Главное управление образования Гомельского облисполкома
Учреждение образования
«Гомельский государственный машиностроительный колледж»

Учебная дисциплина
«Веб-программирование для мобильных устройств»

Инструкция
по выполнению лабораторной работы №18
«Проверка корректности информации, введенной пользователем в элементы формы. Обработка данных формы средствами JavaScript»

Гомель
2021

Составитель: _____ Гаврилова В.М.

Обсуждено и одобрено на заседании цикловой комиссии «Программируемые мобильные системы»

Протокол от «27» августа 2021 № 1

_____ В.М. Гаврилова

Лабораторная работа №18

Тема работы: «Проверка корректности информации, введенной пользователем в элементы формы. Обработка данных формы средствами JavaScript»

1. Цель работы

Научиться выполнять проверку корректности информации, введенной пользователем в элементы формы, организовывать обработку данных форм средствами JavaScript.

2. Задание

Создать форму и скрипт для обработки действий, происходящих на форме, в соответствии с вариантом по списку в журнале.

Варианты:

1. Создать форму, для вычисления площади треугольника по трем сторонам. Площадь вычисляется в результате щелчка по кнопке с параметрами type=”button”.
2. Создать форму, для вычисления площади параллелограмма по двум сторонам и углу между ними. Площадь вычисляется в результате потери фокуса у последнего текстового поля формы.
3. Создать форму, для вычисления объема пирамиды по площади основания и высоте. Площадь вычисляется в результате изменения значения последнего текстового поля формы.
4. Создать форму, для вычисления длины гипотенузы по катету и противолежащему углу. Длина вычисляется в результате щелчка по кнопке с параметрами type= “submit”.
5. Создать форму, для вычисления расстояния между двумя точками. Длина вычисляется в результате выделения текста в первом текстовом поле.
6. Создать форму, для вычисления площади прямоугольного треугольника. Площадь вычисляется в результате потери фокуса у последнего текстового поля.
7. Создать форму, для вычисления площади трапеции. Площадь вычисляется в результате изменения данных в последнем текстовом поле.
8. Создать форму, для вычисления площади прямоугольника. Площадь вычисляется в результате щелчка по последнему текстовому полю.
9. Создать форму, для вычисления площади круга. Площадь вычисляется в результате щелчка по ссылке.
10. Создать форму, для вычисления длины окружности. Длина вычисляется в результате выделения данных текстового поля.
11. Создать форму, для вычисления периметра квадрата. Периметр вычисляется в результате изменения данных в последнем текстовом поле.
12. Создать форму, для вычисления периметра треугольника. Периметр вычисляется в результате наведения мышкой на гиперссылку.
13. Создать форму, для вычисления объема цилиндра. Объем вычисляется в результате отведения мышки от гиперссылки.
14. Создать форму, для вычисления объема конуса. Объем вычисляется в результате наведения мышки на гиперссылку.
15. Создать форму, для вычисления объема цилиндра. Объем вычисляется в результате щелчка мышкой по кнопке.

16. Создать форму, для вычисления объема куба. Объем вычисляется при получении фокуса 1-м текстовым полем.

17. Создать форму, для вычисления площади боковой поверхности пирамиды. Площадь вычисляется при изменении значения текстового поля.

18. Создать форму, для вычисления площади боковой поверхности призмы. Площадь вычисляется в результате выделения в текстовом поле.

19. Создать форму, для вычисления площади n-го члена арифметической прогрессии. Вычисление происходит в результате наведения мыши на гиперссылку текстовом поле.

20. Создать форму, для вычисления среднего балла за сессию. Вычисление происходит в результате щелчка мышью на гиперссылку текстовом поле.

21. Создать форму, для вычисления среднего арифметического трех введенных чисел. Вычисление происходит в результате изменения значения третьего числа.

22. Создать форму, ввести названия и стоимость трех купленных в последнее время книг. Вычислить стоимость трех книг при нажатии на кнопку.

23. Создать форму, ввести расстояние от вашего дома до колледжа и время, затрачиваемое вами на преодоление этого пути. Вычислить скорость вашего движения после ввода времени.

24. Создать форму, ввести значение температуры в помещении по Цельсию. Определите значение температуры по Фаренгейту при потере фокуса.

3. Оснащение работы

Технические средства обучения:

- IBM – совместимый компьютер;

Электронные средства обучения:

- веб-браузер (Opera, Google Chrome, Mozilla Firefox и др.);
- html-редакторы (Notepad++, Sublime Text и др.).

4. Основные теоретические сведения

Элементы управления, такие как <form>, <input> и другие имеют большое количество своих важных свойств и ссылок.

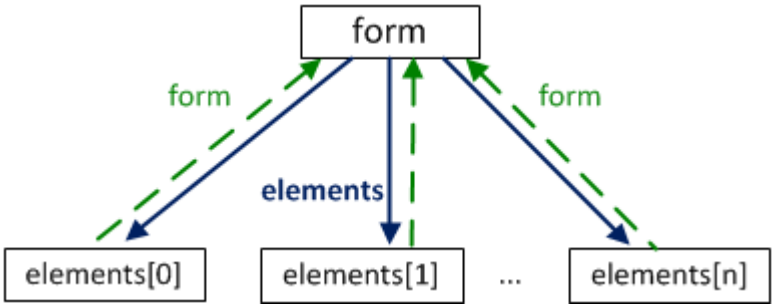
Псевдомассив form.elements

Элементы FORM можно получить по имени или номеру, используя свойство document.forms[name/index].

Например:

```
document.forms.my -- форма с именем 'my'
document.forms[0] -- первая форма в документе
```

Любой *элемент* формы form можно получить аналогичным образом, используя свойство form.elements.



Например:

```
<body>
  <form name="my">
    <input name="one" value="1">
    <input name="two" value="2">
  </form>
  <script>
    var form = document.forms.my; // можно document.forms[0]
    var elem = form.elements.one; // можно form.elements[0]
    alert( elem.value ); // 1
  </script>
</body>
```

Может быть несколько элементов с *одинаковым именем*. В таком случае form.elements[name] вернет коллекцию элементов, например:

```
<body>
<form>
  <input type="radio" name="age" value="10">
  <input type="radio" name="age" value="20">
</form>

<script>
var form = document.forms[0];
var elems = form.elements.age;
alert(elems[0].value); // 10, первый input
</script>
</body>
```

Эти ссылки не зависят от окружающих тегов. Элемент может быть «зарыт» где-то глубоко в форме, но он всё равно доступен через form.elements.

Свойство elements также есть у элементов <fieldset>. Вот пример:

```
<body>
  <form>
    <fieldset name="set">
      <legend>fieldset</legend>
      <input name="text" type="text">
    </fieldset>
  </form>

  <script>
    var form = document.forms[0];
    alert( form.elements.text ); // INPUT
    alert( form.elements.set.elements.text ); // INPUT
  </script>
</body>
```

Спецификация: HTML5 Forms.

Доступ form.name тоже работает, но с особенностями

Получить доступ к элементам формы можно не только через form.elements[name/index], но и проще: form[index/name].

Этот способ короче, но обладает одной неприятной особенностью: если к элементу обратиться по его name, а потом свойство name изменить, то он по-прежнему будет доступен под старым именем.

Звучит странно, поэтому посмотрим на примере.

```
<form name="myform">
  <input name="text">
</form>

<script>
```

```
  var form = document.forms.myform;
  alert( form.elements.text == form.text ); // true, это тот самый INPUT
  form.text.name = "new-name"; // меняем name ему
  // нет больше элемента с таким именем
  alert( form.elements.text ); // undefined
  alert( form.text ); //  INPUT (а должно быть undefined!)
</script>
```

Ссылка на форму element.form

По элементу можно получить его форму, используя свойство element.form.

Пример:

```
<body>
<form>
  <input type="text" name="surname">
</form>

<script>
var form = document.forms[0];
var elem = form.elements.surname;
alert(elem.form == form); // true
</script>
</body>
```

Элемент label

Элемент label – один из самых важных в формах.

Клик на label засчитывается как фокусировка или клик на элементе формы, к которому он относится.

Это позволяет посетителям кликать на большой красивой метке, а не на маленьком квадратике input type=checkbox (radio). Конечно, это очень удобно.

Есть два способа показать, какой элемент относится к label:

1. Дать метке атрибут for, равный id соответствующего input:

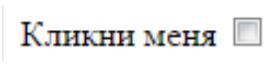
```
<table>
  <tr>
    <td>
      <label for="agree">Согласен с правилами</label>
    </td>
    <td>
      <input id="agree" type="checkbox">
    </td>
  </tr>
  <tr>
    <td>
      <label for="not-a-robot">Я не робот</label>
    </td>
    <td>
      <input id="not-a-robot" type="checkbox">
    </td>
  </tr>
</table>
```

Согласен с правилами ☐

Я не робот ☐

2. Завернуть элемент в label. В этом случае можно обойтись без дополнительных атрибутов:

```
<label>Кликни меня <input type="checkbox"></label>
```



Элементы input и textarea

Для большинства типов input значение ставится/читается через свойство value.

```
input.value = "Новое значение";
textarea.value = "Новый текст";
```

Не используйте textarea.innerHTML

Для элементов textarea также доступно свойство innerHTML, но лучше им не пользоваться: оно хранит только HTML, изначально присутствовавший в элементе, и не меняется при изменении значения.

Исключения – input type="checkbox" и input type="radio"

Текущее «отмеченное» состояние для checkbox и radio находится в свойстве checked (true/false).

```
if (input.checked) {
  alert( "Чекбокс выбран" );
}
```

Элементы select и option

Селект в JavaScript можно установить двумя путями: поставив значение select.value, либо установив свойство select.selectedIndex в номер нужной опции:

```
select.selectedIndex = 0; // первая опция
```

Установка selectedIndex = -1 очистит выбор.

Список элементов-опций доступен через select.options.

Если select допускает множественный выбор (атрибут multiple), то значения можно получить/установить, сделав цикл по select.options. При этом выбранные опции будут иметь свойство option.selected = true.

Пример:

```
<form name="form">
  <select name="genre" multiple>
    <option value="blues" selected>Мягкий блюз</option>
    <option value="rock" selected>Жёсткий рок</option>
    <option value="classic">Классика</option>
  </select>
</form>

<script>
var form = document.forms[0];
var select = form.elements.genre;
for (var i = 0; i < select.options.length; i++) {
  var option = select.options[i];
  if(option.selected) {
    alert( option.value );
  }
}
</script>
```

new Option

В стандарте the option element есть любопытный короткий синтаксис для создания элемента с тегом option:

```
option = new Option(text, value, defaultSelected, selected);
```

Параметры:

- text – содержимое,
- value – значение,
- defaultSelected и selected поставьте в true, чтобы сделать элемент выбранным.

Его можно использовать вместо document.createElement('option'), например:

```
var option = new Option("Текст", "value");
// создаст <option value="value">Текст</option>
```
Такой же элемент, но выбранный:
```
var option = new Option("Текст", "value", true, true);
```
Дополнительные свойства option

У элементов option также есть особые свойства, которые могут оказаться полезными:

selected - выбрана ли опция

index - номер опции в списке селекта

text - Текстовое содержимое опции (то, что видит посетитель).

Итого

Свойства для навигации по формам:

document.forms

Форму можно получить как document.forms[name/index].

form.elements

Элементы в форме: form.elements[name/index]. Каждый элемент имеет ссылку на форму в свойстве form. Свойство elements также есть у <fieldset>.

Значение элементов читается/ставится через value или checked.

Для элемента select можно задать опцию по номеру через select.selectedIndex и перебрать опции через select.options. При этом выбранные опции (в том числе при мультиселекте) будут иметь свойство option.selected = true.

Фокусировка: focus/blur

Говорят, что элемент «получает фокус», когда посетитель фокусируется на нём. Обычно фокусировка автоматически происходит при нажатии на элементе мышкой, но также можно перейти на нужный элемент клавиатурой – через клавишу `Tab`, нажатие пальцем на планшете и так далее.

Момент получения фокуса и потери очень важен.

При получении фокуса мы можем подгрузить данные для автодополнения, начать отслеживать изменения. При потере – проверить данные, которые ввёл посетитель.

Кроме того, иногда полезно «вручную», из JavaScript перевести фокус на нужный элемент, например, на поле в динамически созданной форме.

События focus/blur

Событие `focus` вызывается тогда, когда пользователь фокусируется на элементе, а `blur` – когда фокус исчезает, например посетитель кликает на другом месте экрана.

Давайте сразу посмотрим на них в деле, используем для проверки («валидации») введённых в форму значений.

В примере ниже:

- Обработчик `onblur` проверяет, что в поле введено число, если нет – показывает ошибку.
- Обработчик `onfocus`, если текущее состояние поля ввода – «ошибка» – скрывает её (потом при `onblur` будет повторная проверка).

В примере ниже, если набрать что-нибудь в поле «возраст» и завершить ввод, нажав `Tab` или кликнув в другое место страницы, то введённое значение будет автоматически проверено:

```
<style> .error { border-color: red; } </style>
Введите ваш возраст: <input type="text" id="input">
<div id="error"></div>

<script>
input.onblur = function() {
  if (isNaN(this.value)) { // введено не число
    // показать ошибку
    this.className = "error";
```

```
    error.innerHTML = 'Вы ввели не число. Исправьте, пожалуйста.'
  }
};
input.onfocus = function() {
  if (this.className == 'error') { // сбросить состояние "ошибка", если оно есть
    this.className = "";
    error.innerHTML = "";
  }
};
</script>
```

Введите ваш возраст:

Методы focus/blur

Методы с теми же названиями переводят/уводят фокус с элемента.

Для примера модифицируем пример выше, чтобы при неверном вводе посетитель просто не мог уйти с элемента:

```
<style>
  .error {     background: red;   }
</style>

<div>Возраст:
  <input type="text" id="age">
</div>
<div>Имя:
  <input type="text">
</div>

<script>
  age.onblur = function() {
    if (isNaN(this.value)) { // введено не число
      // показать ошибку
      this.classList.add("error");
      //... и вернуть фокус обратно
      age.focus();
    } else {
      this.classList.remove("error");
    }
  };
</script>
```

Возраст:

Имя:

Этот пример работает во всех браузерах, кроме Firefox (ошибка).

Если ввести что-то нецифровое в поле «возраст», и потом попытаться табом или мышкой перейти на другой `<input>`, то обработчик `onblur` вернёт фокус обратно.

Обратим внимание – если из `onblur` сделать `event.preventDefault()`, то такого же эффекта не будет, потому что `onblur` срабатывает уже *после* того, как элемент потерял фокус.

HTML5 и CSS3 вместо focus/blur

Прежде чем переходить к более сложным примерам, использующим JavaScript, мы рассмотрим три примера, когда его использовать не надо, а достаточно современного HTML/CSS.

Подсветка при фокусировке

Стилизация полей ввода может быть решена средствами CSS (CSS2.1), а именно – селектором `:focus`:

```
<style>
input:focus {
  background: #FA6;
```

```
  outline: none;  /* убрать рамку */
}
</style>
<input type="text">

<p>Селектор :focus выделит элемент при фокусировке на нем и уберёт рамку, которой браузер выделяет этот элемент по умолчанию.</p>
```

Селектор :focus выделит элемент при фокусировке на нем и уберёт рамку, которой браузер выделяет этот элемент по умолчанию.

Автофокус

При загрузке страницы, если на ней существует элемент с атрибутом `autofocus` – браузер автоматически фокусируется на этом элементе. Работает во всех браузерах, кроме IE9-.

```
<input type="text" name="search" autofocus>
```

Если нужны старые IE, то же самое может сделать JavaScript:

```
<input type="text" name="search">
<script>
  document.getElementsByName('search')[0].focus();
</script>
```

Как правило, этот атрибут используется при изначальной загрузке, для страниц поиска и так далее, где главный элемент очевиден.

Плейсхолдер

Плейсхолдер – это значение-подсказка внутри `INPUT`, которое автоматически исчезает при фокусировке и существует, пока посетитель не начал вводить текст.

Во всех браузерах, кроме IE9-, это реализуется специальным атрибутом `placeholder`:

```
<input type="text" placeholder="E-mail">
```

E-mail

В некоторых браузерах этот текст можно стилизовать:

```
<style>
.my::-webkit-input-placeholder {
  color: red;
  font-style: italic;
}
.my::-moz-input-placeholder {
  color: red;
  font-style: italic;
}
.my::-ms-input-placeholder {
  color: red;
  font-style: italic;
}
</style>

<input class="my" type="text" placeholder="E-mail">
Стилизованный плейсхолдер
```

E-mail Стилизованный плейсхолдер

Разрешаем фокус на любом элементе: tabindex

По умолчанию не все элементы поддерживают фокусировку.

Перечень элементов немного рознится от браузера к браузеру, например, список для IE описан в MSDN, одно лишь верно всегда – заведомо поддерживают `focus/blur` те элементы, с которыми посетитель может взаимодействовать: `<button>`, `<input>`, `<select>`, `<a>` и т.д.

С другой стороны, на элементах для форматирования, таких как , , <table> – по умолчанию сфокусироваться нельзя. Впрочем, существует способ включить фокусировку и для них.

В HTML есть атрибут tabindex.

Его основной смысл – это указать номер элемента при переборе клавишей Tab.

То есть, если есть два элемента, первый имеет tabindex="1", а второй tabindex="2", то нажатие Tab при фокусе на первом элементе – переведёт его на второй.

Исключением являются специальные значения:

- tabindex="0" делает элемент всегда последним.
- tabindex="-1" означает, что клавиша Tab будет элемент игнорировать.

Любой элемент поддерживает фокусировку, если у него есть tabindex.

В примере ниже есть список элементов. Кликните на любой из них и нажмите «tab».

Кликните на первый элемент списка и нажмите Tab. Внимание! Дальнейшие нажатия Tab могут вывести за границы iframe'а с примером.

```
<ul>
  <li tabindex="1">Один</li>
  <li tabindex="0">Ноль</li>
  <li tabindex="2">Два</li>
  <li tabindex="-1">Минус один</li>
</ul>

<style>
  li { cursor: pointer; }
  :focus { outline: 1px dashed green; }
</style>
```

Кликните на первый элемент списка и нажмите Tab. Внимание! Дальнейшие нажатия Tab могут вывести за границы iframe'а с примером.

- Один
- Ноль
- Два
- Минус один

Порядок перемещения по клавише «Tab» в примере выше должен быть таким: 1 - 2 - 0 (ноль всегда последний). Продвинутые пользователи частенько используют «Tab» для навигации, и ваше хорошее отношение к ним будет вознаграждено :)

Обычно <li> не поддерживает фокусировку, но здесь есть tabindex.

Делегирование с focus/blur

События focus и blur не всплывают.

Это грустно, поскольку мы не можем использовать делегирование с ними. Например, мы не можем сделать так, чтобы при фокусировке в форме она вся подсвечивалась:

```
<!-- при фокусировке на форме ставим ей класс -->
<form onfocus="this.className='focused'">
  <input type="text" name="name" value="Ваше имя">
  <input type="text" name="surname" value="Ваша фамилия">
</form>

<style> .focused { outline: 1px solid red; } </style>
```

Пример выше не работает, т.к. при фокусировке на любом <input> событие focus срабатывает только на этом элементе и не всплывает наверх. Так что обработчик onfocus на форме никогда не сработает.

Что делать? Неужели мы должны присваивать обработчик каждому полю <input>?

Это забавно, но хотя focus/blur не всплывают, они могут быть пойманы на фазе перехвата.

Вот так сработает:

```
<form id="form">
  <input type="text" name="name" value="Ваше имя">
  <input type="text" name="surname" value="Ваша фамилия">
```

```
</form>

<style>
  .focused {
    outline: 1px solid red;
  }
</style>

<script>
  // ставим обработчики на фазе перехвата, последний аргумент true
  form.addEventListener("focus", function() {
    this.classList.add('focused');
  }, true);

  form.addEventListener("blur", function() {
    this.classList.remove('focused');
  }, true);
</script>
```

События focusin/focusout

События `focusin/focusout` – то же самое, что и `focus/blur`, только они всплывают.

У них две особенности:

- Не поддерживаются Firefox (хотя поддерживаются даже старейшими IE).
- Должны быть назначены не через `on`-свойство, а при помощи `elem.addEventListener`.

Из-за отсутствия подержки Firefox эти события используют редко. Получается, что во всех браузерах можно использовать `focus` на стадии перехвата, ну а `focusin/focusout` – в IE8-, где стадии перехвата нет.

Подсветка формы в примере ниже работает во всех браузерах.

```
<form name="form">
  <input type="text" name="name" value="Ваше имя">
  <input type="text" name="surname" value="Ваша фамилия">
</form>
<style>
  .focused {
    outline: 1px solid red;
  }
</style>

<script>
  function onFormFocus() {
    this.className = 'focused';
  }
  function onFormBlur() {
    this.className = '';
  }
  var form = document.forms.form;
  if (form.addEventListener) {
    // focus/blur на стадии перехвата срабатывают во всех браузерах
    // поэтому используем их
    form.addEventListener('focus', onFormFocus, true);
    form.addEventListener('blur', onFormBlur, true);
  } else {
    // ветка для IE8-, где нет стадии перехвата, но есть focusin/focusout
    form.onfocusin = onFormFocus;
    form.onfocusout = onFormBlur;
  }
</script>
```

Итого

События `focus/blur` происходят при получении и снятия фокуса с элемента.

У них есть особенности:

- Они не всплывают. Но на фазе перехвата их можно перехватить. Это странно, но это так, не спрашивайте почему.

Везде, кроме Firefox, поддерживаются всплывающие альтернативы `focusin/focusout`.

- По умолчанию многие элементы не могут получить фокус. Например, если вы кликните по `DIV`, то фокусировка на нем не произойдет.

Но это можно изменить, если поставить элементу атрибут `tabIndex`. Этот атрибут также дает возможность контролировать порядок перехода при нажатии `Tab`.

Текущий элемент: `document.activeElement`

Кстати, текущий элемент, на котором фокус, доступен как `document.activeElement`.

Изменение: change, input, cut, copy, paste

На элементах формы происходят события клавиатуры и мыши, но есть и несколько других, особенных событий.

Событие change

Событие change происходит по окончании изменении значения элемента формы, когда это изменение зафиксировано.

Для текстовых элементов это означает, что событие произойдёт не при каждом вводе, а при потере фокуса.

Например, пока вы набираете что-то в текстовом поле ниже – события нет. Но как только вы уведёте фокус на другой элемент, например, нажмёте кнопку – произойдет событие `onchange`.

```
<input type="text" onchange="alert(this.value)">
<input type="button" value="Кнопка">
```

Для остальных же элементов: `select`, `input type=checkbox/radio` оно срабатывает сразу при выборе значения.

Поздний `onchange` в IE8-

В IE8- `checkbox/radio` при изменении мышью не инициируют событие сразу, а ждут потери фокуса.

Для того, чтобы видеть изменения `checkbox/radio` тут же – в IE8- нужно повесить обработчик на событие `click` (оно произойдет и при изменении значения с клавиатуры) или воспользоваться событием `propertychange`, описанным далее.

Событие input

Событие `input` срабатывает *тут же* при изменении значения текстового элемента и поддерживается всеми браузерами, кроме IE8-.

В IE9 оно поддерживается частично, а именно – *не возникает при удалении символов* (как и `onpropertychange`).

Пример использования (не работает в IE8-):

```
<input type="text"> oninput: <span id="result"></span>
<script>
  var input = document.body.children[0];

  input.oninput = function() {
    document.getElementById('result').innerHTML = input.value;
  };
</script>
```

В современных браузерах `oninput` – самое главное событие для работы с элементом формы. Именно его, а не `keydown/keypress` следует использовать.

Если бы ещё не проблемы со старыми IE… Впрочем, их можно решить при помощи события `propertychange`.

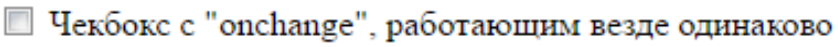
IE10-, событие propertychange

Это событие происходит только в IE10-, при любом изменении свойства. Оно позволяет отлавливать изменение тут же. Оно нестандартное, и его основная область использования – исправление недочётов обработки событий в старых IE.

Если поставить его на `checkbox` в IE8-, то получится «правильное» событие `change`:

```
<input type="checkbox"> Чекбокс с "onchange", работающим везде одинаково
<script>
 var checkbox = document.body.children[0];

 if ("onpropertychange" in checkbox) {
  // старый IE
  checkbox.onpropertychange = function() {
   // проверим имя изменённого свойства
   if (event.propertyName == "checked") {
    alert( checkbox.checked );
   }
  };
 } else {
  // остальные браузеры
  checkbox.onchange = function() {
   alert( checkbox.checked );
  };
 }
</script>
```



Это событие также срабатывает при изменении значения текстового элемента. Поэтому его можно использовать в старых IE вместо `oninput`.

К сожалению, в IE9 у него недочёт: оно не срабатывает при удалении символов. Поэтому сочетания `onpropertychange` + `oninput` недостаточно, чтобы поймать любое изменение поля в старых IE. Далее мы рассмотрим пример, как это можно сделать иначе.

События cut, copy, paste

Эти события используются редко. Они происходят при вырезании/вставке/копировании значения.

К сожалению, кросс-браузерного способа получить данные, которые вставляются/копируются, не существует, поэтому их основное применение – это отмена соответствующей операции.

Например, вот так:

```
<input type="text" id="input"> event: <span id="result"></span>
<script>
 input.oncut = input.oncopy = input.onpaste = function(event) {
  result.innerHTML = event.type + ' ' + input.value;
  return false;
 };
</script>
```

event:

Пример: поле с контролем СМС

Как видим, событий несколько и они взаимно дополняют друг друга.

Посмотрим, как их использовать, на примере.

Сделаем поле для СМС, рядом с которым должно показываться число символов, обновляющееся при каждом изменении поля.

Как такое реализовать?

Событие `input` идеально решит задачу во всех браузерах, кроме IE9-. Собственно, если IE9- нам не нужен, то на этом можно и остановиться.

IE9-

В IE8- событие `input` не поддерживается, но, как мы видели ранее, есть `onpropertychange`, которое может заменить его.

Что же касается IE9 – там поддерживаются и `input` и `onpropertychange`, но они оба не работают при удалении символов. Поэтому мы будем отслеживать удаление при помощи `keyup` на `Delete` и `BackSpace`. А вот удаление командой «вырезать» из меню – сможет отловить лишь `oncut`.

Получается вот такая комбинация:

```
<input type="text" id="sms"> символов: <span id="result"></span>
<script>
 function showCount() {
  result.innerHTML = sms.value.length;
 }

 sms.onkeyup = sms.oninput = showCount;
 sms.onpropertychange = function() {
  if (event.propertyName == "value") showCount();
 }
 sms.oncut = function() {
  setTimeout(showCount, 0); // на момент oncut значение еще старое
 };
</script>
```

символов:

Здесь мы добавили вызов `showCount` на все события, которые могут приводить к изменению значения. Да, иногда изменение будет обрабатываться несколько раз, но зато с гарантией.

Есть и совсем другой простой, но действенный вариант: через `setInterval` регулярно проверять значение и, если оно слишком длинное, обрезать его.

Чтобы сэкономить ресурсы браузера, мы можем начинать отслеживание по `onfocus`, а прекращать – по `onblur`, вот так:

```
<input type="text" id="sms"> символов: <span id="result"></span>

<script>
 var timerId;

 sms.onfocus = function() {

  var lastValue = sms.value;
  timerId = setInterval(function() {
```

```
    if (sms.value != lastValue) {
      showCount();
      lastValue = sms.value;
    }
  }, 20);
};

sms.onblur = function() {
  clearInterval(timerId);
};

function showCount() {
  result.innerHTML = sms.value.length;
}
</script>
```

Обратим внимание – весь этот «танец с бубном» нужен только для поддержки IE8-, в которых не поддерживается `oninput` и IE9, где `oninput` не работает при удалении.

Итого

События изменения данных:

Событие	Описание	Особенности
`change`	Изменение значения любого элемента формы. Для текстовых элементов срабатывает при потере фокуса.	В IE8- на чекбоксах ждет потери фокуса, поэтому для мгновенной реакции ставят также `onclick`-обработчик или `onpropertychange`.
`input`	Событие срабатывает только на текстовых элементах. Оно не ждет потери фокуса, в отличие от `change`.	В IE8- не поддерживается, в IE9 не работает при удалении символов.
`propertychange`	Только для IE10-. Универсальное событие для отслеживания изменения свойств элементов. Имя изменённого свойства содержится в `event.propertyName`. Используют для мгновенной реакции на изменение значения в старых IE.	В IE9 не срабатывает при удалении символов.
`cut/copy/paste`	Срабатывают при вставке/копировании/удалении текста. Если в их обработчиках отменить действие браузера, то вставки/копирования/удаления не произойдёт.	Вставляемое значение получить нельзя: на момент срабатывания события в элементе всё ещё *старое* значение, а новое недоступно.

Ещё особенность: в IE8- события `change`, `propertychange`, `cut` и аналогичные не всплывают. То есть, обработчики нужно назначать на сам элемент, без делегирования.

Формы: отправка, событие и метод submit

Событие submit возникает при отправке формы. Наиболее частое его применение – это *валидация* (проверка) формы перед отправкой.

Метод submit позволяет инициировать отправку формы из JavaScript, без участия пользователя. Далее мы рассмотрим детали их использования.

Событие submit

Чтобы отправить форму на сервер, у посетителя есть два способа:

1. **Первый – это нажать кнопку <input type="submit"> или <input type="image">.**
2. **Второй – нажать Enter, находясь на каком-нибудь поле.**

Какой бы способ ни выбрал посетитель – будет сгенерировано событие submit. Обработчик в нём может проверить данные и, если они неверны, то вывести ошибку и сделать event.preventDefault() – тогда форма не отправится на сервер.

Например, в таком HTML оба способа выведут alert, форма не будет отправлена:

```
<form onsubmit="alert('submit!');return false">
  Первый: Enter в текстовом поле <input type="text" value="Текст"><br>
  Второй: Нажать на "Отправить": <input type="submit" value="Отправить">
</form>
```

<form onsubmit="alert('submit!');return false">
Первый: Enter в текстовом поле <input type="text" value="Текст">
Второй: Нажать на "Отправить": <input type="submit" value="Отправить">
</form>

Ожидаемое поведение:

1. Перейдите в текстовое поле и нажмите Enter, будет событие, но форма не отправится на сервер благодаря return false в обработчике.
2. То же самое произойдет при клике на <input type="submit">.

Взаимосвязь событий submit и click

При отправке формы путём нажатия Enter на текстовом поле, на элементе <input type="submit"> везде, кроме IE8-, генерируется событие click.

Это довольно забавно, учитывая что клика-то и не было.

```
<form onsubmit="alert('submit');return false">
 <input type="text" size="30" value="При нажатии Enter будет click">
 <input type="submit" value="Submit" onclick="alert('click')">
</form>
```

<form onsubmit="alert('submit');return false">
<input type="text" size="30" value="При нажатии Enter будет click">
<input type="submit" value="Submit" onclick="alert('click')">
</form>

Метод submit

Чтобы отправить форму на сервер из JavaScript – нужно вызвать на элементе формы метод form.submit().

При этом само событие submit не генерируется. Предполагается, что если программист вызывает метод form.submit(), то он выполнил все проверки.

Это используют, в частности, для искусственной генерации и отправки формы.

ГРАФИЧЕСКИЕ КОМПОНЕНТЫ

Первый и главный шаг в наведении порядка – это оформить код в объекты, каждый из которых будет решать свою задачу.

Здесь мы сосредоточимся на графических компонентах, которые также называют «виджетами».

В браузерах есть встроенные виджеты, например <select>, <input> и другие элементы, о которых мы даже и не думаем, «как они работают». Они «просто работают»: показывают значение, вызывают события…

Виджет Menu

Мы начнём работу с виджета, который предусматривает уже готовую разметку.

То есть, в нужном месте HTML находится DOM-структура для меню – заголовок и список опций:

```
<div class="menu" id="sweets-menu">
  <span class="title">Сладости</span>
  <ul>
    <li>Торт</li>
    <li>Пончик</li>
    <li>...</li>
  </ul>
</div>
```

Далее она может дополняться, изменяться, но в начале – она такая.

Обратим внимание на важные соглашения виджета:

Вся разметка заключена в корневой элемент .

Это очень удобно: вынул этот элемент из DOM – нет меню, вставил в другое место – переместил меню. Кроме того, можно удобно искать подэлементы.

Внутри корневого элемента – только классы, не id.

Документ вполне может содержать много различных меню. Они не должны конфликтовать между собой, поэтому для разметки везде используются классы.

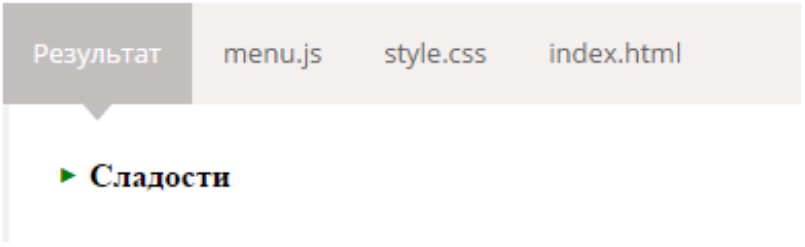
Исключение – корневой элемент. В данном случае мы предполагаем, что данное конкретное «меню сладостей» в документе только одно, поэтому даём ему id.

Класс виджета

Для работы с разметкой будем создавать объект new Menu и передавать ему корневой элемент. В конструкторе он поставит необходимые обработчики:

```
function Menu(options) {
  var elem = options.elem;
  elem.onmousedown = function() {
    return false;
  }
  elem.onclick = function(event) {
    if (event.target.closest('.title')) {
      elem.classList.toggle('open');
    }
  };
}
// использование
var menu = new Menu({
  elem: document.getElementById('sweets-menu')
});
```

Меню:



Это, конечно, только первый шаг, но уже здесь видны некоторые важные соглашения в коде.

У конструктора только один аргумент – объект options.

Это удобно, так как у графических компонентов обычно много настроек, большинство из которых имеют разумные значения «по умолчанию». Если передавать аргументы через запятую – их будет слишком много.

Обработчики назначаются через делегирование.

Вместо того, чтобы найти элемент и поставить обработчик на него:

```
var titleElem = elem.querySelector('.title');

titleElem.onclick = function() {
  elem.classList.toggle('open');
}
```

…Мы ставим обработчик на корневой elem и используем делегирование:

```
elem.onclick = function(event) {
  if (event.target.closest('.title')) {
    elem.classList.toggle('open');
  }
};
```

Это ускоряет инициализацию, так как не надо искать элементы, и даёт возможность в любой момент менять DOM внутри, в том числе через innerHTML, без необходимости переставлять обработчик.

В этот код лучше добавить дополнительную проверку на то, что найденный .title находится внутри elem:

```
elem.onclick = function(event) {
  var closestTitle = event.target.closest('.title');
  if (closestTitle && elem.contains(closestTitle)) {
    elem.classList.toggle('open');
  }
};
```

Публичные методы

Уважающий себя компонент обычно имеет публичные методы, которые позволяют управлять им снаружи.

Рассмотрим повнимательнее этот фрагмент:

```
if (event.target.closest('.title')) {
  elem.classList.toggle('open');
}
```

Здесь в обработчике события сразу код работы с элементом. Пока одна строка – всё понятно, но если их будет много, то при чтении понадобится долго и упорно вникать: «А что же, всё-таки, такое делается при клике?»

Для улучшения читаемости выделим обработчик в отдельную функцию toggle, которая к тому же станет полезным публичным методом:

```
function Menu(options) {
  var elem = options.elem;

  elem.onmousedown = function() {
    return false;
  }

  elem.onclick = function(event) {
    if (event.target.closest('.title')) {
      toggle();
    }
```

```
  };

  function toggle() {
    elem.classList.toggle('open');
  }

  this.toggle = toggle;
}
```

Теперь метод toggle можно использовать и снаружи:

```
var menu = new Menu(...);
menu.toggle();
```

Генерация DOM-элемента

До этого момента меню «оживляло» уже существующий HTML.

Но далеко не всегда в HTML уже есть готовая разметка. В сложных интерфейсах намного чаще её нет, а есть данные, на основе которых компонент генерирует разметку.

В случае меню, данные – это набор пунктов меню, которые передаются конструктору.

Для генерации DOM добавим меню три метода:

- render() – генерирует корневой DOM-элемент и заголовок меню.
- renderItems() – генерирует DOM для списка опций ul/li.
- getElem() – возвращает DOM-элемент меню, при необходимости запуская генерацию, публичный метод.

Функция генерации корневого элемента с заголовком render отделена от генерации списка renderItems. Почему – будет видно чуть далее.

Новый способ использования меню:

```
// создать объект меню с данным заголовком и опциями
var menu = new Menu({
  title: "Сладости",
  items: [
    "Торт",
    "Пончик",
    "Пирожное",
    "Шоколадка",
    "Мороженое"
  ]
});

// получить сгенерированный DOM-элемент меню
var elem = menu.getElem();

// вставить меню в нужное место страницы
document.body.appendChild(elem);
```

Код Menu с новыми методами:

```
function Menu(options) {
  var elem;

  function getElem() {
    if (!elem) render();
    return elem;
  }

  function render() {
```

```
  elem = document.createElement('div');
  elem.className = "menu";

  var titleElem = document.createElement('span');
  elem.appendChild(titleElem);
  titleElem.className = "title";
  titleElem.textContent = options.title;

  elem.onmousedown = function() {
    return false;
  };

  elem.onclick = function(event) {
    if (event.target.closest('.title')) {
      toggle();
    }
  }

}

function renderItems() {
  var items = options.items || [];
  var list = document.createElement('ul');
  items.forEach(function(item) {
    var li = document.createElement('li');
    li.textContent = item;
    list.appendChild(li);
  });
  elem.appendChild(list);
}

function open() {
  if (!elem.querySelector('ul')) {
    renderItems();
  }
  elem.classList.add('open');
};

function close() {
  elem.classList.remove('open');
};

function toggle() {
  if (elem.classList.contains('open')) close();
  else open();
};

this.getElem = getElem;
this.toggle = toggle;
this.close = close;
this.open = open;
```

}

Отметим некоторые особенности этого кода.

Обработчики отделяются от реальных действий.

В обработчике onclick мы «ловим» событие и выясняем, что именно произошло. Возможно, нужно проверитьevent.target, координаты, клавиши-модификаторы, и т.п. Это всё можно делать здесь же.

Выяснив, что нужно сделать, обработчик onclick не делает это сам, а вызывает для этого соответствующий метод. Этот метод уже не знает ничего о событии, он просто делает что-то с виджетом. Его можно вызвать и отдельно, не из обработчика.

Здесь есть ряд важных плюсов:

- Обработчик onclick не «распухает» чрезмерно.
- Код гораздо лучше читается.
- Метод можно повторно использовать, в том числе и сделать публичным, как в коде выше.

Генерация DOM, по возможности, должна быть «ленивой».

Мы стараемся откладывать работу до момента, когда она реально нужна. Например, когда new Menu создаётся, то переменная elem лишь объявляется. DOM-дерево будет сгенерировано только при вызове getElem() функцией render().

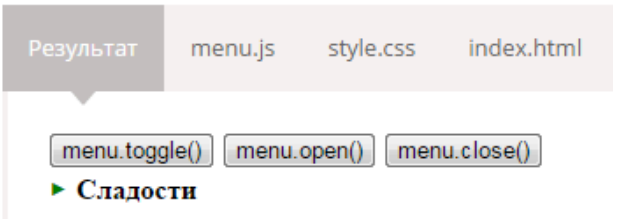
Более того! Пока меню закрыто – достаточно заголовка. Кроме того, возможно, посетитель вообще никогда не раскроет это меню, так зачем генерировать список раньше времени? А при первом открытии open() вызовет функцию renderItems(), которая специально для этого выделена отдельно от render().

Фаза инициализации очень чувствительна к производительности, так как обычно в сложном интерфейсе создаётся много всего.

Если изначально подходить к оптимизации на этой фазе «спустя рукава», то потом поправить долгий старт может быть сложно. Тем более, что инициализация – это фундамент, начало работы виджета, её оптимизация в будущем может потребовать сильных изменений кода.

Конечно, здесь, как и везде в оптимизации – без фанатизма. Бывают ситуации, когда гораздо удобнее что-то сделать сразу. Если это один элемент, то оптимизация здесь ни к чему. А если большой фрагмент DOM, который, как в случае с меню, прямо сейчас не нужен – то лучше отложить.

В действии:



Итого

Мы начали создавать компонент «с чистого листа», пока без дополнительных библиотек.

Основные принципы:

- Виджет – это объект, который либо контролирует готовое дерево DOM, либо создаёт своё.
- В конструктор виджета передаётся объект аргументов options.
- Виджет при необходимости создаёт элемент или «оживляет» готовый. Внутри в разметке не используются id.
- Обработчики назначаются через делегирование – для производительности и упрощения виджета.

- Обработчики событий вызывают соответствующий метод, не пытаются делать всё сами.
- При инициализации, если существенный участок работы можно отложить до реального задействования виджета – откладываем его.

Вёрстка графических компонентов

При создании графических компонент («виджетов») в первую очередь придумывается их HTML/CSS-структура.

Как будет выглядеть виджет в обычном состоянии? Как будет меняться в процессе взаимодействия с посетителем?

Чтобы разработка виджета была удобной, при вёрстке полезно соблюдать несколько простых, но очень важных соглашений.

Семантическая вёрстка

HTML-разметка и названия CSS-классов должны отражать не оформление, а смысл.

Например, сообщение об ошибке можно сверстать так:

```
<div style="color:red; border: 1px solid red">
  Плохая вёрстка сообщения об ошибке: атрибут style!
</div>
```

…Или так:

```
<div class="red red-border">
  Плохая вёрстка сообщения об ошибке: несемантический class!
</div>
```

В обоих случаях вёрстка не является семантической. В первом случае – стиль, а во втором – класс содержат информацию об *оформлении*.

При семантической вёрстке классы описывают смысл («что это?» – меню, кнопка…) и состояние (открыто, закрыто, отключено…) компонента.

Например:

```
<div class="error">
  Сообщение об ошибке (error), правильная вёрстка!
</div>
```

У предупреждения будет класс warning и так далее, по смыслу.

```
<div class="warning">
  Предупреждение  (warning), правильная вёрстка!
</div>
```

Семантическая верстка упрощает поддержку и развитие CSS, упрощает взаимодействие между членами команды.

Такая верстка удобна для организации JS-кода. В коде мы просто ставим нужный класс, остальное делает CSS.

Состояние виджета – класс на элементе

Зачастую компонент может иметь несколько состояний. Например, меню может быть открыто или закрыто.

Состояние должно добавляться CSS-классом не на тот элемент, который нужно скрыть/показать/…, а на тот, к которому оно «по смыслу» относится, обычно – на корневой элемент.

Например, меню в закрытом состоянии скрывает свой список элементов. Класс open нужно добавлять не к списку опций <ul>, который скрывается-показывается, а к *корневому элементу* виджета, поскольку это состояние касается всего меню:

```
<div class="menu open">
  <span class="title">Заголовок меню</span>
  <ul>
    <li>Список элементов</li>
```

```
  </ul>
</div>
```

Или, к примеру, разметка для индикатора загрузки может выглядеть так:

```
<div class="indicator loading">
  <span class="progress">Тут показывается прогресс</span>
</div>
```

Состояние индикатора может быть «в процессе» (loading) или «загрузка завершена» (complete). С точки зрения оформления оно может влиять только на показ внутреннего span, но ставить его нужно всё равно на внешний элемент, ведь это – состояние всего компонента.

Из примеров выше можно подумать, что классы, описывающие состояние, всегда ставятся на корневой элемент. Но это не так.

Возможно и такое, что состояние относится к внутреннему элементу. Например, для дерева состояние открыт/закрыт относится к узлу, соответственно, класс должен быть на узле.

Например:

```
<ul class="tree">
  <li class="closed">
    Закрытый узел дерева
  </li>
  <li class="open">
    Открытый узел дерева
  </li>
  ...
</ul>
```

Префиксы компонента у классов

Рассмотрим пример вёрстки «диалогового окна»:

```
<div class="dialog">
  <h2 class="title">Заголовок</h2>
  <div class="content">
    HTML-содержимое.
  </div>
  <div class="close">Закрыть</div>
</div>

<style>
.dialog {
  background: lightgreen;
  border: lime 2px solid;
  border-radius: 10px;
  padding: 4px;
  position: relative;
}

.dialog .title {
  margin: 0;
  font-size: 24px;
  color: darkgreen;
}

.dialog .content {
  padding: 10px 0 0 0;
```

```
}

.dialog .close {
  position: absolute;
  right: 4px;
  top: 4px;
  font-size: 10px;
}
</style>
```

Диалоговое окно может иметь любое HTML-содержимое.

А что будет, если в этом содержимом окажется меню – да-да, то самое, которое рассмотрели выше, со ?

Правило .dialog .title применяется ко всем .title внутри .dialog, а значит – и к нашему меню тоже. Будет конфликт стилей с непредсказуемыми последствиями.

Конечно, можно попытаться бороться с этим. Например, жёстко задать вложенность – использовать класс .dialog > .title. Это сработает в данном конкретном примере, но как быть в тех местах, где между .dialog и .title есть другие элементы? Длинные цепочки вида .dialog > ... > .title страшновато выглядят и делают вёрстку ужасно негибкой. К счастью, есть альтернативный путь.

Чтобы избежать возможных проблем, все классы внутри виджета начинают с его имени.

Здесь имя dialog, так что все, относящиеся к диалогу, будем начинать с dialog__

Получится так:

```
<div class="dialog">
  <h2 class="dialog__title">Заголовок</h2>
  <div class="dialog__content">
    HTML-содержимое.
  </div>
  <div class="dialog__close">Закрыть</div>
</div>

<style>
  .dialog { ... }
  .dialog__title { стиль заголовка }
  .dialog__content { стиль содержимого  }
  ...
</style>
```

Здесь двойное подчёркивание __ служит «стандартным» разделителем. Можно выбрать и другой разделитель, но при этом стоит иметь в виду, что иногда имя класса может состоять из нескольких слов. Например title-picture. С двойным подчёркиванием: dialog__title-picture, очень наглядно видно где что.

Есть ещё одно полезное правило, которое заключается в том, что стили должны вешаться на класс, а не на тег. То есть, не h2 { ... }, а .dialog__title { ... }, где .dialog__title – класс на соответствующем заголовке.

Это позволяет и избежать конфликтов на вложенных h2, и использовать всегда те теги, которые имеют правильный смысл, не оглядываясь на встроенные стили (которые можно обнулить своими).

Итого

- Вёрстка должна быть семантической, использовать соответствующие смыслу информации теги и классы.
- Класс, описывающий состояние всего компонента, нужно ставить на его корневом элементе, а не на том, который нужно «украсить» в этом состоянии. Если состояние относится не ко всему компоненту, а к его части – то на соответствующем «по смыслу» DOM-узле.
- Классы внутри компонента должны начинаться с префикса – имени компонента.

Это не всегда строго необходимо, но позволяет избежать проблем в случаях, когда компонент может содержать произвольный DOM, как например диалоговое окно с произвольным HTML-текстом.

Использование .dialog__title вместо .dialog .title гарантирует, что CSS не применится по ошибке к какому-нибудь другому .title внутри диалога.

5. Порядок выполнения работы

1. Создайте сценарий в соответствии с заданием.
2. Подключите сценарий к html-файлу.
3. Просмотрите с помощью web-браузера созданный документ.

6. Форма отчета о работе

Номер учебной группы _____
Фамилия, инициалы обучающегося _____
Дата выполнения работы _____
Тема работы: _____
Цель работы: _____
Задание: _____
Оснащение работы: _____
Результат выполнения работы: _____

7. Контрольные вопросы и задания

1. Каким образом можно получить элементы FORM?
2. Объяснить назначение событий focus и blur?
3. Объяснить назначение событий change, input?
4. Дайте понятие плейсхолдера?

Рекомендуемая литература

1 Васильев, А.Н. JavaScript в примерах и задачах / А.Н. Васильев. – М.: Эксмо, 2018.

2 Вахтуров, В.В. JavaScript. Освой на примерах / В.В. Вахтуров. – СПб.:БХВ-Петербург, 2007.

3 Макфарланд, Д. JavaScript и jQuery. Исчерпывающее руководство / Д. Макфарланд. - М.: Эксмо, 2012.

4 Интернет-ресурс https://learn.javascript.ru/