будет присвоено соответствующее строковое значение из массива

```
mon=arr[i];
```

после чего цикл будет остановлен инструкцией **break**.

Теперь соберем дату в единое целое и выведем ее на страницу:

```
document.getElementById("god").innerHTML=den.getDate()+
                                    "   "+mon+"   "+god.getFullYear()+"
года";
```

Обратите внимание: в этом примере дважды сделан перенос части кода на вторую строку (из-за недостатка ширины страницы). В реальном сценарии элементы массива и код вывода данных на страницу записываются одной строкой. Запомните правило: все переносы строк кода существуют только в их типографском воспроизведении. Если вы в дальнейшем столкнетесь с подобной ситуацией, учитывайте данный аспект.

Прокомментируем, что мы написали. Метод **getDate()** объекта **Date** вычисляет текущее число, а **getFullYear()** – текущий год в формате XXXX (например, 2021). Применив эти методы к переменным **den** и **god**, мы получаем искомые параметры. Название месяца у нас уже имеется. Соединяем эти данные пробелами и выводим на страницу в строку с **id="god"**.

Следующий этап – определение времени. Ему посвящена вторая часть функции **tim**.

Вновь создаем три новых экземпляра объекта **Date**:

```
let cha=new Date();
let min=new Date();
let sek=new Date();
```

Они нужны для вычисления часа, минут и секунд. Первым делом мы определяем количество минут методом **getMinutes()**:

```
let m=min.getMinutes();
```

Затем переводим числовой показатель в строку:

```
let mi=""+m;
```

Теперь вычисляем длину строки с показаниями минут:

```
mi.length
```

Если число минут состоит из одной цифры, добавляем к ней цифру «0»:

```
if(mi.length==1)
   {
   m="0"+m;
   }
```

Благодаря этому получается, что таймер все время показывает двухзначное число минут, что выглядит аккуратнее. Кроме того, теперь таймер имеет стабильный внешний вид, а не смещается в стороны, когда число минут меняется с одной цифры на две.

Такие же манипуляции проделываем и с показаниями секунд:

```
let s=sek.getSeconds();
let se=""+s;
if(se.length==1)
   {
   s="0"+s;
   }
```

Методом **getSeconds()** определяем количество секунд, переводим показания в строку и добавляем «0», если строка состоит из одной цифры.

Когда предварительная подготовка завершена, выводим данные на страницу в строку с **id="min"**:

```
document.getElementById("min").innerHTML=cha.getHours()+":"+m+":"+s;
```

Как видите, текущий час мы вычисляем методом **getHours()** непосредственно при выводе показаний:

```
cha.getHours()
```

Однако сценарий еще не завершен. Если мы оставим все, как есть, то после загрузки страницы увидим то время, в которое произошла эта загрузка. И все – показания таймера больше не будут меняться. Чтобы заставить работать таймер в непрерывном режиме, добавим еще одну строку в самый конец нашей функции:

```
window.setTimeout(tim, 1000);
```

Здесь мы использовали метод **setTimeout** окна **window**, который позволяет запустить тот или иной процесс через определенный промежуток времени. В нашем случае функция **tim** запустится вновь через 1 секунду. Произойдет новое вычисление времени, а затем опять «сработает» строка

```
window.setTimeout(tim, 1000);
```

и функция запустится в очередной раз. Словом, получится замкнутый цикл, который будет продолжаться до тех пор, пока страница загружена в браузер. Благодаря этому таймер «оживает» и начинает непрерывно показывать текущие дату и время с точностью до секунды.

Вот теперь можно подвести итог. Соберем отдельные строки кода в единое целое:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие load</title>

<script>
window.addEventListener("load", tim);

function tim()
{
let den=new Date();
let mes=new Date();
let mon="";
let  arr=["января",  "февраля",  "марта",  "апреля",  "мая",  "июня",
"июля",
          "         августа",    "сентября",   "октября",   "ноября",
"декабря"];

for(let i=0; i<12; i++)
   {
   if(mes.getMonth()==i)
      {
      mon=arr[i];
      break;
      }
   }

let god=new Date();
document.getElementById("god").innerHTML=den.getDate()+
                                    "    "+mon+"    "+god.getFullYear()+"
года";

let cha=new Date();
let min=new Date();
let sek=new Date();
let m=min.getMinutes();
let mi=""+m;
if(mi.length==1)
   {
   m="0"+m;
   }
let s=sek.getSeconds();
let se=""+s;
if(se.length==1)
   {
   s="0"+s;
   }
document.getElementById("min").innerHTML=cha.getHours()+":"+m+":"+s;

window.setTimeout(tim, 1000);
```

```
}
</script>
</head>

<body>

<p id="god"></p>
<p id="min"></p>

</body>
</html>
```

В первых двух примерах мы подробно разбирали каждый шаг по созданию страницы и написанию сценария. Думаю, читатели уже поняли, что во всех остальных примерах книги будет тот же алгоритм: сначала создаем головную часть и тело документа, потом добавляем в тело необходимое содержимое, затем вписываем в заголовочную часть теги <script> </script> и уже в них размещаем наш сценарий. Поскольку в каждом случае последовательность одинакова, мы больше не будем заострять на ней внимание.

3.3. Событие click. Переход по ссылке

На мой взгляд, событие **click** относится к разряду тех, что наиболее часто «подвергаются» обработке. Щелчки на ссылках, изображениях, кнопках – типичное явление для многих-многих-многих страниц. Поэтому вполне естественно, что эти события у нас идут под вторым номером после **load**. И если последнее далеко не во всех случаях сразу запускает какой-то процесс, то событие **click**, наоборот, почти всегда приводит или к изменению «внешнего вида» документа, или к изменению содержимого страницы, или к загрузке новой.

Для события **click** у нас будет наибольшее количество примеров – 6. Пять из них мы рассмотрим по порядку в следующих пяти параграфах, а шестой – ближе к концу книги, так как он будет работать в связке с другим обработчиком, очередь до которого дойдет несколько позже.

И опять мы начнем с самого простого случая. Вы, возможно, не раз наблюдали такую картину при посещении некоторых сайтов: на странице есть ссылка на сторонний ресурс. Вы щелкаете на этой ссылке – и тут появляется диалоговое окно с вопросом: «Вы действительно хотите покинуть этот сайт?» В окне – две кнопки: «ОК» (то есть «да») и «Отмена» (в Яндекс.Браузере кнопки имеют надписи «Да» и «Нет»). В зависимости от того, какую из них вы нажмете, вы либо останетесь на прежней странице, либо перейдете на другой ресурс. По сути, такой механизм сродни просьбе администратора текущего сайта: «А может не стоит покидать эти прекрасные страницы? Оставайтесь!»

В нашем первом примере мы реализуем подобный механизм. Только вопрос наш будет звучать несколько иначе: «Вы действительно хотите покинуть эту страницу?» И ссылка будет вести не на сторонний ресурс, а на главную страницу сайта поддержки книги. Пример демонстрируется в действии на странице **https://testjs.ru/p3.html**.

Итак, у нас есть текстовая ссылка (рис. 3.3.1). Ее мы создали, поместив в тело документа вот такой фрагмент:

```
<a href="#" id="con">Тестовая ссылка</a>
```

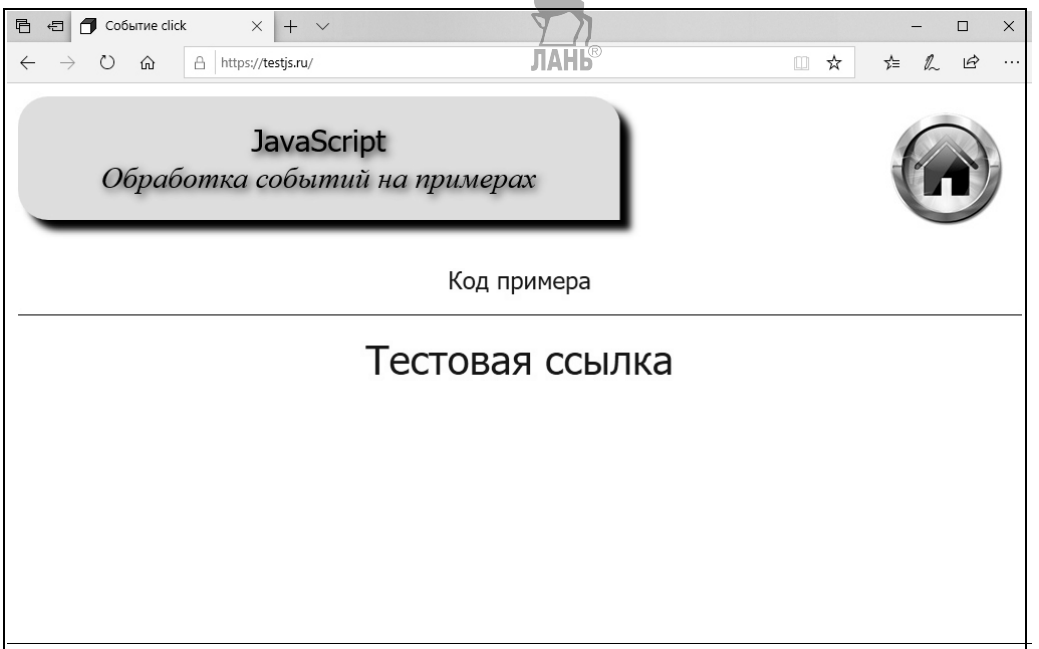

Рис. 3.3.1. Страница с размещенной на ней ссылкой

Как видите, ссылка не имеет адреса – он появится в нужный момент в коде сценария. А вот и сам сценарий:

```
window.addEventListener("load", function()
{
document.getElementById("con").addEventListener("click", function()
   {
   if(window.confirm("Вы действительно хотите покинуть эту
страницу?")==true)
      {
      window.location="http://testjs.ru/";
      }

   return false;
   });
});
```

После загрузки страницы регистрируется обработчик события **click**, роль которого выполняет анонимная функция:

```
document.getElementById("con").addEventListener("click", function()
```

```
{
...
});
```

Когда посетитель нажимает ссылку, начинается выполнение сценария – появляется диалоговое окно **window.confirm** с вопросом: «Вы действительно хотите покинуть эту страницу?» (рисунок 3.3.2). Окно «ждет», какой выбор сделает посетитель. Если он передумал покидать страницу и нажимает кнопку «Отмена», окно закрывается и дальше ничего не происходит. Поэтому в окне браузера по-прежнему остается та же страница.

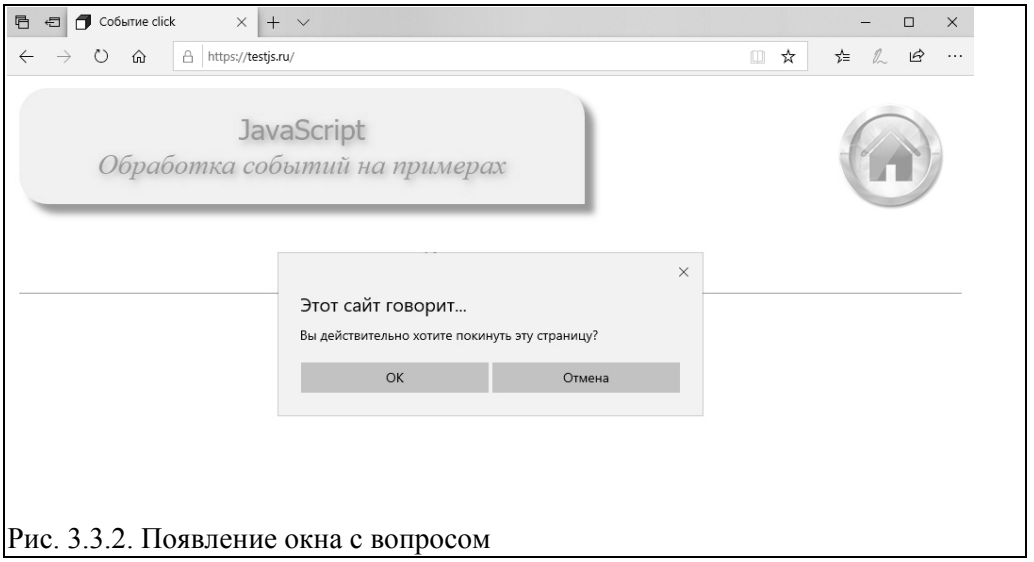

Рис. 3.3.2. Появление окна с вопросом

Если посетитель подтверждает желание уйти на другой сайт, условие

```
if(window.confirm("Вы действительно хотите покинуть эту
страницу?")==true)
```

оказывается истинным и запускается механизм перенаправления:

```
window.location="http://testjs.ru/";
```

Здесь мы использовали объект **location** окна **window**. Данный объект содержит адрес уже открытой страницы. Однако, меняя значение свойства объекта, можно изменить адрес страницы, загруженной браузером. Что мы и делаем. Таким образом

```
window.location="http://testjs.ru/";
```

это, фактически, команда браузеру изменить адрес загруженного документа. В результате мы оказываемся на домашней странице сайта книги.

Отдельно остановимся на строке

```
return false;
```

Для чего она? Помните – адрес ссылки у нас выглядит так: **href="#"**? В наших примерах аналогичная запись попадется еще не раз. Такую «конструкцию» применяют, когда надо показать посетителю сайта, что перед ним ссылка и ее можно нажать. Только нажатие приводит не к переходу на другую страницу, а к запуску программы на JavaScript (что и требуется в нашем случае). При клике на такой ссылке страница не станет перезагружаться. Но зато в конце адресной строки браузера появится лишний символ #, что исказит реальный адрес, а сама страница прокрутится вверх. Чтобы избавится от этих нежелательных эффектов, в конце функции, обрабатывающей клик на такой ссылке, необходимо добавить строку **return false;**. Что мы и сделали.

И напоследок полный код страницы с примером:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие click</title>

<script>
window.addEventListener("load", function()
{
document.getElementById("con").addEventListener("click", function()
   {
   if(window.confirm("Вы действительно хотите покинуть эту
страницу?")==true)
      {
      window.location="http://testjs.ru/";
      }

   return false;
   });
});
</script>
</head>

<body>

<a href="#" id="con">Тестовая ссылка</a>
</body>
</html>
```

3.4. Событие click. Растет цветочек

Следующий пример чуть-чуть сложнее, но не намного. Мы рассмотрим ситуацию, когда событие **click** произошло на изображении.

Обычно для таких случаев авторы предлагают сценарии, в которых при клике на картинке появляется ее увеличенное изображение, которое показыва-

ется на темном (чаще) или на светлом (реже) фоне, закрывающем все окно браузера. Так, например, демонстрируются большие фото в популярных социальных сетях. У нас тоже будет нечто похожее, но только позже. Сейчас мы пойдем другим путем.

Зайдите на страницу **https://testjs.ru/p4.html** сайта книги. Там вы увидите снимок цветка. В исходном состоянии изображение выглядит так, как показано на рисунке 3.4.1. Наведите указатель мыши на фотографию. Внешний вид указателя изменится со стрелки на «руку», показывая тем самым, что это ссылка, которую можно кликнуть. Что мы и сделаем. На наших глазах изображение плавно увеличится в 2,5 раза (рис. 3.4.2). Если теперь снова щелкнуть на картинке, она уменьшится до исходного состояния. Можно даже немного побаловаться, быстро щелкая мышью на изображении еще до того, как оно «вырастет» до максимума и «сожмется» до исходного состояния. В этом случае снимок будет плавно, но быстро менять размеры, немного увеличиваясь и уменьшаясь около некоего среднего положения.

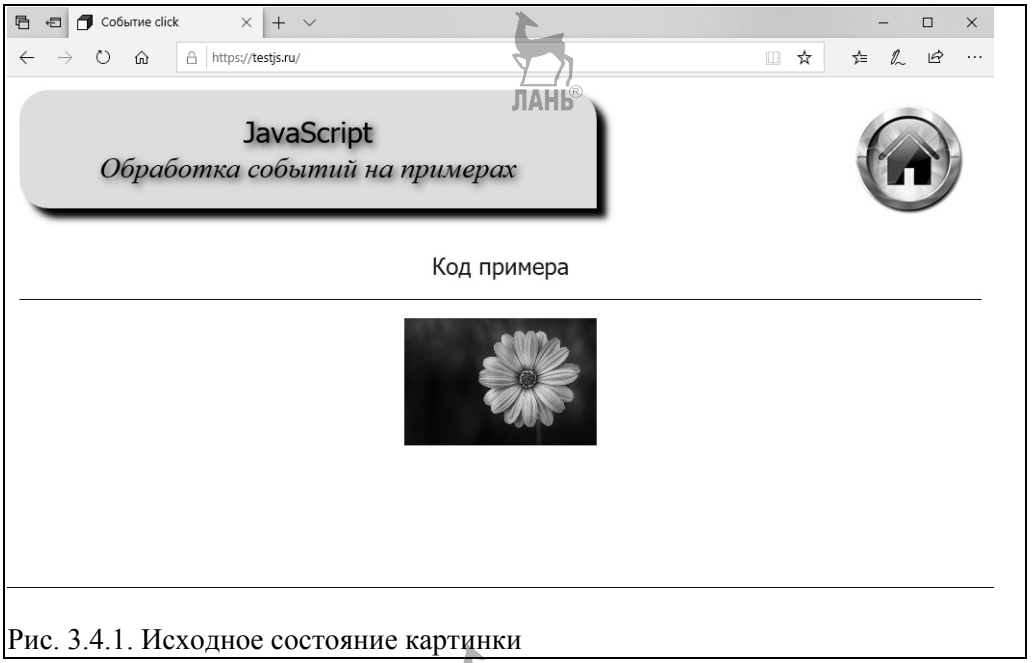

Рис. 3.4.1. Исходное состояние картинки

Ну а теперь о серьезном. Как устроена эта программа?

В теле документа есть фотография цветка:

```
<a href="#"><img id="flo" src="pict/flower.jpg" alt="Фото"></a>
```

Изображение «обернуто» в ссылку, чтобы при наведении указателя мыши на фото посетитель понимал, что на нем можно щелкнуть.

Обратите внимание: в теге **img** присутствует альтернативный текст **alt="Фото"**, необходимый на случай, когда у посетителя отключена демонстрация картинок. Согласно спецификации HTML5 наличие этого атрибута явля-

ется обязательным для изображений. Мы с вами неукоснительно следуем принятым стандартам.

Зададим начальный размер снимка, поместив в заголовочную часть документа настройки стиля:

```
<style>
#flo {width: 200px;}
</style>
```

Увеличением и уменьшением фотографии будет управлять один и тот же обработчик. Зарегистрируем его:

```
window.addEventListener("load", function()
{
document.getElementById("flo").addEventListener("click", tra);
});
```

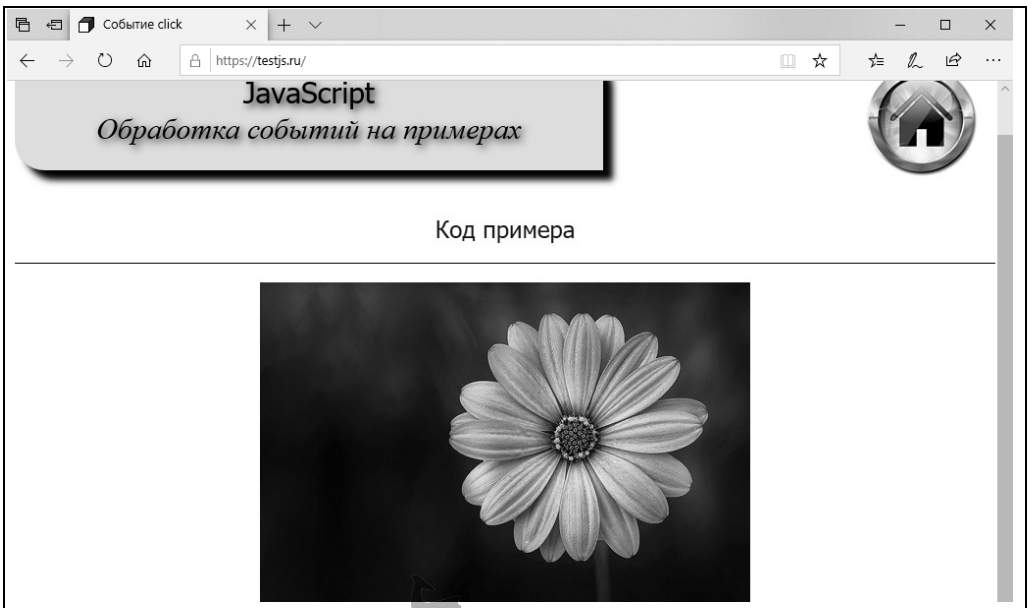

Рис. 3.4.2. Изображение цветка увеличилось в два с половиной раза

Объявим переменную **i**:

```
let i=1;
```

Она будет выполнять роль идентификатора, который указывает направление изменения размеров: увеличиваться картинке или уменьшаться. Как действует эта переменная, мы разберемся ниже.

Для плавного изменения фото мы используем свойство стиля **transition**. Оно определяет особенности перехода элемента из одного состояния в другое. Зададим изменяемый параметр и время перехода:

```
transition="width 1s"
```

Как видите, у снимка меняется ширина, что автоматически приводит к пропорциональному изменению высоты. Время перехода – 1 секунда.

Когда значение переменной **i** равно единице, в результате события **click** происходит увеличение изображения с 200 пикселей до 500 пикселей:

```
if(i==1)
   {
   document.getElementById("flo").style.transition="width 1s";
   document.getElementById("flo").style.width="500px";
   i=2;
   }
```

Внутри блока в первой строке задаются условия перехода, а во второй – конечный параметр, по достижении которого переход должен завершиться.

После этого в третьей строке значение идентификатора **i** меняется с 1 на 2:

```
i=2;
```


Теперь следующий клик на снимке приведет к его уменьшению до начального значения:

```
else
   {
   document.getElementById("flo").style.transition="width 1s";
   document.getElementById("flo").style.width="200px";
   i=1;
   }
```

Первые две строки кода из блока **else** аналогичны уже разбиравшимся выше: они задают условия уменьшения картинки и размер, при достижении которого переход завершается. В третьей строке переменной **i** присваивается исходное значение 1. Таким образом, сценарий вновь готов увеличивать фотографию, если будет новый клик. После чего идентификатор опять изменит значение, подготавливая условия для обратного процесса. И так до бесконечности, пока это развлечение не надоест посетителю.

Как видите, мы достигли нужного результата, использовав довольно простой сценарий.

Соберем все части в единое целое:


```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие click</title>

<style>
#flo {width: 200px;}
</style>
```

```
<script>
window.addEventListener("load", function()
{
document.getElementById("flo").addEventListener("click", tra);
});

let i=1;

function tra()
{
if(i==1)
   {
   document.getElementById("flo").style.transition="width 1s";
   document.getElementById("flo").style.width="500px";
   i=2;
   }
else
   {
   document.getElementById("flo").style.transition="width 1s";
   document.getElementById("flo").style.width="200px";
   i=1;
   }

return false;
}
</script>
</head>

<body>

<a href="#"><img id="flo" src="pict/flower.jpg" alt="Фото"></a>

</body>
</html>
```

Как и в предыдущем примере, наша ссылка, добавленная к фото, имеет адрес **href="#"**? И так же как в предыдущем случае, последняя инструкция, завершающая описание обработчика, это **return false;**. Повторю: подобные приемы в данной книге использованы неоднократно. Я думаю, читатель уже запомнил эту особенность некоторых наших сценариев, поэтому дополнительные объяснения больше не понадобятся. Обнаружив похожие фрагменты кода, вы будете знать, для чего они нужны.

3.5. Событие click. Кнопочки для выбора пушистика

Если мы считаем, что событие **click** – одно из наиболее часто обрабатываемых, то нажатие кнопки – один из наиболее часто встречающихся вариантов этого события. В следующем примере мы посмотрим, как можно обрабатывать событие **click**, которое происходит на кнопках.

Данный пример вы можете посмотреть на странице **https://testjs.ru/p5.html**.

У нас есть пять кнопок для демонстрации фотографий тех или иных домашних животных – кошки, собаки, хорька, шиншиллы и хомяка (и все они как

на подбор пушистые – так и просятся, чтобы их погладили). Нажатие одной из кнопок приводит к появлению соответствующего изображения на странице. Вот и все – так обрабатывается событие **click** в нашей новой программе.

Начнем с того, что разместим кнопки на странице:

```
<div id="pic">
<input type="button" value="Кошка" id="cat">
<input type="button" value="Собака" id="dog">
<input type="button" value="Хорек" id="fre">
<input type="button" value="Шиншилла" id="chin">
<input type="button" value="Хомяк" id="hams">
<br><br>
<img id="im" src="pict/net.jpg" alt="Фото">
</div>
```

Как видите, каждая кнопка имеет персональный id, на который в дальнейшем будет ориентироваться наш сценарий. Все кнопки размещены внутри слоя с **id="pic"**. В этом же слое есть изображение:

```
<img id="im" src="pict/net.jpg" alt="Фото">
```

Это так называемая «заглушка». А еще его можно назвать платформой для вывода фотографии того или иного животного. Подобные «заглушки» в этой книге встретятся не раз.

В чем ее смысл? Это элемент, в который будет загружаться необходимый снимок при щелчке на кнопке. Сразу после загрузки элемент остается пустым, ведь мы еще не выбрали интересующее нас животное. Но такая ситуация противоречит стандартам HTML5, потому что каждый тег **img** должен изначально иметь заполненный атрибут **src**. Чтобы избежать конфликтов со спецификацией HTML5, мы создали рисунок **net.jpg**, который представляет собой точку размером 1x1 пиксель белого цвета. Такая точка «теряется» на белом фоне, и посетитель после загрузки страницы видит только кнопки (рис. 3.5.1). В нужный момент «заглушка» будет заменяться реальной фотографией.

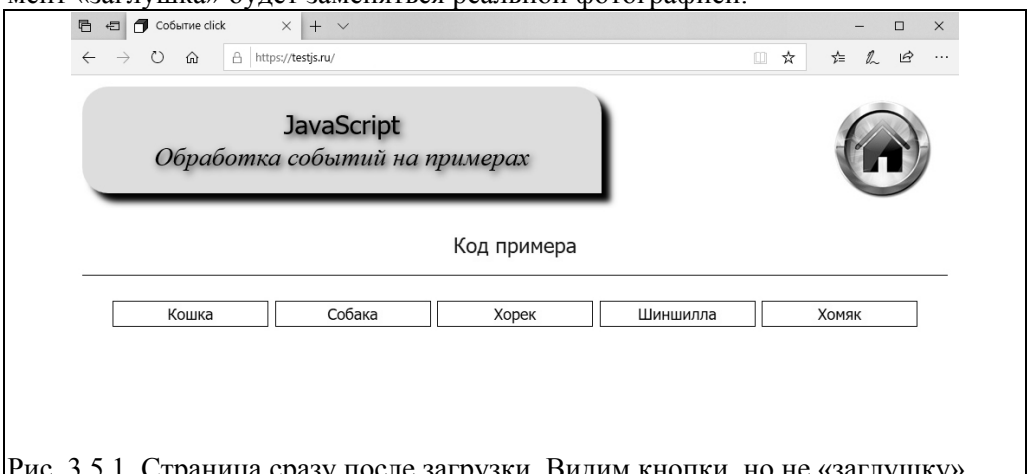

Рис. 3.5.1. Страница сразу после загрузки. Видим кнопки, но не «заглушку»

Смотрим дальше. После загрузки документа регистрируется обработчик события **click**, который объявляет анонимную функцию:

```
window.addEventListener("load", function()
{
document.getElementById("pic").addEventListener("click", function()
   {
   ...
   });
});
```

Обратите внимание: обработчик регистрируется для щелчков на слое, а не на кнопках. Сделано это для «экономии кода». Но обрабатываться будет именно нажатие кнопки. Такая вот особенность у нашего сценария. Если бы мы поступили иначе, пришлось бы объявлять один и тот же обработчик пять раз или писать пять отдельных обработчиков. Или применять еще какой-нибудь способ с не самой лаконичной «формулировкой». Словом, на мой взгляд, нами выбрано оптимальное решение.

Операторы обработчика выглядят следующим образом:

```
if(event.target.tagName=="INPUT")
      {
      let a=event.target.id;
      document.getElementById("im").src="anim/"+a+".jpg";
      document.getElementById("im").style.border="1px        solid
#000000";
      }
```

В первую очередь мы проверяем, где был произведен щелчок мышью:

```
if(event.target.tagName=="INPUT")
      {
      ...
      }
```

Поясним: объект **event** – это конструкция языка JavaScript, содержащая информацию о произошедшем на странице событии: перемещении мыши, щелчке на ссылке, получении элементом фокуса и многих других. Объект имеет свойство **target**, указывающее, на каком элементе произошло событие. Это свойство позволяет получить доступ к дополнительной информации, например, узнать имя тега, на котором был выполнен клик, или его ID.

Если щелчок произошел на кнопке (**tagName=="INPUT"**), то мы определяем ее **id**:

```
let a=event.target.id;
```

Пример составлен так, чтобы **id** фото соответствовал ее адресу. Например, если **id** снимка **dog**, то само изображение имеет адрес **anim/dog.jpg**. Получив **id** картинки, мы можем вывести ее на экран компьютера вместо «заглушки»:

```
document.getElementById("im").src="anim/"+a+".jpg";
```

Ну и для качественного оформления фото добавим ему рамку:

```
document.getElementById("im").style.border="1px solid #000000";
```

Если теперь щелкнуть на другом снимке, событие получит новый **id** и на странице появится следующая фотография (рис. 3.5.2).

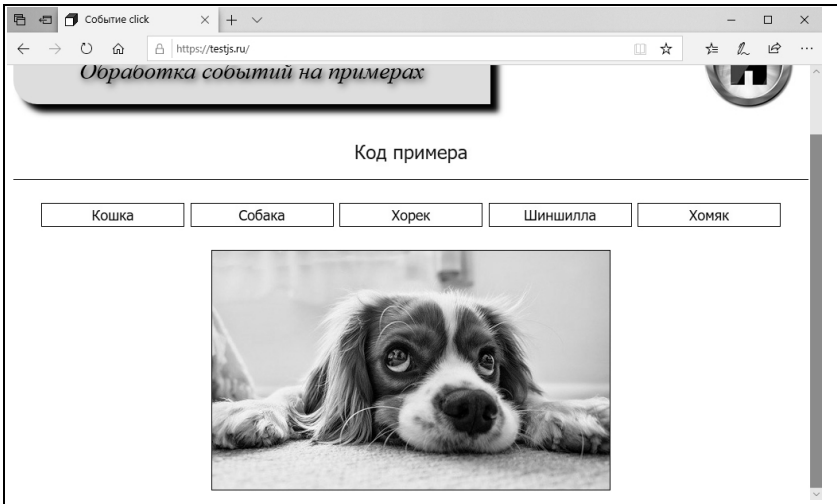

Рис. 3.5.2. После щелчка на ссылке появляется изображение домашнего животного

Пример вышел простым и коротким. Вот полный код страницы:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие click</title>

<script>
window.addEventListener("load", function()
{
document.getElementById("pic").addEventListener("click", function()
    {
    if(event.target.tagName=="INPUT")
        {
        let a=event.target.id;
        document.getElementById("im").src="anim/"+a+".jpg";
        document.getElementById("im").style.border="1px        solid
#000000";
        }
    });
});
</script>
</head>
```

```
<body>

<div id="pic">
<input type="button" value="Кошка" id="cat">
<input type="button" value="Собака" id="dog">
<input type="button" value="Хорек" id="fre">
<input type="button" value="Шиншилла" id="chin">
<input type="button" value="Хомяк" id="hams">
<br><br>
<img id="im" src="pict/net.jpg" alt="Фото">
</div>

</body>
</html>
```

3.6. Событие click. Not a crazy frog

Помните, была такая серия разных мелодий и клипов под общим названием «Crazy frog»? В нашем следующем примере тоже появится лягушка. Только вести себя она будет более адекватно и предсказуемо. И даже станет слушаться команд посетителя сайта. И все это – в рамках обработки события **click**.

Зайдите на страницу **https://testjs.ru/p6.html**. Если сейчас у вас нет подключения к Интернету, посмотрите на рисунок 3.6.1. На странице в ее левой части находится симпатичная лягушка. Наведите на нее указатель мыши и сделайте щелчок. Теперь переместите указатель в произвольную точку документа и щелкните еще раз. Послушная лягушка перепрыгнет в это место. Щелкните в другой точке – и лягушка тут же окажется там. Сколько щелчков – столько прыжков лягушки. Более того, она не просто прыгает, а поворачивается в направлении прыжка, точно настоящая (рис. 3.6.2).

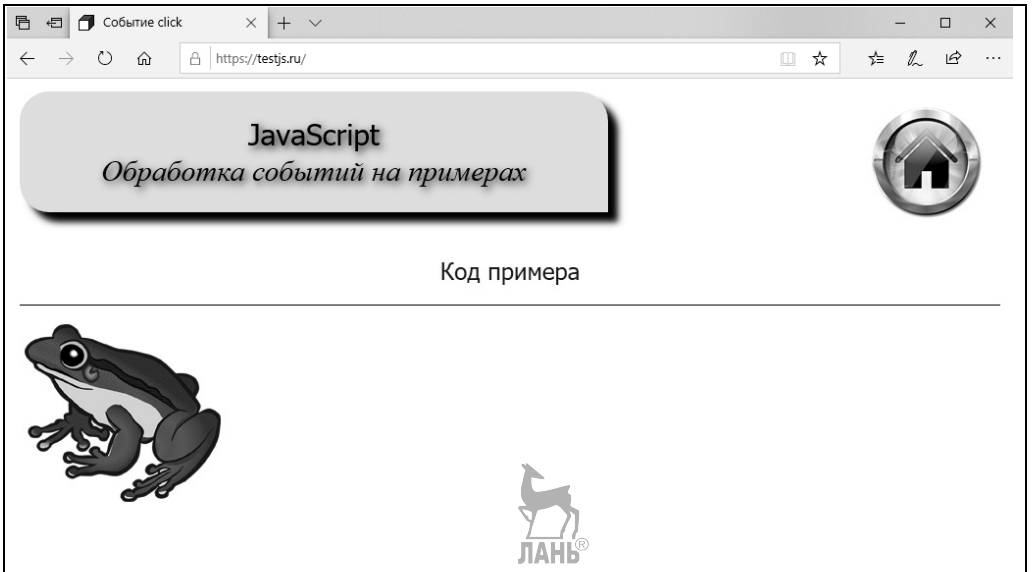

Рис. 3.6.1. Страница с изображением лягушки

На этом примере мы наблюдаем оригинальный вариант обработки события **click**.

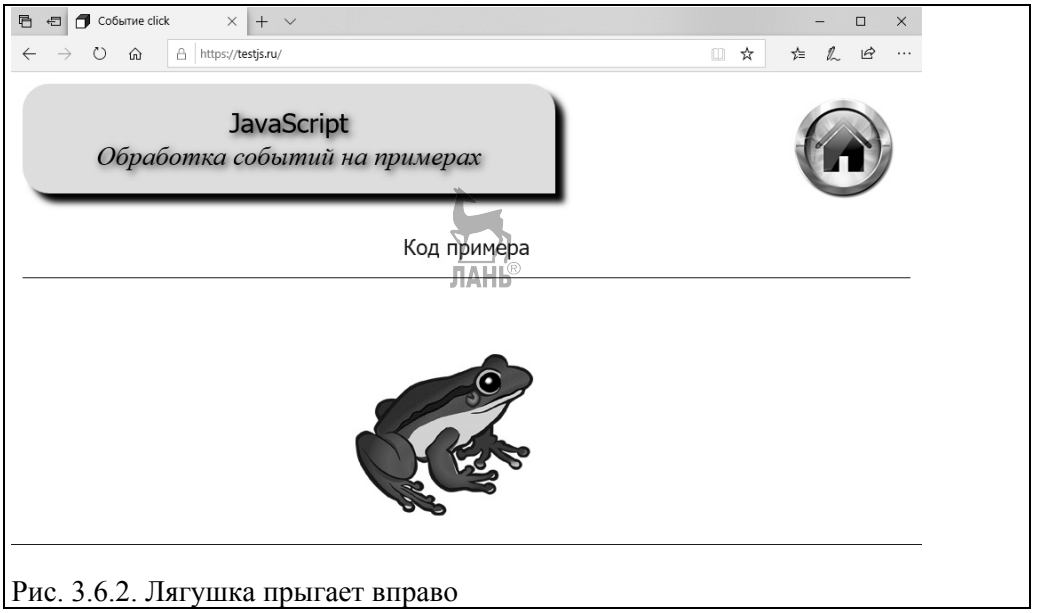

Рис. 3.6.2. Лягушка прыгает вправо

Откроем страницу с исходным кодом примера. Что мы обнаружим? Рисунок лягушки в теле документа

```
<img src="pict/frog_b.png" id="fr" alt="Фото">
```

с настройками

```
#fr {position: absolute; top: 0px; left: 0px;}
```

В данном сценарии регистрируется сразу 2 обработчика события **click**:

```
window.addEventListener("click", frog);
window.addEventListener("load", function()
{
document.getElementById("fr").addEventListener("click", coor);
});
```

Первый обработчик отслеживает события уровня окна и реагирует на клики мышью в любой точке страницы. Второй «привязан» к щелчкам на изображении лягушки.

У нас есть 2 функции – **frog** и **coor** (название произошло от английского coordinate), а также объявлены 4 глобальных переменных, доступных в обеих функциях:

```
let h=0;
let v=0;
```

```
let i=1;
let c=0;
```

Переменные **h** и **v** нужны для сохранения в них координат самого первого щелчка на лягушке. Переменная **i** – идентификатор, по значению которого сценарий определяет, какую из функций запускать после щелчка мышью, **c** – элемент хранения промежуточной горизонтальной координаты, по которой программы определяет, в каком направлении «скачет» картинка. Исходя из этого направления сценарий «поворачивает» изображение вправо или влево.

Пока условие **i==1** истинно, любые клики не на лягушке не дадут никакого результата, так как в функции управления лягушкой **frog** условие **i==2** ложно.

Наконец, произошел щелчок на картинке. Сразу будет запущена функция **coor**. Ее задача определить координаты указателя мыши во время этого щелчка и разрешить выполнение второй функции:

```
function coor()
{
if(i==1)
   {
   h=event.pageX;
   v=event.pageY;

   i=2;

   c=h;
   }
}
```

Здесь переменные **h** и **v** получают значения координат по оси X и Y. Идентификатору **i** присваивается число **2**, а переменной **c** – значение горизонтальной координаты. После этого функция **coor** перестает работать, так как условие ее выполнения **i==1** теперь ложно. Сделано это так, потому что функция **coor** нам больше не нужна.

Если теперь кликнуть в произвольном месте документа, запустится функция **frog**, условие в которой **i==2** стало истинным.

Определяем координаты нового щелчка

```
let nh=event.pageX;
let nv=event.pageY;
```

и рассчитываем разницу между координатами предыдущего и настоящего кликов:

```
let ih=nh-h;
let iv=nv-v;
```

Дальше сравниваем значения переменной **c** (горизонтальная координата предыдущего щелчка) и переменной **nh** (горизонтальная координата текущего щелчка). Если предыдущая горизонтальная координата меньше текущей (**c<nh**), это

значит, что лягушка «прыгнула» вправо, следовательно, выводим на страницу изображение лягушки, смотрящей вправо:

```
if(c<nh)
   {
   document.getElementById("fr").src="pict/frog_a.png";
   }
```

Если лягушка прыгнула влево, значит выводим на страницу изображение лягушки, смотрящей влево:

```
else
   {
   document.getElementById("fr").src="pict/frog_b.png";
   }
```

Операция сравнения **c** и **nh** выполняется при каждом вызове функции **frog**, благодаря чему лягушка все время обращена в сторону прыжка.

«Повернув» лягушку в нужном направлении (или сохранив направление – это в зависимости от вектора прыжков), переместим ее в новое место:

```
document.getElementById("fr").style.left=ih+"px";
document.getElementById("fr").style.top=iv+"px";
```

Как видите, смещение координат изображения лягушки равно смещению координат текущего клика по отношению к предыдущему. Благодаря этому лягушка прыгает строго в соответствии с местами щелчков.

В конце функции присваиваем переменной **c** значение текущей горизонтальной координаты **nh**:

```
c=nh;
```

При следующем клике текущее значение координаты станет предыдущим и сравнение переменных **c** и **nh**, где **nh** уже будет новой координатой, повторится. А в результате будет выполнено новое определение направления движения лягушки. И так при каждом щелчке.

После того, как мы разобрались в работе сценария, напишем полный код страницы:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие click</title>

<style>
#fr {position: absolute; top: 0px; left: 0px;}
</style>

<script>
window.addEventListener("click", frog);
```

```
window.addEventListener("load", function()
{
document.getElementById("fr").addEventListener("click", coor);
});

let h=0;
let v=0;
let i=1;
let c=0;

function coor()
{
if(i==1)
   {
   h=event.pageX;
   v=event.pageY;

   i=2;

   c=h;
   }
}

function frog()
{
if(i==2)
   {
   let nh=event.pageX;
   let nv=event.pageY;

   let ih=nh-h;
   let iv=nv-v;

   if(c<nh)
      {
      document.getElementById("fr").src="pict/frog_a.png";
      }
   else
      {
      document.getElementById("fr").src="pict/frog_b.png";
      }

   document.getElementById("fr").style.left=ih+"px";
   document.getElementById("fr").style.top=iv+"px";

   c=nh;
   }
}
</script>
</head>

<body>

<img src="pict/frog_b.png" id="fr" alt="Фото">

</body>
</html>
```

3.7. Событие click. Галерея

И опять у нас пример для **click** – настолько интересны и многообразны варианты обработки этого события. Как и в случае с лягушкой, мы напишем сразу 2 обработчика. Однако теперь это будут равнозначные обработчики для двух равнозначных событий, происходящих на двух равнозначных элементах. Единственное отличие – результаты, которые будут достигаться кликами на этих элементах, противоположны.

Перейдем к делу. Мы попробуем написать сценарий фотогалереи, которая «замкнута» в кольцо. Проще говоря, достигнув конечного снимка, программа снова будет выводить на экран первый – и далее по кругу. Или в обратном направлении: достигнув первой фотографии, скрипт вернется к последней – и все по новой.

Как выглядит страница с такой программой, вы можете видеть на рисунке 3.7.1. В качестве снимков использованы изображения природы.

Проверить работу такой галереи можно на сайте поддержки книги на странице **https://testjs.ru/p7.html**.

Сразу после загрузки документа в окне браузера демонстрируется снимок лесной дороги. Щелкнем правой кнопкой с изображением стрелки. Появится следующее фото – озеро на закате. Еще щелчок – возникает изображение горного массива. И так далее по порядку увеличения номера фото. В галерее всего 9 картинок. Последняя – одинокое дерево на фоне облаков. Если теперь щелкнуть на правой стрелке, вновь появится снимок лесной дороги. Круг замкнулся. Дальнейшие щелчки на этой кнопке будут выводить на экран уже знакомые нам фотографии в том же порядке. Такая замкнутая «круговерть» может продолжаться, пока у посетителя хватает интереса и терпения. Если щелкать левой кнопкой со стрелкой, то последовательность демонстрации фотографий будет обратной – и тоже по замкнутому кругу.

Рис. 3.7.1. Галерея фотографий природы

Итак, все отличие двух событий **click**, происходящих на разных кнопках, – в направлении пролистывания фото.

Чтобы создать такую галерею, разместим на странице таблицу с тремя ячейками:

```
<table>
<tr><td>
<a id="nz" href="#"><img src="nat/nz.png" alt="Назад"></a></td>
<td>
<img id="photo" src="nat/0.jpg" alt="Фото"></td>
<td>
<a          id="vp"          href="#"><img          src="nat/vp.png"
alt="Вперед"></a></td></tr>
</table>
```

В левой ячейке находится кнопка пролистывания снимков назад, в средней – фотографии, в правой – кнопка пролистывания снимков вперед. Чтобы клиент понимал, что стрелки можно «нажимать», они «обернуты» в ссылки:

```
<a href="#">...</a>
```

А чтобы в строке адреса после щелчка не появлялись знаки #, мы применим уже знакомый нам способ, добавив в обработчики такие инструкции:

```
return false;
```

Наверное вы заметили, что в исходном состоянии загружена фотография с адресом **nat/0.jpg**. Все остальные фото тоже имеют цифровые имена. Так сделано для упрощения программы.

Зарегистрируем обработчики:

```
window.addEventListener("load", function()
{
document.getElementById("nz").addEventListener("click", nz);
document.getElementById("vp").addEventListener("click", vp);
});
```

Как видите, у нас есть две функции, которые управляют пролистыванием снимков назад (**nz**) и вперед (**vp**). Кроме того, есть глобальная переменная **i**, доступная в обеих функциях. Ее задача – хранить номер изображения, которое в данный момент загружено на страницу. Так как в исходном состоянии в галерее показывается картинка с номером **0**, то и переменная **i** имеет начальное значение **0**:

```
let i=0;
```

Для разнообразия начнем с ситуации, когда фотографии демонстрируются в обратном порядке. Эти манипуляции выполняет функция **nz**:

```
function nz()
{
if(i>0)
   {
   i--;
   document.getElementById("photo").src="nat/"+i+".jpg";
   }
else
   {
   i=8;
   document.getElementById("photo").src="nat/"+i+".jpg";
   }
return false;
}
```

Действия обработчика зависят от истинности или ложности условия **if(i>0)**. Так как в момент первого клика на кнопке «Назад» данное условие ложно, то выполняются операторы второго блока:

```
i=8;
document.getElementById("photo").src="nat/"+i+".jpg";
```

Переменной **i** присваивается значение **8**, и на страницу выводится последний снимок. После этого условие **if(i>0)** становится истинным. В результате при следующем клике «в дело» вступит первый блок:

```
i--;
document.getElementById("photo").src="nat/"+i+".jpg";
```

Переменная **i** уменьшится на единицу, и появится фото с номером **7**. Новый щелчок опять уменьшает переменную **i**, благодаря чему демонстрируется картинка под номером **6**. Так происходит до тех пор, пока **i>0**. Как только на экране появится «нулевая» фотография, очередной клик приведет к тому, что вновь будет запущен второй блок, переменная **i** получит значение **8**, цикл замкнется и просмотр изображений пойдет по второму кругу.

Функция перемотки снимков вперед выглядит похоже:

```
function vp()
{
if(i<8)
   {
   i++;
   document.getElementById("photo").src="nat/"+i+".jpg";
   }
else
   {
   i=0;
   document.getElementById("photo").src="nat/"+i+".jpg";
   }
return false;
}
```

Здесь проверяется условие **if(i<8)**. Когда оно истинно, работает первый блок инструкций. Каждый новый клик увеличивает переменную **i** на единицу, а в ре-

зультате демонстрируется следующая по возрастанию номеров фотография. Когда на странице оказывается последний снимок, очередной щелчок мышью приводит в действие второй блок инструкций, в котором переменная **i** получает значение **0** и вновь показывается самая первая фотография. Так замыкается цикл прямого направления прокрутки снимков.

В завершение добавлю, что менять направление просмотра фотографий можно в любой момент – обработчики легко справятся с этой задачей, так как текущее значение переменной **i** доступно в обеих функциях.

Когда мы соберем все части, получится вот такая программа:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие click</title>

<script>
window.addEventListener("load", function()
{
document.getElementById("nz").addEventListener("click", nz);
document.getElementById("vp").addEventListener("click", vp);
});

let i=0;

function nz()
{
if(i>0)
   {
   i--;
   document.getElementById("photo").src="nat/"+i+".jpg";
   }
else
   {
   i=8;
   document.getElementById("photo").src="nat/"+i+".jpg";
   }
return false;
}

function vp()
{
if(i<8)
   {
   i++;
   document.getElementById("photo").src="nat/"+i+".jpg";
   }
else
   {
   i=0;
   document.getElementById("photo").src="nat/"+i+".jpg";
   }
return false;
}
</script>
</head>
```

```
<body>

<table>
<tr><td>
<a id="nz" href="#"><img src="nat/nz.png" alt="Назад"></a></td>
<td>
<img id="photo" src="nat/0.jpg" alt="Фото"></td>
<td>
<a          id="vp"          href="#"><img          src="nat/vp.png"
alt="Вперед"></a></td></tr>
</table>

</body>
</html>
```

Теперь мы временно завершим написание примеров для события **click**. Но это не последний сценарий, посвященный обработчикам данного типа. Несколько позже мы рассмотрим еще один пример для события **click**.

3.8. Событие mouseover. Анимируем кнопки

Продолжаем знакомиться с событиями компьютерной мыши. На очереди у нас **mouseover** – событие наведения курсора на какой-либо элемент страницы. Так как подобные события похожи для кнопок, рисунков, слоев, таблиц, ссылок и т. д., мы ограничимся примером с наведением мыши на кнопки. У нас будет сценарий, который меняет оформление этих элементов. Кстати, сценарий довольно простой.

Страница **https://testjs.ru/p8.html** демонстрирует пример в действии. Как видите, в окне браузера есть три кнопки. Если навести на одну из них указатель мыши, кнопка, ее рамка и ее надпись изменят цвет, а вокруг появится небольшая тень (рис. 3.8.1). Такие действия еще иногда называют подсвечиванием элементов. Это – очень распространенный прием среди разработчиков web-ресурсов. Если указатель покинул пределы кнопки, ей возвращается изначальный вид. Как видите, действительно все очень просто.

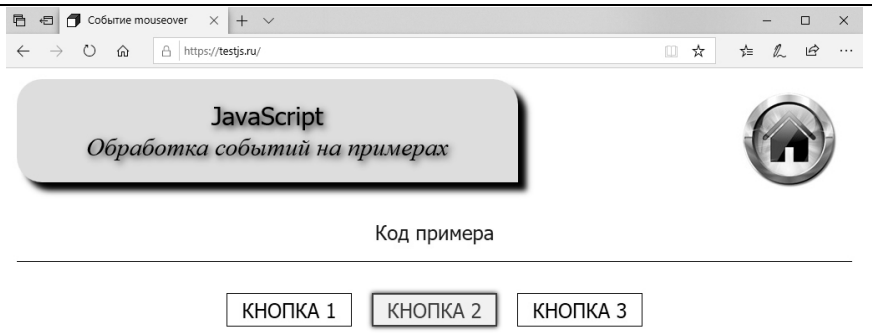

Рис. 3.8.1. Подсвечивание кнопки, на которую наведен указатель мыши

Поскольку прием очень распространенный, научимся добиваться подобного эффекта программными методами. (Кстати, данного эффекта также можно достичь и без помощи JavaScript, используя только возможности CSS. Но у нас задача – наглядно продемонстрировать обработку события **mouseover**. Для такого случая наш пример подходит, как нельзя лучше.)

Приступим. Добавим на страницу три кнопки:

```
<input id="bu1" type="button" value="КНОПКА 1" class="inp">
<input id="bu2" type="button" value="КНОПКА 2" class="inp">
<input id="bu3" type="button" value="КНОПКА 3" class="inp">
```

Настроим их внешний вид:

```
.inp {background: #FFFFFF; border: 1px solid #000000;}
```

По умолчанию надписи на кнопках будут черного цвета, а тени отсутствовать, так как они не указаны в стилях.

Зарегистрируем обработчик события **mouseover** и сразу назначим ему анонимную функцию:

```
window.addEventListener("load", function()
{
window.addEventListener("mouseover", function()
   {
   if(event.target.id.indexOf("bu")==0)
      {
      let e=event.target.id;
      document.getElementById(e).style.border="2px solid #CC0000";
      document.getElementById(e).style.background="#E6F7F8";
      document.getElementById(e).style.color="#CC0000";
      document.getElementById(e).style.boxShadow="#000000  0px  0px
8px";
      }
   else
      {
      for(let i=0; i<3; i++)
         {
         document.getElementsByTagName("input")[i].style.border=
                                                    "1px    solid
#000000";
document.getElementsByTagName("input")[i].style.background=

"#FFFFFF";
document.getElementsByTagName("input")[i].style.color="#000000";
document.getElementsByTagName("input")[i].style.boxShadow=
                                                  "#000000    0px
0px 0px";
         }
      }
   });
});
```

Как видно из сценария, обработчик отслеживает событие **mouseover** на уровне окна. Если указатель мыши оказался наведен на кнопку, ID которой начинается с символов **bu**

```
if(event.target.id.indexOf("bu")==0)
```

то происходит определение ID кнопки:

```
let e=event.target.id;
```

после чего меняется внешний вид этой самой кнопки. Сначала меняем толщину и цвет рамки. Она становится красной:

```
document.getElementById(e).style.border="2px solid #CC0000";
```

Далее изменяем фоновый цвет с белого на голубой:

```
document.getElementById(e).style.background="#E6F7F8";
```

Потом сделаем текст надписи красным:

```
document.getElementById(e).style.color="#CC0000";
```

и создаем тень:

```
document.getElementById(e).style.boxShadow="#000000 0px 0px 8px";
```

Все, внешний вид кнопки изменился, как нам того хотелось. Но! Если оставить все как есть, мы столкнемся с одной неприятной ситуацией: когда указатель мыши покинет кнопку, она так и останется подсвеченной. Что делать, ведь у нас в этом сценарии предусмотрен обработчик только одного события? Написать вторую часть обработчика, которая будет вступать в действие, если событие **mouseover** происходит не на кнопке:

```
else
   {
   for(let i=0; i<3; i++)
      {
      document.getElementsByTagName("input")[i].style.border=
                                                    "1px    solid
#000000";
document.getElementsByTagName("input")[i].style.background=
"#FFFFFF";
document.getElementsByTagName("input")[i].style.color="#000000";
document.getElementsByTagName("input")[i].style.boxShadow=
                                                  "#000000    0px
0px 0px";
      }
   }
```

Внутри блока мы написали цикл **for**, который проходит по всем кнопкам и возвращает им исходную толщину и цвет рамки, цвет фона и текста, а также удаляет тень. Поскольку на странице 3 кнопки, операции цикла выполняются, пока верно условие **i<3**. Как вы помните, счет элементов страницы всегда начинается с числа 0. В нашем случае элементы имеют индексы 0, 1 и 2.

Подведем итоговую черту под нашим примером:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие mouseover</title>

<style>
.inp {background: #FFFFFF; border: 1px solid #000000;}
</style>

<script>
window.addEventListener("load", function()
{
window.addEventListener("mouseover", function()
   {
   if(event.target.id.indexOf("bu")==0)
      {
      let e=event.target.id;
      document.getElementById(e).style.border="2px solid #CC0000";
      document.getElementById(e).style.background="#E6F7F8";
      document.getElementById(e).style.color="#CC0000";
      document.getElementById(e).style.boxShadow="#000000  0px  0px
8px";
      }
   else
      {
      for(let i=0; i<3; i++)
         {
         document.getElementsByTagName("input")[i].style.border=
                                                        "1px     solid
#000000";

document.getElementsByTagName("input")[i].style.background=

"#FFFFFF";

document.getElementsByTagName("input")[i].style.color="#000000";

document.getElementsByTagName("input")[i].style.boxShadow=
                                                       "#000000     0px
0px 0px";
         }
      }
   });
});
</script>
</head>

<body>

<input id="bu1" type="button" value="КНОПКА 1" class="inp">
```

```
<input id="bu2" type="button" value="КНОПКА 2" class="inp">
<input id="bu3" type="button" value="КНОПКА 3" class="inp">

</body>
</html>
```

В этом параграфе мы разобрали случай применения обработчика для события **mouseover**. Однако чаще всего событие наведения мыши на элемент тесно связано с другим – уходом указателя мыши с элемента. В следующем примере мы рассмотрим взаимосвязанное применение обработчиков для обоих событий.

3.9. События mouseover и mouseout. Раскрашиваем кораблик

В этом параграфе рассмотрим прекрасный и весьма наглядный пример обработки взаимосвязанных событий **mouseover и mouseout**. Как явствует из названия, черно-белое фото корабля мы будем превращать в цветное.

Пример в действии можно посмотреть на странице **https://testjs.ru/p9.html**. Это один из самых простых и коротких сценариев книги.

К сожалению, черно-белая печать не позволяет передать эффект «обретения» цвета, поэтому иллюстрирует пример только изображение исходного состояния фотографии корабля (рис. 3.9.1).

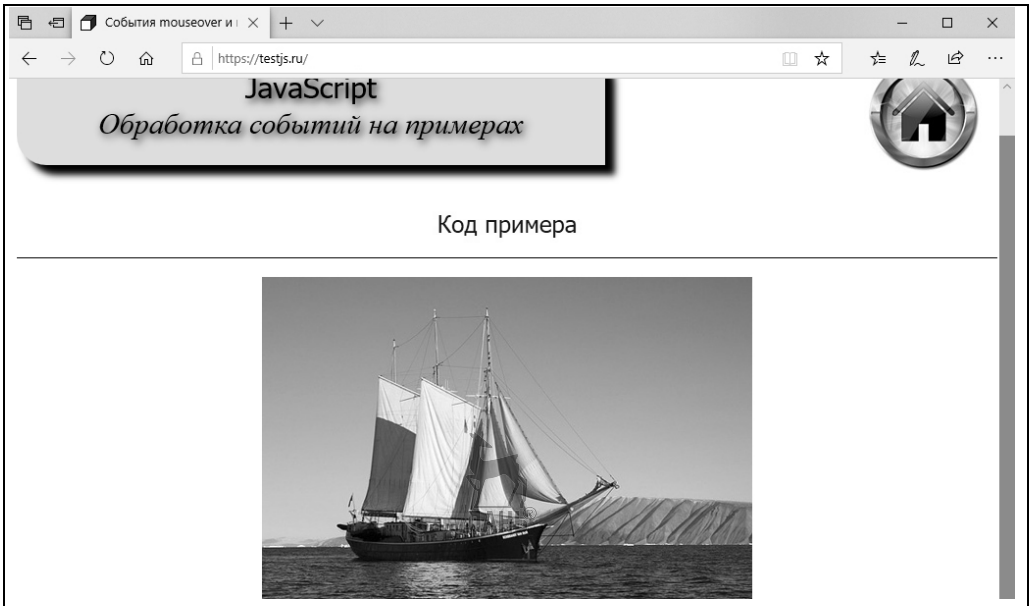

Рис. 3.9.1

Начнем. В теле документа есть указанный снимок:

```
<img src="pict/ship_a.jpg" id="sh" alt="Фото">
```

А в головной части страницы размещен очень простой сценарий:

```
window.addEventListener("load", function()
{
document.getElementById("sh").addEventListener("mouseover", function()
   {
   document.getElementById("sh").src="pict/ship_b.jpg";
   });
document.getElementById("sh").addEventListener("mouseout", function()
   {
   document.getElementById("sh").src="pict/ship_a.jpg";
   });
});
```

В качестве обработчиков событий выступают две анонимные функции. Первая откликается на событие наведения указателя мыши на картинку:

```
document.getElementById("sh").addEventListener("mouseover", function()
   {
   document.getElementById("sh").src="pict/ship_b.jpg";
   });
```

Когда это происходит, начальное черно-белое фото заменяется на аналогичное цветное:

```
document.getElementById("sh").src="pict/ship_b.jpg";
```

Когда указатель мыши покидает границы снимка, запускается второй обработчик:

```
document.getElementById("sh").addEventListener("mouseout", function()
   {
   document.getElementById("sh").src="pict/ship_a.jpg";
   });
```

Он меняет цветное фото на исходное черно-белое:

```
document.getElementById("sh").src="pict/ship_a.jpg";
```

Как видите, трудно придумать более простой случай демонстрации обработки событий **mouseover и mouseout**. «Суммарный» код страницы подтверждает это:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>События mouseover и mouseout</title>
<script>
window.onload=function()
{
```

```
document.getElementById("sh").addEventListener("mouseover", function()
    {
    document.getElementById("sh").src="pict/ship_b.jpg";
    });
document.getElementById("sh").addEventListener("mouseout", function()
    {
    document.getElementById("sh").src="pict/ship_a.jpg";
    });
});
</script>
</head>

<body>

<img src="pict/ship_a.jpg" id="sh" alt="Фото">

</body>
</html>
```

3.10. Событие mousemove. «Проявляем» самолет

Начиная с этого примера, мы рассмотрим несколько случаев обработки события **mousemove**.

Первым будет сценарий, манипулирующий с настройками изображения авиационной техники. Если вы когда-то самостоятельно печатали фотографии, то знакомы с эффектом медленного появления снимка на фотобумаге в проявителе. Нечто похожее будем делать и мы. Только в нашей программе можно будет не только «проявлять» самолет, но и наоборот, плавно сводить его изображение на «нет».

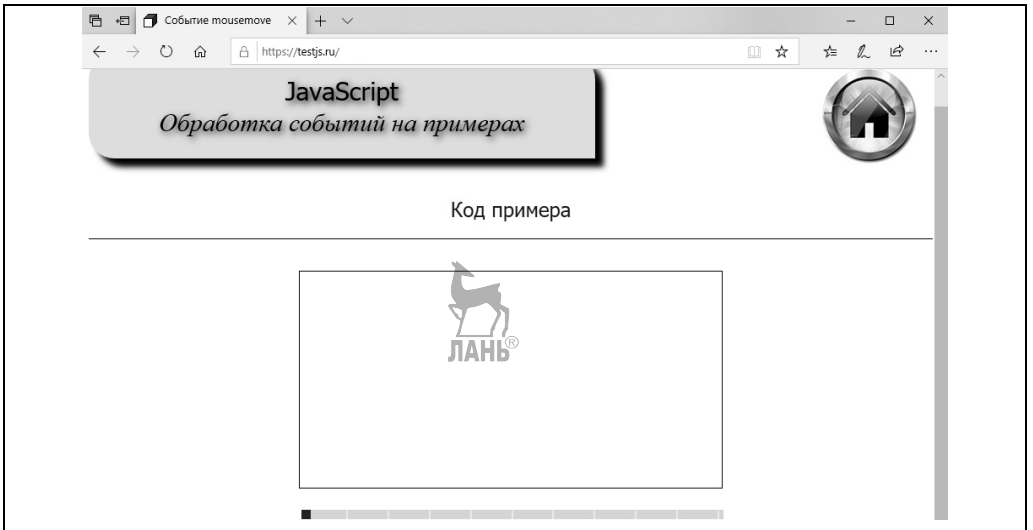

Рис. 3.10.1. После загрузки страницы самолет не виден

Кстати, как и в предыдущем параграфе, новый сценарий тоже будет простым и коротким.

Зайдите на страницу **https://testjs.ru/p10.html**. Перед вами прямоугольная область, предназначенная для размещения картинки, а под ней – ползунок со шкалой делений (рис. 3.10.1). Наведите курсор на ползунок, нажмите левую кнопку мыши и плавно ведите указатель по делениям шкалы слева направо. Вы увидите, как изображение самолета медленно проявится сквозь белый фон. Если теперь провести указателем в обратном направлении, изображение плавно «растворится». Поэкспериментируйте, перемещая ползунок вправо-влево. Вы увидите, что можно фиксировать промежуточные состояния прозрачности фото (рис. 3.10.2).

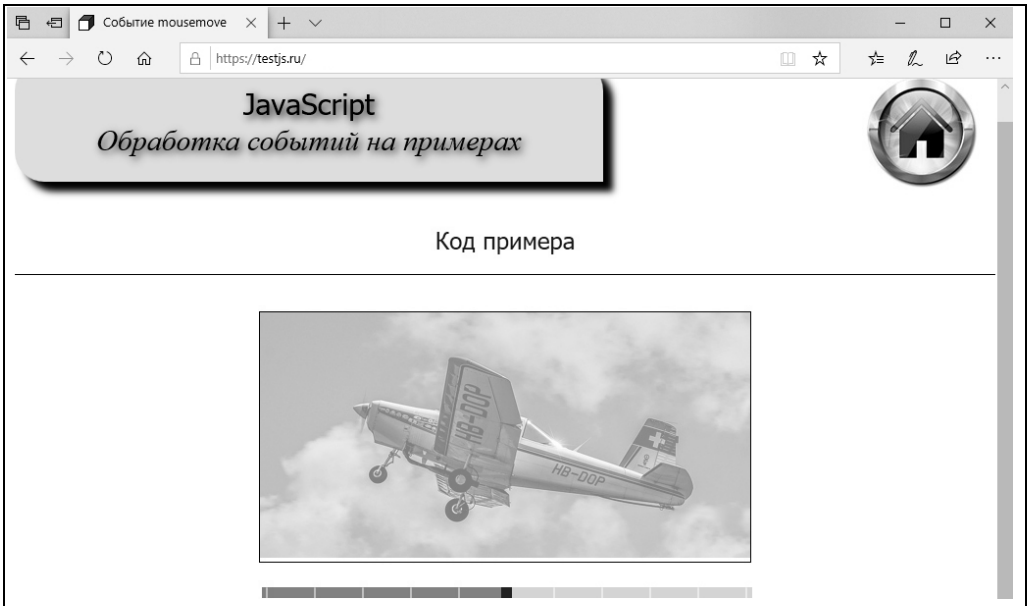

Рис. 3.10.2. Промежуточная стадия «проявления» самолета

Если посмотреть исходный код страницы, то можно увидеть, что в ее теле располагается снимок самолета

```
<img src="pict/air.jpg" id="air" alt="Фото">
```

непрозрачность которого в начальном состоянии равна нулю:

```
#air {opacity: 0;}
```

Под фотографией размещена шкала с ползунком:

```
<input type="range" min="0" max="1" step="0.01" value="0"
id="ran">
```

Шкала имеет минимальное значение 0, а максимальное – 1. Шаг значений – 0.01 (одна сотая). Именно с таким шагом будет изменяться показатель непрозрачности снимка. Размер шкалы – например, 500 пикселей:

```
#ran {width: 500px;}
```

Регулировкой непрозрачности управляет анонимная функция, которая «откликается» на событие **mousemove** – то есть на перемещение курсора мыши по шкале с ползунком:

```
window.addEventListener("load", function()
{
document.getElementById("ran").addEventListener("mousemove", function()
    {
    document.getElementById("air").style.opacity=this.value;
    });
});
```

При этом непрозрачность фотографии меняется в соответствии с положением ползунка на шкале значений:

```
document.getElementById("air").style.opacity=this.value;
```

В крайней левой точке шкала имеет значение 0, поэтому непрозрачность рисунка равна 0 – то есть он не виден. В крайней правой точке шкала имеет значение 1, поэтому клиент видит четкое изображение самолета. В промежуточных положениях самолет виден нечетко.

Ключевое слово this указывает на элемент, к которому привязан обработчик. Соответственно, с помощью конструкции this.value мы получаем текущее положение ползунка на шкале в заданных единицах.

Страница с примером выглядит так:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие mousemove</title>

<style>
#air {opacity: 0;}
#ran {width: 500px;}
</style>

<script>
window.addEventListener("load", function()
{
document.getElementById("ran").addEventListener("mousemove", function()
    {
    document.getElementById("air").style.opacity=this.value;
    });
});
```

```
</script>
</head>

<body>

<img src="pict/air.jpg" id="air" alt="Фото"><br>
<input   type="range"   min="0"   max="1"   step="0.01"   value="0"
id="ran">

</body>
</html>
```

3.11. Событие mousemove. Раритетное авто

Следующий пример обработки события **mousemove** будет несколько сложнее предыдущего. Мы станем проявлять изображение раритетного авто уже без использования ползунка. Если так можно выразиться, событие **mousemove** в новом примере работает в «чистом виде», без «примеси» вспомогательных элементов.

Страница с этой программой расположена по адресу **https://testjs.ru/p11.html**.

После загрузки документа мы видим только рамку изображения, которое необходимо «проявить» (рис. 3.11.1). Пока указатель мыши не попадет внутрь рамки, ничего не происходит.

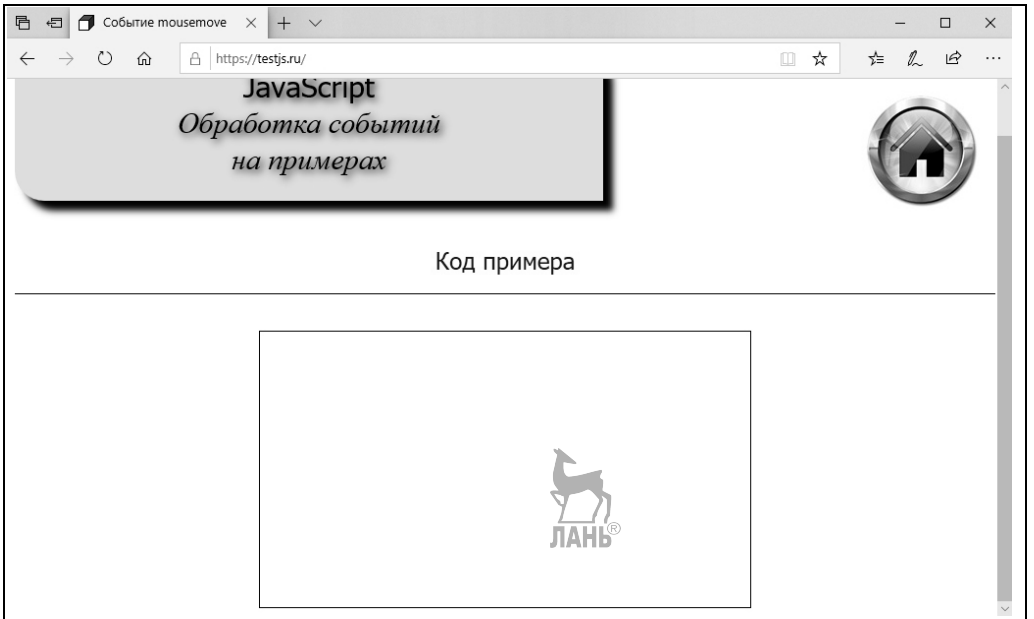

Рис. 3.11.1. Автомобиля совсем не видно

Теперь медленно проведите курсором внутри рамки слева направо. Вы увидите, как на белом фоне плавно возникнет фотография старинного автомобиля (рис. 3.11.2). Проведите курсором в обратном направлении – справа налево. Фотография также плавно «растворится». Наилучший эффект достигается именно при медленном движении мыши. Если быстро провести указателем по рабочей области, фотография проявится или растворится только частично. То же произойдет, если указатель начал двигаться не от боковых сторон рамок, а, например, от середины изображения.

Не правда ли, интересный пример? Посмотрим, как он запрограммирован. А для этого откроем код примера. Что мы увидим? На странице есть слой

```
<div id="pic">...</div>
```

на котором размещен снимок авто:

```
<div id="pic"><img src="pict/oldcar.jpg" id="car" alt="Фото"></div>
```

Слой имеет ширину 500 пикселей и рамку толщиной 1 пиксель:

```
#pic {border: 1px solid #000000; width: 500px;}
```

У фотографии задан нулевой уровень непрозрачности:

```
#car {opacity: 0;}
```

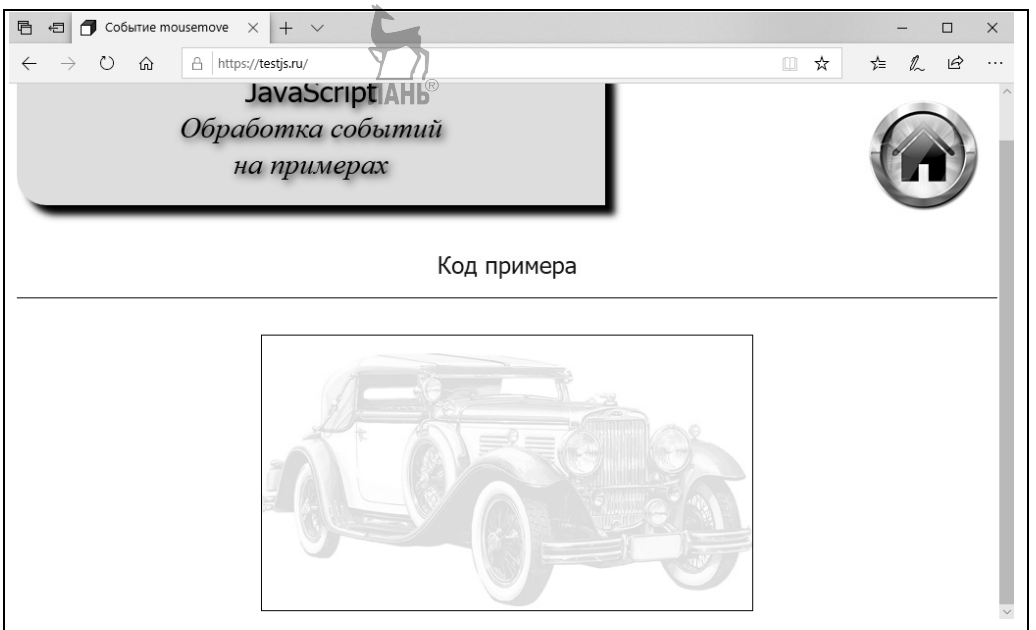

Рис. 3.11.2. Рисунок начинает менять прозрачность

Его изменением управляет функция **opa** (название происходит от свойства opacity, что переводится как непрозрачность):

```
window.addEventListener("load", function()
{
document.getElementById("car").addEventListener("mousemove", opa);
});
```

Прежде чем разбирать принципы работы этой функции, зададимся вопросом: а зачем нам слой **div**? Только для демонстрации рамки? Но ведь ее можно создать непосредственно у фотоснимка. Да, можно. Только тогда в исходном состоянии она не будет видна также, как и картинка. Поэтому нам трудно будет обнаружить, где находится снимок.

Вернемся к сценарию. У нас объявлены две переменные:

```
let d=0;
let i=0;
```

Переменная **d** предназначена для хранения предыдущей по отношению к текущей координате мыши по оси X. Это определение станет более понятным в процессе разбора сценария. Переменная **i** – счетчик уровня непрозрачности.

Как только событие **mousemove** произойдет над рисунком (то есть указатель мыши начнет движение над рисунком), будет выполнен запуск обработчика. Вот его код:

```
function opa()
{
let h=event.pageX;

if(h-d>3)
   {
   if(i<=1)
      {
      i+=0.01;
      document.getElementById("car").style.opacity=i;
      d=h;
      }
   else
      {
      i=1;
      }
   }
if(d-h>3)
   {
   if(i>=0)
      {
      i-=0.01;
      document.getElementById("car").style.opacity=i;
      d=h;
      }
   else
      {
      i=0;
      d=0;
```

```
      }
   }
}
```

Начинается сценарий с определения горизонтальной координаты указателя:

```
let h=event.pageX;
```

Давайте считать, что сейчас курсор двигается слева направо. Первым делом проверяется условие **if(h-d>3)**. В начальный момент **d** имеет значение **0**, поэтому разность между **h** и **d** удовлетворяет условию. Таким образом, при попадании указателя мыши на фото будут выполняться инструкции первого блока **if**.

Поясним, что шаг в **3** пикселя выбран путем экспериментов. При таком значении разницы текущей и предыдущей координаты программа работает наиболее стабильно и точно.

Продолжим. Внутри первого блока выполняется проверка еще одного условия:

```
if(i<=1)
```

Так как исходное значение переменной **i** равно **0**, то это условие тоже будет истинным. Таким образом станут выполняться три инструкции:

```
i+=0.01;
document.getElementById("car").style.opacity=i;
d=h;
```

В первой строке счетчик увеличивается на **0.01**. Во второй строке его значение присваивается свойству непрозрачности изображения. В третьей выполняется операция присвоения текущей координаты мыши переменной **d**.

Смотрим, что произойдет дальше. Курсор мыши продолжает движение вправо, а значит вторично запускается функция **opa**. Опять выполняется проверка условия **if(h-d>3)**. Как видите, теперь сравнивается новая координата указателя со старой, полученной на предыдущем шаге и помещенной в переменную **d**. Если условие истинно, показатель непрозрачности опять увеличится. Продолжаем вести мышь слева направо – значение переменной **i** все увеличивается и увеличивается. В какой-то момент уровень непрозрачности достигнет **1** и фотография станет видна полностью. Заметим, что переменная **i** может увеличиться до значений, немного превышающих **1**. Но это неважно, так как на подобный случай у нас есть соответствующие инструкции:

```
if(i<=1)
   {
   ...
   }
else
```

```
    {
    i=1;
    }
```

Сколько бы раз вы не проводили мышью по картинке слева направо (минуя обратное движение над изображением), счетчик **i** всегда будет иметь максимальное значение **1** – и не больше.

Этот момент мы разберем подробнее. Если бы не было таких ограничений по значению счетчика, мы обнаружили бы неприятное поведение сценария. Представьте, что вы один раз провели по фото слева направо. Оно полностью проявилось (то есть **i** стала равна единице или даже чуть большему числу). Теперь вы убираете мышь с рисунка, перемещаете ее указатель вновь к левому краю и опять проводите по снимку. Внешне ничего не поменялось, но переменная **i** уже достигнет значения 2 или немного больше. Еще несколько подобных манипуляций и значение счетчика станет равно 5, или 8, или вообще двухзначному числу. При этом у картинки по-прежнему ничего не изменится, так как свойство **opacity** воспринимает все значения выше единицы как полную непрозрачность. Если теперь один раз провести мышью по картинке в обратном направлении, мы с удивлением обнаружим, что изображение не исчезает. А все потому, что число, хранимое счетчиком, хотя и уменьшились на единицу, но стало, допустим, 4 вместо накрученных 5. Инструкция

```
else
    {
    i=1;
    }
```

избавляет нас от такого неприятного эффекта.

Кстати, похожую проверку выполняет и вторая часть функции, не допуская уменьшения значения счетчика ниже отметки **0**.

Теперь разберемся, что будет происходить во время движения указателя мыши по картинке в обратном направлении. Как раз вторая часть функции и предназначена для обработки таких действий посетителя:

```
if(d-h>3)
    {
    if(i>=0)
        {
        i-=0.01;
        document.getElementById("car").style.opacity=i;
        d=h;
        }
    else
        {
        i=0;
        d=0;
        }
    }
```

В данном случае проверяется обратное условие:

```
if(d-h>3)
```

То есть, если предыдущая координата мыши больше текущей, значит указатель двигается в обратном направлении и показатель непрозрачности уменьшается на 0.01:

```
i-=0.01;
document.getElementById("car").style.opacity=i;
d=h;
```

Эти инструкции выполняются до тех пор, пока уровень непрозрачности больше или равен **0**:

```
if(i>=0)
```

По достижении нулевой «отметки» рисунок полностью исчезает, скручивание счетчика прекращается и переменным **i** и **d** присваиваются исходные значения:

```
else
   {
   i=0;
   d=0;
   }
```

Открывать и прятать снимок можно сколько угодно – программа составлена так, что допускает неограниченное число однотипных действий.

В завершении рассказа полный код страницы:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие mousemove</title>

<style>
#pic {border: 1px solid #000000; width: 500px;}
#car {opacity: 0;}
</style>

<script>
window.addEventListener("load", function()
{
document.getElementById("car").addEventListener("mousemove", opa);
});

let d=0;
let i=0;

function opa()
{
let h=event.pageX;

if(h-d>3)
   {
   if(i<=1)
      {
```

```
        i+=0.01;
        document.getElementById("car").style.opacity=i;
        d=h;
        }
    else
        {
        i=1;
        }
    }
if(d-h>3)
    {
    if(i>=0)
        {
        i-=0.01;
        document.getElementById("car").style.opacity=i;
        d=h;
        }
    else
        {
        i=0;
        d=0;
        }
    }
}
</script>
</head>

<body>

<div       id="pic"><img       src="pict/oldcar.jpg"       id="car"
alt="Фото"></div>

</body>
</html>
```

3.12. Событие mousemove. Упряжка за шторкой

Продолжаем создавать оригинальные примеры, иллюстрирующие обработку события **mousemove**. В новом сценарии большое значение имеет позиционирование элементов. Поэтому в коде документа будет непривычно много настроек стилей.

Откроем страницу **https://testjs.ru/p12.html**. Ее начальный вид очень напоминает страницу из предыдущего примера (рис. 3.12.1). Опять есть рамка. Внутри нее что-то скрывается? Нет, не внутри, а за ней. Рамка – это границы белой шторки, за которой «спряталось» изображение кареты с упряжкой лошадей.

Как и в предыдущем примере, медленно проведите курсором по шторке слева направо. На ваших глазах шторка плавно сдвинется к правому краю, а за ней обнаружится обещанное изображение (рис. 3.12.2). Если теперь провести указателем мыши по шторке в обратном направлении, она вновь раздвинется и фотография упряжки скроется за ней. Не торопитесь, ведите мышь плавно. Если поспешить и сделать резкое движение, указатель «соскочит» со шторки и она перестанет открываться. Правда, в этом нет ничего страшного. Достаточно

вернуть курсор на место и продолжить плавное движение, чтобы шторка полностью открылась или закрылась. (Совет: скройте боковую панель в браузере Opera, чтобы пример работал максимально точно.)

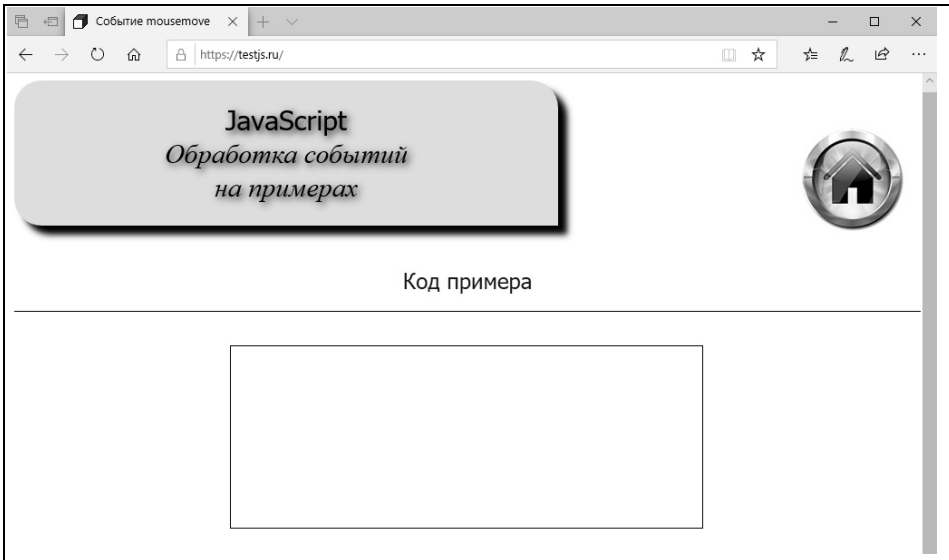

Рис. 3.12.1. Начальный вид документа очень напоминает страницу из предыдущего примера

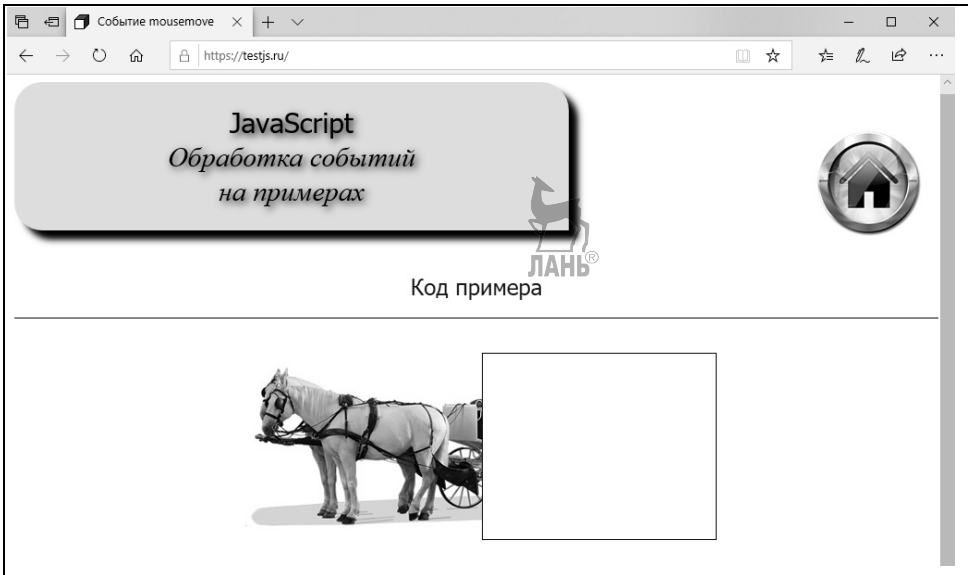

Рис. 3.12.2. Начинаем отодвигать шторку

В теле документа есть два слоя и фотография упряжки:

```
<div id="pic"><img src="pict/horse.jpg" id="horse" alt="Фото">
<div id="lay"></div></div>
```

Слой с **id="pic"** служит контейнером для остальных двух элементов. Его свойства:

```
#pic {position: relative; width: 800px; margin: 0px auto;}
```

Наиболее важное значение для правильной работы сценария имеет позиционирование слоя по центру страницы:

```
margin: 0px auto;
```

Свойства фотографии:

```
#horse {position: absolute; right: 150px; width: 500px; z-index:
10;}
```

Поверх снимка располагается слой **lay**, который выполняет роль шторки. Его настройки:

```
#lay {position: absolute; right: 140px; width: 520px; height:
200px;
      border: 1px solid #000000; background: #FFFFFF; z-index: 20;}
```

Обратите внимание – позиционирование шторки отсчитывается от правого края слоя-контейнера:

```
right: 140px;
```

Это тоже важно для правильной работы программы.

Сам код сценария довольно короткий. Регистрируем обработчик события **mousemove**:

```
window.addEventListener("load", function()
{
document.getElementById("lay").addEventListener("mousemove", wid);
});
```

Объявляем переменную **w**. Она необходима для определения размеров сжатия шторки. Эту переменную мы не просто задаем, а рассчитываем в зависимости от разрешения монитора:

```
let w=screen.width/2+270;
```

Для этого выясняем ширину экрана компьютера

```
screen.width
```

делим ее на **2** и получаем координату середины страницы, а значит и координату середины шторки. К полученному результату прибавляем число **270**. Вообще-то, ширина шторки 520 пикселей, а значит, вроде бы, чтобы полностью убрать ее, нужно в эти расчеты добавлять число 260 (то есть половину ширины слоя **lay**). Но если мы полностью уберем шторку, то после этого уже не сможем раздвинуть ее обратно. Поэтому мы взяли значение с запасом, чтобы после остановки скрипта у нас еще оставались 10 пикселей несвернутой шторки, от которых программа и будет «плясать» во время обратного движения мыши.

Подозреваю, что эти объяснения пока выглядят довольно туманно. Но вы поймете их, разбирая принцип работы функции **wid**.

Вот ее код:

```
function wid()
{
let h=event.pageX;
let p=w-h;

if(p<520)
   {
   document.getElementById("lay").style.width=p+"px";
   }
}
```

Первым делом определяется горизонтальная координата мыши

```
let h=event.pageX;
```

а затем вычисляется ширина шторки в данный момент:

```
let p=w-h;
```

Если значение переменной **p** меньше **520** (а это как раз максимальная ширина шторки), то выполняется инструкция

```
document.getElementById("lay").style.width=p+"px";
```

Пусть указатель двигается слева. В это время размеры шторки уменьшаются, так как **w** – постоянная величина, а переменная **h** все время возрастает. Поскольку позиционирование слоя со шторкой отсчитывается от правого края контейнера, то при движении мыши слева слой **lay** будет «сворачиваться» вправо, вслед за курсором. При обратном движении мыши значение переменной **p** будет увеличиваться и шторка станет разворачиваться.

Теперь необходимые пояснения к переменной **w**. По сути она представляет собой координату правого края слоя **lay** плюс еще 10 пикселей. **w=screen.width/2+270** или тоже самое, только в другой записи: **w=screen.width/2+260+10**. **screen.width/2** – координата середины страницы, а **260** – половина ширины слоя. В сумме они и дают координату правого края слоя. **10** – это наш «запас». Допустим, правая координата этого слоя равна

700 пикселей. Тогда **w** будет иметь значение 710. Представим, что курсор двигается слева направо. 670 пикселей от левого края, 680, 690 и, наконец, 700. Дальше курсор уйдет со слоя и выполнение программы прекратиться. Чему будет равна ширина слоя **lay**, когда курсор уйдет с него? Ширина вычисляется так: **p=w-h**. Подставим в это уравнение реальные числа **p=710-700**. Получим 10 пикселей. Такова будет ширина полностью развернутой шторки. Кстати, «подхватывание» шторки в тот момент, когда курсор начнет движение по ней слева, произойдет на 10 пикселей «позже». В этом легко убедиться, выполнив аналогичные расчеты для крайней левой координаты слоя или в результате соответствующего эксперимента на странице с примером. И естественно, что шторка полностью закроется при обратном движении мыши на 10 пикселей «раньше», чем мышь покинет слой **lay**.

Осталось добавить, что условие **if(p<520)** ограничивает движение шторки в обратном направлении.

У нас получился короткий сценарий с не самой простой логикой.

Скомпонуем страницу:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие mousemove</title>

<style>
#pic {position: relative; width: 800px; margin: 0px auto;}
#horse {position: absolute; right: 150px; width: 500px; z-index:
10;}
#lay {position: absolute; right: 140px; width: 520px; height:
200px;
      border: 1px solid #000000; background: #FFFFFF; z-index: 20;}
</style>

<script>
window.addEventListener("load", function()
{
document.getElementById("lay").addEventListener("mousemove", wid);
});

let w=screen.width/2+270;

function wid()
{
let h=event.pageX;
let p=w-h;

if(p<520)
   {
   document.getElementById("lay").style.width=p+"px";
   }
}
</script>
</head>
<body>
```

```
<div id="pic"><img src="pict/horse.jpg" id="horse" alt="Фото">
<div id="lay"></div></div>

</body>
</html>
```

3.13. События mousedown, mouseup и mousemove. Бабочка

В этом примере у нас задействованы обработчики сразу трех событий. К уже «апробированному» **mousemove** мы добавим два события: нажатия (**mousedown**) и отпускания (**mouseup**) кнопки мыши. При этом трудно сказать, какой из этих трех обработчиков важнее – без любого из них программа не сможет функционировать.

Пример в действии можно посмотреть на странице **https://testjs.ru/p13.html**.

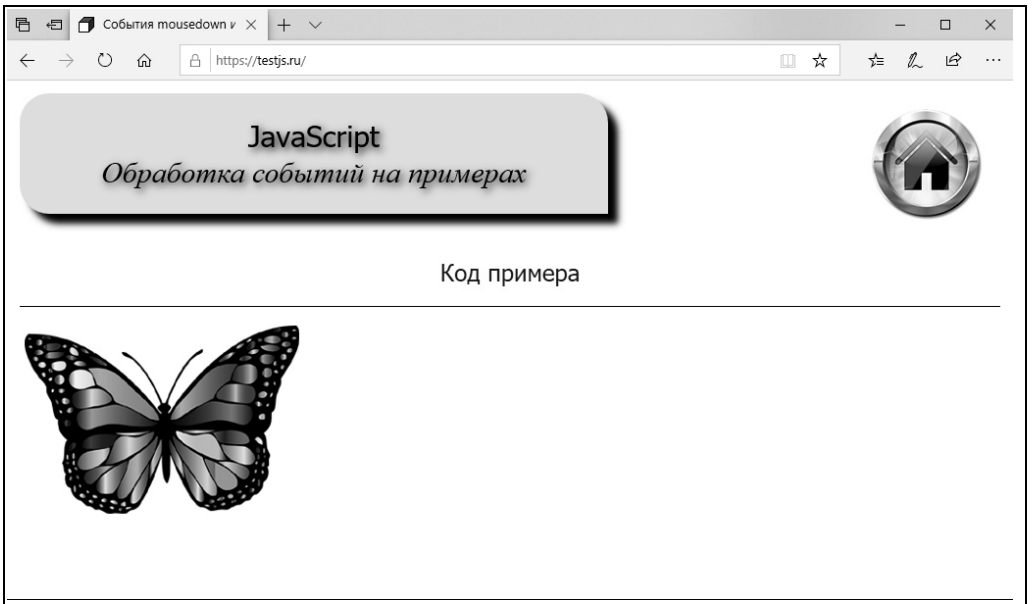

Рис. 3.13.1. Положение бабочки после загрузки страницы

После загрузки документа вы увидите в левой верхнем углу браузера изображение бабочки (рис. 3.13.1). Наведите на нее курсор и нажмите левую кнопку мыши. Теперь бабочку можно перетащить в любое место страницы. Отпустите кнопку мыши. Бабочка останется на том месте, где вы ее «бросили» (рис. 3.13.2). Снова нажмите на изображении левую кнопку и перетащите бабочку в другое место, а затем отпустите кнопку. Бабочка опять поменяла свое местоположение. Таким способом вы можете передвигать картинку в пределах окна браузера.

Некоторые из вас могут подумать, что в этом сценарии реализована технология drag-and-drop. Но это не так. В данной программе мы применили другие методы. А с технологией drag-and-drop мы познакомимся в следующем примере.

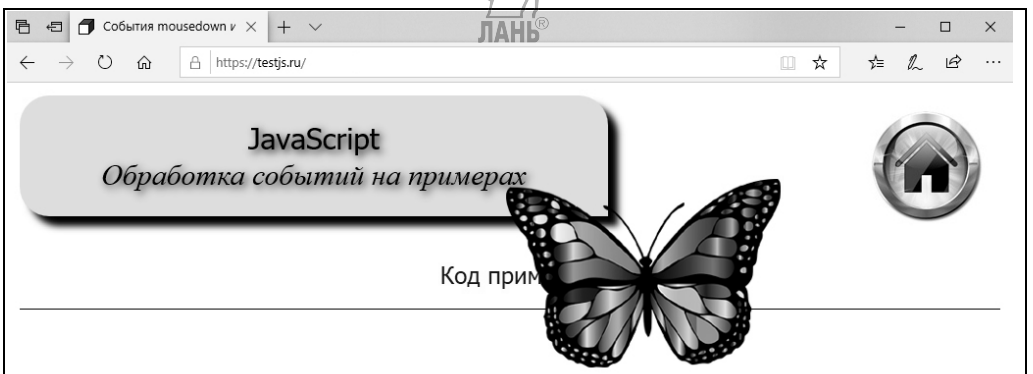

Рис. 3.13.2. Перемещаем бабочку по странице

Написание программы начнем с размещения на странице изображения бабочки:

```
<a href="#"><img src="pict/bat.png" id="bat" alt="Фото"></a>
```

Ее начальные координаты:

```
#bat {position: absolute; top: 0px; left: 0px;}
```

Чтобы читатель по изменению вида указателя мыши понял, что с бабочкой можно что-то делать, мы добавили картинке ссылку без адреса.

Перейдем к разбору примера. Как всегда, в самом начале сценария регистрируем обработчики. Их у нас три:

```
window.addEventListener("load", function()
{
document.getElementById("bat").addEventListener("mousedown", coor);
document.getElementById("bat").addEventListener("click", function()
    {
    return false;
    });
});
window.addEventListener("mouseup", stco);
```

Неожиданное вклинивание обработчика **click** я сейчас поясню. Что касается события **mousemove**, то его обработчик не нуждается в предварительной регистрации, так как будет запускаться в процессе работы сценария из другого обработчика – события **onmousedown**. Иными словами, программа обрабатывает событие **mousemove** только при условии, что произошло событие **onmousedown**.

Вернемся к событию **click**. Поскольку у нас есть ссылка без адреса, необходимо, чтобы обработка клика на ней завершалась инструкцией **return false**. Такой прием мы использовали уже неоднократно.

Пойдем дальше. Объявим две глобальные переменные:

```
let dh=0;
let dv=0;
```

Их задача сохранять разности координат между начальным положением бабочки и местом первого щелчка мышью на ней. Необходимость этих переменных станет ясна позже. Рассмотрим функцию **coor**:

```
let h=event.pageX;
let v=event.pageY;

let sh=document.getElementById("bat").offsetLeft;
let sv=document.getElementById("bat").offsetTop;

dh=h-sh;
dv=v-sv;

window.onmousemove=neco;
```

Она запускается после нажатия кнопки мыши на изображении. В первых двух строках определяются координаты указателя по осям X и Y во время нажатия кнопки. Затем выясняются координаты левого верхнего угла картинки. Теперь самое главное – вычисляем разницу между координатами указателя и рисунка:

```
dh=h-sh;
dv=v-sv;
```

Эти значения учитываются сценарием в дальнейших действиях. Каким образом? Когда мы будем вести мышью по странице, верхний левый угол рисунка будет все время находится на одном и том же – только что вычисленном – расстоянии от указателя. Собственно, благодаря этому и обеспечивается перетаскивание бабочки – она следует за указателем на постоянном расстоянии. Обеспечивает такое движение функция, которая запускается последней инструкцией обработчика **coor**:

```
window.onmousemove=neco;
```

Функция **neco** – это обработчик события **onmousemove**:

```
function neco()
{
let nh=event.pageX;
let nv=event.pageY;

let fh=nh-dh;
let fv=nv-dv;

document.getElementById("bat").style.left=fh+"px";
document.getElementById("bat").style.top=fv+"px";

return false;
}
```

В первых двух строках мы выясняем новые координаты указателя после того, как мышь начала движение по странице. Затем вычисляем смещение, которое должна получить бабочка при данном смещении курсора

```
let fh=nh-dh;
let fv=nv-dv;
```

и перемещаем бабочку в новое место:

```
document.getElementById("bat").style.left=fh+"px";
document.getElementById("bat").style.top=fv+"px";
```

Такие инструкции выполняются непрерывно, пока указатель «скользит» по странице. За счет этого бабочка тоже «порхает» за курсором.

Последняя инструкция

```
return false;
```

«защищает» сценарий от некорректной обработки события **onmousemove**. Не будь этой инструкции, бабочка начала бы двигаться по совершенно иной схеме (от себя добавлю, совершенно непрезентабельной).

Теперь представим, что мы перетащили бабочку и отпустили кнопку мыши. Произойдет событие **mouseup**, на которое откликнется функция **stco**:

```
function stco()
{
window.onmousemove=null;
}
```

Ее единственная задача – прекратить обработку события **onmousemove**, благодаря чему после отпускания кнопки мыши изображение останется на новом месте.

Еще раз переместите бабочку, выбрав другую точку на картинке. Красавица будет следовать за указателем мыши на новом, но опять постоянном расстоянии.

Можно пробовать разные варианты переноса бабочки: тянуть ее за кончик крылышка, за усик или хвостик. Во всех этих случаях вы увидите, что выбранный орган бабочки все время находится строго под курсором.

Завершим рассказ о такой необычной программе, написав ее полный код:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>События mousedown и mouseup</title>

<style>
#bat {position: absolute; top: 0px; left: 0px;}
</style>

<script>
window.addEventListener("load", function()
{
document.getElementById("bat").addEventListener("mousedown", coor);
document.getElementById("bat").addEventListener("click", function()
   {
   return false;
   });
};;
window.addEventListener("mouseup", stco);

let dh=0;
let dv=0;

function coor()
{
let h=event.pageX;
let v=event.pageY;

let sh=document.getElementById("bat").offsetLeft;
let sv=document.getElementById("bat").offsetTop;

dh=h-sh;
dv=v-sv;

window.onmousemove=neco;
}

function neco()
{
let nh=event.pageX;
let nv=event.pageY;

let fh=nh-dh;
let fv=nv-dv;

document.getElementById("bat").style.left=fh+"px";
document.getElementById("bat").style.top=fv+"px";

return false;
}

function stco()
{
```

```
window.onmousemove=null;
}
</script>
</head>

<body>

<a href="#"><img src="pict/bat.png" id="bat" alt="Фото"></a>

</body>
</html>
```

3.14. События dragstart, dragover и drop. За нектаром

Наши эксперименты с бабочкой продолжаются. В данном примере мы опробуем в действии технологию drag-and-drop.

Небольшое отступление. Не раз видел на разных сайтах в Интернете, как гуру программирования, рассказывая начинающим о данной технологии, приводят такие сложные примеры, что этим самым начинающим разобраться в них крайне непросто. Между тем в drag-and-drop нет ничего сложного. Получить желаемый эффект можно, написав минимум кода. Рассматривая пример из данного параграфа, читатели убедятся в этом.

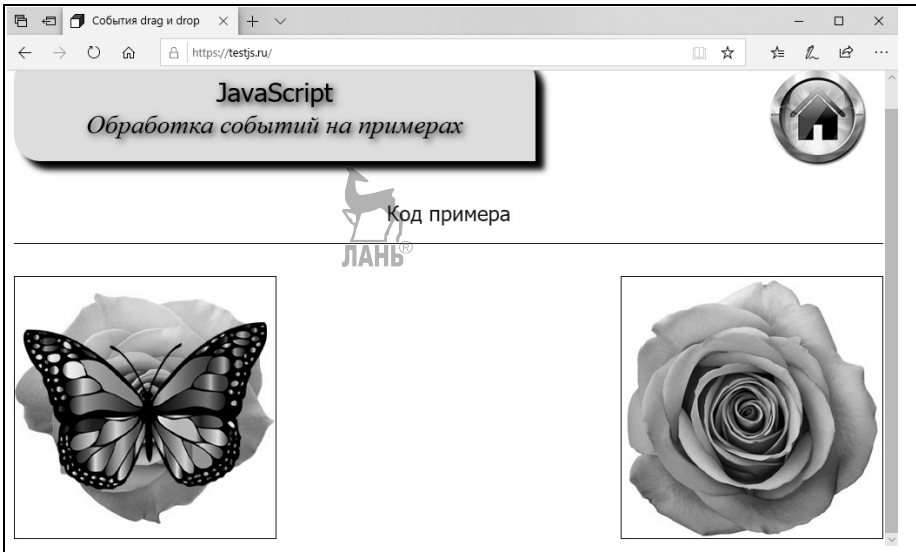

Рис. 3.14.1. После загрузки страницы бабочка «сидит» на левом цветке

Зайдите на страницу **https://testjs.ru/p14.html** сайта поддержки книги. Вы увидите картинку, аналогичную той, что изображена на рисунке 3.14.1. Слева и справа изображения двух разноцветных роз. На левой сидит уже знакомая нам бабочка. Цветы находятся в рамках для более наглядных объяснений принципов работы примера.

Наведите на бабочку указатель мыши, нажмите левую кнопку, перетащите бабочку на правую розу и отпустите кнопку. Бабочка останется на правом цветке (рис. 3.14.2). Что и требовалось в нашем примере. Можно провести эксперимент: перетаскивая картинку, отпустить ее между двумя розами. Поскольку изображение не достигло узла-приемника, оно останется на левой розе. Так будет во всех упомянутых во «Введении» браузерах, кроме, как вы, наверное, уже догадались, Firefox. В нем после такого эксперимента загрузится страница с изображением бабочки на черном фоне – что неудивительно для этого «особенного» web-обозревателя. Но в остальном пример работает в Firefox корректно.

Чтобы написать программу перемещения бабочки, нам необходимо создать обработчики для трех событий: **dragstart, dragover и drop**.

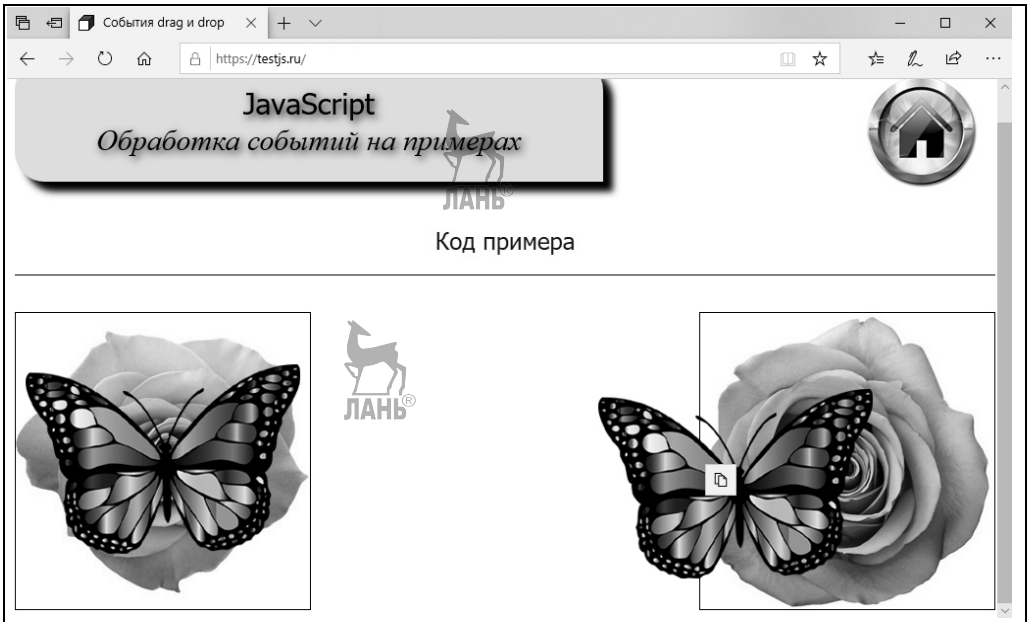

Рис. 3.14.2. Перемещаем бабочку на правый цветок

Но начнем мы как всегда с создания необходимых элементов в теле страницы. Для размещения изображений и создания между ними достаточного интервала, поместим в документ таблицу с тремя ячейками:

```
<table>
<tr>
<td id="td1"><img src="pict/bat.png" draggable="true" id="im"

alt="Фото"></td>
<td id="td2"></td>
<td id="td3"></td>
</tr>
</table>
```

В левой ячейке находится рисунок бабочки

```
<td id="td1"><img src="pict/bat.png" draggable="true" id="im"
alt="Фото"></td>
```

а в качестве фона – изображение розы:

```
#td1 {width: 280px; height: 280px; background: url(pict/ros1.jpg);
padding: 10px; border: 1px solid #000000;}
```

Средняя ячейка создает зазор в **400** пикселей между крайними ячейками:

```
#td2 {width: 400px;}
```

Правая ячейка в исходном состоянии пустая. Она будет выполнять роль приемника для перемещаемого изображения. В этой ячейке фоном тоже служит изображение цветка:

```
#td3 {width: 280px; height: 280px; background: url(pict/ros2.jpg);
      padding: 10px; border: 1px solid #000000;}
```

Крайние ячейки имеют рамки, показывая тем самым, что слева находится контейнер, содержащий исходный рисунок, а справа – узел, служащий приемником для этого рисунка при его перетаскивании.

Необходимые элементы расположились на странице, поэтому теперь можно зарегистрировать обработчики событий:

```
window.addEventListener("load", function()
{
document.getElementById("im").addEventListener("dragstart", sta);
document.getElementById("td2").addEventListener("dragover", mid);
document.getElementById("td3").addEventListener("dragover", mid);
document.getElementById("td3").addEventListener("drop", fin);
});
```

Разберем обработчики по порядку.

Первый – для события **dragstart** – запускается, когда мы подхватываем цветок мышью:

```
function sta()
{
event.dataTransfer.setData("image", "im");
}
```

Объект **DataTransfer** устанавливает операцию перетаскивания к указанным данным и их типу методом **setData()**. Тип данных у нас **image**, а на сами данные указывает ID изображения **im**.

Перетаскивание бабочки началось. Оно проходит по средней и правой ячейке. По умолчанию браузеры блокируют перетаскивание элементов. Чтобы отменить такое поведение web-обозревателя, у нас есть функция **mid**, которая обрабатывает события **ondragover**, происходящие в центральной и правой ячейках:

```
function mid()
{
event.preventDefault();
}
```

Метод **preventDefault()** как раз и отменяет блокировку перетаскивания.

Наконец последний этап – обработка события **drop**. Оно происходит, когда вы отпускаете изображение в узле-приемнике. Для этого события у нас есть функция **fin**:

```
function fin()
{
event.preventDefault();
docu-
ment.getElementById("td3").appendChild(document.getElementById("im
"));
}
```

Метод **appendChild()** добавляет элемент в конец списка дочерних элементов данного родительского узла. В качестве родительского узла или контейнера-приемника у нас выступает правая ячейка, ID которой **td3**.

Наш пример для упрощения кода написан для перетаскивания бабочки только в одном направлении. Вернуть бабочку на левый цветок уже не получится (если только не перезагрузить страницу).

Как видите, сценарий получился очень простым и понятным. Вот код страницы:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>События drag и drop</title>

<style>
#td1 {width: 280px; height: 280px; background: url(pict/ros1.jpg);
      padding: 10px; border: 1px solid #000000;}
#td2 {width: 400px;}
#td3 {width: 280px; height: 280px; background: url(pict/ros2.jpg);
      padding: 10px; border: 1px solid #000000;}
</style>

<script>
window.addEventListener("load", function()
{
document.getElementById("im").addEventListener("dragstart", sta);
document.getElementById("td2").addEventListener("dragover", mid);
document.getElementById("td3").addEventListener("dragover", mid);
```

```
document.getElementById("td3").addEventListener("drop", fin);
});

function sta()
{
event.dataTransfer.setData("image", "im");
}

function mid()
{
event.preventDefault();
}

function fin()
{
event.preventDefault();
document.getElementById("td3").appendChild(document.getElementById
("im"));
}
</script>
</head>

<body>

<table>
<tr>
<td id="td1"><img src="pict/bat.png" draggable="true" id="im"

alt="Фото"></td>
<td id="td2"></td>
<td id="td3"></td>
</tr>
</table>

</body>
</html>
```

В завершении рассказа еще один небольшой комментарий. На самом деле технология drag-and-drop настолько проста, что мы могли бы сократить в нашем сценарии функцию, связанную с событием **dragstart**. И все равно сценарий продолжал бы работать корректно. Но я не стал этого делать, чтобы продемонстрировать вам обработку трех событий **dragstart, dragover и drop**, без которых в более сложных программах не обойтись.

3.15. Событие resize. Меняем кегль

Сначала определимся, что такое кегль. Это размер буквы или знака шрифта по вертикали. Наша новая программа будет менять размер шрифта в зависимости от размера окна браузера. Событие изменения размера окна браузера именуется **resize**. Вот мы и займемся обработкой этого события.

Пример можно посмотреть по адресу **https://testjs.ru/p15.html**.

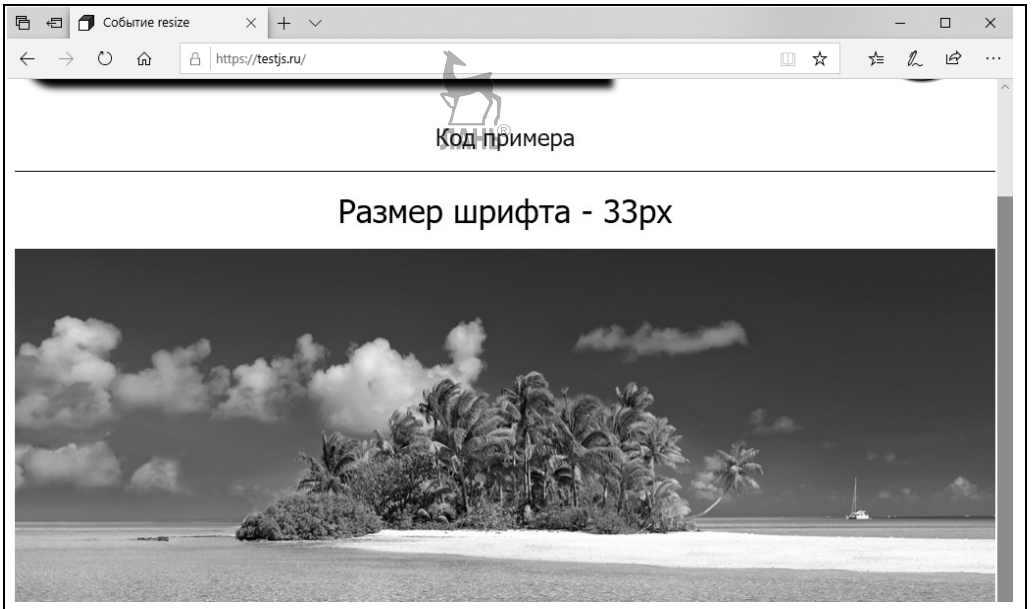

Рис. 3.15.1. Размер шрифта при горизонтальной ширине страницы 1000 пикселей

На странице есть изображение кораллового острова и сверху надпись, сообщающая о текущем размере шрифта в пикселях на данный момент (рис. 3.15.1):

```
<span id="te">Размер шрифта - <span id="re"></span></span><br><br>
<img src="pict/land.jpg" id="im" alt="Фото">
```

Если теперь в правом верхнем углу браузера нажать изображение прямоугольника (или двух прямоугольников – в разных web-обозревателях по разному; эта кнопка еще называется «Свернуть в окно»), то вы увидите, что размер картинки остался прежним и она уже не помещается в окне, а шрифт изменился – теперь он стал, например, 24 пикселя (рис. 3.15.2). То есть изменение размера шрифта пропорционально изменению размера окна, что особенно заметно на фоне стабильно крупной фотографии. Если продолжать сворачивать окно web-обозревателя дальше, то в какой-то момент шрифт достигнет размера 20 пикселей и его уменьшение на этом прекратится.

Можно снова начать разворачивать окно браузера – размер шрифта станет увеличиваться до тех пор, пока окно браузера не займет весь экран.

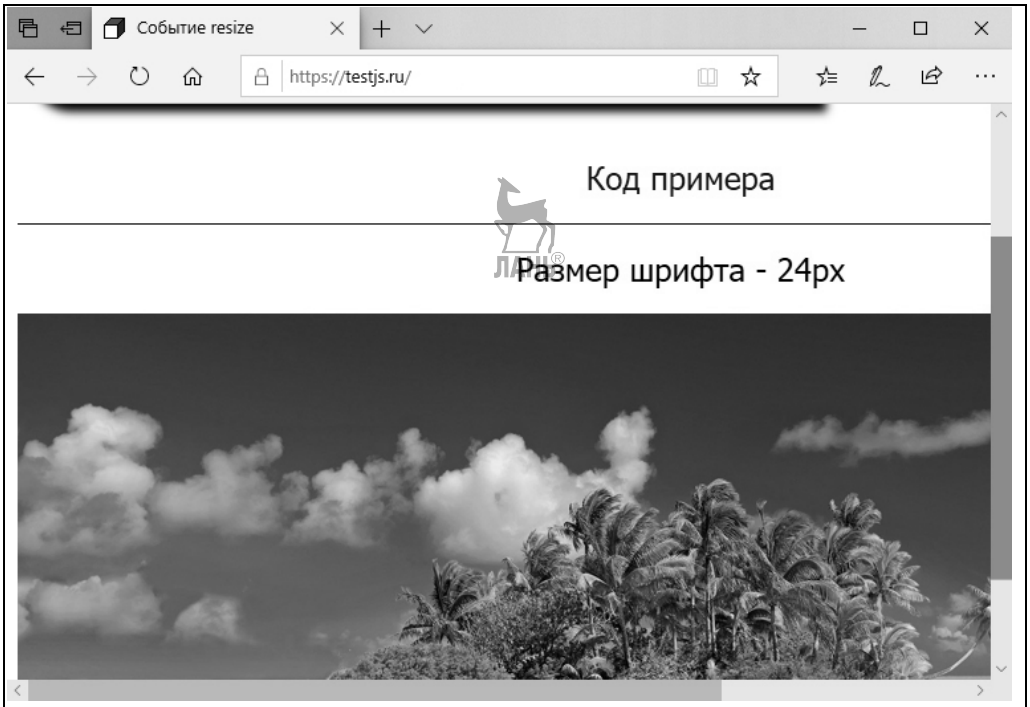

Рис. 3.15.2. Размер шрифта уменьшается по мере уменьшения окна браузера

Управляет таким поведением надписи довольно простой сценарий. В нем есть один обработчик, который запускается двумя событиями уровня окна – **load** и **resize**:

```
window.addEventListener("load", res);
window.addEventListener("resize", res);
```

Оба обращаются к функции **res**, которая производит необходимые манипуляции со шрифтом.

Первый раз функция срабатывает после загрузки документа, формируя надпись, размер шрифта которой зависит от размеров окна браузера. Во втором и последующих случаях функция запускается при изменении размеров окна браузера. Функция выполняет следующие инструкции:

```
let w=document.body.clientWidth;
let a=Math.round(w/30);

if(w>600)
   {
   document.getElementById("te").style.fontSize=a+"px";
   document.getElementById("re").innerHTML=a+"px";
   }
else
   {
```

```
    document.getElementById("te").style.fontSize="20px";
    document.getElementById("re").innerHTML="20px";
    }
```

Сначала определяется текущий размер окна браузера:

```
let w=document.body.clientWidth;
```

Затем высчитывается размер шрифта, который должен соответствовать данному размеру окна:

```
let a=Math.round(w/30);
```

Мы выбрали соотношение между размерами шрифта и окна 1 к 30. Метод **round()** объекта **Math** округляет результат деления до ближайшего целого значения.

На следующем шаге проверяется условие **if(w>600)**. Если оно истинно, то выполняются операторы первого блока. Сначала мы меняем размера шрифта информационной надписи

```
document.getElementById("te").style.fontSize=a+"px";
```

а потом числовой показатель:

```
document.getElementById("re").innerHTML=a+"px";
```

В результате на странице мы видим сообщение о текущих параметрах текста.

Если условие **if(w>600)** ложно, это означает, что окно браузера достигло пограничного размера, при уменьшении которого высота шрифта уже не будет меняться. В этом случае выполняются инструкции второго блока:

```
document.getElementById("te").style.fontSize="20px";
document.getElementById("re").innerHTML="20px";
```

Подведем черту:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие resize</title>

<script>
window.addEventListener("load", res);
window.addEventListener("resize", res);

function res()
{
let w=document.body.clientWidth;
let a=Math.round(w/30);
```

```
if(w>600)
   {
   document.getElementById("te").style.fontSize=a+"px";
   document.getElementById("re").innerHTML=a+"px";
   }
else
   {
   document.getElementById("te").style.fontSize="20px";
   document.getElementById("re").innerHTML="20px";
   }
}
</script>
</head>

<body>

<span id="te">Размер шрифта - <span id="re"></span></span><br><br>
<img src="pict/land.jpg" id="im" alt="Фото">

</body>
</html>
```

3.16. Событие resize. «Махнемся» грибами

Попробуем еще как-нибудь манипулировать событием **resize**. Например, менять изображение на странице в зависимости от изменения размеров окна браузера. А попутно сообщать результаты сворачивания окна в процентах. Для этого создан пример, демонстрируемый по адресу **https://testjs.ru/p16.html**. В нем высчитывается отношение ширины рабочей области окна web-обозревателя к ширине экрана компьютера и, в зависимости от этого показателя, меняется изображение.

Зайдите на страницу с примером. Если окно браузера развернуто на весь экран, вы увидите изображение белого гриба, а над ним короткий текст, сообщающий, что ширина окна браузера около 100%. Скорее всего, будет указано 99 или 98%. Такое отклонение от максимального показателя получается за счет боковых рамок окна браузера.

Попробуем понемногу уменьшать размеры окна. Пока процентный показатель выше 70, вы по-прежнему видите изображение белого гриба (рис. 3.16.1). Если свернуть окно меньше, чем на 70%, появится фото семейства мухоморов (рис. 3.16.2). Теперь как не уменьшай размеры окна, на странице будут оставаться мухоморы. Развернем окно больше, чем на 70%, – и снова появится снимок белого гриба.

Разберем пример.

В теле документа у нас есть изображение и текст сообщения:

```
Ширина окна браузера - <span id="re"></span><br><br>
<img id="photo" src="pict/net.jpg" alt="Фото">
```

Поскольку программе заранее неизвестно, насколько будет развернуто окно браузера в момент загрузки данных, в качестве исходного фото мы выбрали уже известную вам «заглушку»:

```
src="pict/net.jpg"
```

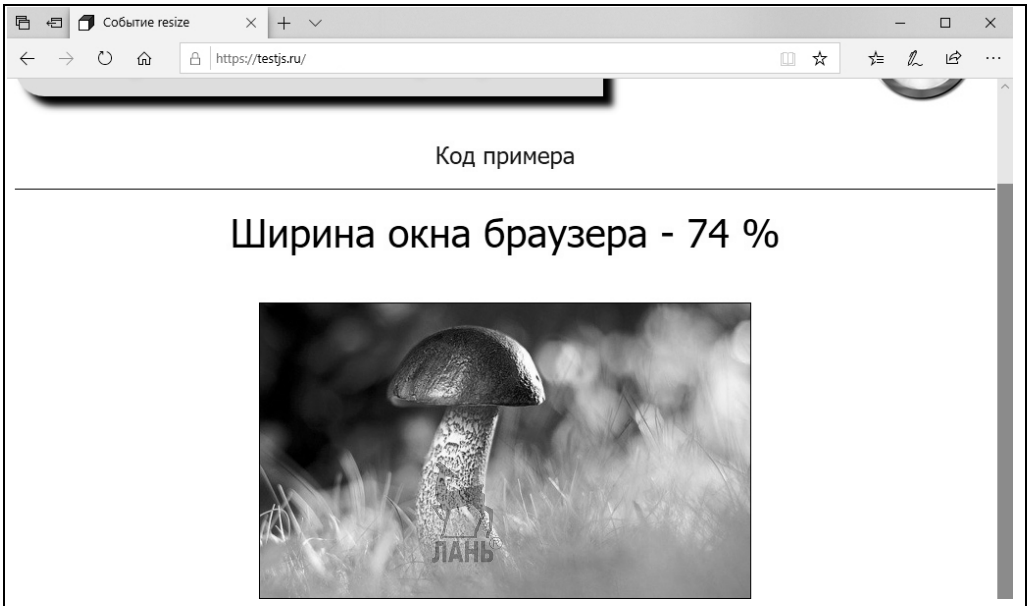

Рис. 3.16.1. При ширине окна браузера 1000 пикселей на странице показывается фото белого гриба

Как и в предыдущем случае, у нас вновь один обработчик для двух событий уровня окна – **load** и **resize**:

```
window.addEventListener("load", res);
window.addEventListener("resize", res);
```

Оба запускают функцию **res**, которая определяет размер окна и какую картинку необходимо вывести на экран.

Как и в примере со шрифтом, сначала выполняются необходимые вычисления:

```
let s=screen.width;
let w=document.body.clientWidth;
let a=Math.round(100*w/s);
```

Первым делом узнаем ширину экрана:

```
let s=screen.width;
```

Следом определяем ширину окна браузера:

```
let w=document.body.clientWidth;
```

А затем высчитываем проценты:

```
let a=Math.round(100*w/s);
```

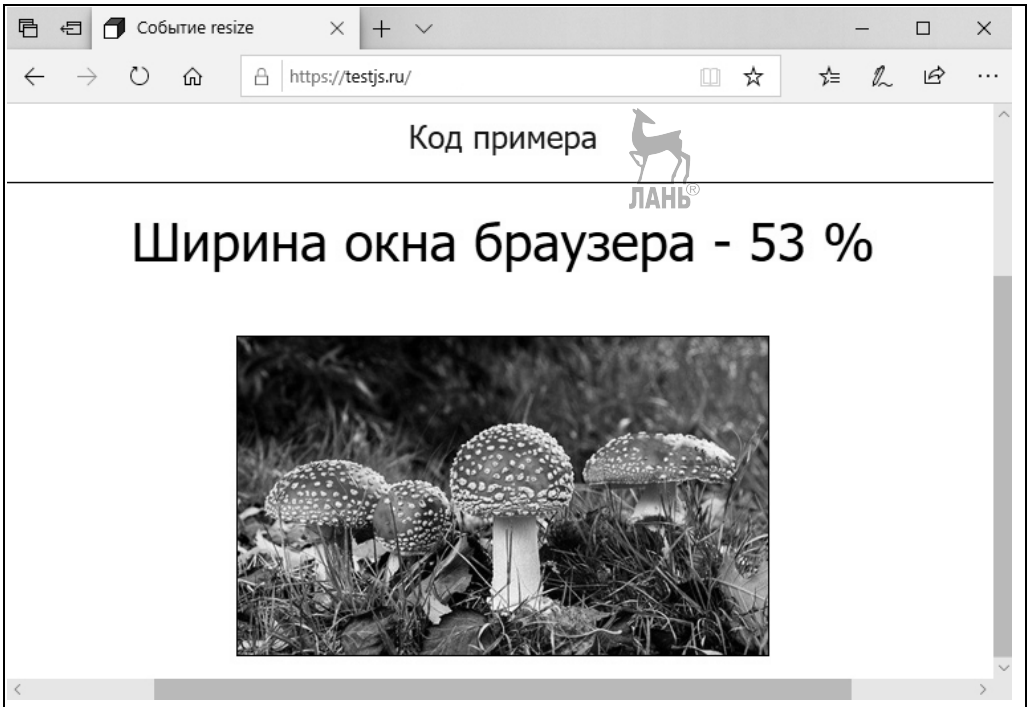

Рис. 3.16.2. Ширина окна браузера уменьшилась почти наполовину и теперь появилось фото семейства мухоморов

Выводим сообщение о ширине окна браузера:

```
document.getElementById("re").innerHTML=a+" %";
```

Проверяем условие **if(a<70)**. Если ширина окна меньше 70%, выводим на страницу фотографию мухоморов:

```
document.getElementById("photo").src="pict/matr.jpg";
```

Если ширина окна больше или равна 70%, демонстрируем снимок белого гриба:

```
document.getElementById("photo").src="pict/mush.jpg";
```

Наш второй пример с событием **resize** получился таким же простым, как и первый. В этом легко убедиться, взглянув на полный код страницы:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие resize</title>
```

```
<script>
window.addEventListener("load", res);
window.addEventListener("resize", res);

function res()
{
let s=screen.width;
let w=document.body.clientWidth;
let a=Math.round(100*w/s);

document.getElementById("re").innerHTML=a+" %";
if(a<70)
   {
   document.getElementById("photo").src="pict/matr.jpg";
   }
else
   {
   document.getElementById("photo").src="pict/mush.jpg";
   }
}
</script>
</head>

<body>

Ширина окна браузера - <span id="re"></span><br><br>
<img id="photo" src="pict/net.jpg" alt="Фото">

</body>
</html>
```

3.17. События focus и blur. Фокус с подсказкой

В этом параграфе мы будем экспериментировать с обработчиками двух взаимосвязанных событий: **focus** и **blur**. Как вы помните, событие **focus** возникает при получении элементом фокуса. Событие **blur** происходит, когда элемент теряет фокус.

Мы используем данные события для вывода подсказок о том, как правильно заполнить поля формы перед ее отправкой.

У нас есть форма, показанная на демонстрационном сайте: **https://testjs.ru/p17.html** (рис. 3.17.1). Предположим, вы решили заполнить поле «Ваше имя». Как только вы установили курсор мыши в этом поле, над ним появляется подсказка о том, что вы можете ввести не только настоящее имя, но и свой псевдоним (рис. 3.17.2). Если переместить курсов в нижнее поле, над ним появится подсказка «Введите адрес вашей электронной почты». Обе подсказки облегчат посетителю правильное заполнение формы.

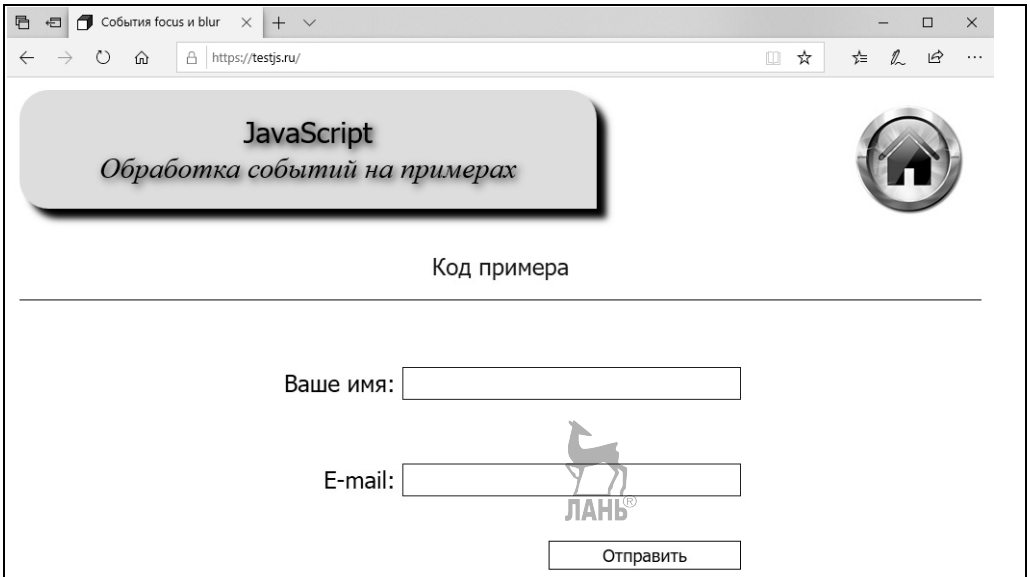

Рис. 3.17.1. Внешний вид страницы до получения фокуса одним из полей формы

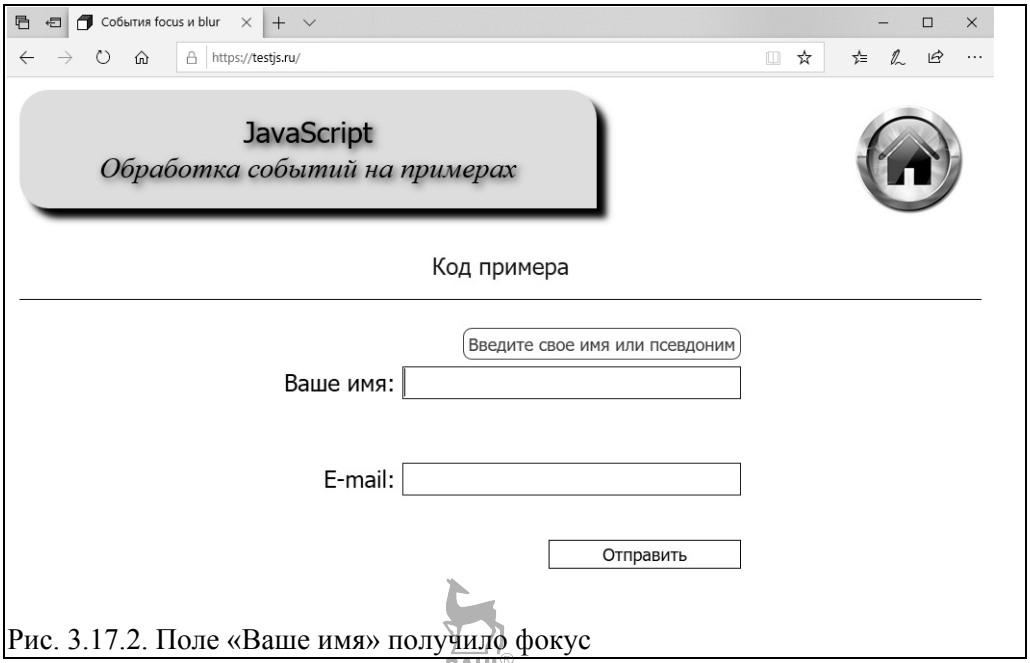

Рис. 3.17.2. Поле «Ваше имя» получило фокус

Полезность такого взаимодействия с клиентом, думаю, очевидна каждому. Ведь вы ждете от посетителя данных, не содержащих ошибок и неточностей.

Перейдем к делу. Поместим на странице текстовые поля

```
<div id="for1">Ваше имя: <input type="text" id="f1"
maxlength="30"></div>
<div id="for2">E-mail: <input type="text" id="f2"
maxlength="30"></div>
```

а под ними кнопку отправки данных:

```
<input type="button" id="bu" value="Отправить">
```

Так как нам желательно добиться правильного взаимного расположения полей и подсказок, добавим элементам формы необходимое позиционирование:

```
#for1 {position: absolute; top: 50px;}
#for2 {position: absolute; top: 100px;}
#bu {position: absolute; top: 150px;}
```

Поместим в тело документа подсказки:

```
<span id="att1">Введите свое имя или псевдоним</span>
<span id="att2">Введите адрес вашей электронной почты</span>
```

Оформим их позиционирование:

```
#att1 {position: absolute; top: 30px; display: none;}
#att2 {position: absolute; top: 80px; display: none;}
```

В исходном состоянии подсказки не видны, так как их свойства **display** имеют значения **none**.

Зарегистрируем четыре обработчика событий **focus** и **blur** (по два на каждое поле), в роли которых выступят анонимные функции:

```
window.addEventListener("load", function()
{
document.getElementById("f1").addEventListener("focus", function()
   {
   document.getElementById("att1").style.display="inline";
   });
document.getElementById("f1").addEventListener("blur", function()
   {
   document.getElementById("att1").style.display="none";
   });

document.getElementById("f2").addEventListener("focus", function()
   {
   document.getElementById("att2").style.display="inline";
   });
document.getElementById("f2").addEventListener("blur", function()
   {
   document.getElementById("att2").style.display="none";
   });
});
```

Обрабатываются события предельно просто. Когда курсор устанавливается в поле «Ваше имя», подсказка для него становится видна

```
document.getElementById("att1").style.display="inline";
```

Когда курсор покидает данное поле, подсказка исчезает:

```
document.getElementById("att1").style.display="none";
```

Похожим образом все происходит и с полем «E-mail». Курсор в поле – появляется подсказка:

```
document.getElementById("att2").style.display="inline";
```

Поле лишилось фокуса – подсказка исчезла:

```
document.getElementById("att2").style.display="none";
```

Вот таким простым способом можно оказать помощь клиенту при заполнении формы.

Страница примера целиком:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>События focus и blur</title>

<style>
#for1 {position: absolute; top: 50px;}
#for2 {position: absolute; top: 100px;}
#bu {position: absolute; top: 150px;}
#att1 {position: absolute; top: 30px; display: none;}
#att2 {position: absolute; top: 80px; display: none;}
</style>

<script>
window.addEventListener("load", function()
{
document.getElementById("f1").addEventListener("focus", function()
   {
   document.getElementById("att1").style.display="inline";
   });
document.getElementById("f1").addEventListener("blur", function()
   {
   document.getElementById("att1").style.display="none";
   });

document.getElementById("f2").addEventListener("focus", function()
   {
   document.getElementById("att2").style.display="inline";
   });
document.getElementById("f2").addEventListener("blur", function()
   {
```

```
   document.getElementById("att2").style.display="none";
   });
});
</script>
</head>

<body>

<span id="att1">Введите свое имя или псевдоним</span>
<span id="att2">Введите адрес вашей электронной почты</span>
<div id="for1">Ваше имя: <input type="text" id="f1"
maxlength="30"></div>
<div id="for2">E-mail: <input type="text" id="f2"
maxlength="30"></div>
<input type="button" id="bu" value="Отправить">

</body>
</html>
```

3.18. Событие submit. Проверка формы

Событие **submit** возникает при отправке формы. В предыдущем примере мы научились давать клиенту необходимые подсказки по заполнению полей формы. В этом примере мы покажем, как использовать событие **submit** для проверки корректности заполнения формы. Добавлю, что проверка не будет исчерпывающей. Для упрощения сценария мы будем выяснять только количество знаков, введенных в полях, и сравнивать их с минимальными требованиями.

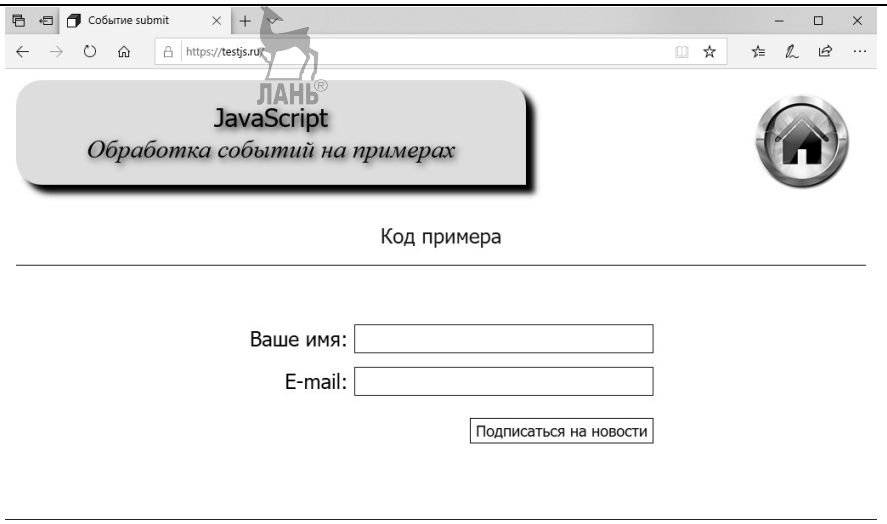

Рис. 3.18.1. Так выглядит форма после загрузки страницы

Есть демонстрационная страница с уже знакомой нам формой: **https://testjs.ru/p18.html** (рис. 3.18.1). Только вместо кнопки «Отправить» у нас

теперь кнопка «Подписаться на новости». То есть мы имитируем ситуацию, когда посетитель подписывается на рассылку новостей с нашего сайта. Обращаю ваше внимание, что этот пример – учебный. В нем не происходит реальная отправка формы. Вместо этого при отправке загружается страница-заглушка (наподобие рисунков-заглушек). Ее задача показать, что форма заполнена правильно и данные могут быть отправлены. Если вы хотите реально отправлять сообщения от посетителей, вам необходимо освоить один из языков написания серверных программ – PHP, Python, Perl или другой. Тогда вы сможете писать программы, принимающие данные из ваших web-страниц.

Итак, если мы ввели недостаточное количество знаков в одном из полей, сценарий выдаст предупреждение об этом. Чтобы предупреждения были особенно наглядными, все сообщения мы будем выводить в качестве надписи кнопки отправки данных (рис. 3.18.2). Если оба поля содержат необходимое количество символов, после отправки формы будет загружаться страница с текстом «Ваша подписка зарегистрирована!».

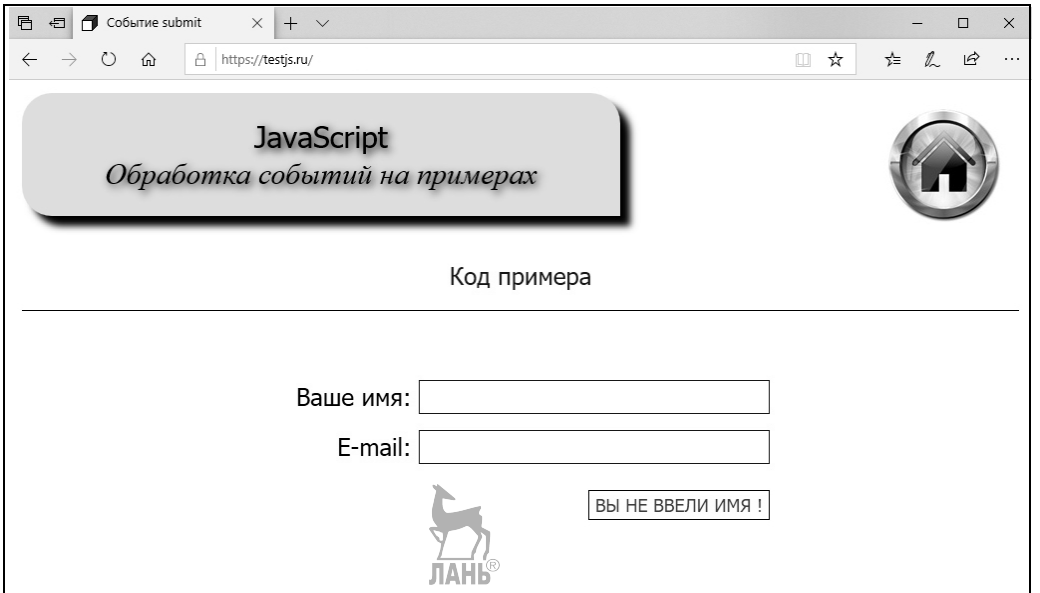

Рис. 3.18.2. Попытка отправить незаполненную форму приводит к появлению предупреждения

Для такого примера необходимо создать документ, содержащий форму с двумя полями:

```
<form method="POST" action="sub.html" name="fo">
Ваше имя: <input type="text" id="f1" maxlength="30"><br><br>
E-mail: <input type="text" id="f2"  maxlength="30"><br><br>
<input type="submit" id="bu" value="Подписаться на ново-
сти"></form>
```

Позиционирование теперь не играет роли, поэтому элементы формы просто находятся один под другим.

Регистрируем обработчик события **submit**, который станет проверять корректность заполнения полей:

```
<script>
window.addEventListener("load", function()
{
document.forms["fo"].onsubmit=test;
});
```

Начинаем проверку поля «Ваше имя»:

```
if(document.getElementById("f1").value.length<2)
   {
   ...
   }
```

Самые короткие имена состоят из двух букв (например, Ян, Ия). Если количество введенных знаков меньше **2**, то на кнопку выводится предупреждение

```
document.getElementById("bu").value="ВЫ НЕ ВВЕЛИ ИМЯ !";
```

которое оформляется красным цветом:

```
document.getElementById("bu").style.color="#CC0000";
```

после чего сценарий отказывает в отправке формы:

```
return false;
```

Если количество введенных букв **2** или больше, наступает следующий этап – проверка количества знаков в поле «E-mail»:

```
if(document.getElementById("f2").value.length<8)
```

Самый короткий адрес электронной почты должен иметь длину не менее 8 символов: два в начале + @ + два символа доменного имени второго уровня + точка + два символа доменного имени первого уровня (теоретически возможны и более короткие имена, но они настолько уникальное явление, что учитывать их существование не имеет смысла). Если это условие не выполнено, выводится соответствующее предупреждение (и тоже красным цветом – для привлечения внимания клиента):

```
document.getElementById("bu").style.color="#CC0000";
document.getElementById("bu").value="ВЫ НЕ ВВЕЛИ E-MAIL !";
```

и вновь сценарий отказывает в отправке формы:

```
return false;
```

Ели же оба поля заполнены верно, отправка сообщения разрешается:

```
else
   {
   return true;
   }
```

Пример целиком:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие submit</title>

<script>
window.addEventListener("load", function()
{
document.forms["fo"].onsubmit=test;
});

function test()
{
if(document.getElementById("f1").value.length<2)
   {
   document.getElementById("bu").style.color="#CC0000";
   document.getElementById("bu").value="Вы НЕ ВВЕЛИ ИМЯ !";
   return false;
   }
else
   {
   if(document.getElementById("f2").value.length<8)
      {
      document.getElementById("bu").style.color="#CC0000";
      document.getElementById("bu").value="Вы НЕ ВВЕЛИ E-MAIL !";
      return false;
      }
   else
      {
      return true;
      }
   }
}
</script>
</head>

<body>

<form method="POST" action="sub.html" name="fo">
Ваше имя: <input type="text" id="f1" maxlength="30"><br><br>
E-mail: <input type="text" id="f2"  maxlength="30"><br><br>
<input type="submit" id="bu" value="Подписаться на ново-
сти"></form>

</body>
</html>
```

Хочу предупредить вас, что проверка, которую мы рассмотрели в данном примере, действенна, если на сайт зашел добросовестный клиент. А если ее посетил недоброжелатель? Он легко может создать аналогичную форму без проверочного кода и отправить на ваш сайт все, что пожелает. Таким «товарищам» надо создать непреодолимые барьеры. А для этого все данные требуется проверять не только на странице сайта, но и на сервере. Причем, контроль на стороне сервера должен быть более строгим, ведь именно здесь ставится реальный барьер для недоброжелателей (так как добраться до серверной программы недоброжелатель не в состоянии). Надо обязательно проверять объем входящей информации, типы файлов, наличие исполняемого кода и запрещенных символов, соответствие ссылок требуемому формату и разрешенным адресам.

3.19. Событие change. Водоемы

Вспомним, что событие **change** происходит на элементах **select**, **input** и **textarea**, когда меняются их значения. Мы для знакомства с обработкой этого события выберем элемент **select**.

Страница с примером находятся по адресу **https://testjs.ru/p19.html**.

Итак, есть выпадающий список с перечнем различных типов водоемов: Море – Река – Озеро – Пруд – Бассейн – Лужа. А под списком соответствующая картинка – как на рисунке 3.19.1.

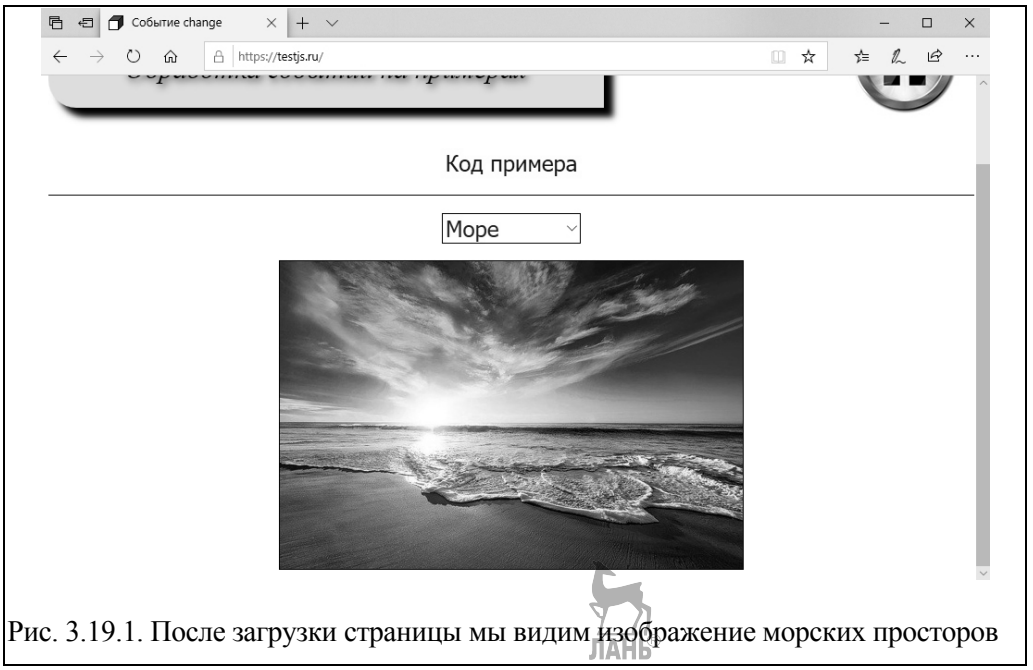

Рис. 3.19.1. После загрузки страницы мы видим изображение морских просторов

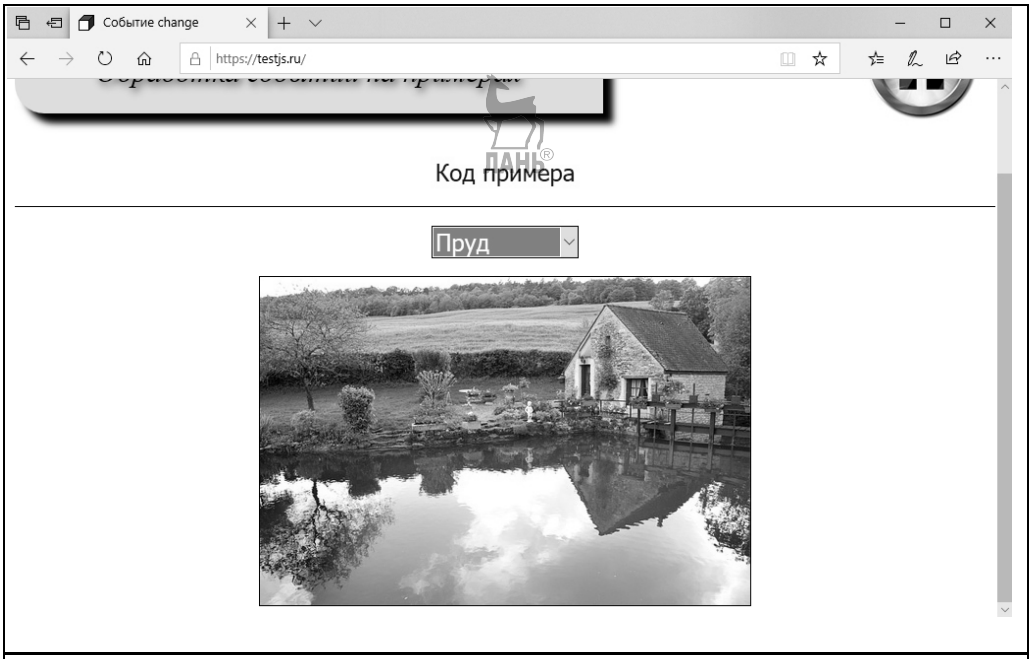

Рис. 3.19.2. Сейчас мы выбрали изображение сельского пруда

Выбирайте в любом порядке пункты списка – и каждый раз на экране будет появляться фотография соответствующего водоема (рис. 3.19.2). Все это происходит благодаря обработчику события **change**, который «внедрен» в код страницы с примером. Посмотрим на него.

В теле документа есть выпадающий список с перечнем водоемов из шести пунктов:

```
<select id="sel"><option selected value="1">Море</option>
<option value="2">Река</option>
<option value="3">Озеро</option>
<option value="4">Пруд</option>
<option value="5">Бассейн</option>
<option value="6">Лужа</option></select>
```

Каждому из пунктов присвоено цифровое значение от **1** до **6**. В исходном состоянии выбран пункт со значением **1** – «Море».

Ниже расположилось фото:

```
<img src="reser/1.jpg" id="photo" alt="Фото">
```

Сразу скажу, что все фотографии имеют в адресе цифры, соответствующие значениям списка.

Действия посетителя на язык компьютерных команд переводит обработчик

```
window.addEventListener("load", function()
{
```

```
document.getElementById("sel").addEventListener("change", function()
   {
   let a=document.getElementById("sel").value;
   document.getElementById("photo").src="reser/"+a+".jpg";
   });
});
```

При каждом событии **change** первым делом получаем значение выбранного пункта из списка

```
let a=document.getElementById("sel").value;
```

а затем формируем новый адрес фотографии:

```
document.getElementById("photo").src="reser/"+a+".jpg";
```

Как видите, проще некуда.

Полная страница выглядит так:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие change</title>

<script>
window.addEventListener("load", function()
{
document.getElementById("sel").addEventListener("change", function()
   {
   let a=document.getElementById("sel").value;
   document.getElementById("photo").src="reser/"+a+".jpg";
   });
});
</script>
</head>

<body>

<select id="sel"><option selected value="1">Море</option>
<option value="2">Река</option>
<option value="3">Озеро</option>
<option value="4">Пруд</option>
<option value="5">Бассейн</option>
<option value="6">Лужа</option></select><br><br>
<img src="reser/1.jpg" id="photo" alt="Фото">

</body>
</html>
```

Первый сценарий, написанный нами для события **change**, был максимально простым. Следующие будут немного сложнее.

3.20. Событие change. Тормозим Винни-Пуха

В новом примере событие **change** будет происходить на элементе **input**.

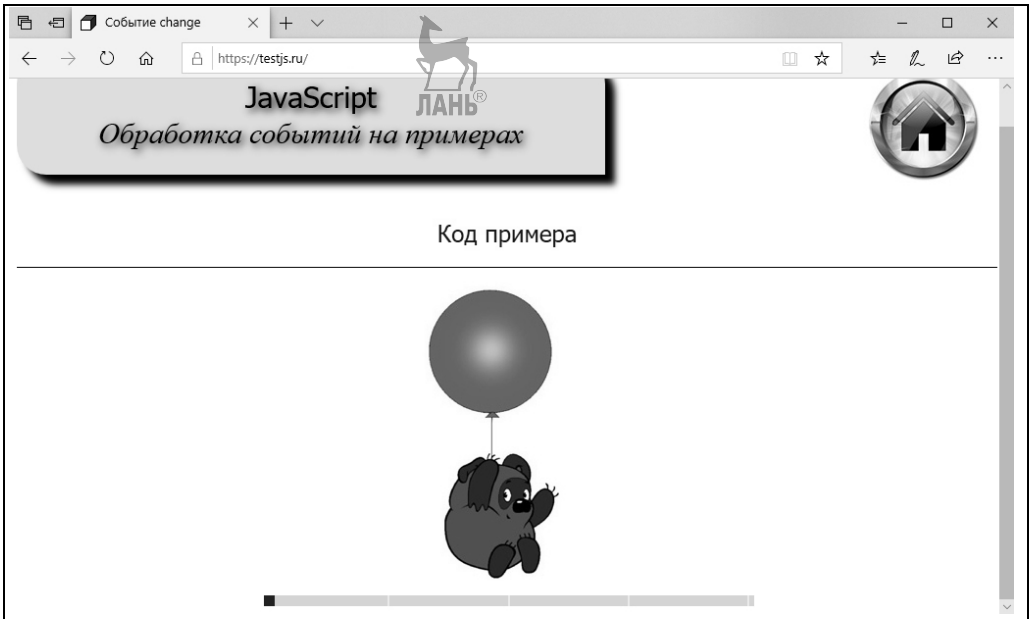

Рис. 3.20.1. Для регулировки скорости полета Винни-Пуха у нас есть шкала с ползунком

Зайдите на страницу **https://testjs.ru/p20.html** сайта поддержки книги. Перед вами изображение Винни-Пуха с воздушным шариком в лапе. Медвежонок с большой скоростью перемещается вперед-назад (словно спасаясь от пчел, охраняющих свой мед). Под изображением находится шкала с ползунком. Сейчас ползунок в крайнем левом положении (рис. 3.20.1). Щелкните мышью по шкале на пару сантиметров правее. Ползунок сместится на одно деление вправо, а скорость полета Винни-Пуха немного замедлится. Сделайте новый щелчок еще правее. Ползунок переместится еще на одно деление и одновременно скорость полета медвежонка вновь немного уменьшится. Щелкайте дальше, смещаясь по шкале к правому краю. В результате экспериментов вы обнаружите, что у медвежонка есть пять «скоростных режимов». После того, как вы притормозили картинку, можно вновь разогнать ее. Скорость полета будет увеличиваться по мере смещения щелчков к левому краю. Как вам уже, наверное, понятно, вы можете щелкать в любой точке шкалы в любой последовательности – тем самым переключая картинку в один из скоростных режимов.

На странице у нас два элемента. Картинка с Винни-Пухом

```
<img src="pict/1ball.jpg" id="ball" alt="Фото">
```

и поле **input** с ползунком:

```
<input type="range" min="10" max="90" step="20" value="10"
id="ran">
```

Шкала имеет минимальное значение **10**, максимальное **90** и шаг **20** единиц. Такое распределение как раз и дает пять скоростей перемещения рисунка.

Параметры элементов страницы:

```
#ball {position: absolute; left: 0px;}
#ran {position: absolute; top: 350px; width: 500px;}
```

Регистрируем обработчик для события **change** и одновременно запускаем функцию **balo**, которая управляет движением рисунка вперед:

```
window.addEventListener("load", function()
{
balo();
document.getElementById("ran").addEventListener("change", function()
   {
   ...
   });
});
```

Обработку события **change** мы разберем чуть позже, а пока остановимся на функции **balo**. Ей необходимы две переменные, которые мы объявим в первую очередь:

```
let z=0;
let t=10;
```

z – это показатель смещения рисунка от левого края документа, а **t** – временной интервал в миллисекундах.

Как мы уже сказали, первой запускается функция **balo**:

```
function balo()
{
z+=10;

if(z<840)
   {
   document.getElementById("ball").style.left=z+"px";
   window.setTimeout(balo, t);
   }
else
   {
   document.getElementById("ball").src="pict/2ball.jpg";
   window.setTimeout(rev, t);
   }
}
```

И сразу же увеличивает переменную **z** на **10** пикселей:

```
z+=10;
```

Если в этот момент выполняется условие **if(z<840)**, то есть смещение рисунка не превышает **840** пикселей, то его позиции присваивается новое значение переменной **z**:

```
document.getElementById("ball").style.left=z+"px";
```

а это значит, что Винни-Пух передвигается на **10** пикселей правее. Затем через интервал времени **t** (в исходном состоянии это **10** миллисекунд) функция **balo** вызывается вновь:

```
window.setTimeout(balo, t);
```

z увеличивается еще раз на **10**, а смещение картинки на этом шаге достигнет уже 20 пикселей. Теперь инструкция

```
window.setTimeout(balo, t);
```

выполняется вновь.

Так происходит до тех пор, пока смещение рисунка от левого края не достигнет **840** пикселей. После этого условие **if(z<840)** становится ложным и выполняются альтернативные инструкции:

```
else
   {
   document.getElementById("ball").src="pict/2ball.jpg";
   window.setTimeout(rev, t);
   }
```

Рисунок Винни-Пуха, смотрящего вправо, меняется на Винни-Пуха, смотрящего влево, после чего через интервал **t** запускается функция **rev**. Ее назначение – перемещать рисунок от правого края страницы к левому. Функция **rev** почти аналогична **balo**:

```
function rev()
{
z-=10;

if(z>20)
   {
   document.getElementById("ball").style.left=z+"px";
   window.setTimeout(rev, t);
   }
else
   {
   document.getElementById("ball").src="pict/1ball.jpg";
   window.setTimeout(balo, t);
   }
}
```

Но есть важные отличия. Во-первых, показатель смещения рисунка **z** теперь уменьшается:

```
z-=10;
```

За счет этого медвежонок начинает двигаться в обратном направлении. Во-вторых, теперь с интервалом **t** запускается функция обратного движения **rev**.

Винни-Пух летит назад до тех пор, пока истинно условие **if(z>20)**. Когда картинка достигнет предельного левого смещения в **20** пикселей, будут выполнены инструкции

```
document.getElementById("ball").src="pict/1ball.jpg";
window.setTimeout(balo, t);
```

Вновь загрузится рисунок медвежонка смотрящего вправо и запустится функция **balo**. Процесс двинется по второму кругу.

Полет этот бесконечный и продолжается до тех пор, пока открыта страница с примером.

Вернемся к обработке события **change**. Для него у нас всего одна инструкция:

```
document.getElementById("ran").addEventListener("change", function()
   {
   t=this.value;
   });
```

Она присваивает переменной временного интервала **t** текущее значение из поля **input**. Если установленное в поле число небольшое, временной интервал тоже невелик и картинка двигается быстро (так как функции движения вызываются часто). Если значение поля максимальное или близко к максимуму, то временной интервал увеличивается и картинка замедляется (так как функции движения вызываются реже).

Как видите, сценарий перемещения картинки оказался сложнее, чем обработчик события **change**.

Сведем «дебет» с «кредитом» и напишем полный код страницы:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие change</title>

<style>
#ball {position: absolute; left: 0px;}
#ran {position: absolute; top: 350px; width: 500px;}
</style>

<script>
window.addEventListener("load", function()
{
```

```
balo();
document.getElementById("ran").addEventListener("change", function()
   {
   t=this.value;
   });
});

let z=0;
let t=10;

function balo()
{
z+=10;

if(z<840)
   {
   document.getElementById("ball").style.left=z+"px";
   window.setTimeout(balo, t);
   }
else
   {
   document.getElementById("ball").src="pict/2ball.jpg";
   window.setTimeout(rev, t);
   }
}

function rev()
{
z-=10;

if(z>20)
   {
   document.getElementById("ball").style.left=z+"px";
   window.setTimeout(rev, t);
   }
else
   {
   document.getElementById("ball").src="pict/1ball.jpg";
   window.setTimeout(balo, t);
   }
}
</script>
</head>

<body>

<img src="pict/1ball.jpg" id="ball" alt="Фото"><br>
<input type="range" min="10" max="90" step="20" value="10"
id="ran">

</body>
</html>
```

3.21. События change и load. Домашние животные

И вновь пример с участием домашних пушистиков. На этот раз мы посмотрим не только фотографии – к ним будет добавляться короткий текст с краткой информацией о животном.

В сценарии на равных правах работают два события – **change** и **load.** Причем событие **load** понадобится не только для регистрации обработчиков, но и для вывода текста методом **Ajax**.

Данный пример можно посмотреть на странице **https://testjs.ru/p21.html**. Вместо нескольких кнопок используем выпадающий список с необходимыми пунктами (рис. 3.21.1). Визуально все выглядит очень просто: выбираете в списке интересующее вас животное и его фото вместе с описанием появится в нижней части страницы (рис. 3.21.2). Однако за внешней простотой кроется более сложная программа, нежели в примере с кнопками выбора животного.

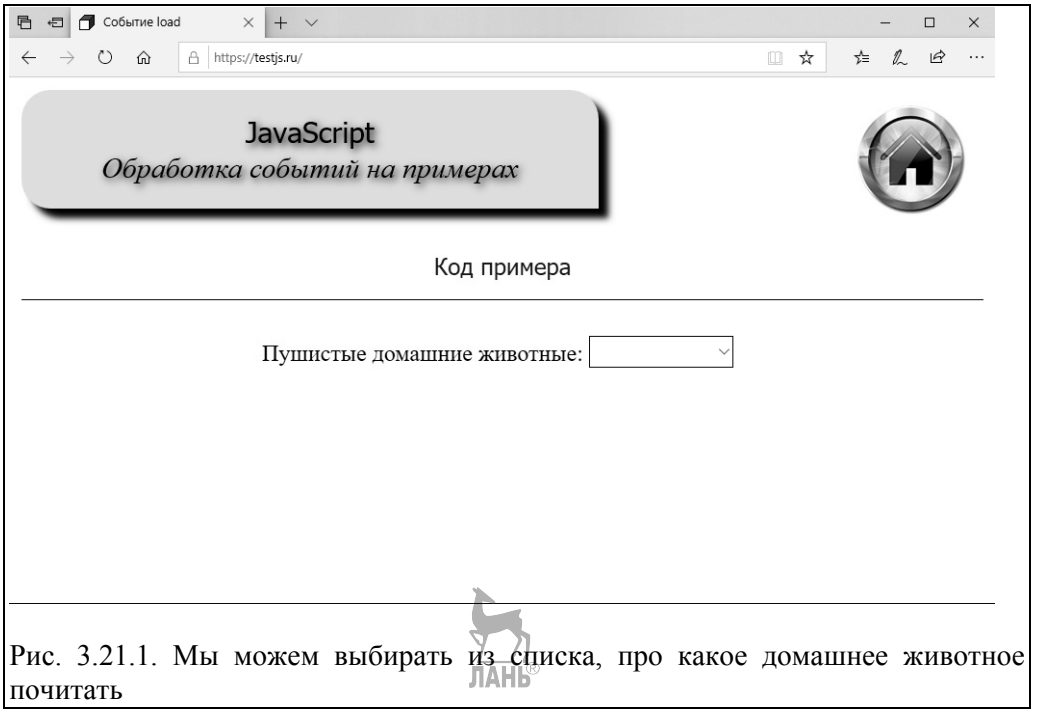

Рис. 3.21.1. Мы можем выбирать из списка, про какое домашнее животное почитать

У нас на странице есть текст «Пушистые домашние животные». Кроме того, здесь расположились 3 главных элемента.

Список:

```
<select id="sel"><option selected value="no"> </option>
<option value="cat">Кошка</option>
<option value="dog">Собака</option>
<option value="fre">Хорек</option>
<option value="chin">Шиншилла</option>
<option value="hams">Хомяк</option></select>
```

ID каждого пункта списка будут использованы нами для определения, о каком животном должны выводиться фото и описание. После загрузки страницы в списке выбрано пустое поле:

```
<option selected value="no"> </option>
```

Прямо под списком будут появляться снимки. В исходном состоянии вместо них стоит «заглушка»:

```
<img src="pict/net.jpg" id="an" alt="Фото">
```

Под изображением отведено место для текста:

```
<div id="tean"></div>
```

В сценарии регистрируется один обработчик – события **change**:

```
window.addEventListener("load", function()
{
document.getElementById("sel").addEventListener("change", anim);
});
```

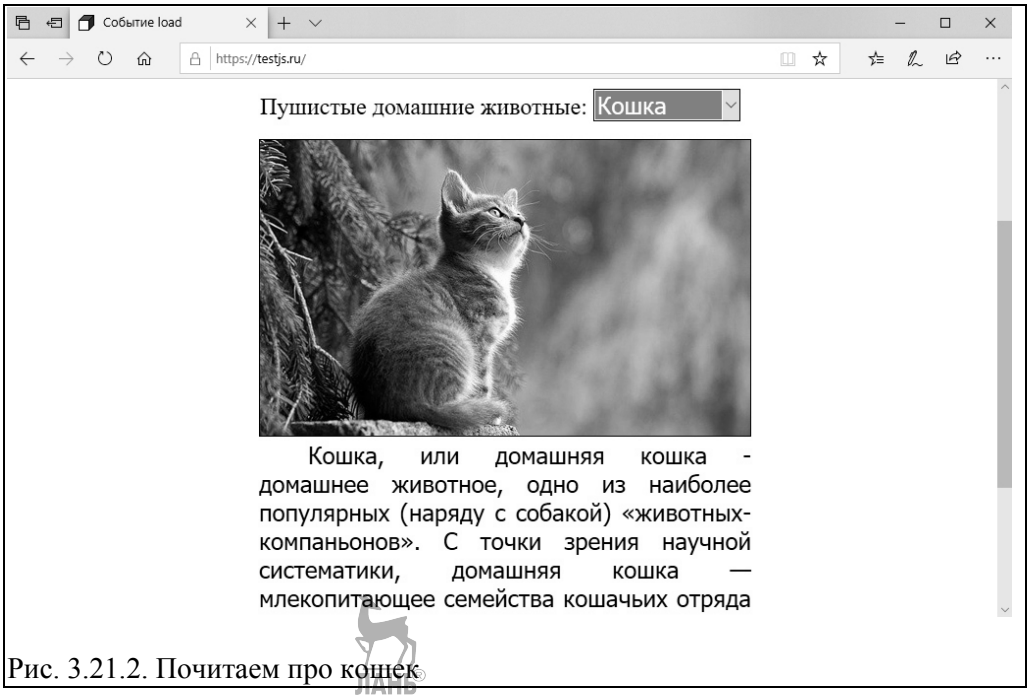

Рис. 3.21.2. Почитаем про кошек

Представим, что вы выбрали один из пунктов списка. Первое, что происходит внутри функции, – это получение выбранного значения:

```
let p=this.value;
```

Затем проверяется условие, не является ли выбранный пункт пустым:

```
if(p!="no")
```

Давайте сначала рассмотрим ситуацию, при которой условие оказывается ложным – то есть ни одно животное не выбрано. Тогда выполняется следующая группа инструкций:

```
document.getElementById("an").src="pict/net.jpg";
document.getElementById("tean").innerHTML="";
```

В первой строке производится операция замены фото на «заглушку», во второй строке текстовое поле

```
<div id="tean"></div>
```

очищается. Этот процесс важен в том случае, если вы хотите удалить со страницы ранее просмотренную информацию и фотографию, не прибегая к просмотру других пунктов из списка. В результате на странице остается только выпадающий список с заголовком.

Самое интересное происходит, когда условие **if(p!="no")** истинно. Первым делом выводим на страницу необходимое изображение:

```
document.getElementById("an").src="anim/"+p+".jpg";
```

А затем, используя технологию **Ajax**, добавляем в документ нужный текст:

```
let re=new XMLHttpRequest();
re.onload=demo;
let adr="text/"+p+".txt";
re.open("GET", adr, true);
re.send();
```

Ajax расшифровывается как «Асинхронный JavaScript и XML» и представляет собой современную технологию получения данных от сервера в фоновом режиме без перезагрузки страницы. А **XMLHttpRequest** из первой строки данной части сценария – это экземпляр интерфейса, имеющего свойства и методы, позволяющие формировать запрос к серверу и получать данные из ответа.

Итак, создаем новый экземпляр **XMLHttpRequest**:

```
let re=new XMLHttpRequest();
```

Дальше сообщаем сценарию, что после получения ответа от сервера и загрузки данных (то есть, когда произойдет событие **load** для этих данных), необходимо запустить функцию **demo**:

```
re.onload=demo;
```

Задача функции **demo** – вывести текст про животное на страницу.

Формируем адрес файла с текстом, который должен быть загружен:

```
let adr="text/"+p+".txt";
```

Подготавливаем запрос, где указано, что информация будет передана методом **GET** по адресу, хранящемуся в переменной **adr**:

```
re.open("GET", adr, true);
```

Отправляем запрос:

```
re.send();
```

Получив ответ, функция **demo** выполняет всего одну инструкцию:

```
document.getElementById("tean").innerHTML=this.responseText;
```

Тем самым мы помещаем текст о животном на предназначенный для него слой. Добавлю, что свойство **responseText** предоставляет доступ к ответу сервера. Значение **responseText** – это текстовая строка.

Подведем итог:

– фотографию на страницу мы поместили, присвоив соответствующий адрес тегу **img** в теле документа;

– для добавления текста нами использован более «продвинутый» метод – загрузка данных из файла методом **Ajax**.

Вот такой у нас получился пример:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие load</title>

<script>
window.addEventListener("load", function()
{
document.getElementById("sel").addEventListener("change", anim);
});

function anim()
{
let p=this.value;
if(p!="no")
   {
   document.getElementById("an").src="anim/"+p+".jpg";

   let re=new XMLHttpRequest();
   re.onload=demo;
   let adr="text/"+p+".txt";
   re.open("GET", adr, true);
   re.send();
   }
else
   {
   document.getElementById("an").src="pict/net.jpg";
   document.getElementById("tean").innerHTML="";
   }
```

```
}

function demo()
{
document.getElementById("tean").innerHTML=this.responseText;
}
</script>
</head>

<body>

Пушистые домашние животные:
<select id="sel"><option selected value="no"> </option>
<option value="cat">Кошка</option>
<option value="dog">Собака</option>
<option value="fre">Хорек</option>
<option value="chin">Шиншилла</option>
<option value="hams">Хомяк</option></select><br>

<img src="pict/net.jpg" id="an" alt="Фото"><br>
<div id="tean"></div>

</body>
</html>
```

3.22. Событие scroll. Разные гитары

На примере музыкальных инструментов продемонстрируем случай фотоленты с ограниченным количеством снимков. А главным действующим «лицом» у нас будет событие **scroll**, которое происходит во время прокручивания страницы в окне браузера или во фрейме. Страница такой фотоленты расположена по адресу **https://testjs.ru/p22.html**.

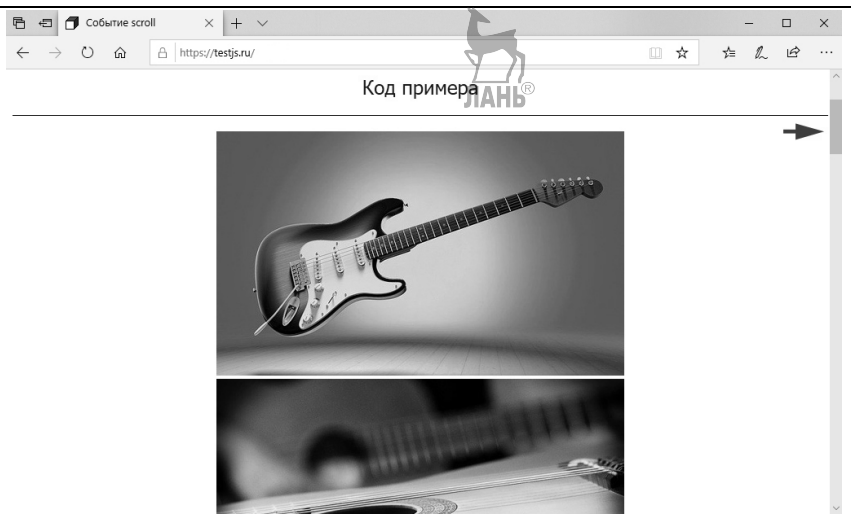

Рис. 3.22.1. Обратите внимание на размеры ползунка сразу после загрузки страницы

В нашей ленте 35 изображений. В исходном состоянии на странице расположены 8 снимков (рис. 3.22.1). По мере прокрутки страницы вниз вы увидите, что появляются новые фото, а ползунок становится все короче и короче (укорачивание ползунка подтверждает, что фото добавляются именно при прокрутке). Когда вся лента «размотана», добавление снимков прекращается, а ползунок останавливается и упирается в нижний край окна браузера (рис. 3.22.2). Такая технология позволяет ускорить загрузку основной части страницы и выводить последующие изображения постепенно. Согласитесь: если бы мы разместили в документе сразу все картинки, страница грузилась бы гораздо дольше. А длительная загрузка не нравится очень многим посетителям сайтов.

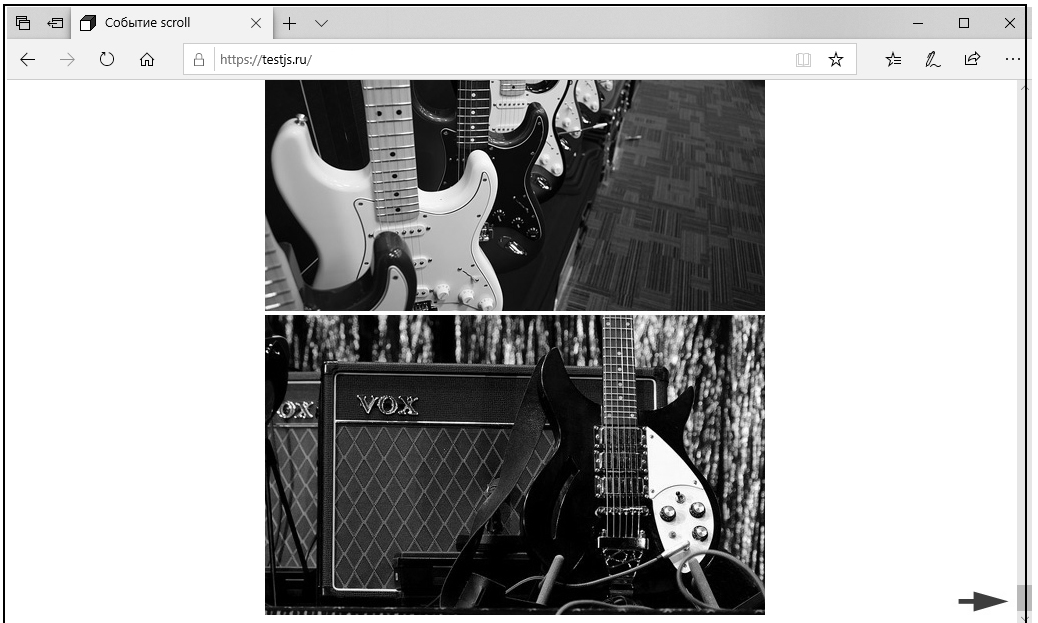

Рис. 3.22.2. Контент добавляется по мере прокрутки страницы и ползунок становится все короче

Перейдем к реализации примера.

У нас есть слой

```
<div id="pic">...</div>
```

на котором размещены 8 снимков:

```
<img src="mus/1.jpg" alt="Фото"><br>
<img src="mus/2.jpg" alt="Фото"><br>
<img src="mus/3.jpg" alt="Фото"><br>
<img src="mus/4.jpg" alt="Фото"><br>
<img src="mus/5.jpg" alt="Фото"><br>
```

```
<img src="mus/6.jpg" alt="Фото"><br>
<img src="mus/7.jpg" alt="Фото"><br>
<img src="mus/8.jpg" alt="Фото">
```

Новые фото будут добавляться по очереди – каждый раз в текущий конец списка.

Зарегистрируем обработчик события **scroll** (напомню, что это событие уровня окна браузера):

```
window.addEventListener("scroll", scro);
```

Создадим счетчик изображений:


```
let i=9;
```

Поскольку 8 фото у нас уже есть на странице, отсчет будет начинаться с числа **9**.

Теперь напишем функцию **scro**:

```
function scro()
{
if(window.pageYOffset+window.innerHeight>=document.body.clientHeig
ht)
   {
   if(i<=35)
      {
      let res='<br><img src="mus/'+i+'.jpg" alt="Фото">';
document.getElementById("pic").insertAdjacentHTML("beforeend",
res);

      i++;
      }
   }
}
```

Обратите внимание на первую строку

```
if(window.pageYOffset+window.innerHeight>=document.body.clientHeig
ht)
```

Здесь выполняется проверка уровня прокрутки документа. Свойство **pageYOffset** содержит информацию о величине вертикальной прокрутки текущей страницы в пикселях. Свойство **innerHeight** содержит информацию о внутренней высоте окна браузера в пикселях (включая полосу горизонтальной прокрутки, если она есть). Свойство **clientHeight** содержит информацию о внутренней высоте элемента или страницы в пикселях (без учета размеров границы и внешних отступов).

Что мы делаем с этими параметрами? Берем величину прокрутки страницы

```
window.pageYOffset
```

складываем с внутренним размером окна браузера

```
window.innerHeight
```

и сравниваем полученный результат с «высотой» документа

```
document.body.clientHeight
```

До тех пор, пока страница прокручена не до конца, условие

```
if(window.pageYOffset+window.innerHeight>=document.body.clientHeig
ht)
```

ложно. Но как только документ окажется прокручен полностью, условие станет истинным и будет выполнен следующий блок инструкций:

```
if(i<=35)
   {
   let res='<br><img src="mus/'+i+'.jpg" alt="Фото">';
   document.getElementById("pic").insertAdjacentHTML("beforeend",
res);

   i++;
   }
```

Как видите, эти инструкции выполняются до тех пор, пока количество фото на странице меньше или равно **35**. По достижении заданного предела функция прекратит свою работу.

Первая инструкция формирует HTML-код очередной фотографии:

```
let res='<br><img src="mus/'+i+'.jpg" alt="Фото">';
```

Во второй строке мы добавляем полученный HTML-код в конец списка уже загруженных снимков:

```
document.getElementById("pic").insertAdjacentHTML("beforeend",
res);
```

Метод **insertAdjacentHTML()** предназначен для добавления данных (они могут быть в виде HTML-кода или простого текста) к какому-либо элементу страницы. Первый аргумент метода указывает на место вставки, второй содержит код или текст. В нашем случае данные добавляются на слой **id="pic"** непосредственно перед закрывающим тегом, на что указывает аргумент **beforeend**.

Последняя операция – увеличение счетчика на единицу:

```
i++;
```

Вот и весь сценарий. Его листинг:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие scroll</title>
```

```
<script>
window.addEventListener("scroll", scro);

let i=9;

function scro()
{
if(window.pageYOffset+window.innerHeight>=document.body.clientHeig
ht)
   {
   if(i<=35)
      {
      let res='<br><img src="mus/'+i+'.jpg" alt="Фото">';
document.getElementById("pic").insertAdjacentHTML("beforeend",
res);

      i++;
      }
   }
}
</script>
</head>

<body>

<div id="pic"><img src="mus/1.jpg" alt="Фото"><br>
<img src="mus/2.jpg" alt="Фото"><br>
<img src="mus/3.jpg" alt="Фото"><br>
<img src="mus/4.jpg" alt="Фото"><br>
<img src="mus/5.jpg" alt="Фото"><br>
<img src="mus/6.jpg" alt="Фото"><br>
<img src="mus/7.jpg" alt="Фото"><br>
<img src="mus/8.jpg" alt="Фото"></div>

</body>
</html>
```

В этом примере мы продемонстрировали простейший случай ленты с контентом. Главное, что послужило основой такой простоты, – это ограниченное количество снимков. Несколько позже мы создадим бесконечную ленту сообщений, основанную на других принципах.

3.23. Событие scroll. «Скручиваем» прозрачность

Этот пример у нас особый. Впервые и только в этом случае я продемонстрирую вам, как одну задачу можно решить двумя разными способами. Первый способ будет довольно прямолинейным и незатейливым, второй – более изящным и универсальным. И вновь мы – уже в третий раз – напишем обработчики для события **scroll**.

Наверное, вы не раз видели сайты, на страницах которых отдельные блоки контента «проявляются» сквозь фон по мере прокрутки страницы вниз. Именно такой эффект мы и попробуем создать двумя разными способами. А для его реализации опять воспользуемся снимками гитар.

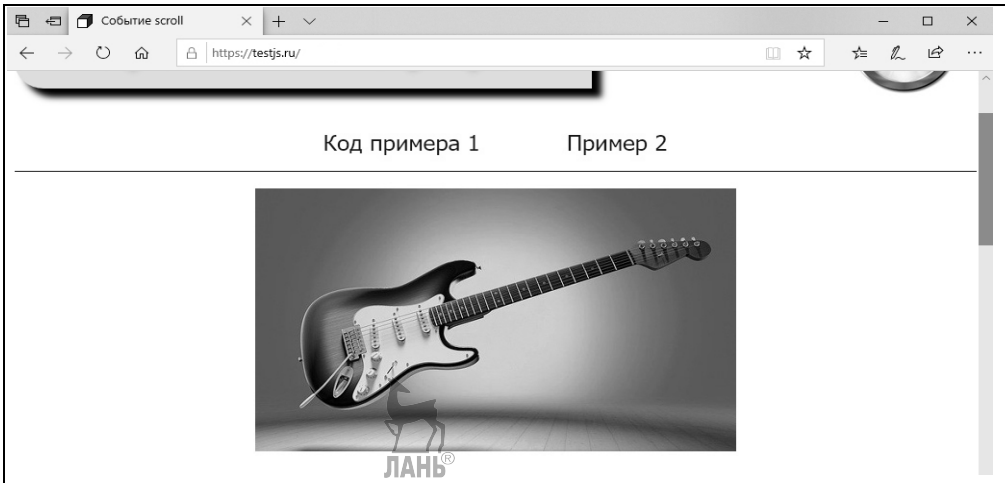

Рис. 3.23.1. Исходный вид страницы

Двинемся по порядку. Для первого варианта у нас есть такая страница: **https://testjs.ru/p23_1.html**. Что мы видим? Ту же самую картину, что и на рисунке 3.23.1. А именно – фото электрогитары. Начнем двигать ползунок вниз – и по мере его прокрутки на странице станут поочередно «проявляться» фотографии еще четырех музыкальных инструментов. Заметим, что проявление будет плавным (рис. 3.23.2). Когда все пять гитар появятся на странице, процесс остановится. Это означает, что весь контент загрузился полностью. Попутно добавим, что для наглядности лучше перемещать ползунок не спеша.

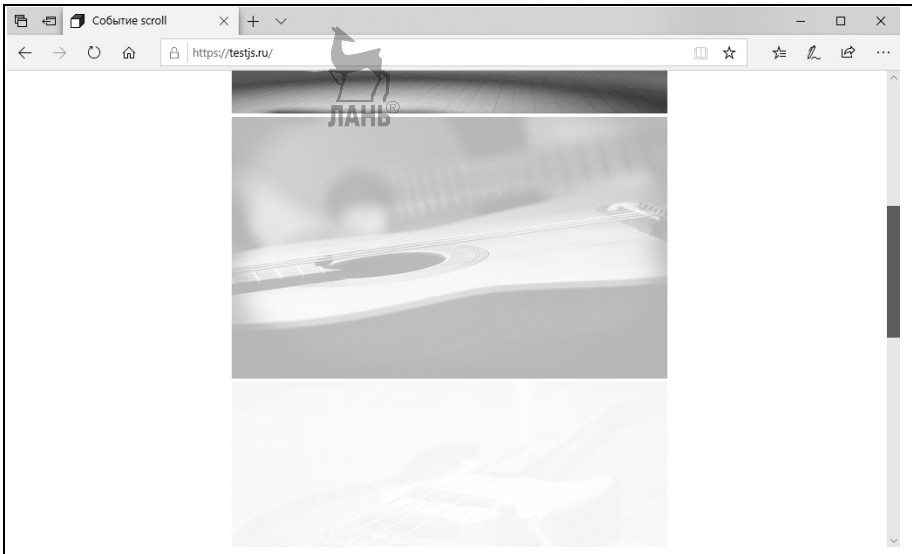

Рис. 3.23.2. Снимки гитар начинают «проявляться»

Для создания первого варианта примера разместим на странице 5 разных фотографий гитар:

```
<img src="mus/1.jpg" alt="Фото" id="im1"><br>
<img src="mus/2.jpg" alt="Фото" id="im2" class="ima"><br>
<img src="mus/3.jpg" alt="Фото" id="im3" class="ima"><br>
<img src="mus/4.jpg" alt="Фото" id="im4" class="ima"><br>
<img src="mus/5.jpg" alt="Фото" id="im5" class="ima">
<br><br><br><br><br>
```

Несколько переводов строки

```
<br><br><br><br><br>
```

нужны для получения необходимой «высоты» документа, от которой напрямую зависит корректная работа сценария.

В исходном состоянии 4 снимка (**class="ima"**) не видны. Их непрозрачность равна нулю:

```
.ima {opacity: 0;}
```

Для «проявления» изображений служит функция **scro**:

```
window.addEventListener("scroll", scro);
```

Она состоит из пяти похожих блоков:

```
function scro()
{
if(window.pageYOffset>10)
    {
    document.getElementById("im2").style.transition="opacity 1s";
    document.getElementById("im2").style.opacity=1;
    }
if(window.pageYOffset>310)
    {
    document.getElementById("im3").style.transition="opacity 1s";
    document.getElementById("im3").style.opacity=1;
    }
if(window.pageYOffset>610)
    {
    document.getElementById("im4").style.transition="opacity 1s";
    document.getElementById("im4").style.opacity=1;
    }
if(window.pageYOffset>910)
    {
    document.getElementById("im5").style.transition="opacity 1s";
    document.getElementById("im5").style.opacity=1;
    }
}
```

В каждом блоке проверяется величина прокрутки страницы. Когда ползунок смещается более чем на 10 пикселей вниз, выполняется первый блок.

Для плавного «проявления» снимка мы использовали уже знакомое нам свойство **transition**. В течение одной секунды оно изменяет непрозрачность второй фотографии с **0** до **1**:

```
document.getElementById("im2").style.transition="opacity 1s";
document.getElementById("im2").style.opacity=1;
```

Продолжаем двигать ползунок вниз. После того, как он сместится более чем на 310 пикселей, выполнится второй блок инструкций:

```
if(window.pageYOffset>310)
   {
   document.getElementById("im3").style.transition="opacity 1s";
   document.getElementById("im3").style.opacity=1;
   }
```

Теперь третий снимок плавно «проявится» в течение 1 секунды.

Дальше все будет происходить похожим образом. Последующие контрольные точки для оставшихся снимков – 610 и 910 пикселей.

Первый вариант готов. Вот его код:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие scroll</title>
<style>
.ima {opacity: 0;}
</style>

<script>
window.addEventListener("scroll", scro);

function scro()
{
if(window.pageYOffset>10)
   {
   document.getElementById("im2").style.transition="opacity 1s";
   document.getElementById("im2").style.opacity=1;
   }
if(window.pageYOffset>310)
   {
   document.getElementById("im3").style.transition="opacity 1s";
   document.getElementById("im3").style.opacity=1;
   }
if(window.pageYOffset>610)
   {
   document.getElementById("im4").style.transition="opacity 1s";
   document.getElementById("im4").style.opacity=1;
   }
if(window.pageYOffset>910)
   {
   document.getElementById("im5").style.transition="opacity 1s";
   document.getElementById("im5").style.opacity=1;
```

```
   }
}
</script>
</head>

<body>

<img src="mus/1.jpg" alt="Фото" id="im1"><br>
<img src="mus/2.jpg" alt="Фото" id="im2" class="ima"><br>
<img src="mus/3.jpg" alt="Фото" id="im3" class="ima"><br>
<img src="mus/4.jpg" alt="Фото" id="im4" class="ima"><br>
<img src="mus/5.jpg" alt="Фото" id="im5" class="ima">
<br><br><br><br><br>

</body>
</html>
```

При всей красоте получаемого эффекта этот вариант имеет один недостаток: если вы захотите увеличить количество «проявляемых» снимков, вам придется добавлять все новые и новые блоки в функцию **scro**. Сценарий будет разрастаться, а следом за этим будет расти и «вес» документа.

От подобного недостатка свободен второй вариант. В нем при необходимости увеличить количество фотографий нужно изменить всего одну цифру в сценарии.

Более совершенный вариант находится по адресу **https://testjs.ru/p23_2.html**. Визуально события на странице этого примера выглядят точно так же, как и в первом случае. Поэтому мы не станем описывать их по новой (так же как и демонстрировать рисунки, почти аналогичные уже приведенным).

Первые изменения коснутся размещения картинок. Теперь они располагаются на слое:

```
<div id="pic"><img src="mus/1.jpg" alt="Фото" id="im1"><br>
<img src="mus/2.jpg" alt="Фото" id="im2" class="ima"><br>
<img src="mus/3.jpg" alt="Фото" id="im3" class="ima"><br>
<img src="mus/4.jpg" alt="Фото" id="im4" class="ima"><br>
<img src="mus/5.jpg" alt="Фото" id="im5" class="ima"></div>
```

Настройки слоя:

```
#pic {height: 1600px; padding-top: 100px;}
```

За счет этих параметров создается необходимая высота документа, требуемая для его корректной работы, а также отступ в **100** пикселей от верхнего края документа, что тоже критично.

Функция, управляющая поведением снимков, как и в предыдущем случае, имеет название **scro**. Только теперь она будет несколько сложнее. Во-первых, для ее работы нам понадобятся две переменные:

```
let i=2;
let c=2;
```

Переменная **i** – счетчик фотографий, а **c** – идентификатор состояния программы. Его назначение вы поймете в процессе разбора функции **scro**. Вот она:

```
function scro()
{
if(i<6)
   {
   let a=300*i-600;

   if(window.pageYOffset>a)
      {
      if(c==1)
         {
         let b="im"+i;
         document.getElementById(b).style.transition="opacity 2s";
         document.getElementById(b).style.opacity=1;

         c=2;
         i++;
         }
      }

   if((window.pageYOffset>a+100)&&(window.pageYOffset<a+150))
      {
      c=1;
      }
   }
}
```

Итак, вы только-только начали двигать ползунок. В этот момент условие **if(i<6)** истинно, поэтому выполняется первая инструкция функции:

```
let a=300*i-600;
```

Здесь рассчитывается контрольная точка смещения ползунка на каждом шаге выполнения программы (самая первая точка равна 0 пикселей, потому что исходное значение переменной **i** – **2**). Так как в начальном состоянии идентификатору **c** присвоено значение **2**, первый блок инструкций пропускается (условие **if(c==1)** ложно).

Продолжаем двигать ползунок. Когда его смещение от первой контрольной точки превысит **100** пикселей, но не достигнет **150**

```
if((window.pageYOffset>a+100)&&(window.pageYOffset<a+150))
```

идентификатору **c** будет присвоено значение **1** и выполнится первый блок инструкций:

```
if(c==1)
   {
   let b="im"+i;
   document.getElementById(b).style.transition="opacity 2s";
   document.getElementById(b).style.opacity=1;
```

```
c=2;
i++;
}
```

Здесь мы для разнообразия увеличили длительность проявления снимка до 2 секунд. Какое фото должно быть «проявлено», вычисляется в выражении

```
let b="im"+i;
```

В самом начале это будет фото номер 2, так как исходное значение переменной **i** – **2**. Дальше счетчик фото **i** увеличится на единицу, а идентификатору **c** опять присваивается стоп-значение **2**.

Начинается следующий цикл сценария. Переменная **a** получает новое значение – на **300** пикселей больше предыдущего. Но пока при дальнейшей прокрутке на странице остаются две фотографии. Это происходит, потому что в данное время первый блок инструкций не может быть выполнен из-за ложности условия **if(c==1)**. Когда смещение ползунка от новой контрольной точки превысит **100** пикселей, но не достигнет **150**, идентификатор **c** снова изменит значение. В результате опять выполнится первый блок инструкций, который «проявит» третью фотографию и увеличит счетчик **i** еще на единицу. Думаю, следующие повторяющиеся шаги уже понятны.

Что мы имеем в итоге? Второе изображение появляется при смещении ползунка на **100** пикселей. Затем каждая последующая прокрутка страницы на **300** пикселей увеличивает значение счетчика фотографий на единицу. То есть следующие рисунки появляются с шагом в **300** пикселей. Идентификатор **c** «следит», чтобы значение счетчика изменялось для одной контрольной точки только один раз. Так как событие **scroll** происходит не на каждом пикселе полосы прокрутки, мы взяли не конкретную точку срабатывания второго блока инструкций, а интервал от **100** до **150** пикселей, в рамках которого событие **scroll** точно произойдет. Наконец, условие **if(i<6)** ограничивает наш сценарий «проявлением» пяти снимков.

Код второго варианта:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие scroll</title>

<style>
#pic {height: 1600px; padding-top: 100px;}
.ima {opacity: 0;}
</style>

<script>
window.addEventListener("scroll", scro);

let i=2;
let c=2;
```

```
function scro()
{
if(i<6)
   {
   let a=300*i-600;

   if(window.pageYOffset>a)
      {
      if(c==1)
         {
         let b="im"+i;
         document.getElementById(b).style.transition="opacity 2s";
         document.getElementById(b).style.opacity=1;

         c=2;
         i++;
         }
      }

   if((window.pageYOffset>a+100)&&(window.pageYOffset<a+150))
      {
      c=1;
      }
   }
}
</script>
</head>

<body>

<div id="pic"><img src="mus/1.jpg" alt="Фото" id="im1"><br>
<img src="mus/2.jpg" alt="Фото" id="im2" class="ima"><br>
<img src="mus/3.jpg" alt="Фото" id="im3" class="ima"><br>
<img src="mus/4.jpg" alt="Фото" id="im4" class="ima"><br>
<img src="mus/5.jpg" alt="Фото" id="im5" class="ima"></div>

</body>
</html>
```

Что надо скорректировать в этом варианте, если вы хотите увеличить количество рисунков? Всего-навсего изменить число в условии **if(i<6)**. Если у вас 8 изображений, значит условие должно выглядеть так:

```
if(i<9)
```

Если у вас 10 изображений, значит условие необходимо записать так:

```
if(i<11)
```

И далее в том же ключе. Думаю, алгоритм понятен. В условии всегда должно стоять число на единицу больше количества снимков. Как видите, перенастраивать второй вариант гораздо проще и рациональнее.

3.24. События scroll и load. Бесконечная лента

Новый пример, как и в некоторых предыдущих случаях, будет демонстрировать совместное применение нескольких обработчиков. На этот раз базовыми станут события **scroll** и **load**. Как я и обещал, мы напишем сценарий бесконечной ленты новостей.

Прежде, чем рассказывать о таком сценарии, небольшой комментарий к нему. Дело в том, что обычно для подобных лент пишется специальная серверная программа на одном из скриптовых языков. Подобная система работает так: со страницы, открытой на компьютере клиента, серверу поступает запрос на передачу очередной порции данных. Серверная программа отбирает эти данные в соответствии с заложенным в ней алгоритмом и отправляет обратно на страницу. Однако изучение скриптовых языков не входит в задачи нашей книги. Поэтому мы пойдем другим путем: будем при каждом запросе отправлять одни и те же данные из одного и того же файла. При этом задача написания бесконечной ленты будет выполнена полноценно.

Оговорюсь, что мы могли бы создать огромное количество, например, пронумерованных файлов с разными текстами и фотографиями, добавить в наш сценарий счетчик и каждый раз отправлять запрос на файл с очередным номером. В этом случае можно создать «натуральную» бесконечную ленту без специальной серверной программы. Но тогда нам пришлось бы выполнить колоссальный объем работы по созданию множества этих самых файлов. Что, конечно же, для примера, служащего всего лишь иллюстрацией, непродуктивно.

Теперь к делу. Бесконечная лента новостей демонстрируется на странице **https://testjs.ru/p24.html**. На ней расположено 5 одинаковых блоков с изображением гитары и коротким описанием этого музыкального инструмента. Начинаем прокрутку страницы (рис. 3.24.1). Как только ползунок достигнет нижнего края окна браузера, на страницу будет добавлено еще несколько блоков подобной информации, а ползунок сместится немного вверх. Снова передвинем его вниз – опять добавляются очередные блоки и ползунок смещается вверх. При этом, поскольку страница становится все длиннее и длиннее, ползунок, наоборот, делается все короче и короче (рис. 3.24.2). Прокручивать страницу можно сколь угодно долго – блоки исправно добавляются в конец документа.

Приступим к созданию страницы и сценария. Как было сказано выше, в исходном состоянии документ содержит 5 одинаковых блоков, состоящих из рисунка и текста:

```
Гитара – старинный щипковый инструмент, известный еще с VI века.
Один из самых популярных музыкальных инструментов в мире.
Используется как сольно, так и для аккомпанемента. Изобретение
электрогитары дало толчок развитию и популяризации
вокально-инструментальных ансамблей и рок-групп.<br>
<img src="mus/9.jpg" alt="Фото"><br><br>
```

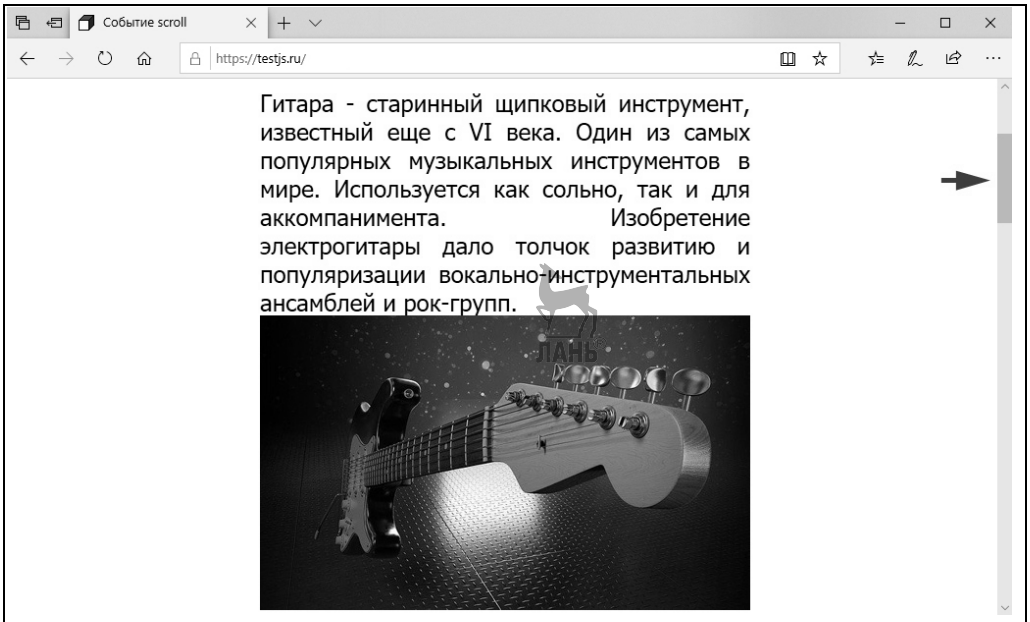

Рис. 3.24.1. Обратите внимание на размеры ползунка сразу после загрузки страницы

Они размещаются на слое

```
<div id="pic">...</div>
```

Регистрируем первый обработчик события

```
window.addEventListener("scroll", scro);
```

и пишем его код:

```
function scro()
{
if(window.pageYOffset+window.innerHeight>=document.body.clientHeig
ht)
   {
   let re=new XMLHttpRequest();
   re.onload=demo;
   re.open("GET", "text/tex.txt", true);
   re.send();
   }
}
```

Смысл условия

```
if(window.pageYOffset+window.innerHeight>=document.body.clientHeig
ht)
```

нам уже знаком из примера 3.22. Как вы помните, тем самым мы проверяем уровень прокрутки документа вниз. Когда ползунок достиг нижнего уровня, выполняются следующие инструкции (тоже знакомые нам, но по примеру 3.21):

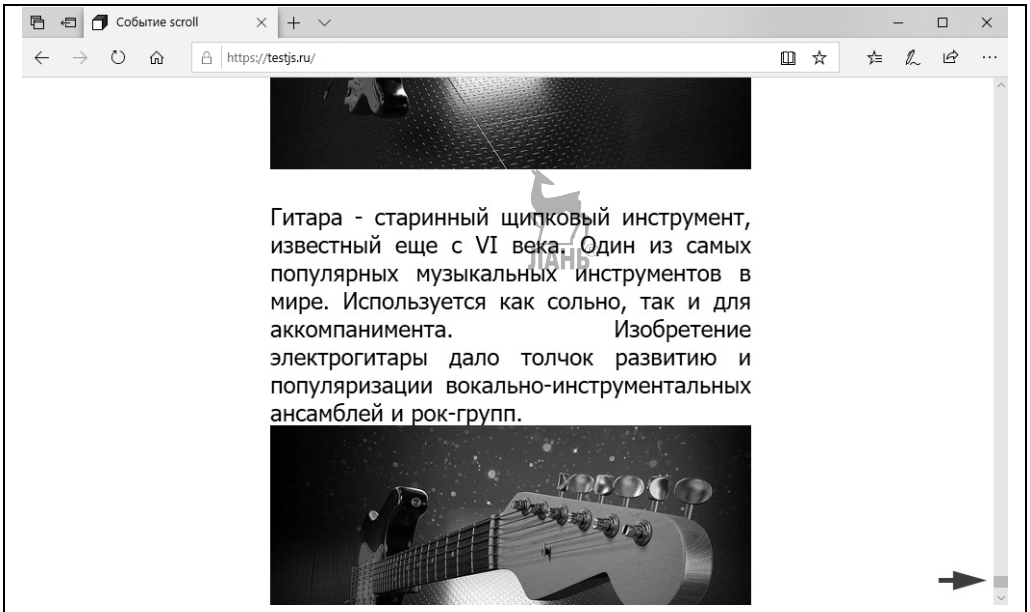

Рис. 3.24.2. Контент добавляется по мере прокрутки страницы и ползунок становится все короче

```
let re=new XMLHttpRequest();
re.onload=demo;
re.open("GET", "text/tex.txt", true);
re.send();
```

Создаем новый экземпляр **XMLHttpRequest**:

```
let re=new XMLHttpRequest();
```

Регистрируем обработчик события загрузки данных с сервера:

```
re.onload=demo;
```

Указываем, данные из какого файла необходимо загрузить:

```
re.open("GET", "text/tex.txt", true);
```

Отправляем запрос:

```
re.send();
```

Пишем функцию **demo**:

```
function demo()
{
document.getElementById("pic").insertAdjacentHTML("beforeend",
this.responseText);
}
```

И опять здесь уже знакомые нам инструкции. Сначала мы получаем HTML-код из файла

```
this.responseText
```

а потом добавляем новый блок в самый конец слоя с ID **pic**:

```
insertAdjacentHTML("beforeend", ...);
```


После этого документ становится длиннее и ползунок немного смещается вверх. Повторно передвинем его вниз. Условие

```
if(window.pageYOffset+window.innerHeight>=document.body.clientHeig
ht)
```

вновь становится истинным, и добавляется еще один блок. Дальше все происходит в том же духе, что и делает процесс добавления контента бесконечным.

Как всегда в завершающей стадии приведем полный код страницы:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие scroll</title>

<style>
#pic {width: 500px;}
</style>

<script>
window.addEventListener("scroll", scro);

function scro()
{
if(window.pageYOffset+window.innerHeight>=document.body.clientHeig
ht)
   {
   let re=new XMLHttpRequest();
   re.onload=demo;
   re.open("GET", "text/tex.txt", true);
   re.send();
   }
}

function demo()
```

```
{
document.getElementById("pic").insertAdjacentHTML("beforeend",

this.responseText);
}
</script>
</head>

<body>

<div id="pic">Гитара - старинный щипковый инструмент, известный
еще с VI века. Один из самых популярных музыкальных инструментов в
мире.
Используется как сольно, так и для аккомпанемента. Изобретение
электрогитары дало толчок развитию и популяризации
вокально-инструментальных ансамблей и рок-групп.<br>
<img src="mus/9.jpg" alt="Фото"><br><br>

...

</div>

</body>
</html>
```

Как видите, совместное использование обработчиков событий **scroll** и **load**, а также технологии **Ajax** позволило создать программу, аналогичную сценариям из социальных сетей. На многих программистских форумах часто обсуждается тема «Как создать бесконечную ленту новостей». Почти всегда авторы сообщений предлагают на удивление громоздкие способы. Наш пример демонстрирует, что на самом деле решение у этой задачи очень простое.

3.25. События click и scroll. Слой вместо фото

В следующем примере опять комбинация событий. На этот раз **scroll** и **click** (как вы, наверное, помните, мы обещали, что один из примеров события **click** будет показан ближе к концу книги; его очередь настала). Обработчик события **click** мы используем для демонстрации фото, а обработчик события **scroll** – для его удаления.

Посмотрите на страницу **https://testjs.ru/p25.html**. На ней расположилась большая картинка с изображением горных сельскохозяйственных угодий (рис. 3.25.1). На этом фото есть шесть маленьких снимков фруктов и ягод. В чем особенность того, что вы видите? Дело в том, что все это единый рисунок. Наша задача добиться того, чтобы мы могли увеличивать фрагменты с фруктами и ягодами для более крупного просмотра. И нам это удается. Наведите указатель мыши, допустим, на изображение клубники. Вы обнаружите, что форма курсора поменяла свой вид – теперь он выглядит как рука. Это означает, что перед нами ссылка, которую можно нажать. Что мы и сделаем. Документ скроется за полупрозрачным белым фоном, на котором появится крупная фотография клубники (рис. 3.25.2). Если теперь щелкнуть мышью на снимке, фоне или

прокрутить страницу вниз, снимок и фон исчезнут и страница вернется к первоначальному виду. То же можно проделать и с остальными фрагментами – то есть вы можете просмотреть все шесть увеличенных фото.

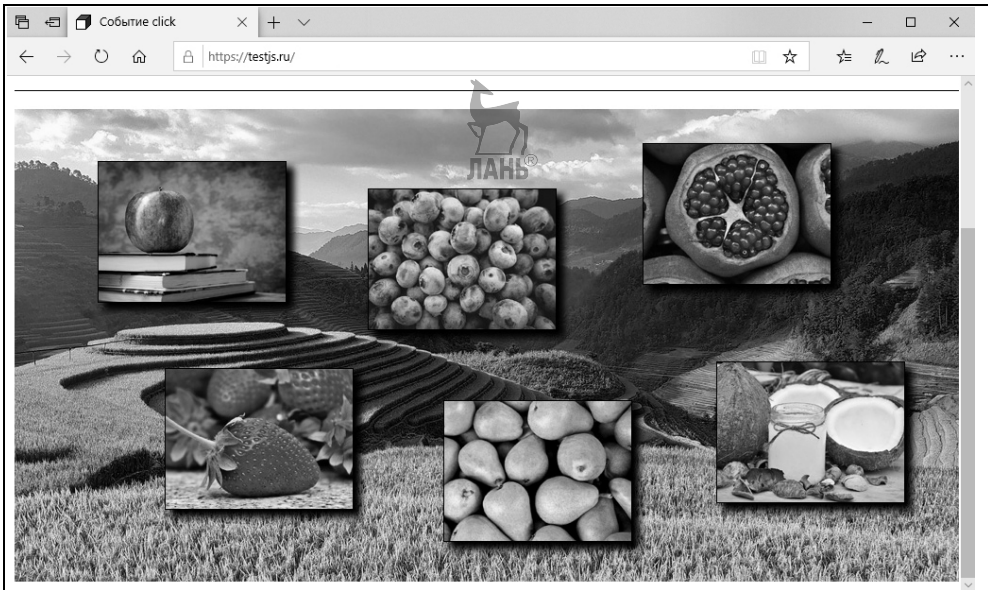

Рис. 3.25.1. Сельскохозяйственные угодья дают вот такие вкусные плоды

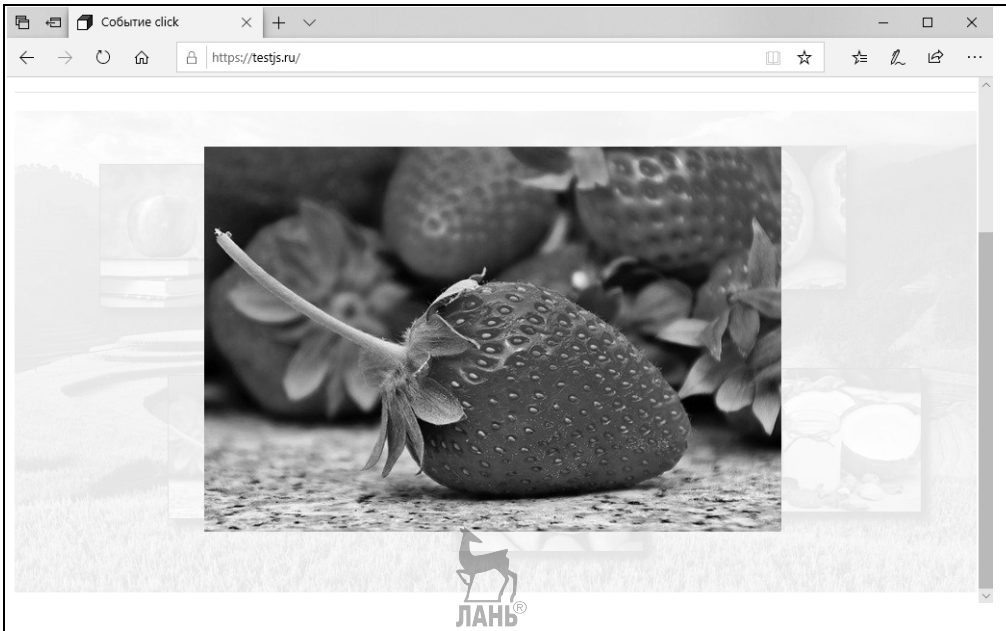

Рис. 3.25.2. Смотрим увеличенное фото

Как нам удалось получить такой результат? Дело в том, что над каждым ягодно-фруктовым кадром находится прозрачный слой, полностью охватывающий площадь данного изображения. Соответственно, программа реагирует на щелчок по слою. А дальше уже запускает сценарий обработки этого щелчка.

Впрочем, давайте по порядку.

У нас есть основной слой

```
<div id="fon">...</div>
```

на котором находится основное изображение

```
<img src="pict/fon.jpg" alt="Фото">
```

и еще шесть дочерних слоев над каждым фрагментом с рисунком фрукта или ягоды:

```
<a href="#"><div id="fr1"></div></a>
<a href="#"><div id="fr2"></div></a>
<a href="#"><div id="fr3"></div></a>
<a href="#"><div id="fr4"></div></a>
<a href="#"><div id="fr5"></div></a>
<a href="#"><div id="fr6"></div></a>
```

При этом на каждом слое имитируется ссылка:

```
<a href="#">...</a>
```

Положение основного слоя относительное

```
#fon {position: relative;}
```

а положение дочерних слоев – абсолютное:

```
#fr1 {position: absolute; top: 56px; left: 88px; width: 200px;
      height: 150px; z-index: 31;}
#fr2 {position: absolute; top: 84px; left: 374px; width: 200px;
      height: 150px; z-index: 32;}
#fr3 {position: absolute; top: 36px; left: 665px; width: 200px;
      height: 150px; z-index: 33;}
#fr4 {position: absolute; top: 274px; left: 158px; width: 200px;
      height: 150px; z-index: 34;}
#fr5 {position: absolute; top: 308px; left: 454px; width: 200px;
      height: 150px; z-index: 35;}
#fr6 {position: absolute; top: 267px; left: 743px; width: 200px;
      height: 150px; z-index: 36;}
```

Они позиционированы таким образом, чтобы находиться строго над соответствующим фрагментом картинки.

Одна заготовка сделана. Теперь надо «соорудить конструкцию» для просмотра увеличенных изображений. Вот она:

```
<div id="bas"></div>
<table id="view">
<tr><td id="tdfo"><img id="photo" src="pict/net.jpg"
alt="Фото"></td></tr>
</table>
```

Первым делом создаем новый слой

```
<div id="bas"></div>
```

с нужными характеристиками:

```
#bas {position: absolute; left: 0px; top: 0px; width: 100%; height:
100%;
      visibility: hidden; background: #FFFFFF; opacity: 0.9; z-
index: 90;}
```

У нас получился слой белого цвета с уровнем непрозрачности **0.9**. Фон занимает все пространство экрана и в исходном состоянии не виден. Слой имеет абсолютное расположение по отношению к телу документа.

Над слоем разместим таблицу, состоящую из одной ячейки:

```
<table id="view">
<tr><td id="tdfo">...</td></tr>
</table>
```

z-index таблицы на единицу больше, чем у слоя, поэтому таблица все время находится выше. Как и фон, таблица занимает все пространство экрана и в исходном состоянии не видна:

```
#view {position: absolute; left: 0px; top: 0px; width: 100%;
height: 100%;
       visibility: hidden; z-index: 91;}
```

В ячейке находится рисунок-заглушка

```
<img id="photo" src="pict/net.jpg" alt="Фото">
```

который, благодаря настройкам ячейки, располагается по центру таблицы:

```
#tdfo {text-align: center; vertical-align: middle;}
```

Наверное, у кого-то из читателей появился вопрос: а почему понадобился отдельный фоновый слой? Не проще ли обойтись только таблицей, указав в ее настройках белый цвет фона и уровень непрозрачности **0.9**? К сожалению, такой способ не подойдет, так как в этом случае изменится прозрачность большой фотографии (как дочернего элемента таблицы). Она станет «просвечивать», чего мы ни в коем случае не хотим.

Пора зарегистрировать обработчики. У нас их будет два – один станет открывать увеличенное изображение, а второй – удалять его:

```
window.addEventListener("load", function()
{
document.getElementById("fon").onclick=look;
document.getElementById("view").onclick=del;
});
window.addEventListener("scroll", del);
```

Функция **look** предназначена для демонстрации снимка и выглядит следующим образом:

```
function look()
{
if(event.target.id.indexOf("fr")==0)
    {
    let ni=event.target.id;
    let sc=window.pageYOffset;
    document.getElementById("bas").style.top=sc+"px";
    document.getElementById("view").style.top=sc+"px";
    document.getElementById("photo").src="pict/"+ni+".jpg";
    document.getElementById("view").style.visibility="visible";
    document.getElementById("bas").style.visibility="visible";
    }
return false;
}
```

Разберемся, как она работает.

Вы щелкнули мышью по фрагменту с рисунком фрукта или ягоды. После этого первым делом проверяется, где выполнен клик:

```
if(event.target.id.indexOf("fr")==0)
```

Если ID элемента, на котором произошло событие **click**, не содержит символов **fr**, то щелчок игнорируется программой. Если же клик выполнен на одном из слоев, то мы определяем, на котором из них:

```
let ni=event.target.id;
```

Выясняем, не прокручена ли страница вниз, и если да, то на сколько пикселей:

```
let sc=window.pageYOffset;
```

Затем смещаем фоновый слой и таблицу на соответствующее расстояние от верхнего края документа так, чтобы они закрыли все окно браузера:

```
document.getElementById("bas").style.top=sc+"px";
document.getElementById("view").style.top=sc+"px";
```

Заменяем адрес рисунка в таблице на увеличенный снимок выбранного нами фрагмента:

```
document.getElementById("photo").src="pict/"+ni+".jpg";
```

и делаем видимыми фоновый слой и таблицу с фото:

```
document.getElementById("view").style.visibility="visible";
document.getElementById("bas").style.visibility="visible";
```

Посмотрели – теперь можно убрать этот рисунок. Этим у нас занимается функция **del**, которая запускается в двух случаях:
– при прокрутке страницы вниз;
– в результате щелчка на поверхности таблицы (на рисунке или на пустом пространстве – без разницы).

Функция **del** выполняет следующие операции:

```
function del()
{
document.getElementById("view").style.visibility="hidden";
document.getElementById("bas").style.visibility="hidden";
document.getElementById("photo").src="pict/net.jpg";
return false;
}
```

Первые две инструкции делают фоновый слой и таблицу (вместе с рисунком на ней) невидимыми, третья инструкция удаляет из таблицы текущее изображение, заменяя его на рисунок-заглушку.

Листинг примера:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие click</title>

<style>
#bas {position: absolute; left: 0px; top: 0px; width: 100%; height:
100%;
      visibility: hidden; background: #FFFFFF; opacity: 0.9; z-
index: 90;}
#view {position: absolute; left: 0px; top: 0px; width: 100%;
height: 100%;
       visibility: hidden; z-index: 91;}
#tdfo {text-align: center; vertical-align: middle;}
#fon {position: relative;}
#fr1 {position: absolute; top: 56px; left: 88px; width: 200px;
      height: 150px; z-index: 31;}
#fr2 {position: absolute; top: 84px; left: 374px; width: 200px;
      height: 150px; z-index: 32;}
#fr3 {position: absolute; top: 36px; left: 665px; width: 200px;
      height: 150px; z-index: 33;}
#fr4 {position: absolute; top: 274px; left: 158px; width: 200px;
```

```
      height: 150px; z-index: 34;}
#fr5 {position: absolute; top: 308px; left: 454px; width: 200px;
      height: 150px; z-index: 35;}
#fr6 {position: absolute; top: 267px; left: 743px; width: 200px;
      height: 150px; z-index: 36;}
</style>

<script>
window.addEventListener("load", function()
{
document.getElementById("fon").onclick=look;
document.getElementById("view").onclick=del;
});
window.addEventListener("scroll", del);

function look()
{
if(event.target.id.indexOf("fr")==0)
    {
    let ni=event.target.id;
    let sc=window.pageYOffset;
    document.getElementById("bas").style.top=sc+"px";
    document.getElementById("view").style.top=sc+"px";
    document.getElementById("photo").src="pict/"+ni+".jpg";
    document.getElementById("view").style.visibility="visible";
    document.getElementById("bas").style.visibility="visible";
    }
return false;
}

function del()
{
document.getElementById("view").style.visibility="hidden";
document.getElementById("bas").style.visibility="hidden";
document.getElementById("photo").src="pict/net.jpg";
return false;
}
</script>
</head>

<body>

<div id="bas"></div>
<table id="view">
<tr><td id="tdfo"><img id="photo" src="pict/net.jpg"
alt="Фото"></td></tr>
</table>

<div id="fon">
<img src="pict/fon.jpg" alt="Фото">
<a href="#"><div id="fr1"></div></a>
<a href="#"><div id="fr2"></div></a>
<a href="#"><div id="fr3"></div></a>
<a href="#"><div id="fr4"></div></a>
<a href="#"><div id="fr5"></div></a>
<a href="#"><div id="fr6"></div></a>
</div>

</body>
</html>
```

Хотя код страницы получился довольно объемным, принцип действия сценария оказался очень простым.

Небольшая ремарка в самом конце. Вообще, мы могли бы получить аналогичный результат, сделав рисунок активной картой и задав соответствующие области в качестве ссылок. Но это давно известный способ, и я посчитал его менее интересным, чем вариант со слоями.

3.26. Событие paste. Вставка текста

Событие **paste** используется программистами довольно редко. Тем не менее мы познакомимся с одним примером его обработчика. Сценарий можно назвать исключительно иллюстративным. Его практическая сторона весьма незначительна, но зато такой пример очень нагляден.

Посмотрите на страницу **https://testjs.ru/p26.html**. Документ содержит короткий текст и поле для его вставки (рис. 3.26.1). Выделите весь текст или его часть, скопируйте и вставьте в поле. Вы увидите, как появится надпись «Произведена вставка текста» (рис. 3.26.2). Появление надписи, заменяющей исходное стихотворение, – это и есть реакция сценария на событие **paste**, которое произошло в текстовом поле.

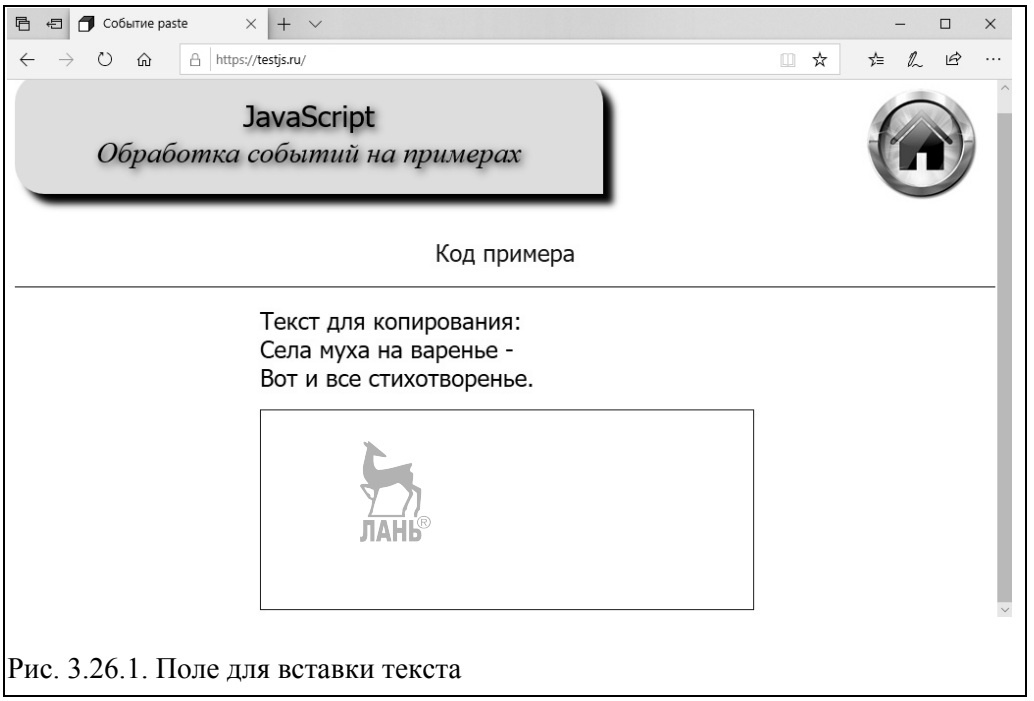

Рис. 3.26.1. Поле для вставки текста

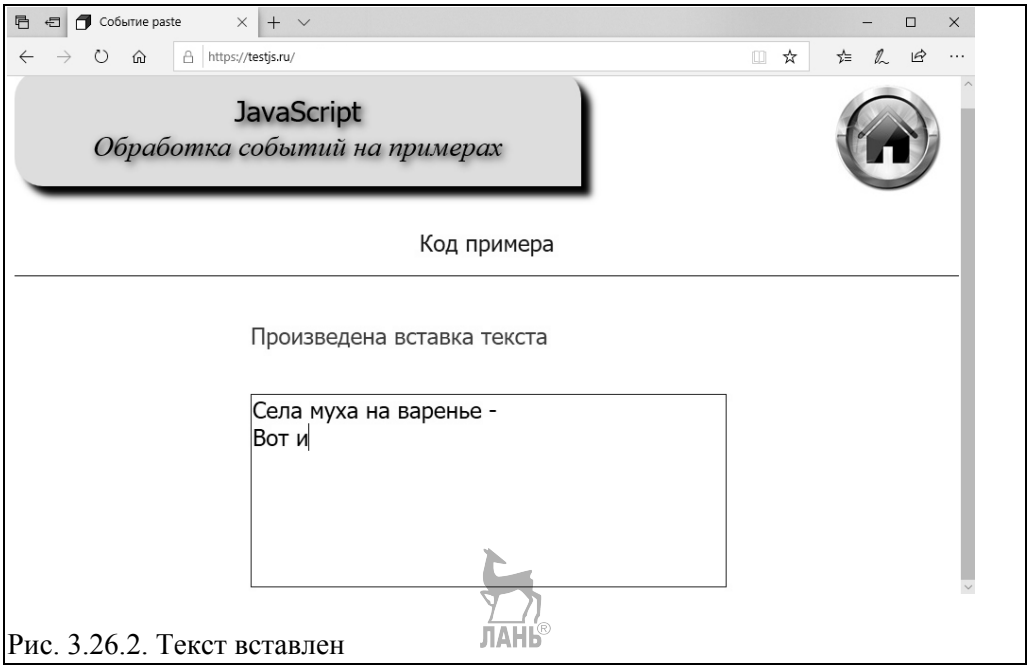

Рис. 3.26.2. Текст вставлен

Чтобы создать подобный пример, добавим в тело документа следующий текст:

```
<span id="sp">Текст для копирования: <br>
Села муха на варенье - <br>
Вот и все стихотворенье.</span>
```

Ниже расположим поле **textarea**:

```
<textarea id="tx"></textarea>
```

И напишем небольшой сценарий, в котором будет всего одна анонимная функция:

```
window.addEventListener("load", function()
{
document.getElementById("tx").addEventListener("paste", function()
   {
   document.getElementById("sp").style.color="#CC0000";
   document.getElementById("sp").innerHTML=" <br>Произведена
                                            вставка
текста<br> ";
   });
});
```

Функция выполняет очень простые операции. Когда пользователь производит вставку, вместо исходного текста появляется сообщение об этом

```
document.getElementById("sp").innerHTML=" <br>Произведена
                                        вставка текста<br> ";
```

которое выводится красным цветом:

```
document.getElementById("sp").style.color="#CC0000";
```

На этом все.

Полный код этого простейшего сценария:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие paste</title>

<script>
window.addEventListener("load", function()
{
document.getElementById("tx").addEventListener("paste", function()
   {
   document.getElementById("sp").style.color="#CC0000";
   document.getElementById("sp").innerHTML=" <br>Произведена
                                                вставка
текста<br> ";
   });
});
</script>
</head>

<body>

<span id="sp">Текст для копирования: <br>
Села муха на варенье - <br>
Вот и все стихотворенье.</span><br><br>
<textarea id="tx"></textarea>

</body>
</html>
```

3.27. Событие keyup. Подсчет знаков

Напомню, что событие **keyup** возникает в момент отпускания любой кнопки на клавиатуре после ее нажатия. Событие очень полезное и нередко используется программистами в различных сценариях. Мы создадим программу, которая подсчитывает знаки, введенные в текстовое поле, и сообщает посетителю, сколько знаков еще можно добавить.

Действующий пример можно посмотреть тут: **https://testjs.ru/p27.html**.

На странице есть два объекта – текстовое поле и табло с сообщением, какое количество знаков может ввести посетитель страницы (рис. 3.27.1). Мы ограничим длину вводимого текста сотней символов. Приступайте к набору произвольного текста. Вы увидите, как показания табло начнут уменьшаться (рис. 3.27.2). Когда до точки ограничения останется 20 знаков или меньше, сообщение на табло станет красного цвета, предупреждая клиента, что установленный

предел близок. Достигнув нулевой отметки, вы увидите, что знаки перестали добавляться в текстовое поле.

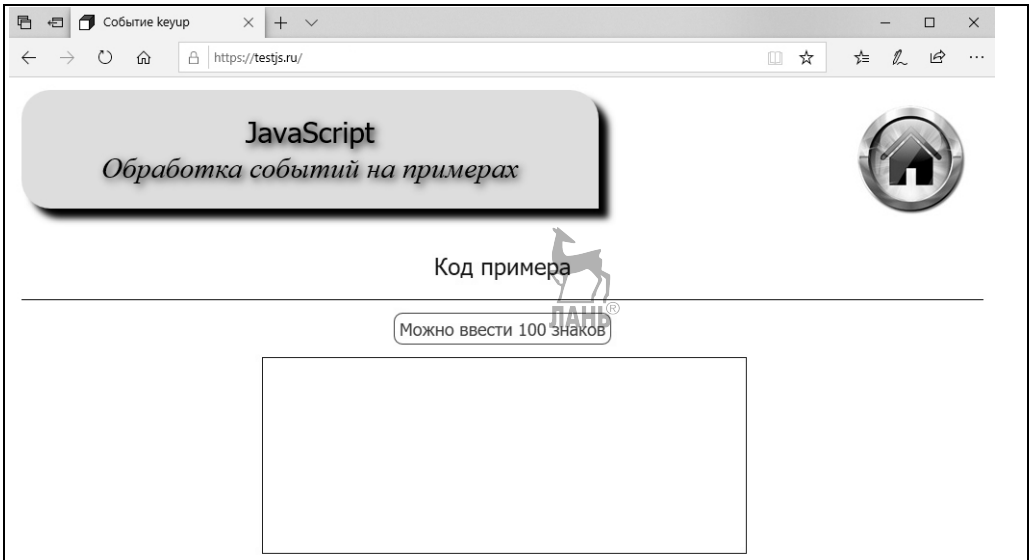

Рис. 3.27.1. Табло подсчета знаков, введенных в текстовое поле

Подобные сценарии часто используются программистами при создании форм – чтобы облегчить клиенту правильное заполнение того или иного поля.

Перейдем к рассмотрению программы.

В теле документа у нас есть табло, роль которого выполняет элемент **span**:

```
<span id="att">Можно ввести 100 знаков</span>
```

Под табло расположилось текстовое поле:

```
<textarea id="tx" maxlength="100"></textarea>
```

Сама программа состоит всего из одной анонимной функции:

```
window.addEventListener("load", function()
{
document.getElementById("tx").addEventListener("keyup", function()
   {
   let t=100-document.getElementById("tx").value.length;
   document.getElementById("att").innerHTML="Осталось: "+t;
   if(t<=20)
      {
      document.getElementById("att").style.color="#CC0000";
      }
   else
      {
```

```
        document.getElementById("att").style.color="#000000";
        }
    });
});
```

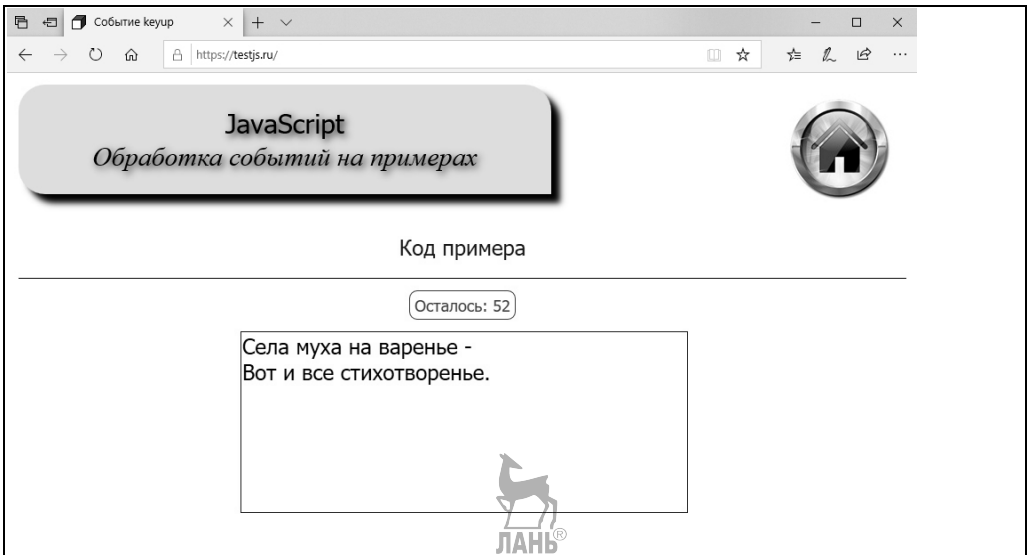

Рис. 3.27.2. Табло сообщает посетителю, сколько еще можно ввести знаков

В первой строке вычисляем количество знаков, которые может ввести клиент. Сначала мы определяем длину введенного текста

```
document.getElementById("tx").value.length;
```

а потом вычитаем полученный результат из числа, определяющего максимальное количество знаков:

```
let t=100-document.getElementById("tx").value.length;
```

Выводим результат вычислений на табло:

```
document.getElementById("att").innerHTML="Осталось: "+t;
```

Если до ограничения осталось 20 знаков или меньше, оформляем сообщения табло красным цветом:

```
if(t<=20)
   {
   document.getElementById("att").style.color="#CC0000";
   }
```

Последний блок

```
else
   {
```

```
   document.getElementById("att").style.color="#000000";
   }
```

необходим, чтобы вернуть тексту табло первоначальный цвет, когда клиент сначала ввел более 80 знаков (при этом табло поменяло цвет на красный), а потом часть из них удалил (а значит, до предельного значения осталось больше 20 знаков).

У нас получилась очень простая и весьма полезная программа:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие keyup</title>

<script>
window.addEventListener("load", function()
{
document.getElementById("tx").addEventListener("keyup", function()
   {
   let t=100-document.getElementById("tx").value.length;
   document.getElementById("att").innerHTML="Осталось: "+t;
   if(t<=20)
      {
      document.getElementById("att").style.color="#CC0000";
      }
   else
      {
      document.getElementById("att").style.color="#000000";
      }
   });
});
</script>
</head>

<body>

<span id="att">Можно ввести 100 знаков</span><br><br>
<textarea id="tx" maxlength="100"></textarea>

</body>
</html>
```

Мы познакомились с интересным вариантом использования события **keyup**. Но это не единственный пример. Дальше мы посмотрим, в каком еще случае может оказаться полезным обработчик этого события.

3.28. Событие keyup. Простейший WYSIWYG

Используя обработку события **keyup**, мы попробуем создать простейший WYSIWYG-редактор. Что это такое? Термин WYSIWYG – это аббревиатура, произошедшая от английского выражения What You See Is What You Get, что в переводе означает «что вы видите, то вы и получаете». В нашем

случае смысл этого выражения таков: что вы видите в редакторе, то и будет на странице сайта. Подобные визуальные редакторы используются практически во всех on-line конструкторах сайтов и системах управления сайтами. Обычно они состоят из двух частей: поля с визуальным представлением и поля textarea, в которое передается HTML-код из визуального редактора. Мы в очередном примере рассмотрим самый простейший случай взаимодействия этих полей.

Демонстрационная страница представлена по адресу **https://testjs.ru/p28.html**. Как видите, у нас есть два поля (рис. 3.28.1). Верхнее – это фрейм, находящийся в режиме визуального редактора. Каким образом мы сделали из фрейма редактор, объясним позже. Ниже расположено обычное текстовое поле.

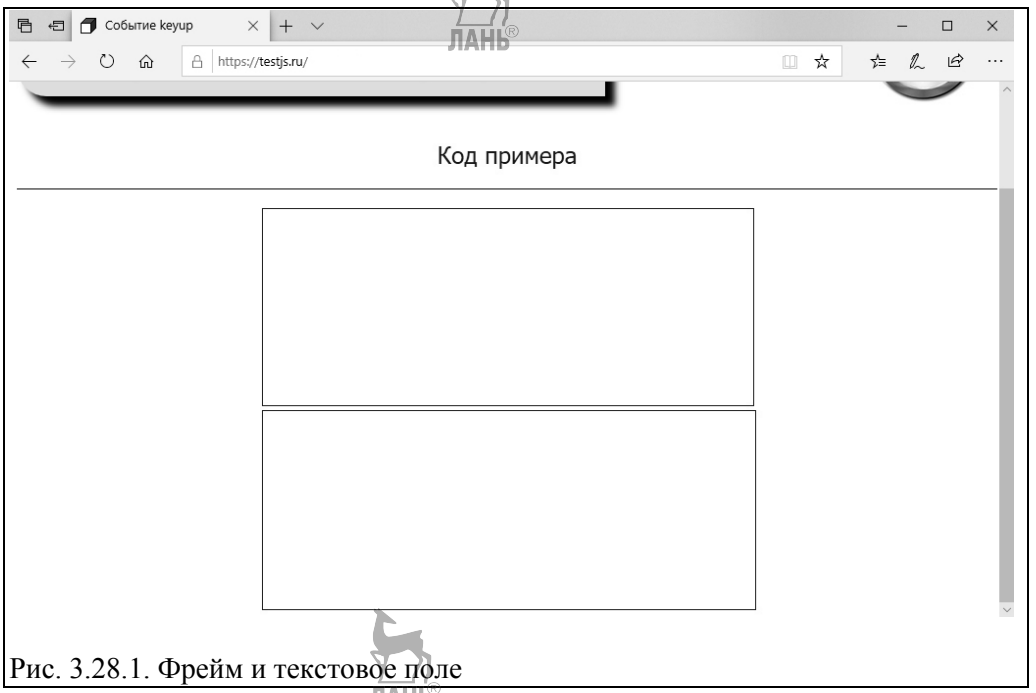

Рис. 3.28.1. Фрейм и текстовое поле

Попробуйте ввести во фрейм несколько строчек произвольного текста (для наглядности необходимы именно несколько строк с переходом на следующую нажатием клавиши «Enter»). Вы увидите, как в textarea появляются аналогичные строки, оформленные соответствующими HTML-тегами (рис. 3.28.2). Это – иллюстрация взаимодействия между полями. Обратите внимание: в подобных взаимодействиях все браузеры добавляют HTML-теги в текстовые поля автоматически, без нашего участия.

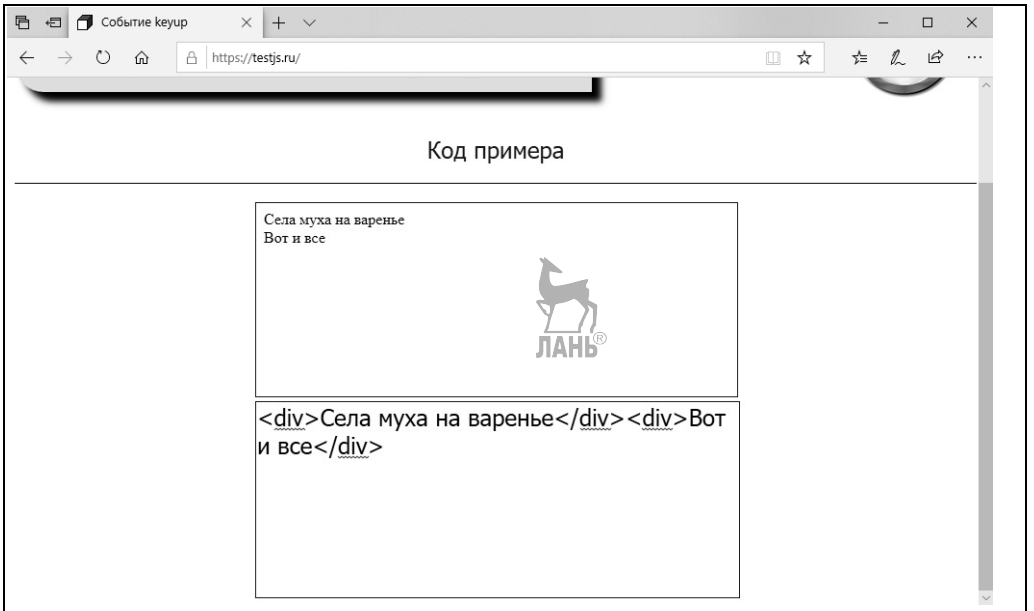

Рис. 3.28.2. Вводим текст во фрейме

Как сделан такой пример? На удивление, довольно просто. На странице есть фрейм

```
<iframe id="fr" name="fr"></iframe>
```

а под ним текстовое поле:

```
<textarea id="tx"></textarea>
```

Перед нами стоит задача превратить обычный фрейм в визуальный редактор. Это можно сделать, воспользовавшись атрибутом **designMode**. Данный атрибут, когда он включен (**designMode="on"**), позволяет редактировать любую HTML-страницу. Эта особенность натолкнула программистов на идею создания визуальных редакторов. Если поместить документ в **iframe**, включить у фрейма режим **designMode**, то можно прямо в браузере менять любую часть содержимого документа или добавлять новое содержимое в пустой документ.

В нашем случае мы включили данный режим пустому фрейму, благодаря чему можем вводить в него любой текст:

```
frames["fr"].document.designMode="on";
```

Передачей этого текста из фрейма управляет вот такая функция:

```
frames["fr"].document.addEventListener("keyup", function()
   {
```

```
   document.getElementById("tx").value=

frames["fr"].document.body.innerHTML;
   });
```

Данные из фрейма

```
frames["fr"].document.body.innerHTML
```

передаются в текстовое поле

```
document.getElementById("tx").value=frames["fr"].document.body.inn
erHTML;
```

каждый раз, когда в **iframe** происходит событие **keyup**. Таким образом, мы одновременно наблюдаем появление текста во фрейме и появление аналогичного текста, оформленного тегами, в **textarea**.

Задача организовать взаимодействие между полями простейшего редактора нами решена. Подведем итог, показав полную страницу примера:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие keyup</title>

<script>
window.addEventListener("load", function()
{
frames["fr"].document.designMode="on";

frames["fr"].document.addEventListener("keyup", function()
   {
   document.getElementById("tx").value=

frames["fr"].document.body.innerHTML;
   });
});
</script>
</head>

<body>

<iframe id="fr" name="fr"></iframe><br>
<textarea id="tx"></textarea>

</body>
</html>
```

3.29. Событие select. Из поля в поле

В следующем примере мы продемонстрируем обратное взаимодействие между полями на основе события **select**. Теперь будем передавать данные из **textarea** в **iframe**. Это также просто, как и в предыдущем случае.

Для демонстрации такого взаимодействия у нас есть страница **https://testjs.ru/p29.html**. Здесь разместились два поля: вверху – текстовое, внизу – фрейм, преобразованный уже известным нам способом в визуальный редактор. В текстовом поле заранее набрано стихотворение про зайчика, который вышел погулять. Отдельные слова стихотворения оформлены разными HTML-тегами (рис. 3.29.1).

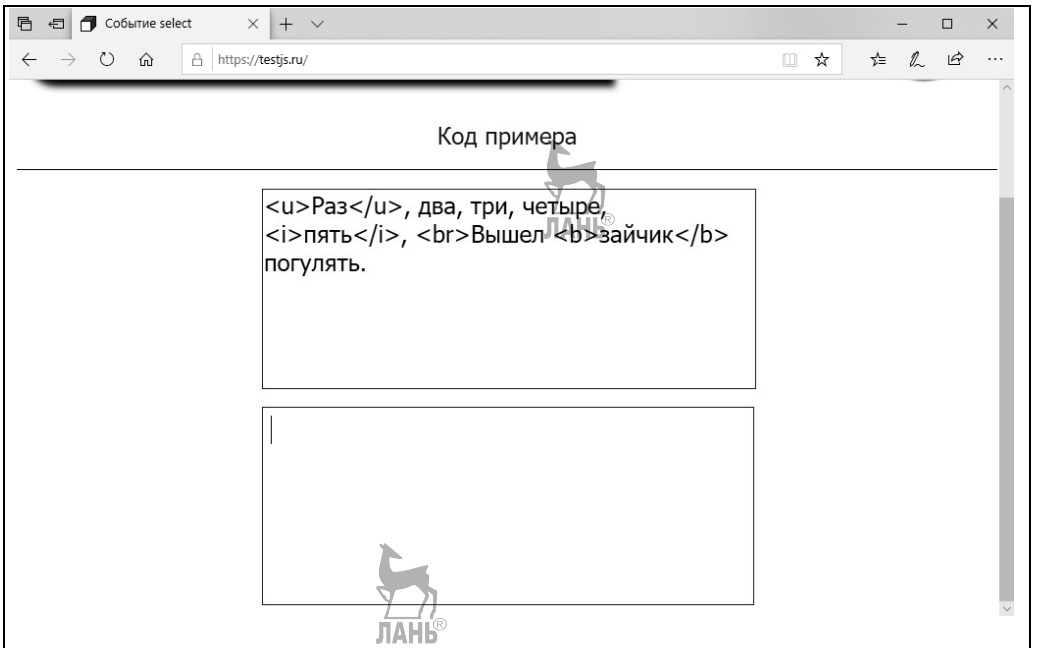

Рис. 3.29.1. Стихотворение с разметкой в текстовом поле

Пример необыкновенно прост. Достаточно выделить набранный текст – и в нижнем поле появятся те же визуально оформленные строки. Слово «раз» во фрейме будет подчеркнутым, слово «пять» – наклонным, а слово «зайчик» – жирным (рис. 3.29.2). Вы можете выделить лишь часть текста – тогда во фрейме появится только этот фрагмент, но по-прежнему с соответствующим оформлением. На таком примере мы демонстрируем еще один прием, используемый при создании WYSIWYG-редакторов.

Подготовить страницу с таким примером несложно. Разместим в теле документа текстовое поле со стихотворением:

```
<textarea id="tx"><u>Раз</u>, два, три, четыре, <i>пять</i>,
<br>Вышел <b>зайчик</b> погулять.</textarea>
```

Как видите, текст уже оформлен необходимыми тегами.

Ниже **textarea** у нас будет **iframe**:

```
<iframe id="fr" name="fr"></iframe>
```

«Превращаем» фрейм в визуальный редактор:

```
frames["fr"].document.designMode="on";
```

Регистрируем анонимный обработчик события **select**:

```
document.getElementById("tx").addEventListener("select", function()
   {
   ...
   });
```

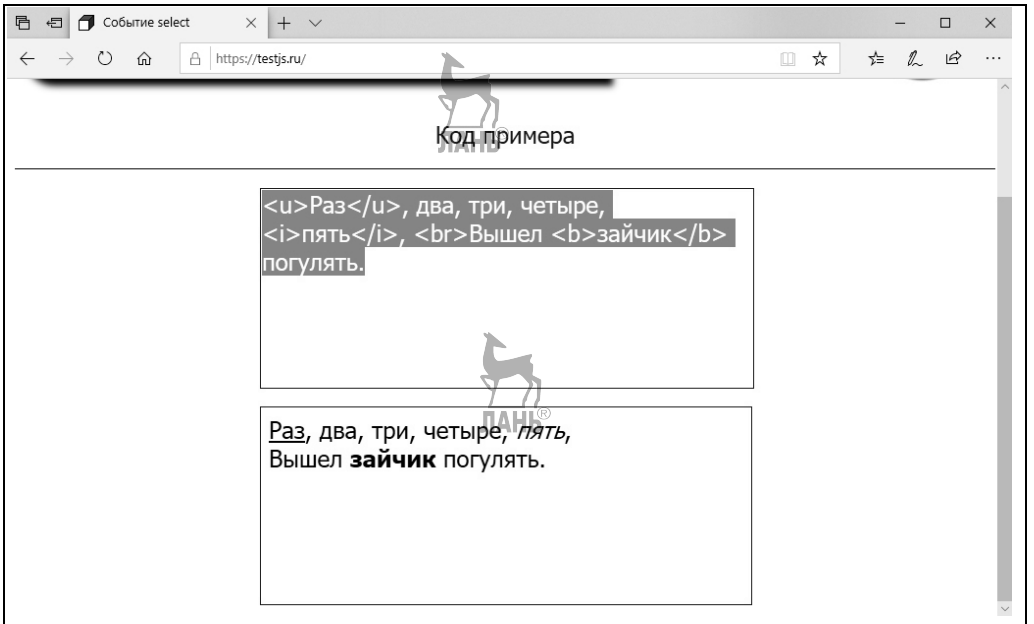

Рис. 3.29.2. Выделенный текст появился во фрейме

Функция содержит такие инструкции:

```
let v=document.getElementById("tx").value;
let st=document.getElementById("tx").selectionStart;
let en=document.getElementById("tx").selectionEnd;
frames["fr"].document.body.innerHTML=v.substring(st, en);
```

В первой строке мы создаем переменную **v** и присваиваем ей содержимое текстового поля:

```
let v=document.getElementById("tx").value;
```

Следующая инструкция определяет точку, с которой начинается выделение:

```
let st=document.getElementById("tx").selectionStart;
```

В третьей строке находим конечную точку выделенного фрагмента:

```
let en=document.getElementById("tx").selectionEnd;
```

Формируем строку, содержащую выделенный фрагмент

```
v.substring(st, en);
```

и передаем ее во фрейм:

```
frames["fr"].document.body.innerHTML=v.substring(st, en);
```

После выполнения этих операций мы видим выделенный текст в **iframe** в его визуальном представлении.

Этот пример также прост, как и предыдущие. Вот его код:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие select</title>

<script>
window.addEventListener("load", function()
{
frames["fr"].document.designMode="on";

document.getElementById("tx").addEventListener("select", function()
   {
   let v=document.getElementById("tx").value;
   let st=document.getElementById("tx").selectionStart;
   let en=document.getElementById("tx").selectionEnd;
   frames["fr"].document.body.innerHTML=v.substring(st, en);
   });
});
</script>
</head>

<body>

<textarea id="tx"><u>Раз</u>, два, три, четыре, <i>пять</i>,
<br>Вышел <b>зайчик</b> погулять.</textarea><br><br>
<iframe id="fr" name="fr"></iframe>

</body>
</html>
```

3.30. События select и click. Делаем редактор

Используя обработчики событий **select** и **click**, попробуем сделать подобие настоящего визуального редактора.

Зайдите на страницу **https://testjs.ru/p30.html** сайта поддержки книги. Здесь представлен редактор, который мы создадим в данном параграфе. Он состоит из двух полей. Вверху – текстовое с уже знакомым стихотворением про зайчика, внизу – фрейм. Над всей этой конструкцией поместились три кнопки для оформления содержимого текстового поля (рис. 3.30.1). Кнопки позволяют сделать выделенный фрагмент жирным, наклонным и подчеркнутым.

Поэкспериментируем. Выделите любую часть текста и нажмите одну из кнопок. Весь текст тут же появится во фрейме, причем выделенный фрагмент будет оформлен в соответствии с нажатой кнопкой (рис. 3.30.2). Попробуйте выделять разные части текста и нажимать разные кнопки. Каждый раз вы будете видеть соответствующее визуальное представление во фрейме. Как видите, у нас получился простейший редактор с минимальным набором функций.

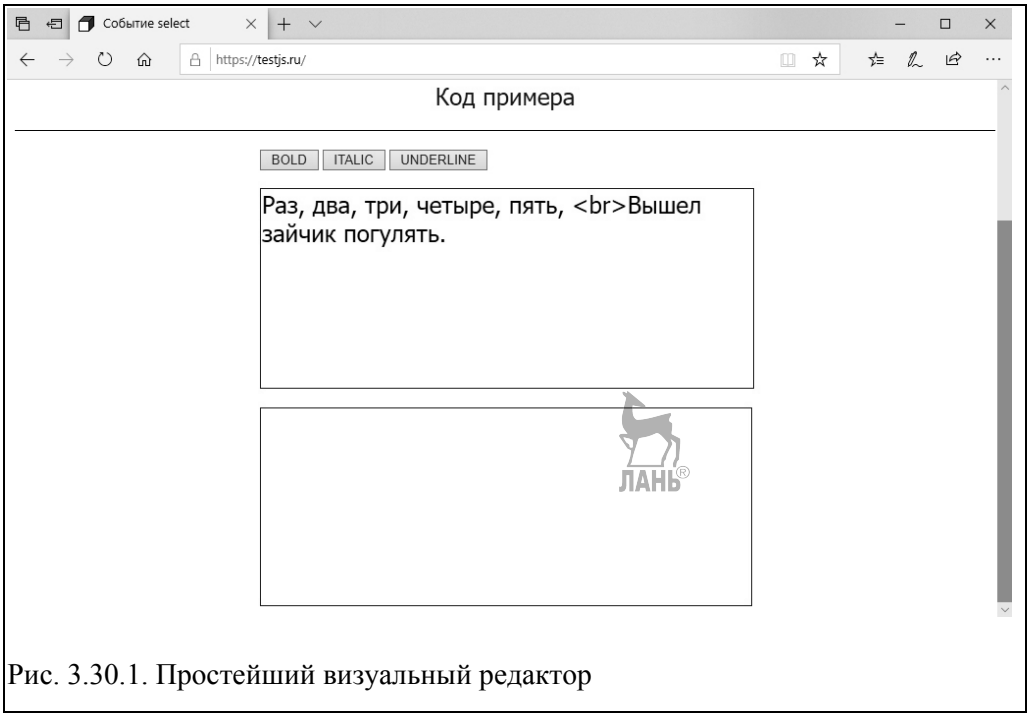

Рис. 3.30.1. Простейший визуальный редактор

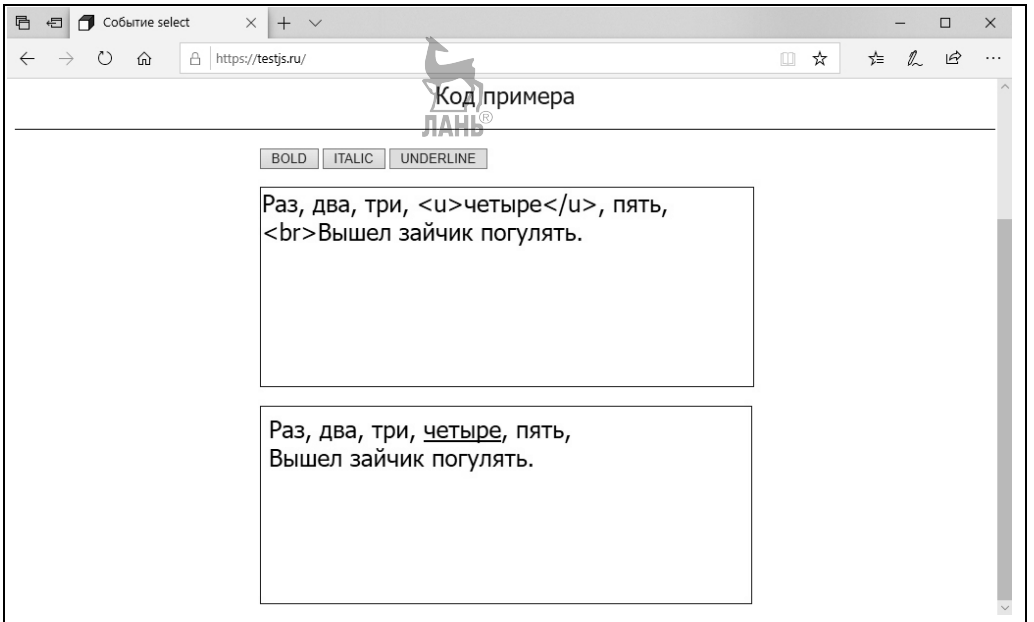

Рис. 3.30.2. Оформляем текст во фрейме

Начнем создание программы с того, что поместим в тело документа три кнопки:

```
<input id="bo" type="button" value=" BOLD ">
<input id="it" type="button" value=" ITALIC ">
<input id="un" type="button" value=" UNDERLINE ">
```

Под ними у нас будет текстовое поле с фрагментом стихотворения:

```
<textarea id="tx">Раз, два, три, четыре, пять,
<br>Вышел зайчик погулять.</textarea>
```

А еще ниже – фрейм:

```
<iframe id="fr" name="fr"></iframe>
```

Включим режим **designMode**:

```
frames["fr"].document.designMode="on";
```

За оформление текста у нас отвечает анонимная функция, которая содержит три похожих блока. Разберем один из них, например, добавляющий к выделенному фрагменту теги **...**:

```
document.getElementById("bo").addEventListener("click", function()
   {
```

```
   let v=document.getElementById("tx").value;
   let st=document.getElementById("tx").selectionStart;
   let en=document.getElementById("tx").selectionEnd;
   let res=v.substring(st, en);
   document.getElementById("tx").value=v.substring(0, st)+
'<b>'+res+'</b>'+v.substring(en);

   frames["fr"].document.body.innerHTML=
document.getElementById("tx").value;
   });
```

В первой строке мы получаем содержимое текстового поля:

```
let v=document.getElementById("tx").value;
```

Затем определяем начальную

```
let st=document.getElementById("tx").selectionStart;
```

и конечную

```
let en=document.getElementById("tx").selectionEnd;
```

точки выделенного фрагмента.

Получаем часть текста, у которого надо изменить оформление

```
let res=v.substring(st, en);
```

и формируем новую строку с тегами **...** у выделенной части:

```
v.substring(0, st)+'<b>'+res+'</b>'+v.substring(en);
```

Изменяем содержимое поля **textarea**:

```
document.getElementById("tx").value=v.substring(0, st)+
'<b>'+res+'</b>'+v.substring(en);
```

Когда подготовка текста завершена, передаем его во фрейм:

```
frames["fr"].document.body.innerHTML=document.getElementById("tx")
.value;
```

В результате мы видим во фрейме текст из поля **textarea** в его визуальном оформлении.

Как уже было сказано выше, теги, отвечающие за наклонный и подчеркнутый шрифт, добавляются аналогичным образом. Не будем разбирать соответствующие блоки программы, а просто продемонстрируем ее листинг «в сборе»:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие select</title>

<script>
window.addEventListener("load", function()
{
frames["fr"].document.designMode="on";

document.getElementById("tx").addEventListener("select", function()
   {
   document.getElementById("bo").addEventListener("click", function()
      {
      let v=document.getElementById("tx").value;
      let st=document.getElementById("tx").selectionStart;
      let en=document.getElementById("tx").selectionEnd;
      let res=v.substring(st, en);
      document.getElementById("tx").value=v.substring(0, st)+
'<b>'+res+'</b>'+v.substring(en);

      frames["fr"].document.body.innerHTML=
document.getElementById("tx").value;
      });

   document.getElementById("it").addEventListener("click", function()
      {
      let v=document.getElementById("tx").value;
      let st=document.getElementById("tx").selectionStart;
      let en=document.getElementById("tx").selectionEnd;
      let res=v.substring(st, en);
      document.getElementById("tx").value=v.substring(0, st)+
'<i>'+res+'</i>'+v.substring(en);

      frames["fr"].document.body.innerHTML=
document.getElementById("tx").value;
      });

   document.getElementById("un").addEventListener("click", function()
      {
      let v=document.getElementById("tx").value;
      let st=document.getElementById("tx").selectionStart;
      let en=document.getElementById("tx").selectionEnd;
      let res=v.substring(st, en);
      document.getElementById("tx").value=v.substring(0, st)+
'<u>'+res+'</u>'+v.substring(en);

      frames["fr"].document.body.innerHTML=
document.getElementById("tx").value;
      });
   });
});
</script>
</head>


<body>
```

```
<input id="bo" type="button" value=" BOLD ">
<input id="it" type="button" value=" ITALIC ">
<input id="un" type="button" value=" UNDERLINE "> <br><br>
<textarea id="tx">Раз, два, три, четыре, пять,
<br>Вышел зайчик погулять.</textarea><br><br>
<iframe id="fr" name="fr"></iframe>

</body>
</html>
```

3.31. События mouseenter и mouseleave. Играем в гольф

Настала пора самого сложного примера в этой книге. Причем, в первую очередь сложен не столько код, сколько логика действия сценария.

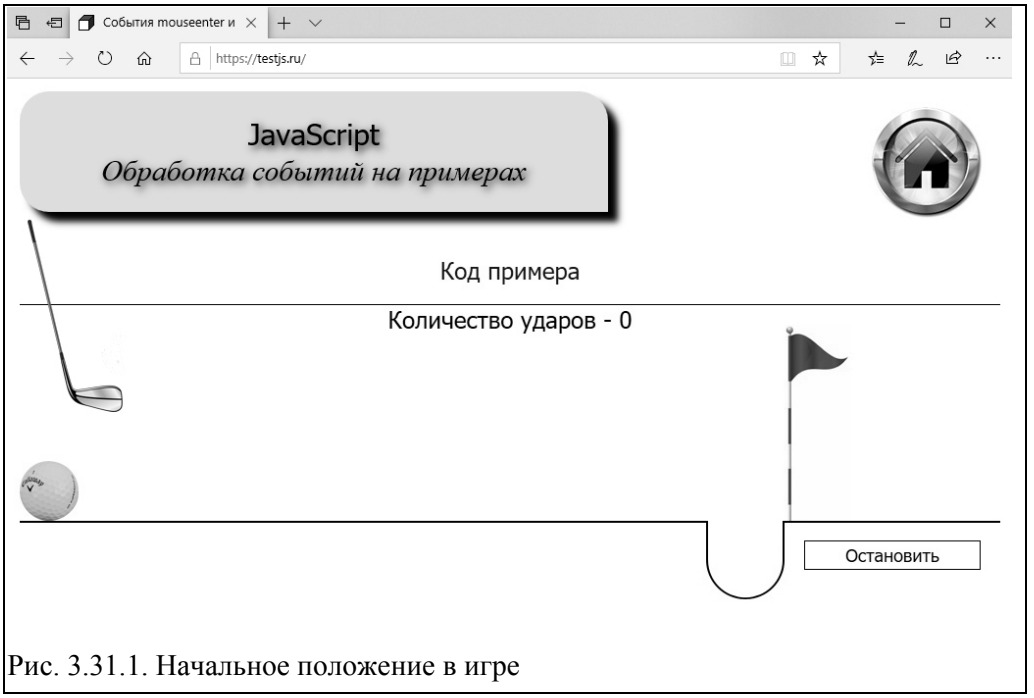

Рис. 3.31.1. Начальное положение в игре

Мы посмотрим в действии обработку событий **mouseenter** и **mouseleave**. О том, чем они отличаются от **mouseover** и **mouseout**, вы уже прочитали в п. 2.2.

Попробуем создать несложную игру, которая станет простейшим аналогом гольфа. У нас будет поле, одна лунка, мяч и клюшка, а также счетчик ударов (и даже флажок около лунки – как на настоящем поле для гольфа). Задача игрока – загнать мяч в лунку минимальным количеством ударов.

Пример такой игры представлен на странице **https://testjs.ru/p31.html**. В случае точных действий игрока мяч падает в лунку, после чего игра прекращается. Запустить вновь ее можно, перезагрузив страницу. Если игрок произве-

дет слишком сильные удары, мяч проскочит лунку и прикатится к краю поля, после чего игра также будет закончена. Возможность нанести удар в обратном направлении при перелете мяча за лунку не предусмотрена. Так сделано, чтобы упростить программный код игры.

Обратите внимание! Игра спроектирована с расчетом, что для ее нормальной работы необходим монитор, имеющий горизонтальное разрешение более 1000 пикселей.

Как выглядит страница в исходном состоянии, показано на рисунке 3.31.1.

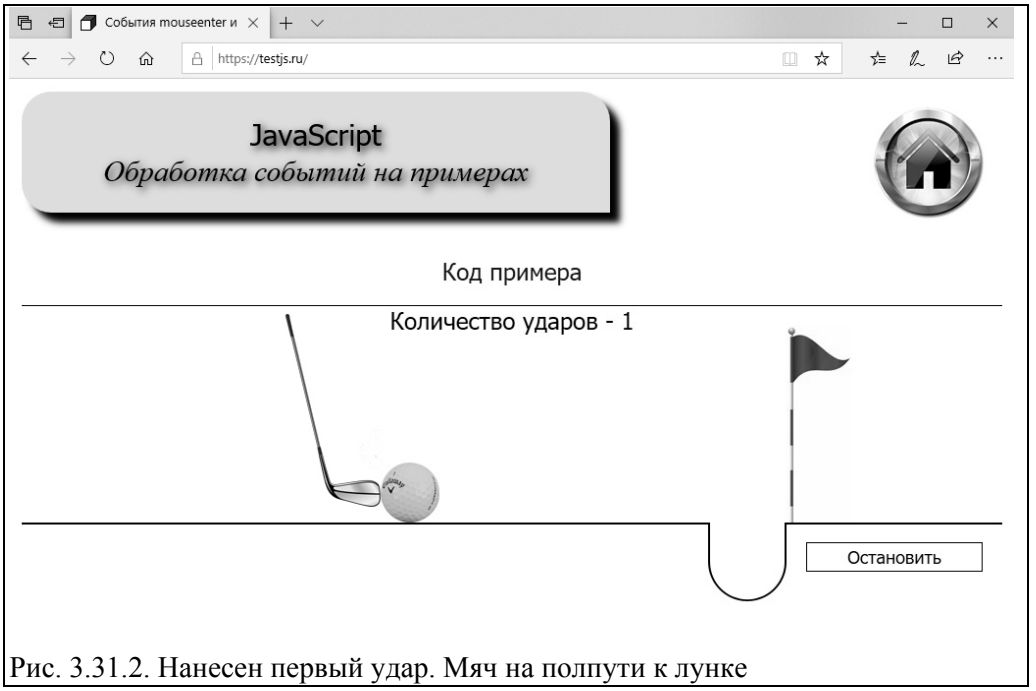

Рис. 3.31.2. Нанесен первый удар. Мяч на полпути к лунке

После того как указатель мыши окажется на странице, к указателю «привяжется» клюшка. Теперь она будет постоянно следовать за курсором. Установите клюшку слева от мяча на некотором расстоянии. Щелкните левой кнопкой мыши – этот клик запускает программу. Теперь имитируйте удар клюшкой по мячу. Вы сразу увидите, что мяч переместился ближе к лунке. Поскольку наша задача не устраивать развлекательные состязания, а разобраться, как работает подобный сценарий, открою вам небольшой секрет: дальность «полета» мяча в 2 раза больше расстояния между точкой щелчка и мячом. Поэтому, нанося удары, старайтесь «на глазок» прикидывать его предполагаемое место «приземления». Например, мой первый удар, как видно из рисунка 3.31.2, сократил расстояние от мяча до лунки сразу наполовину.

Чтобы нанести следующий удар, вновь установите клюшку слева от мяча на некотором расстоянии. Опять щелкните мышью и проведите курсором мыши по мячу.

Если мяч еще не достиг лунки, надо сделать третий удар – и так до тех пор, пока не произойдет попадание в цель. В этот момент мяч упадет в лунку, счетчик ударов остановится и игра завершится (рис. 3.31.3).

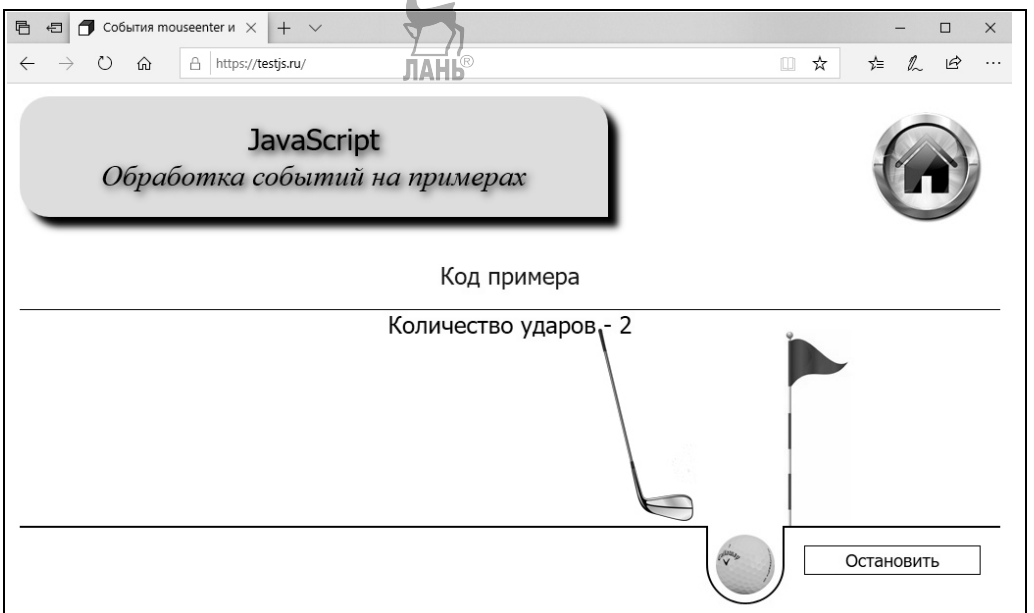

Рис. 3.31.3. После второго удара мяч оказался в лунке

Если мяч перелетел лунку (рис. 3.31.4), вы проиграли, так как в обратную сторону ударить нельзя. Чтобы мяч не улетал слишком далеко, границы его перемещения лежат в пределах поля. То есть при сильном перелете мяч остановится у правого края поля.

Надо отметить один нюанс программы. Клюшка «привязана» к указателю мыши таким образом, что курсор все время оказывается на головке. И щелчки тоже происходят на головке клюшки. Это из-за того, что изображение клюшки имеет довольно высокий z-index в иерархии элементов страницы и, к тому же, так продумана логика действия сценария. Более высокие z-index – у мяча и кнопки «Остановить». С первым все понятно – это чтобы клюшка при ударе проходила за мячом. А вот случай с кнопкой разберем подробнее. Испытав программу, вы решили посмотреть ее код или вернуться на главную страницу сайта. И тут обнаружите, что нажать соответствующие ссылки не получится – мешает изображение клюшки, которое все время оказывается поверх ссылок. Однако ситуация не безвыходная. Наведите мышь на кнопку и клюшка окажется под ней. Нажмите «Остановить». Клюшка застынет на месте, прекратив сопровождение курсора (рис. 3.31.5). Теперь можно посмотреть код примера или перейти на главную страницу демонстрационного сайта. А можно обновить документ и игра запустится вновь.

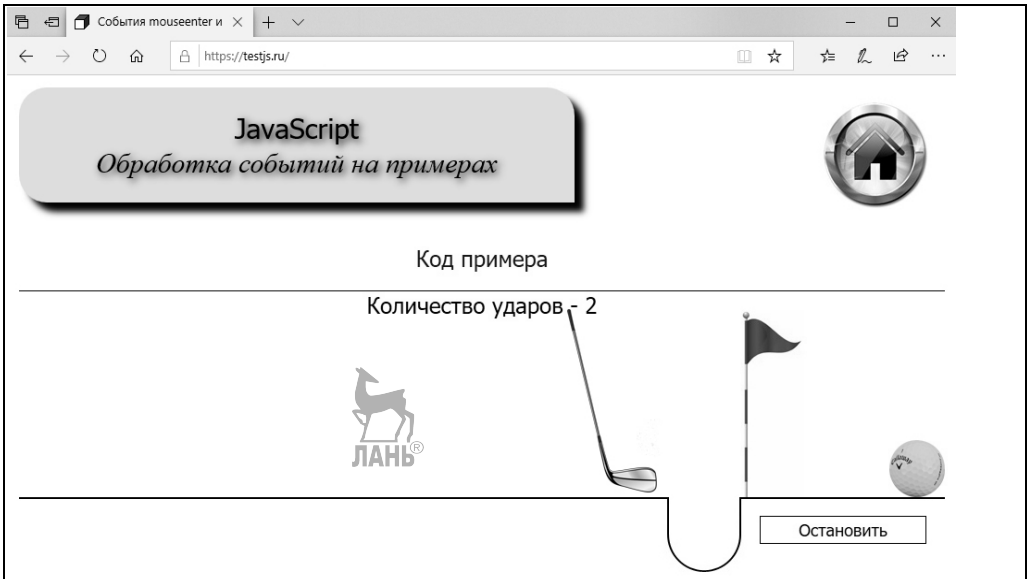

Рис. 3.31.4. От «сильного» удара мяч перелетел лунку

Приступим к созданию такой программы. Как обычно, начинаем с размещения необходимых элементов.

Первый – это клюшка. Так как она должна свободно перемещаться по всей странице, то «пропишем» ее сразу после открывающего тега **<body>**:

```
<img id="golf" src="pict/golf.png" alt="Фото">
```

Настройки стилей изображения клюшки:

```
#golf {position: absolute; left: 0px; top: 0px; z-index: 40;}
```

Теперь создадим игровое поле, добавив в тело документа слой с **id="pic"**:

```
<div id="pic">...</div>
```

Настройки слоя:

```
#pic {position: relative; width: 1000px; height: 300px; margin: 0px auto;}
```

Как видите, слой располагается по середине страницы, его ширина – **1000**, а высота – **300** пикселей.

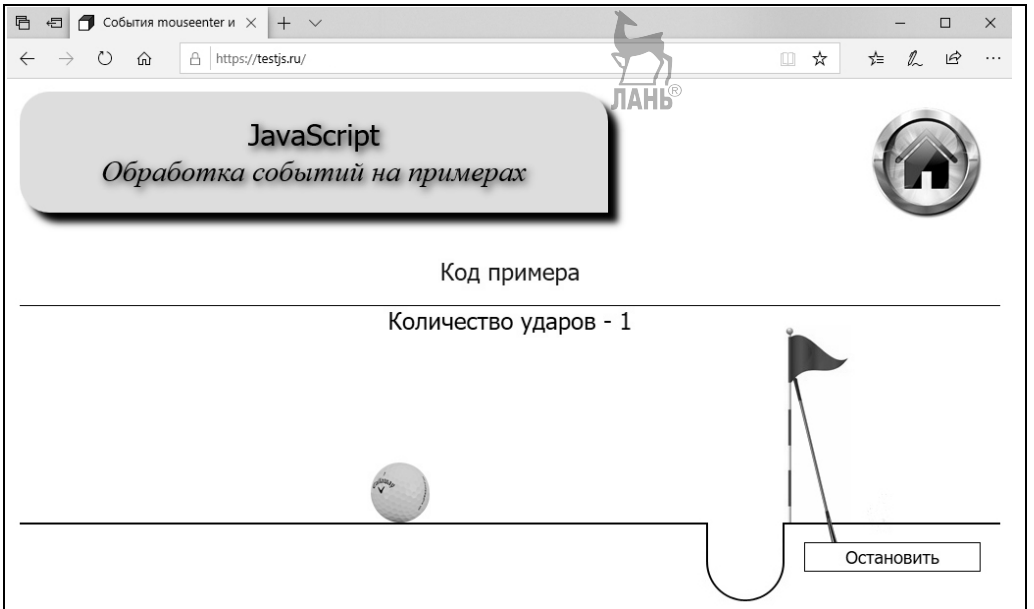

Рис. 3.31.5. Избавляемся от «преследования» клюшкой указателя мыши

Все следующие элементы помещаем на этот слой.

Добавляем счетчик ударов:

```
<div id="te">Количество ударов - 0</div>
```

Счетчик тоже расположим по середине страницы:

```
#te {position: relative; width: 500px; margin: 0px auto;
     text-align: center; z-index: 20;}
```

Займемся формированием игрового поля:

```
<img id="hole" src="pict/hole.jpg" alt="Фото">
```

Его координаты на слое:

```
#hole {position: absolute; left: 0px; bottom: 0px; z-index: 30;}
```

Установим на поле мяч

```
<img id="ball" src="pict/ball.png" alt="Фото">
```

и зададим его начальное положение:

```
#ball {position: absolute; left: 0px; bottom: 80px; z-index: 50;}
```

Для красоты «воткнем» около лунки флажок

```
<img id="flag" src="pict/flag.jpg" alt="Фото">
```

и пропишем его координаты:

```
#flag {position: absolute; left: 780px; bottom: 80px; z-index: 20;}
```

Игровое поле готово. Осталось добавить кнопку «Остановить»

```
<input type="button" value="Остановить" id="bu">
```

и расположить ее где-нибудь в нижней части страницы:

```
#bu {position: absolute; left: 800px; bottom: 30px; z-index: 60;}
```

Правила игры мы уже продумали – приступаем к написанию сценария.

Регистрируем первый обработчик. Он станет реагировать на перемещения мыши по странице и необходим для «привязки» клюшки к указателю.

```
window.addEventListener("mousemove", mom);
```

Регистрируем еще четыре обработчика:

```
window.addEventListener("load", function()
{
document.getElementById("ball").addEventListener("mouseenter", hit);
document.getElementById("ball").addEventListener("mouseleave", sto);
document.getElementById("golf").addEventListener("click", plu);
document.getElementById("bu").addEventListener("click", lin);
});
```

Первый запускается, когда курсор попадает на мяч (событие **onmouseenter**), второй – в момент ухода курсора с мяча (событие **onmouseleave**), третий срабатывает при клике мышью на головке клюшки, четвертый необходим для «отвязывания» клюшки от курсора.

Функция «привязки» **mom** выглядит так:

```
function mom()
{
if(f==1)
   {
   let g=event.pageX-70;
   let v=event.pageY-180;

   document.getElementById("golf").style.left=g+"px";
   document.getElementById("golf").style.top=v+"px";
   }
}
```

Сначала мы определяем координаты мыши на странице, а после добавляем необходимое смещение рисунка, чтобы головка клюшки находилась прямо под курсором:

```
let g=event.pageX-70;
let v=event.pageY-180;
```

Затем полученные значения присваиваем свойствам позиционирования изображения:

```
document.getElementById("golf").style.left=g+"px";
document.getElementById("golf").style.top=v+"px";
```

Конечно, вы обратили внимание, что все эти действия происходят при соблюдении условия

```
if(f==1)
   {
   ...
   }
```

f – это идентификатор состояния игры. Его мы должны объявить до написания функции **mom**:

```
let f=1;
```

Следом за объявлением идентификатора напишем функцию, управляющую его значением:

```
function lin()
{
f=2;
}
```

Что у нас получается? Пока кнопка «Остановить» не нажата, идентификатор **f** сохраняет значение **1** и клюшка свободно перемещается за указателем мыши. После нажатия кнопки значение идентификатора увеличивается до 2, значит, условие

```
if(f==1)
```

функции **mom** становится ложным и перемещение клюшки за курсором прекращается. Теперь игрок может нажимать ссылки на странице.

Дальше мы объявим несколько глобальных переменных:

```
let i=1;
let a=2;
let h=0;
let s=0;
```

Идентификатор **i** контролирует ситуацию попадания мяча в лунку, **a** отвечает за готовность программы к новому удару по мячу, в **h** записывается смещение мяча относительно левого края поля, а **s** – это счетчик ударов.

Сразу напишем функцию, обрабатывающую событие **onmouseleave**, которое происходит, когда указатель мыши покидает изображение мяча:

```
function sto()
{
a=2;
}
```

Для чего нужны идентификатор **a** и функция **sto**? Представим, что их нет. Тогда при ударе по мячу будут происходить интересные вещи. Смотрите: указатель мыши попал на мяч, покинул границы его изображения и по инерции еще движется вперед. Но в этот момент мяч сместился и догнал указатель, а значит происходит следующий удар. Клюшка продолжает смещаться вперед, ее опять догоняет мяч – новый удар. И так до бесконечности. В итоге при определенной ловкости игрок может вести мяч, как в хоккее, и таким образом завести его в лунку. Но ведь у нас не хоккей, а гольф. Поэтому нужно, чтобы после первого удара сценарий останавливался и ждал, когда игрок вновь щелкнет мышью на некотором расстоянии от мяча и совершит новый удар. Для реализации такого решения как раз и необходимы идентификатор **a** и функция **sto**.

В начальном состоянии **a=2**. Это означает, что нанести удар пока невозможно. После первого щелчка мышью данный идентификатор получает значение **1.** Это происходит в следующей функции, которую нам еще предстоит разобрать. После чего можно наносить удар. Как он выглядит в замедленном воспроизведении? Указатель мыши попадает на мяч (событие **onmouseenter**), проходит по мячу и покидает границы его изображения (событие **onmouseleave**). Даже если клюшка продолжит по инерции движение вперед и вновь встретится с мячом, который переместился вправо, повторного удара не произойдет. Достаточно клюшке один раз потерять контакт с мячом, как функция **sto** изменит значение идентификатора **a** с **1** на **2** и сценарий перестанет реагировать на новые удары – до тех пор, пока игрок снова не щелкнет мышью на головке клюшки.

Изменение значения идентификатора **a** с **2** на **1** происходит внутри функции **plu**:

```
function plu()
{
a=1;

let w=screen.width;
let x=event.pageX;
let b=document.getElementById("ball").offsetLeft;

if(b+(w-1000)/2>x)
   {
   h=(b-x+(w-1000)/2)*2+b;
```

```
   }
else
   {
   h=b;
   }
}
```

Как только данная функция запускается, идентификатор **a** получает значение **1**.

Но у функции **plu** есть и другие задачи:

– определить расстояние между местом щелчка и левой границей рисунка мяча;

– рассчитать растояние, на которое должен сместиться мяч после удара.

Так как щелчок мышью может произойти за пределами слоя **pic**, в расчетах надо отталкиваться не от его размеров, а от горизонтального разрешения экрана компьютера. Его мы и определяем в первую очередь:

```
let w=screen.width;
```

Потом узнаем горизонтальную координату щелчка мышью

```
let x=event.pageX;
```

и определяем положение мяча на слое **pic**:

```
let b=document.getElementById("ball").offsetLeft;
```

Теперь надо рассчитать положение мяча относительно границ экрана. Вычисляем расстояние между границами слоя **pic** и краями экрана (напомню, что ширина слоя **pic** у нас **1000** пикселей):

```
w-1000
```

Делим полученный результат на **2**:

```
(w-1000)/2
```

Мы получили расстояние от левой границы экрана до левой границы слоя **pic**. Теперь прибавим к данному числу значение переменной **b** – смещение мяча внутри слоя относительно его левого края:

```
b+(w-1000)/2
```

Итак, мы вычислили правое смещение мяча относительно левой границы экрана. Сравним его с горизонтальной координатой щелчка. Если эта координата левее, то выполняется условие

```
if(b+(w-1000)/2>x)
```

и производится расчет точки, куда мяч должен переместиться после удара:

```
h=(b-x+(w-1000)/2)*2+b;
```

Здесь мы из уже вычисленного текущего положения мяча вычитаем координату щелчка, умножаем полученный результат на **2** (как вы помните из описания игры, такими мы задали пропорции «силы» удара) и к получившейся дальности полета мяча прибавляем его координату до удара. В результате переменная **h** получает новое значение, которое будет использовано следующей функцией.

Если расстояние между точкой клика и мячом равно **0** или имеет отрицательное значение, выполняется вторая часть условия:

```
else
   {
   h=b;
   }
```

Переменная **h** сохраняет текущее значение координаты мяча.

Самая сложная функция программы обрабатывает различные случаи, возникающие на поле:

– мяч еще не достиг лунки;
– мяч попал в лунку;
– мяч улетел за границы поля.

Посмотрите на нее:

```
function hit()
{
if((i==1)&&(a==1))
   {
   let d=document.getElementById("ball").offsetLeft;

   if(d<940)
      {
      document.getElementById("ball").style.left=h+"px";

      s++;
      document.getElementById("te").innerHTML="Количество ударов -
"+s;

      if((h>690)&&(h<780))
         {
         document.getElementById("ball").style.left="710px";
         document.getElementById("ball").style.bottom="10px";
         i=2;
         }
      }

      let c=document.getElementById("ball").offsetLeft;

      if(c>940)
      {
      document.getElementById("ball").style.left="940px";
      i=2;
```

```
      }
   }
}
```

Функция запускается при ударе клюшки по мячу (событие **onmouseenter**).

Если **(i==1)&&(a==1)**, то есть мяч еще не попал в лунку и удар разрешен, выясняем текущее положение мяча относительно границ слоя **pic**:

```
let d=document.getElementById("ball").offsetLeft;
```

Следом проводится сравнение текущего положения мяча с его максимально возможным правым смещением:

```
if(d<940)
```

Если условие истинно, перемещаем мяч в новое положение, увеличиваем счетчик ударов на **1** и выводим его показания на страницу:

```
document.getElementById("ball").style.left=h+"px";
s++;
document.getElementById("te").innerHTML="Количество ударов - "+s;
```

После удара надо выяснить, не попал ли мяч в лунку:

```
if((h>690)&&(h<780))
   {
   document.getElementById("ball").style.left="710px";
   document.getElementById("ball").style.bottom="10px";
   i=2;
   }
```

Если мяч не достиг лунки, игра продолжается. Если значание переменной **h** оказывается в пределах от **690** до **780** пикселей, считаем, что удар был точным. Присваиваем идентификатору **i** значение **2**. Теперь условие **if((i==1)&&(a==1))** будет ложным и игра остановится.

Если мяч не попал в лунку, проверяем, не вылетел ли он за границы поля. Для этого определяем текущее положение мяча относительно границ слоя **pic**:

```
let c=document.getElementById("ball").offsetLeft;
```

Если мяч улетел слишком далеко

```
if(c>940)
```

возвращаем его на правый край поля и останавливаем игру:

```
document.getElementById("ball").style.left="940px";
i=2;
```

Теперь все. Я постарался максимально внятно объяснить принцип работы этого непростого сценария. Надеюсь, читатели разобрались во всех его нюансах.

Страница вместе со сценарием получилась довольно «объемной»:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>События mouseenter и mouseleave</title>

<style>
#pic {position: relative; width: 1000px; height: 300px; margin:
0px auto;}
#golf {position: absolute; left: 0px; top: 0px; z-index: 40;}
#ball {position: absolute; left: 0px; bottom: 80px; z-index: 50;}
#hole {position: absolute; left: 0px; bottom: 0px; z-index: 30;}
#flag {position: absolute; left: 780px; bottom: 80px; z-index: 20;}
#te {position: relative; width: 500px; margin: 0px auto;
     text-align: center; z-index: 20;}
#bu {position: absolute; left: 800px; bottom: 30px; z-index: 60;}
</style>

<script>
window.addEventListener("mousemove", mom);
window.addEventListener("load", function()
{
document.getElementById("ball").addEventListener("mouseenter", hit);
document.getElementById("ball").addEventListener("mouseleave", sto);
document.getElementById("golf").addEventListener("click", plu);
document.getElementById("bu").addEventListener("click", lin);
});

let f=1;

function lin()
{
f=2;
}

function mom()
{
if(f==1)
   {
   let g=event.pageX-70;
   let v=event.pageY-180;

   document.getElementById("golf").style.left=g+"px";
   document.getElementById("golf").style.top=v+"px";
   }
}

let i=1;
let a=2;
let h=0;
let s=0;

function sto()
{
a=2;
```

```
}

function plu()
{
a=1;

let w=screen.width;
let x=event.pageX;
let b=document.getElementById("ball").offsetLeft;

if(b+(w-1000)/2>x)
   {
   h=(b-x+(w-1000)/2)*2+b;
   }
else
   {
   h=b;
   }
}

function hit()
{
if((i==1)&&(a==1))
   {
   let d=document.getElementById("ball").offsetLeft;

   if(d<940)
      {
      document.getElementById("ball").style.left=h+"px";

      s++;
      document.getElementById("te").innerHTML="Количество ударов -
"+s;

      if((h>690)&&(h<780))
         {
         document.getElementById("ball").style.left="710px";
         document.getElementById("ball").style.bottom="10px";
         i=2;
         }
      }

      let c=document.getElementById("ball").offsetLeft;

      if(c>940)
      {
      document.getElementById("ball").style.left="940px";
      i=2;
      }
   }
}
</script>
</head>

<body>
<img id="golf" src="pict/golf.png" alt="Фото">

<div id="pic"><div id="te">Количество ударов - 0</div>
<img id="ball" src="pict/ball.png" alt="Фото">
<img id="hole" src="pict/hole.jpg" alt="Фото">
```

```
<img id="flag" src="pict/flag.jpg" alt="Фото">
<input type="button" value="Остановить" id="bu"></div>

</body>
</html>
```

3.32. Событие orientationchange. Контент в смартфоне

До сих пор мы создавали сценарии, рассчитанные преимущественно на персональные компьютеры. Настало время написать хотя бы один сценарий, предназначенный исключительно для мобильных устройств.

Код данного примера демонстрирует, что можно разрабатывать сайты, которые хорошо смотрятся не только на мониторах компьютеров, но и в различных гаджетах: планшетах и смартфонах. Эта простая программа позволяет адаптировать дизайн сайта под экраны с разным разрешением, а кроме того, менять контент при повороте смартфона или планшета из вертикального положения в горизонтальное и наоборот.

Главным действующим «лицом» у нас выступит событие **orientationchange**, которое, как вы помните, происходит при смене пространственной ориентации мобильного устройства. Напомню также, что в мобильной версии браузера Firefox такое событие не предусмотрено разработчиками и пример работать не будет. В мобильных версиях остальных web-обозревателей, которые упоминались во введении, наш пример работает корректно.

Что будет делать программа? Менять изображение и размер шрифта при повороте смартфона или планшета, тем самым меняя содержимое и настройки стилей документа. Зайдите на страницу **https://testjs.ru/p32.html** с мобильного устройства и посмотрите пример в действии. Если ваш смартфон расположен вертикально, вы увидите изображение моря, а размер шрифта будет 60px, то есть таким, чтобы хорошо читаться на узком экране. Если повернуть смартфон в горизонтальное положение, появится фотография сельскохозяйственных полей на горе, а размер шрифта станет 40px. Такой шрифт хорошо читается на широком экране. Кроме того, вы можете видеть, как меняется содержимое страницы, глядя на рисунки 3.32.1 и 3.32.2.

Приступим к написанию программы.

Первым делом добавим в код пока еще пустой страницы элементы разметки:

```
<span id="pas"></span><br><br>
<img src="pict/net.jpg" id="im" alt="Фото">
```

У нас есть текстовая область **span** (тоже пока пустая) и картинка, роль которой на начальном этапе загрузки выполняет файл-«заглушка» **net.jpg**. С подобной «заглушкой» вы уже встречались в более ранних примерах.

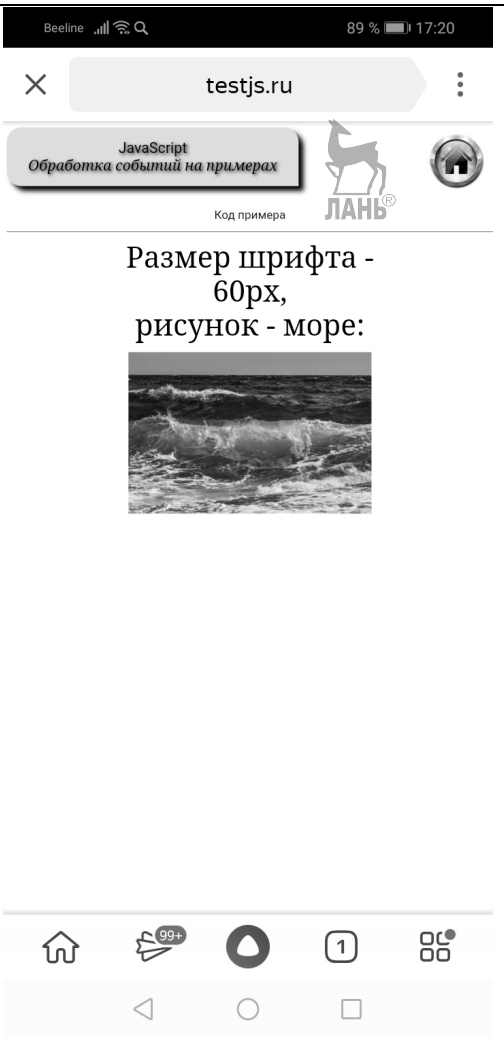

Рис. 3.32.1. Вертикальное положение смартфона

Сам сценарий начинается с регистрации одного обработчика, который запускается в случае двух событий – загрузки страницы или смены пространственной ориентации устройства:

```
window.addEventListener("load", orien);
window.addEventListener("orientationchange", orien);
```

В качестве обработчика используется функция **orien**. Написать ее довольно просто:

```
function orien()
{
```

```
let w=screen.width;
if(w<600)
   {
   document.getElementById("pas").style.fontSize="60px";
   document.getElementById("pas").innerHTML="Размер шрифта - 60px,
                                                <br>рисунок   -
море:";
   document.getElementById("im").src="pict/sea.jpg";
   }
else
   {
   document.getElementById("pas").style.fontSize="40px";
   document.getElementById("pas").innerHTML="Размер шрифта - 40px,
                                                <br>рисунок   -
суша:";
   document.getElementById("im").src="pict/earth.jpg";
   }
}
```

Рис. 3.32.2. Горизонтальное положение смартфона

Как видите, у нас есть блок условий **if** и блок условий **else**, внутри которых и выполняются все действия.

После загрузки страницы происходит событие **load**, которое запускает функцию **orien**. В этот момент она формирует первоначальный вид страницы на экране.

Сначала определяется размер экрана по ширине:

```
let w=screen.width;
```

Затем это значение проверяется на соответствие одному из условий. Мы выбрали пограничную ширину экрана 600px. Все параметры меньше соответствуют вертикальному положению гаджета, все параметры больше – горизонтальному.

Допустим, ваш смартфон находится в вертикальном положении и ширина его экрана в этот момент определяется как 360px. Тогда «сработает» первый блок. Размеру шрифта текстовой строки будет присвоено значение 60px:

```
document.getElementById("pas").style.fontSize="60px";
```

В самой строке появится сообщение «Размер шрифта – 60px, рисунок – море»:

```
document.getElementById("pas").innerHTML="Размер шрифта - 60px,
                                                    <br>рисунок   -
море:";
```

после чего на страницу выводится соответствующая фотография:

```
document.getElementById("im").src="pict/sea.jpg";
```

Если теперь повернуть смартфон в горизонтальное положение, то ширина экрана станет, допустим, 720px. «Сработает» условие **else**. Размер шрифта уменьшится до 40px, изменится текстовое сообщение и загрузится фотография суши:

```
document.getElementById("pas").style.fontSize="40px";
document.getElementById("pas").innerHTML="Размер шрифта - 40px,
                                                    <br>рисунок   -
суша:";
document.getElementById("im").src="pict/earth.jpg";
```

Как вы могли убедиться, необходимый результат был достигнут очень простым способом.

Собираем отрывки вместе и получаем полную страницу с нашим последним примером:

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="utf-8">
<title>Событие orientationchange</title>

<script>
window.addEventListener("load", orien);
window.addEventListener("orientationchange", orien);

function orien()
{
let w=screen.width;
if(w<600)
   {
   document.getElementById("pas").style.fontSize="60px";
   document.getElementById("pas").innerHTML="Размер шрифта - 60px,
<br>рисунок - море:";
   document.getElementById("im").src="pict/sea.jpg";
   }
else
```

```
   {
   document.getElementById("pas").style.fontSize="40px";
   document.getElementById("pas").innerHTML="Размер шрифта - 40px,
<br>рисунок - суша:";
   document.getElementById("im").src="pict/earth.jpg";
   }
}
</script>
</head>

<body>

<span id="pas"></span><br><br>
<img src="pict/net.jpg" id="im" alt="Фото">

</body>
</html>
```

4. Дополнительное программное обеспечение

4.1. Создание локального хостинга

Первая ошибка, которую допускают многие начинающие программисты, состоит в попытках создать локальный хостинг, следуя описаниям из Интернета, где предлагаются «навороченные» методы. Их авторы выделяют на диске С отдельные сектора, создают кучу дополнительных директорий, вносят большое количество изменений в конфигурационные файлы. В результате неопытный человек начинает путаться в этих сложных описаниях, ошибаться в настройках программного обеспечения и лишь путем многочисленных проб и консультаций с автором с энной попытки наконец получает требуемый результат (а иногда так и не получает, из-за чего начинает искать другие описания на данную тему).

Между тем создание хостинга на своем ПК – дело несложное. Если, конечно, применить решения, соизмеримые с простотой этой задачи.

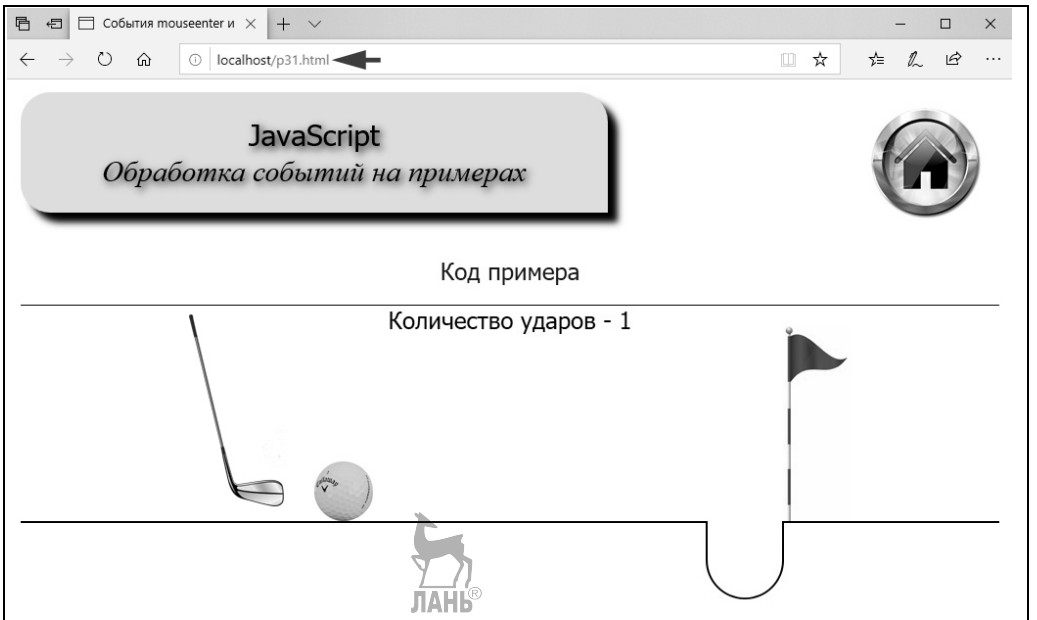

Рис. 4.1.1. Тестируем программы из книги на локальном хостинге

Если вы хотите тестировать только HTML-страницы с внедренными в них сценариями на JavaScript, то можно ограничиться установкой на свой компьютер http-сервера. Если планируете создавать страницы не просто загружаемые с сервера, а взаимодействующие с другими серверными скриптами (например, с программами обработки данных из форм), надо, кроме того, установить интерпретатор какого-либо языка, например PHP.

Выбор за вами. Со своей стороны рекомендую вам начать с сервера Apache и интерпретатора PHP. Описание технологии их установки – в следующих разделах.

Обратите внимание: все процедуры описаны для компьютера с операционной системой Windows 10.

Итак, приступим к созданию локального хостинга. А начнем мы с установки сервера Apache.

4.2. Установка сервера Apache

Первым делом выясните разрядность вашей операционной системы. Для этого щелкните на кнопке «Пуск», а затем на кнопке «Параметры». В открывшемся окне выберите пункт меню «Система». Откроется следующая вкладка, в которой необходимо кликнуть на строке «О системе». После этого на вкладке «Характеристики устройства» посмотрите строку «Тип системы».

Теперь скачайте на рабочий стол компьютера с сайта **https://www.apachelounge.com/download/** zip-архив, соответствующий разрядности вашей ОС. Для 64-разрядной системы – архив с пометкой **Win64**, для 32-разрядной – с пометкой **Win32**. Распакуйте архив. Скопируйте папку **Apache24** (только ее) непосредственно на диск **C**, чтобы ее адрес был **C:\Apache24**.

Рис. 4.2.1. Кнопка «Параметры» в меню

Откройте приложение «Командная строка» от имени администратора. Для этого нажмите кнопку «Пуск», в меню выберите «Служебные – Windows», найдите пункт «Командная строка» и щелкните на нем правой (правой!) кнопкой мыши. В выпадающем списке выберите «Дополнительно», а затем «Запуск от имени администратора». Откроется окно программы.

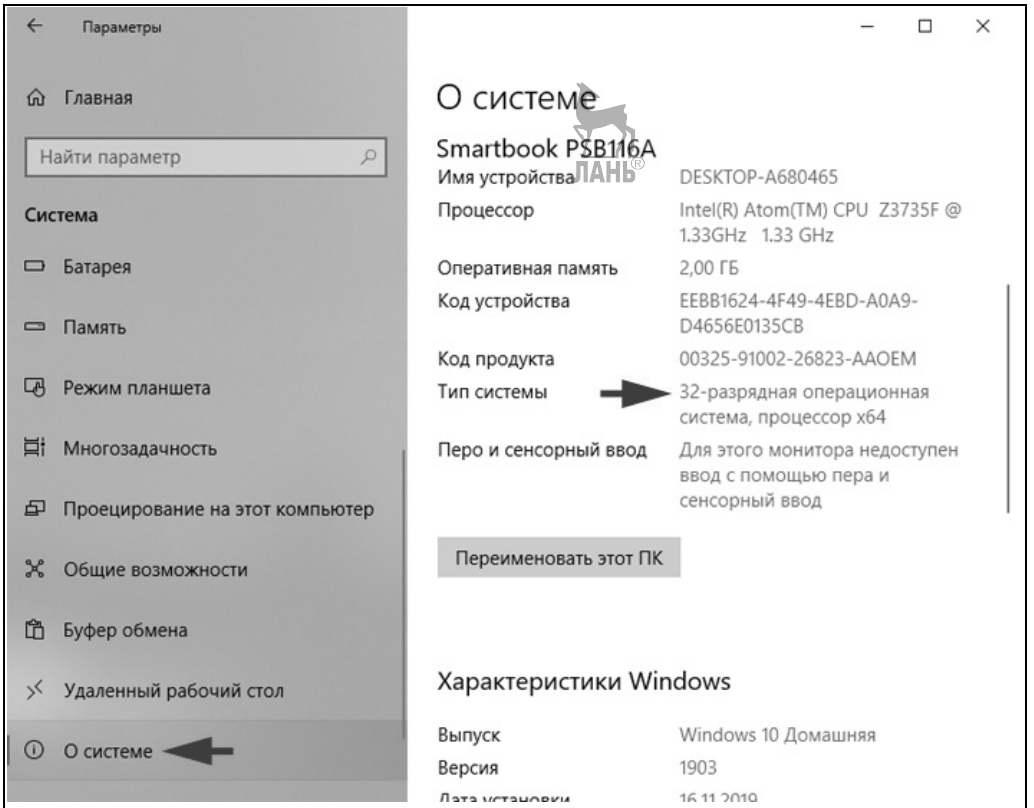

Рис. 4.2.2. Разрядность или тип операционной системы

Рис. 4.2.3. Сайт с zip-архивом сервера

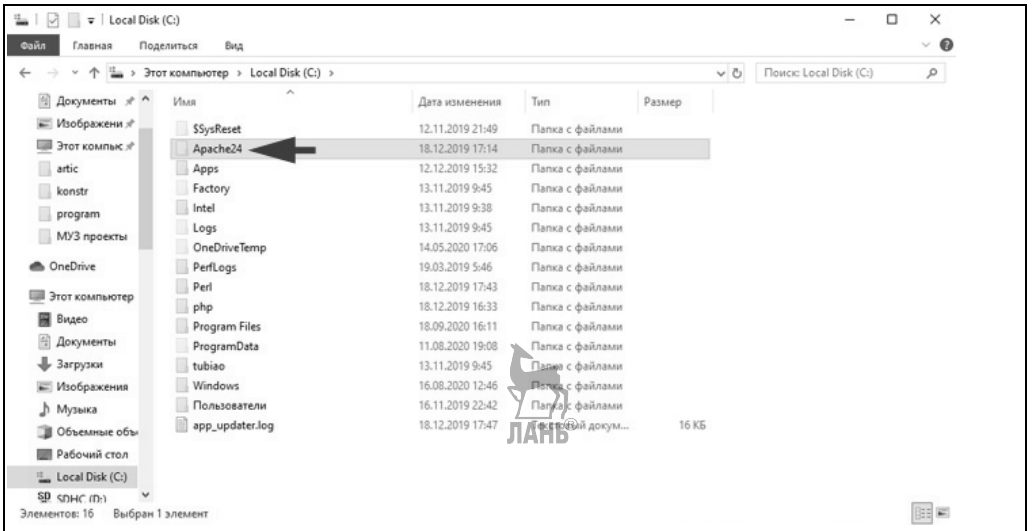

Рис. 4.2.4. Копируем папку Apache24 на диск C

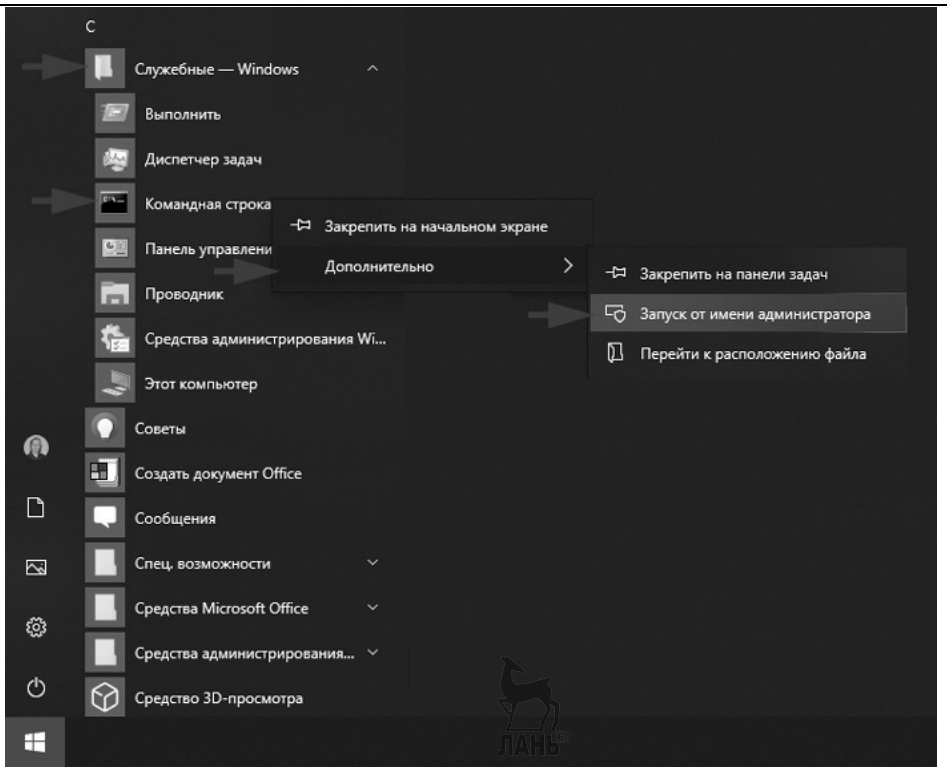

Рис. 4.2.5. Запуск командной строки

Теперь надо инсталлировать, а затем запустить сервер. Для этого в окне командной строки сразу после

```
C:\WINDOWS\System32>
```

наберите

```
C:\Apache24\bin\httpd.exe -k install
```

Должно получиться

```
C:\WINDOWS\System32>C:\Apache24\bin\httpd.exe -k install
```

Нажмите «Enter». Когда процесс инсталляции закончится, в окне программы вновь появится строка

```
C:\WINDOWS\System32>
```

После нее введите

```
C:\Apache24\bin\httpd.exe -k start
```

Должно получиться

```
C:\WINDOWS\System32>C:\Apache24\bin\httpd.exe -k start
```

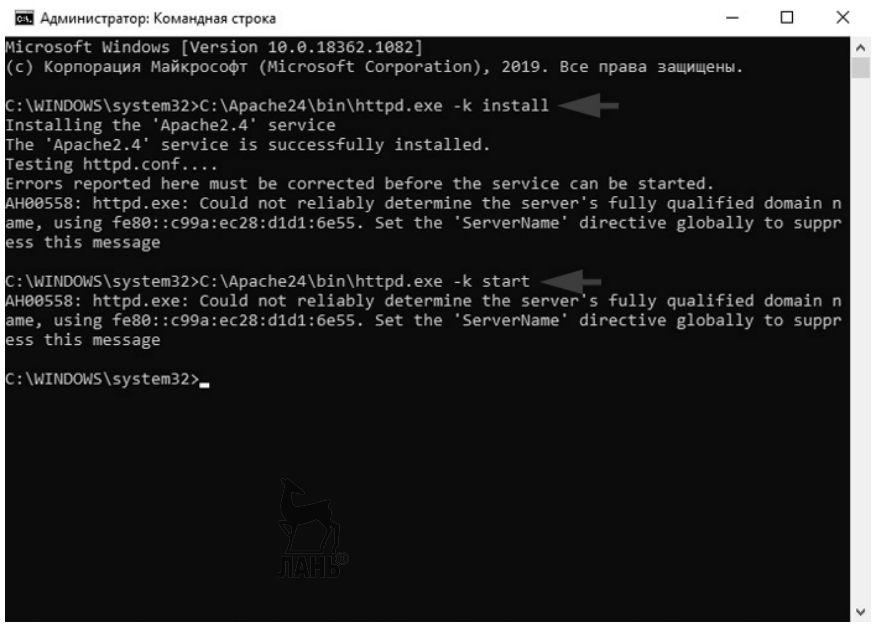

Рис. 4.2.6. Инсталляция и запуск сервера из приложения «Командная строка»

Снова нажмите «Enter». Дождитесь окончания процесса. **ВНИМАНИЕ! Откроется окно брандмауэра с запросом на разрешение работы «Apache».** Разрешите доступ во всех сетях.

Нам надо убедиться, что сервер заработал. Для этого откройте ваш браузер и введите в строке адреса **http://localhost/**. Если появилось сообщение «It works», значит все в порядке – сервер функционирует как положено.

Осталось добавить, что файлы ваших проектов необходимо помещать в папку **htdocs** по адресу **C:\Apache24\htdocs**, а просматривать готовые страницы сайтов по адресу **http://localhost/** или **http://localhost/имя_страницы.html**, например **http://localhost/primer.html** или **http://localhost/test/primer.html**.

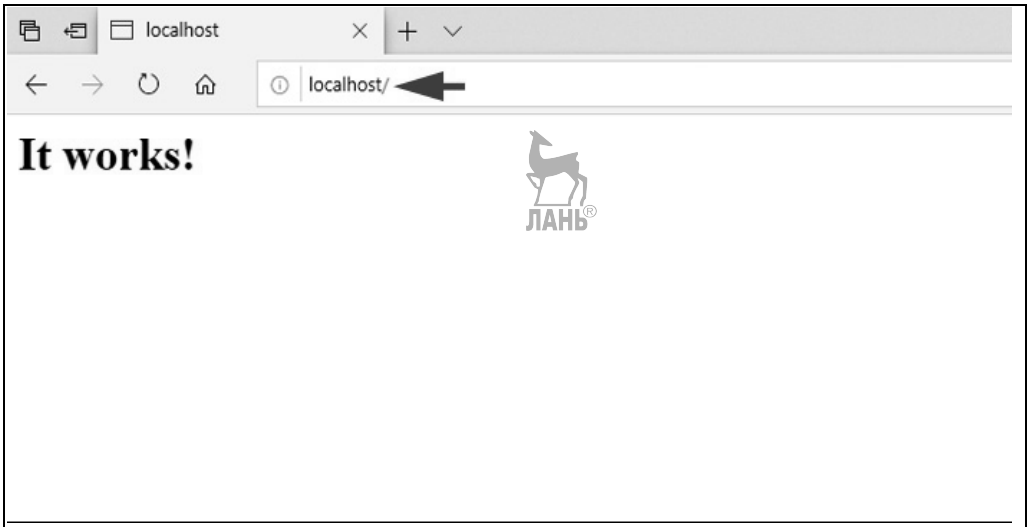

Рис. 4.2.7. Сервер подтверждает, что он работает

4.3. Установка PHP

Скачайте на рабочий стол компьютера с сайта **https://windows.php.net/download/** zip-архив, соответствующий разрядности вашей ОС. Для 64-разрядной системы – архив с пометкой **x64**, для 32-разрядной – с пометкой **x86**. Обратите внимание: так как мы будем устанавливать PHP в качестве модуля сервера Apache, скачивать надо дистрибутив **Thread Safe**.

Распакуйте архив. Переименуйте распакованную папку на **php** и переместите ее непосредственно на диск **C**, чтобы ее адрес был **C:\php**.

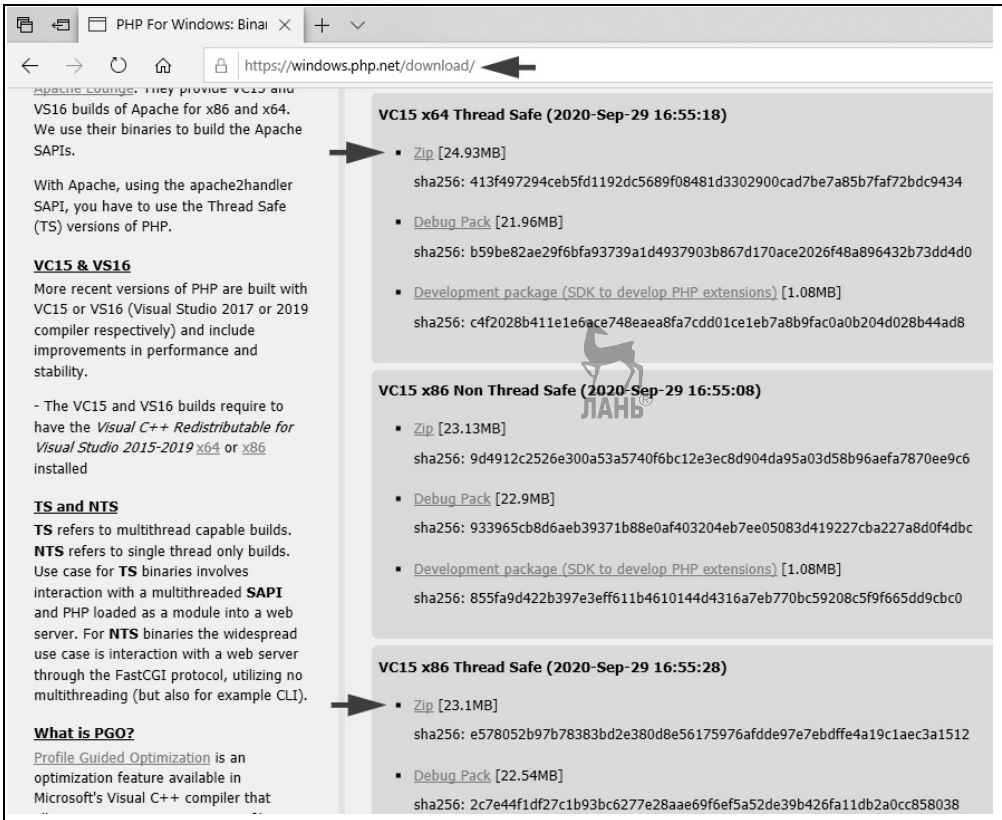

Рис. 4.3.1. Сайт с zip-архивом PHP

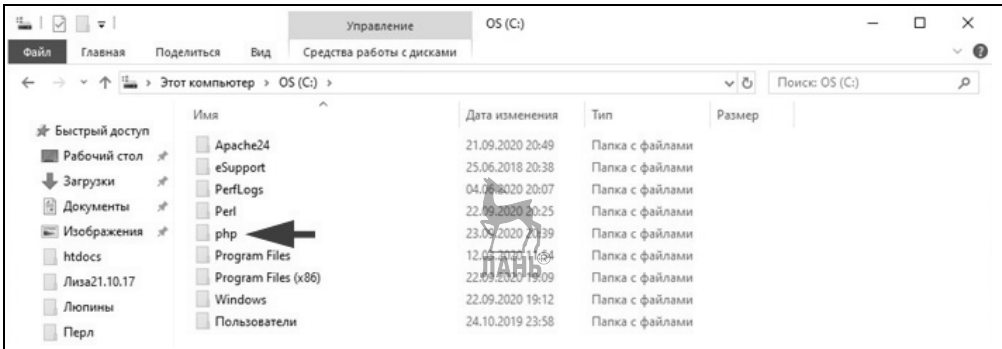

Рис. 4.3.2. Перемещаем папку php на диск C

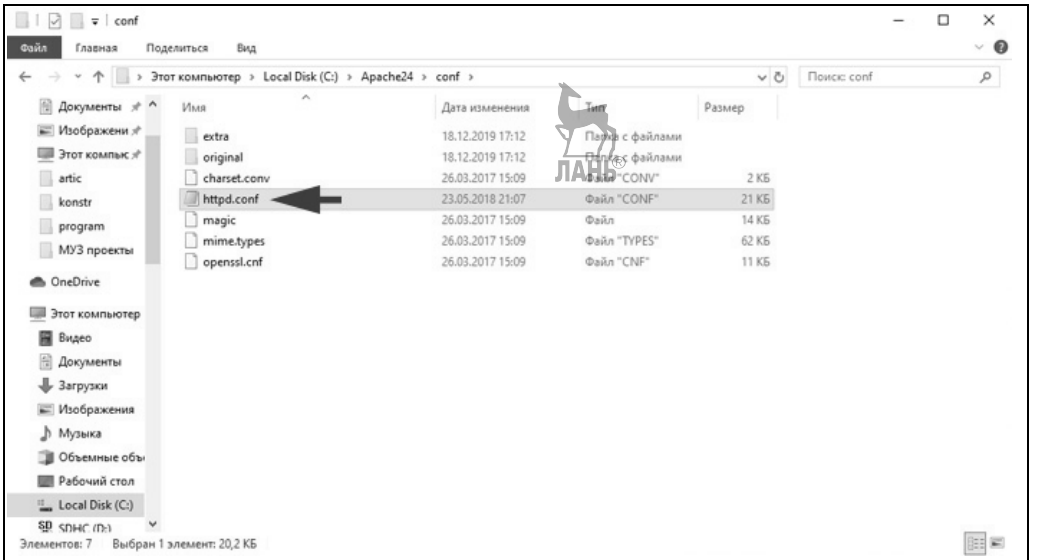

Рис. 4.3.3. Файл httpd.conf

В папке **C:\Apache24\conf** с помощью текстового редактора «Блокнот» откройте файл **httpd.conf** и найдите в нем строку (у меня она 285-ая)

```
DirectoryIndex index.html
```

и добавьте к ней

```
index.php
```

Должно получиться

```
DirectoryIndex index.html index.php
```

Теперь в самый конец файла **httpd.conf** добавьте 2 строки:

```
AddHandler application/x-httpd-php .php
LoadModule php7_module "C:/php/php7apache2_4.dll"
```

Перезагрузите компьютер.

Убедимся, что PHP заработал. Для этого в папке **C:\Apache24\htdocs** создайте текстовый файл и запишите в него следующий код:

```
<?php
echo "Good !";
```

Сохраните этот файл под именем, например, **p1.php**. Откройте ваш браузер и введите в строке адреса **http://localhost/p1.php**. Если появилось сообщение «Good !», значит все в порядке – PHP работает.

Рис. 4.3.4. Проверяем работу PHP

PHP-файлы ваших проектов необходимо помещать в папку **htdocs** по адресу **C:\Apache24\htdocs**, а просматривать готовые страницы сайтов по адресу **http://localhost/** (для файла **index.php**) или **http://localhost/имя_страницы.php**, например **http://localhost/primer.php** или **http://localhost/test/primer.php**.

4.4. Тестирование программ

Проверка сценариев должна начинаться еще на этапе их написания. Тщательно проверяйте набранный код. Типичные ошибки на этой стадии – пропущенные точки с запятой, кавычки, объявления переменных, банальные опечатки. Эти ошибки легко устранить при должной внимательности.

Рис. 4.4.1. Валидатор Javascript обнаружил «недостачу» точки с запятой

Отличный способ поиска ошибок – это тестирование сценариев в валидаторах, о которых говорилось во введении. Напомню о них:

– валидатор HTML5 – **https://validator.w3.org/**;

– валидатор CSS3 – **http://jigsaw.w3.org/css-validator/**;

– валидатор JavaScript – **http://beautifytools.com/javascript-validator.php**.

Особенность этих валидаторов в том, что они находят все ошибки в разметке, в таблицах стилей, в коде программ, в том числе пропущенные символы, опечатки (но не в тексте), необъявленные переменные и т. д.

А иногда бывает другая ситуация. Страница проверена во всех валидаторах, ошибок нет или они устранены, а сценарий ведет себя совсем не так, как вы того ожидаете. Чаще всего это означает, что при составлении программы была неправильно продумана логика ее выполнения. Как обнаружить место, где «смысл» программы оказался искажен?

Могу предложить два похожих способа.

Способ первый. В те блоки программы, в которых логика работы нарушается, поочередно добавляйте в разные точки вызов метода alert(). Например, у вас есть «сомнительный» фрагмент:

```
...
let c=a+b;
let t=v/d;
if(c<t/2)
   {
   n++;
   document.getElementById("pict").style.left=n+"px";
   }
else
   {
   n=0;
   c=document.getElementById("img").offsetLeft;
   }
...
```

Вставьте в разные точки этого блока вызов alert(), например, так:

```
...
let c=a+b;

alert(c);

let t=v/d;
...
```

Потом так:

```
...
let t=v/d;

alert(t);

if(c<t/2)
...
```

А потом вот так:

```
...
if(c<t/2)
   {
   n++;

alert(n);

   document.getElementById("pict").style.left=n+"px";
   }
...
```

Или сразу во все три точки. Таким образом, вы последовательно увидите, как меняются ваши переменные, и обнаружите, где происходит отклонение от заданной линии поведения скрипта.

Второй способ предпочтительнее в ситуации с быстро меняющимися параметрами, например при перемещении указателя мыши по странице. Чтобы отследить поток значений, внедрите в разметку страницы элемент визуального отображения этих значений. Например, так:

```
<div id="test"><div>
```

Найдите фрагмент, вызывающий сомнения, например:

```
...
function coor()
{
let h=event.pageX;
let v=event.pageY;

let sh=document.getElementById("bat").offsetLeft;
let sv=document.getElementById("bat").offsetTop;

dh=h-sh;
dv=v+sv;

window.onmousemove=neco;
}
...
```

Добавьте в код сценария команду выведения данных:

```
...
dh=h-sh;
dv=v+sv;

document.getElementById("test").innerHTML=dh+" "+dv;

window.onmousemove=neco;
}
...
```

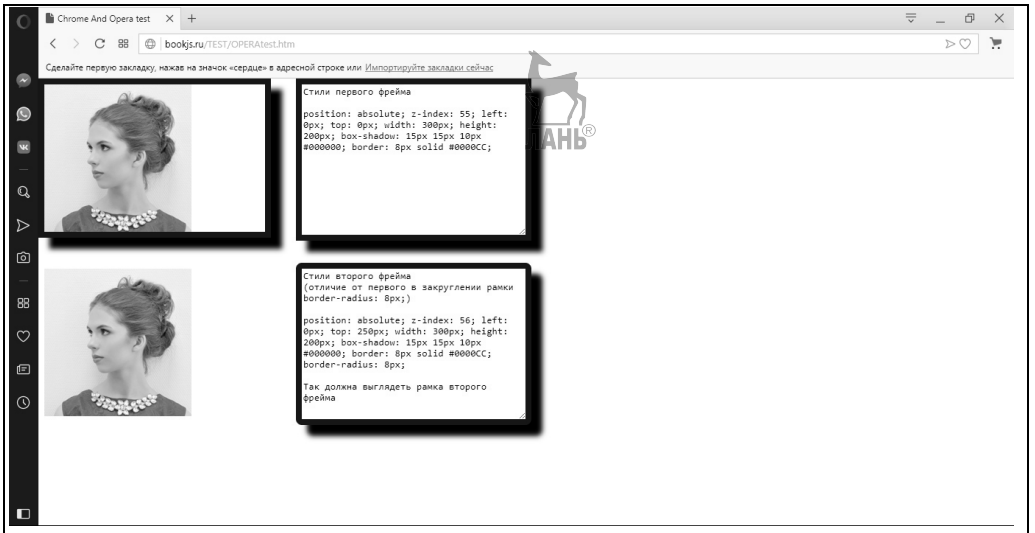

Рис. 4.4.2. В 2018 году браузеры Google Chrome и Opera при определенных настройках стилей некорректно отображали фреймы

Анализируя показания координат, вы сразу обнаружите, что переменная **dv** растет вместо того, чтобы оставаться постоянной. А значит в выражении **dv=v+sv;** знак плюс нужно поменять на минус.

Обязательно проверяйте свои сценарии в разных браузерах. Хотя они более-менее одинаково обрабатывают разметку, стили и JavaScript-код, все-таки некоторые различия в них есть. Да и ошибки бывают. Например, есть свои особенности в Firefox, о чем я уже рассказывал в п. 1.4. В 2018 году, тестируя группу программ, я обнаружил абсолютно одинаковые ошибки в Google Chrome и Opera, связанные с обработкой таблиц стилей. В Chrome эти ошибки были исправлены в течении месяца, а вот в Opera они оставались еще долго.

На сегодняшний день наиболее популярны следующие браузеры: Google Chrome, Mozilla Firefox, Microsoft Edge, Opera и Яндекс.Браузер. На мой взгляд, это минимальный набор web-обозревателей, в которых нужно тестировать свои программы. Желательно добиваться того, чтобы сценарии в любом браузере давали одинаковый результат.

5. Заключение

Повествование о событиях в web-браузерах подошло к концу. Мы с вами рассмотрели 22 разных события, познакомились с практическим применением обработчиков, написав 32 сценария. Однако главная тема книги этим не исчерпывается. Во-первых, событий, для которых можно создать сценарии на JavaScript, заметно больше. Мы экспериментировали только с самыми распространенными. Во-вторых, вариантов создания обработчиков поистине бесконечное множество. И это хорошо. Ибо означает, что вам есть, что еще изучать, а поле для творчества – бескрайне.

Мы также поговорили о способах тестирования и настройки программ. За рамками рассмотрения книги остались приемы отладки сценариев при помощи инструментов, встроенных во многие web-обозреватели. Тема эта достаточно обширная, и, если читатель захочет изучить такие инструменты, он легко найдет недостающую информацию в Интернете.

Как видите, вы можете освоить еще много интересных вещей. Мне остается пожелать вам удачи.

Валерий Викторович ЯНЦЕВ
JAVASCRIPT
ОБРАБОТКА СОБЫТИЙ НА ПРИМЕРАХ
Учебное пособие

Зав. редакцией
литературы по информационным технологиям
и системам связи *О. Е. Гайнутдинова*
Ответственный редактор *Н. А. Кривилёва*
Корректор *О. В. Федорова*
Выпускающий *В. А. Иутин*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com
196105, Санкт-Петербург, пр. Юрия Гагарина, д.1, лит. А.
Тел.: (812) 336-25-09, 412-92-72.
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 17.05.21.
Бумага офсетная. Гарнитура Школьная. Формат 70×100 $^{1}/_{16}$.
Печать офсетная. Усл. п. л. 14,30. Тираж 50 экз.

Заказ № 525-21.

Отпечатано в полном соответствии
с качеством предоставленного оригинал-макета
в АО «Т8 Издательские Технологии»
109316, г. Москва, Волгоградский пр., д. 42, к. 5.