МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

ГЛАВНОЕ УПРАВЛЕНИЕ ПО ОБРАЗОВАНИЮ

МОГИЛЕВСКОГО ОБЛАСТНОГО ИСПОЛНИТЕЛЬНОГО КОМИТЕТА

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«МОГИЛЕВСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ КОЛЛЕДЖ»

|  |  |
| --- | --- |
| Специальность | 2– 40 01 01 |
| Учебная группа | ПО-455 |
|  |  |

Учебная дисциплина Программные средства

создания интернет-

приложений

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №9**

**ВНЕДРЕНИЕ СЦЕНАРИЕВ В HTML-ДОКУМЕНТ.**

**ОБРАБОТКА СОБЫТИЙ С ИСПОЛЬЗОВАНИЕМ DHTML**

Выполнил Савич А.О.

Проверил Денисовец Д. А.

2022

**1 Цель работы**

1.1 Формирование умений внедрять сценарии в HTML-документ, обрабатывать события с использованием DHTML

**2 Индивидуальное задание**

**Вариант 4**

1 Создайте документ Lab8.html, содержащий таблицу следующего вида:

|  |  |  |
| --- | --- | --- |
| Элемент1 | Элемент2 | Элемент3 |
| Элемент4 | Элемент5 | Элемент6 |

2 Определите следующие классы:

- cl1 – текст фиолетового цвета, шрифт MV Boli,11 px;

- cl2 – текст синего цвета, полужирный;

- cl3 – текст голубого цвета, шрифт Tekton Pro Ext, подчеркнутое начертание;

элемент таблицы

- cl4 – текст красного цвета, в желтой рамке (например, );

- cl5 – шрифт зеленого цвета, курсивный, жирное начертание.

3 Ко всем элементам таблицы примените класс cl1.

4 При наведении указателя мыши на «Элемент1» примените класс cl2. Когда указатель мыши выходит за пределы данного элемента таблицы, должен применяться класс cl5.

5 При двойном щелчке на «Элемент2» примените класс cl3.

6 При одинарном щелчке на «Элемент3» примените класс cl4.

7 При наведении курсора мыши на «Элемент4» измените цвет текста на розовый.

<html>

<head>

<title>

заголовок

</title>

<style type="text/css">

.clsr {

font-family: "MV Boli";

font-size:11px;

color: purple;

}

#cl1:hover{

color:blue ;

font-weight: bold;s

}

#cl2:hover{

color:skyblue;

font-family: "Tekton Pro Ext";

text-overflow-decoration: underline;

}

#cl3:hover{

color:red;

border-color: yellow;

border-width: 4px ;

}

#cl1{

color:green;

font-weight: bold;

font-style: italic;

}

#cl4:hover{

color:pink;

}

</style>

</head>

<body>

<table border="2" class="clsr">

<tr>

<td id="cl1">Элемент1</td>

<td id="cl2">Элемент2</td>

<td id="cl3">Элемент3</td>

</tr>

<tr>

<td id="cl4">Элемент4</td>

<td id="cl5">Элемент5</td>

<td id="cl6">Элемент6</td>

</tr>

</table>

</body>

</html>

3 Контрольные вопросы

1 Поясните суть механизма DHML.

**Dynamic HTML** или **DHTML** — это способ (подход) создания интерактивного использующий сочетание статичного языка разметки встраиваемого (и выполняемого на стороне клиента

Он может быть использован для создания приложения например для более простой навигации или для придания интерактивности форм. DHTML может быть использован для динамического перетаскивания элементов по экрану. Также он может служить как инструмент для создания основанных на браузере видеоигр.

2 Опишите клиентские и серверные сценарии.

Сценарии – программы, созданные на сценарном язык программирования.

Ядро JavaScript содержит набор основных объектов, таких как Array, Date и Math, и основной набор элементов языка, таких как операции, управляющие структуры и операторы. Ядро JavaScript может быть расширено для различных целей путём дополнения его новыми объектами.

Клиентский JavaScript расширяет ядро языка, добавляя объекты управления браузером (Navigator или другой аналогичный web-браузер) и Document Object Model/Объектную Модель Документа (DOM). Они позволяют приложению размещать элементы на HTML-форме и реагировать на пользовательские события, такие как щелчок мышью, ввод данных в форму или навигация по страницам.

Серверный JavaScript расширяет ядро языка, добавляя объекты, относящиеся к запуску JavaScript на сервере. Например, серверные расширения позволяют приложению взаимодействовать с реляционной базой данных, сохраняя информацию между вызовами приложения, или выполнять манипуляции с файлами на сервере

JavaScript позволяет создавать приложения, работающие по всей сети Internet. Клиентские приложения работают в браузере, таком как Netscape Navigator, а серверные приложения - на сервере Netscape Enterprise Server. Используя JavaScript, можно создавать динамические HTML-страницы, обрабатывающие пользовательский ввод и данные, используя специальные объекты, файлы и реляционные БД.

Клиентский и серверный JavaScript имеют следующие общие элементы:

- ключевые слова;

- синтаксис операторов и грамматику;

- правила написания выражений, переменных и литералов;

- объектная модель;

- предопределённые объекты и функции, такие как Array, Date и Math.

Web-браузеры могут интерпретировать операторы клиентского JavaScript, внедрённые в HTML-страницу. Когда браузер (или клиент) запрашивает такую страницу, сервер высылает клиенту по сети полное содержимое документа, включая HTML и операторы JavaScript. Браузер читает страницу сверху вниз, отображая результат работы HTML и выполняя операторы JavaScript по мере их обнаружения. Операторы клиентского JavaScript, встроенного в HTML-страницу, могут реагировать на пользовательские события, такие как щелчок мыши, ввод данных в форму и навигация по страницам. Например, Вы можете написать функцию JavaScript для проверки ввода пользователем правильной информации в форму, запрашивающую телефонный номер или zip-код. Без передачи по сети внедрённый JavaScript на HTML-странице может проверить введённые данные и вывести диалоговое окно, если пользователь ввёл неверные данные. На сервере можно внедрять JavaScript в HTML-страницы. Серверные операторы могут соединяться с реляционными базами данных разных производителей, разделять информацию между пользователями приложения, получать доступ к файловой системе сервера или взаимодействовать с другими приложениями через LiveConnect и Java. HTML-страницы с серверным JavaScript могут содержать также клиентский JavaScript.

В отличие от страниц с чисто клиентским JavaScript, HTML-страницы, использующие серверный JavaScript, компилируются в байт-кодовые исполняемые файлы. Эти исполняемые приложения запускаются на выполнение web-сервером, имеющим машину времени выполнения JavaScript. Исходя из этого, создание приложений JavaScript это процесс из двух этапов.

На первом этапе Вы создаёте HTML-страницы (которые могут содержать операторы как клиентского, так и серверного JavaScript) и файлы JavaScript. Затем Вы компилируете все эти файлы в единый исполняемый блок.

На втором этапе страница приложения запрашивается клиентским браузером. Машина выполнения использует исполняемый блок для просмотра исходной страницы и динамической генерации HTML-страницы, возвращаемой клиенту. Она выполняет все найденные на странице операторы серверного JavaScript. Выполнение этих операторов может добавить новые операторы HTML или операторы клиентского JavaScript в HTML-страницу. Машина выполнения отсылает затем окончательный вариант страницы по сети Navigator-клиенту, который выполняет клиентский JavaScript и отображает результат.

В отличие от стандартных программ Common Gateway Interface (CGI), все исходники JavaScript интегрированы непосредственно в HTML-страницы, ускоряя разработку и облегчая обслуживание. Служба Session Management Service серверного JavaScript содержит объекты, которые Вы можете использовать для работы с данными, существующими между клиентскими запросами, у нескольких клиентов или нескольких приложений. Служба LiveWire Database Service серверного JavaScript предоставляет объекты для доступа к БД, служащие интерфейсом для серверов Structured Query Language (SQL) /11/.

3 Перечислите события для элементов, получающих фокус ввода.

Элемент получает фокус, когда пользователь кликает по нему или использует клавишу Tab. Также существует HTML-атрибут autofocus, который устанавливает фокус на элемент, когда страница загружается. Есть и другие способы получения фокуса, о них – далее.

Фокусировка обычно означает: «приготовься к вводу данных на этом элементе», это хороший момент, чтобы инициализовать или загрузить что-нибудь.

Момент потери фокуса («blur») может быть важнее. Это момент, когда пользователь кликает куда-то ещё или нажимает Tab, чтобы переключиться на следующее поле формы. Есть другие причины потери фокуса, о них – далее.

Потеря фокуса обычно означает «данные введены», и мы можем выполнить проверку введённых данных или даже отправить эти данные на сервер и так далее.

В работе с событиями фокусировки есть важные особенности. Мы постараемся разобрать их далее.

События focus/blur

Событие focus вызывается в момент фокусировки, а blur – когда элемент теряет фокус.

Используем их для валидации(проверки) введённых данных.

В примере ниже:

Обработчик blur проверяет, введён ли email, и если нет – показывает ошибку.

Обработчик focus скрывает это сообщение об ошибке (в момент потери фокуса проверка повторится):

<style>

.invalid { border-color: red; }

#error { color: red }

</style>

Ваш email: <input type="email" id="input">

<div id="error"></div>

<script>

input.onblur = function() {

if (!input.value.includes('@')) { // не email

input.classList.add('invalid');

error.innerHTML = 'Пожалуйста, введите правильный email.'

}

};

input.onfocus = function() {

if (this.classList.contains('invalid')) {

// удаляем индикатор ошибки, т.к. пользователь хочет ввести данные заново

this.classList.remove('invalid');

error.innerHTML = "";

}

};

</script>

Современный HTML позволяет делать валидацию с помощью атрибутов required, pattern и т.д. Иногда – это всё, что нам нужно. JavaScript можно использовать, когда мы хотим больше гибкости. А ещё мы могли бы отправлять изменённое значение на сервер, если оно правильное.

Методы focus/blur

Методы elem.focus() и elem.blur() устанавливают/снимают фокус.

Например, запретим посетителю переключаться с поля ввода, если введённое значение не прошло валидацию:

<style>

.error {

background: red;

}

</style>

Ваш email: <input type="email" id="input">

<input type="text" style="width:280px" placeholder="введите неверный email и кликните сюда">

<script>

input.onblur = function() {

if (!this.value.includes('@')) { // не email

// показать ошибку

this.classList.add("error");

// ...и вернуть фокус обратно

input.focus();

} else {

this.classList.remove("error");

}

};

</script>

Это сработает во всех браузерах, кроме Firefox (bug).

Если мы что-нибудь введём и нажмём Tab или кликнем в другое место, тогда onblur вернёт фокус обратно.

Отметим, что мы не можем «отменить потерю фокуса», вызвав event.preventDefault() в обработчике onblur потому, что onblur срабатывает после потери фокуса элементом.

Однако на практике следует хорошо подумать, прежде чем внедрять что-то подобное, потому что мы обычно должны показывать ошибки пользователю, но они не должны мешать пользователю при заполнении нашей формы. Ведь, вполне возможно, что он захочет сначала заполнить другие поля.

**Потеря фокуса, вызванная JavaScript**

Потеря фокуса может произойти по множеству причин.

Одна из них – когда посетитель кликает куда-то ещё. Но и JavaScript может быть причиной, например:

alert переводит фокус на себя – элемент теряет фокус (событие blur), а когда alert закрывается – элемент получает фокус обратно (событие focus).

Если элемент удалить из DOM, фокус также будет потерян. Если элемент добавить обратно, то фокус не вернётся.

Из-за этих особенностей обработчики focus/blur могут сработать тогда, когда это не требуется.

Используя эти события, нужно быть осторожным. Если мы хотим отследить потерю фокуса, которую инициировал пользователь, тогда нам следует избегать её самим.

Включаем фокусировку на любом элементе: tabindex

Многие элементы по умолчанию не поддерживают фокусировку.

Какие именно – зависит от браузера, но одно всегда верно: поддержка focus/blur гарантирована для элементов, с которыми посетитель может взаимодействовать: <button>, <input>, <select>, <a> и т.д.

С другой стороны, элементы форматирования <div>, <span>, <table> – по умолчанию не могут получить фокус. Метод elem.focus() не работает для них, и события focus/blur никогда не срабатывают.

Это можно изменить HTML-атрибутом tabindex.

Любой элемент поддерживает фокусировку, если имеет tabindex. Значение этого атрибута – порядковый номер элемента, когда клавиша Tab (или что-то аналогичное) используется для переключения между элементами.

То есть: если у нас два элемента, первый имеет tabindex="1", а второй tabindex="2", то находясь в первом элементе и нажав Tab – мы переместимся во второй.

Порядок перебора таков: сначала идут элементы со значениями tabindex от 1 и выше, в порядке tabindex, а затем элементы без tabindex (например, обычный <input>).

При совпадающих tabindex элементы перебираются в том порядке, в котором идут в документе.

Есть два специальных значения:

tabindex="0" ставит элемент в один ряд с элементами без tabindex. То есть, при переключении такие элементы будут после элементов с tabindex ≥ 1.

Обычно используется, чтобы включить фокусировку на элементе, но не менять порядок переключения. Чтобы элемент мог участвовать в форме наравне с обычными <input>.

tabindex="-1" позволяет фокусироваться на элементе только программно. Клавиша Tab проигнорирует такой элемент, но метод elem.focus() будет действовать.

Например, список ниже. Кликните первый пункт в списке и нажмите Tab:

Кликните первый пункт в списке и нажмите Tab. Продолжайте следить за порядком. Обратите внимание, что много последовательных нажатий Tab могут вывести фокус из iframe 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>

Порядок такой: 1 - 2 - 0. Обычно <li> не поддерживает фокусировку, но tabindex включает её, а также события и стилизацию псевдоклассом :focus.

**Свойство elem.tabIndex тоже работает**

Мы можем добавить tabindex из JavaScript, используя свойство elem.tabIndex. Это даст тот же эффект.

События focusin/focusout

События focus и blur не всплывают.

Например, мы не можем использовать onfocus на <form>, чтобы подсветить её:

<!-- добавить класс при фокусировке на форме -->

<form onfocus="this.className='focused'">

<input type="text" name="name" value="Имя">

<input type="text" name="surname" value="Фамилия">

</form>

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

Пример выше не работает, потому что когда пользователь перемещает фокус на <input>, событие focus срабатывает только на этом элементе. Это событие не всплывает. Следовательно, form.onfocus никогда не срабатывает.

У этой проблемы два решения.

Первое: забавная особенность – focus/blur не всплывают, но передаются вниз на фазе перехвата.

Это сработает:

<form id="form">

<input type="text" name="name" value="Имя">

<input type="text" name="surname" value="Фамилия">

</form>

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

<script>

// установить обработчик на фазе перехвата (последний аргумент true)

form.addEventListener("focus", () => form.classList.add('focused'), true);

form.addEventListener("blur", () => form.classList.remove('focused'), true);

</script>

Второе решение: события focusin и focusout – такие же, как и focus/blur, но они всплывают.

Заметьте, что эти события должны использоваться с elem.addEventListener, но не с on<event>.

Второй рабочий вариант:

<form id="form">

<input type="text" name="name" value="Имя">

<input type="text" name="surname" value="Фамилия">

</form>

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

<script>

form.addEventListener("focusin", () => form.classList.add('focused'));

form.addEventListener("focusout", () => form.classList.remove('focused'));

</script>

Итого

События focus и blur срабатывают на фокусировке/потере фокуса элемента.

Их особенности:

Они не всплывают. Но можно использовать фазу перехвата или focusin/focusout.

Большинство элементов не поддерживают фокусировку по умолчанию. Используйте tabindex, чтобы сделать фокусируемым любой элемент.

Текущий элемент с фокусом можно получить из document.activeElement.

4 Перечислите события, используемые для отдельных элементов.

Ранее вы уже ознакомились с деревом и узлами DOM, а также научились обращаться, перемещаться, добавлять, удалять и изменять узлы и элементы с помощью консоли разработчика.

На данный момент вы можете вносить в DOM практически любые изменения, но с точки зрения пользователя это не очень полезно, потому что вы делаете это только вручную. Узнав немного больше о событиях в JavaScript, вы поймете, как связать все вместе, чтобы сделать интерактивный сайт.

События – это действия в браузере, которые могут быть инициированы пользователем или самим браузером. Ниже приведены несколько примеров общих событий, которые могут произойти на веб-сайте:

Страница заканчивает загружаться

Пользователь нажимает кнопку

Пользователь наводит мышь на выпадающее меню

Пользователь заполняет форму

Пользователь нажимает клавишу на своей клавиатуре

Кодируя ответы JavaScript, которые выполняются при событии, разработчики могут отображать пользователям сообщения, проверять данные, реагировать на нажатие кнопки и выполнять многие другие действия.

В этой статье вы узнаете, что такое обработчики, прослушиватели и объекты событий. Также мы рассмотрим нескольких наиболее распространенных событий и три разных способа написания кода для их обработки. Эти знания помогут вам сделать более интерактивный веб-интерфейс для конечных пользователей.

Обработчики и прослушиватели событий

Когда пользователь нажимает кнопку или клавишу, происходит событие. Такие события называются событием клика или событием нажатия клавиши соответственно.

Обработчик событий – это функция JavaScript, которая запускается при срабатывании события.

Прослушиватель событий присоединяет чувствительный интерфейс к элементу, который позволяет этому элементу ждать начала конкретного события.

Существует три способа присвоения событий элементам:

Встроенные (inline) обработчики событий

Свойства обработчика событий

Прослушиватели событий

Мы рассмотрим все три метода, а затем обсудим плюсы и минусы каждого из них.

Inline-обработчики событий

Сначала рассмотрим обработчик событий inline. Начнем с очень простого примера, состоящего из элемента кнопки (button ) и элемента p. Мы хотим, чтобы пользователь нажал кнопку, чтобы изменить текстовое содержимое p.

Начнем с HTML-страницы с кнопкой в теле. Сошлитесь на файл JavaScript, в который вы добавите немного кода.

<!DOCTYPE html>  
<html lang="en-US">  
<head>  
<title>Events</title>  
</head>  
<body>  
<!-- Add button -->  
<button>Click me</button>  
<p>Try to change me.</p>  
</body>  
<!-- Reference JavaScript file -->  
<script src="js/events.js"></script>  
</html>

Непосредственно в button добавьте атрибут onclick. Значением атрибута будет функция changeText().

<!DOCTYPE html>  
<html lang="en-US">  
<head>  
<title>Events</title>  
</head>  
<body>  
<button onclick="changeText()">Click me</button>  
<p>Try to change me.</p>  
</body>  
<script src="js/events.js"></script>  
</html>

Теперь создайте файл events.js, который мы разместили здесь в каталоге js/. В нем нужно создать функцию changeText (), которая изменит textContent элемента p.

// Function to modify the text content of the paragraph  
const changeText = () => {  
const p = document.querySelector('p');  
p.textContent = "I changed because of an inline event handler.";  
}

Откройте events.html в браузере, и вы увидите страницу с кнопкой и текстом:

Try to change me

Однако, когда вы или другой пользователь нажимаете на кнопку, текст тэга p будет меняться: вместо «Try to change me» появится «I changed because of an inline event handler.».

Inline-обработчики событий – это простой способ начать работать с событиями, но обычно они не используются нигде, кроме тестирования и учебных задач.

Сравнить inline обработчики событий можно с встроенными стилями CSS в HTML-элементе. Гораздо практичнее поддерживать отдельную таблицу стилей, чем создавать встроенные стили для каждого элемента; аналогично, более целесообразно поддерживать JavaScript, который обрабатывается полностью через отдельный файл сценария, чем добавлять обработчики к каждому элементу.

Свойства обработчика событий

Следующим шагом являются свойства обработчика события. Это очень похоже на inline-обработчик, только вместо атрибута в HTML устанавливается свойство элемента в JavaScript.

Настройка будет выглядеть почти так же, за исключением того, что вам больше не нужно включать строку onclick=”changeText ()”.

...  
<body>  
<button>Click me</button>  
<p>I will change.</p>  
</body>  
...

Функция останется аналогичной, но теперь вам нужно получить доступ к элементу button в JavaScript. Можно просто получить доступ к onclick так же, как при обращении к стилю, идентификатору или любому другому свойству элемента, а затем присвоить ссылку функции.

// Function to modify the text content of the paragraph  
const changeText = () => {  
const p = document.querySelector('p');  
p.textContent = "I changed because of an event handler property.";  
}  
// Add event handler as a property of the button element  
const button = document.querySelector('button');  
button.onclick = changeText;

**Примечание**: Обработчики событий не следуют соглашению camelCase, которого придерживается большинство JavaScript-кодов. Обратите внимание, что нужно писать onclick, а не onClick.

При первой загрузке страницы браузер отобразит кнопку Click me и сообщение «I will change.»

Нажмите кнопку, после чего вместо текущего сообщения появится текст «I changed because of an inline event handler.»

Обратите внимание, что при передаче ссылки функции на свойство onclick мы не включаем круглые скобки, так как мы не вызываем функцию в этот момент, а только передаем ссылку на нее.

Свойство обработчика событий немного более удобно, чем inline обработчик, но этот метод имеет похожие недостатки. Например, попытка установить несколько отдельных свойств onclick приведет к перезаписыванию всех свойств, кроме последнего.

const p = document.querySelector('p');  
const button = document.querySelector('button');  
const changeText = () => {  
p.textContent = "Will I change?";  
}  
const alertText = () => {  
alert('Will I alert?');  
}  
// Events can be overwritten  
button.onclick = changeText;  
button.onclick = alertText;

В приведенном выше примере нажатие кнопки будет отображать только предупреждение, а текст p не изменится, поскольку код alert() был добавлен в свойство последним.

Прослушиватели событий

Последним дополнением к обработчикам JavaScript являются прослушиватели событий. Прослушиватель наблюдает за событием элемента. Вместо того чтобы присваивать событие непосредственно свойству элемента, для прослушивания события нужно использовать метод addEventListener().

addEventListener() принимает два обязательных параметра – событие, которое он должен прослушивать, и функцию обратного вызова прослушивателя.

HTML для прослушивателя событий будет таким:.

...  
<button>Click me</button>  
<p>I will change.</p>  
...

Мы по-прежнему используем ту же функцию changeText(), что и раньше. Метод addEventListener() нужно добавить к кнопке.

// Function to modify the text content of the paragraph  
const changeText = () => {  
const p = document.querySelector('p');  
p.textContent = "I changed because of an event listener.";  
}  
// Listen for click event  
const button = document.querySelector('button');  
button.addEventListener('click', changeText);

Обратите внимание, что при первых двух методах событие click называлось onclick, но в прослушивателе событий оно называется click. Прослушиватели событий не используют слово on . В следующем разделе мы рассмотрим примеры других событий.

Когда вы перезагрузите страницу с таким кодом JavaScript, вы получите кнопку Click me и сообщение «I changed because of an event listener».

На первый взгляд, прослушиватели событий кажутся очень похожими на свойства обработчика событий, но у них есть несколько преимуществ. К примеру, вы можете установить несколько прослушивателей событий на один и тот же элемент, как показано в примере ниже.

const p = document.querySelector('p');  
const button = document.querySelector('button');  
const changeText = () => {  
p.textContent = "Will I change?";  
}  
const alertText = () => {  
alert('Will I alert?');  
}  
// Multiple listeners can be added to the same event and element  
button.addEventListener('click', changeText);  
button.addEventListener('click', alertText);

В этом примере оба события будут срабатывать, предоставляя пользователю оба сообщения (и предупреждение, и измененный текст после нажатия).

Часто вместо ссылки на функцию для прослушивателя событий используются анонимные функции. Анонимные функции – это функции, которые не называются.

// An anonymous function on an event listener  
button.addEventListener('click', () => {  
p.textContent = "Will I change?";  
});

Также для удаления одного или всех событий из элемента можно использовать функцию removeEventListener().

// Remove alert function from button element  
button.removeEventListener('click', alertText);

Кроме того, вы можете использовать addEventListener() для объекта document и window .

Прослушиватели событий в настоящее время являются наиболее распространенным и предпочтительным способом обработки событий в JavaScript.

Основные типы событий

Теперь вы знаете, что такое inline обработчики событий, свойства обработчика событий и прослушиватели событий. Все эти методы  мы рассмотрели на примере событие click, но в JavaScript есть еще много событий. Мы рассмотрим несколько наиболее распространенных типов.

События мыши

События мыши являются одними из наиболее часто используемых событий. К ним относятся нажатия кнопок мыши, наведение и перемещение курсора мыши. Эти события также включают аналогичные действия на сенсорном устройстве.

| **Событие** | **Описание** |
| --- | --- |
| click | Запускается при нажатии и отпускании кнопки мыши на элементе |
| dblclick | Запускается при двойном клике по элементу |
| mouseenter | Запускается, когда курсор входит в элемент |
| mouseleave | Запускается, когда курсор покидает элемент |
| mousemove | Запускается каждый раз, когда курсор перемещается внутри элемента |

Событие click – клик, сложное событие, состоящее из комбинированных событий mousedown и mouseup.

Использование mouseenter и mouseleave в связке воссоздает эффект зависания, который продолжается до тех пор, пока указатель мыши находится на элементе.

События формы

События формы – это действия, которые относятся к формам (например, выбор или удаление элементов ввода, отправка формы и т.п.).

| **Событие** | **Описание** |
| --- | --- |
| submit | Запускается при отправке формы |
| focus | Запускается, когда элемент (например, input) находится в фокусе |
| blur | Запускается, когда элемент теряет фокус |

Фокус достигается, когда пользователь выбирает элемент, например, с помощью щелчка мыши или перехода к нему с помощью клавиши TAB.

JavaScript часто используется для отправки форм и значений на язык бэкэнда. Преимущество использования JavaScript для отправки форм заключается в том, что он не требует перезагрузки страницы для отправки формы. Также JavaScript может использоваться для проверки необходимых полей ввода.

События клавиатуры

События клавиатуры используются для обработки действий клавиатуры, таких как нажатие клавиши, подъем и удержание клавиши.

| **Событие** | **Описание** |
| --- | --- |
| keydown | Запускается один раз при нажатии клавиши |
| keyup | Запускается один раз, когда клавиша отпускается |
| keypress | Постоянно срабатывает при нажатии клавиши |

Хотя keydown и keypress выглядят одинаково, они не имеют доступа к одним и тем же клавишам. keydown будет подтверждать каждую нажатую клавишу, а keypress пропускает клавиши, которые не создают символ, например SHIFT, ALT или DELETE.

События клавиатуры имеют особые свойства для доступа к отдельным клавишам.

Если параметр, известный как объект event, передается прослушивателю событий, вы можете получить дополнительную информацию о действии, которое произошло. К объектам клавиатуры относятся свойства keyCode, key, и code.

Например, если пользователь нажимает на клавишу «А» на своей клавиатуре, свойства будут выглядеть следующим образом:

| **Свойства** | **Описание** | **Пример** |
| --- | --- | --- |
| keyCode | Число, относящееся к ключу | 65 |
| key | Представляет символ | a |
| code | Представляет нажатую физическую клавишу | KeyA |

Чтобы показать, как собирать эту информацию через консоль JavaScript, можно написать следующий код.

// Test the keyCode, key, and code properties  
document.addEventListener('keydown', event => {  
console.log('key: ' + event.keyCode);  
console.log('key: ' + event.key);  
console.log('code: ' + event.code);  
});

Когда вы нажмете Enter в консоли, вы сможете нажать клавишу на клавиатуре (в этом примере это будет a).

keyCode: 65  
key: a  
code: KeyA

Свойство keyCode – это число, которое относится к нажатой клавише. Свойство key – это имя символа, которое может измениться (например, нажатие клавиши a с помощью SHIFT приведет к символу A). Свойство code представляет собой физическую клавишу на клавиатуре.

**Примечание**: Обратите внимание, что keyCode находится в процессе устаревания, и в новых проектах предпочтительно использовать свойство code.

Полный список событий вы найдете на Mozilla Developer Network.

Объекты Event

Объект Event состоит из свойств и методов, к которым могут обращаться все события. В дополнение к универсальному объекту Event каждый тип события имеет свои собственные расширения, такие как KeyboardEvent и MouseEvent.

Объект Event передается в качестве параметра через функцию прослушивателя. Обычно он записывается как event или e. Мы можем получить доступ к свойству code  события keydown для репликации элементов управления клавиатурой ПК.

Попробуйте создать базовый HTML-файл с тегами <p> и загрузите его в браузер.

<!DOCTYPE html>  
<html lang="en-US">  
<head>  
<title>Events</title>  
</head>  
<body>  
<p></p>  
</body>  
</html>

Затем введите следующий код JavaScript в консоль.

**Читайте также**: Использование консоли разработчика JavaScript

// Pass an event through to a listener  
document.addEventListener('keydown', event => {  
var element = document.querySelector('p');  
// Set variables for keydown codes  
var a = 'KeyA';  
var s = 'KeyS';  
var d = 'KeyD';  
var w = 'KeyW';  
// Set a direction for each code  
switch (event.code) {  
case a:  
element.textContent = 'Left';  
break;  
case s:  
element.textContent = 'Down';  
break;  
case d:  
element.textContent = 'Right';  
break;  
case w:  
element.textContent = 'Up';  
break;  
}  
});

Когда вы нажмете одну из клавиш – a, s, d или w – вы увидите в консоли Left, Down, Right или Up.

Теперь вы можете продолжить работу над тем, как браузер будет реагировать на клавиши, и создать более динамичный веб-сайт.

Давайте рассмотрим одно из наиболее часто используемых свойств события: свойство target. В следующем примере есть три элемента div внутри одного section.

<!DOCTYPE html>  
<html lang="en-US">  
<head>  
<title>Events</title>  
</head>  
<body>  
<section>  
<div id="one">One</div>  
<div id="two">Two</div>  
<div id="three">Three</div>  
</section>  
</body>  
</html>

Используя event.target в консоли браузера, вы можете разместить один прослушиватель событий на элементе внешнего section  и получить вложенный элемент.

const section = document.querySelector('section');  
// Print the selected target  
section.addEventListener('click', event => {  
console.log(event.target);  
});

Клик по любому из этих элементов возвращает результат соответствующего элемента в консоль, используя event.target. Это очень удобно, так как позволяет использовать всего один прослушиватель событий для доступа ко многим вложенным элементам.

С помощью объекта Event можно настроить действия, связанные со всеми событиями.

Заключение

События – это действия, которые происходят на веб-сайте, такие как клик, наведение курсора, отправка формы, загрузка страницы или нажатие клавиши на клавиатуре. JavaScript становится по-настоящему интерактивным и динамичным, когда вы можете заставить веб-сайты реагировать на действия пользователей.

В этом мануале вы узнали, что такое события. Также мы рассмотрели примеры общих событий, объяснили разницу между обработчиками и прослушивателями событий. Используя эти знания, вы сможете создавать динамические веб-сайты и приложения.

5 Как изменить стиль элемента при возникновении определенного события?

**CSS-переходы и их события**

Как вы уже знаете для создания простой анимации в CSS можно использовать **переходы**. Суть здесь заключается в том, что с помощью них мы можем указать браузеру то, что **значение какого-то CSS-свойства нужно изменить не сразу, а плавно в течении некоторого количества времени**. После этого, для того чтобы переход начался, нам нужно будет всего лишь изменить значение этого CSS-свойства, и браузер сам выполнит эту анимацию.

Очень часто на сайтах и в веб-приложениях запуск CSS-перехода выполняется при наступлении каких-то определенных событий. Например, при клике на элемент или при наведении на него курсора.

Для **задания CSS-перехода используются следующие свойства**:

transition-property – для указания CSS-свойства, которое нужно анимировать;

transition-duration – длительность перехода;

transition-timing-function – задает как должна изменяться скорость во время CSS-перехода;

transition-delay – время задержки перед началом перехода.

Например, мы хотим, чтобы изменения цвета фона (background-color) происходило плавно в течение 1 секунды. При этом нам нужно, чтобы этот процесс начинался не сразу, а через 0,1 секунды после создания CSS-перехода. Кроме этого, необходимо чтобы переход начинался и заканчивался медленно, или другими словами, чтобы анимация выполнялась посредством функции ease-in-out:

HTMLКопировать

<style>

.box {

background-color: #fffde7;

transition: background-color;

transition-duration: 1s;

transition-timing-function: ease-in-out;

transition-delay: 0.1s;

}

.box:hover {

background-color: #ffeb3b;

}

</style>

<div class="box">Lorem</div>

В CSS также можно записать все эти свойства кратко, используя transition:

CSSКопировать

.box {

background-color: #fffde7;

transition: background-color 1s ease-in-out 0.1s;

}

**При CSS-переходах могут возникать следующие события в DOM**:

transitionrun – при создании CSS-перехода, то есть до того, как он реально начался;

transitionstart – когда CSS-переход фактически начался;

transitionend – при завершении выполнения перехода;

transitioncancel – при отмене CSS-перехода.

Событие transitionrun возникает при создании CSS-перехода, то есть перед transition-delay. В то время transitionstart возникает, когда CSS-переход уже начался, то есть сразу после окончания transition-delay.

Событие transitionend возникает после завершения CSS-перехода. Если CSS-переход удаляется до завершения, то событие transitionend сгенерировано не будет. Например, если мы во время перехода удалили свойство transition или установили для элемента свойство display со значением none.

При удалении анимации до её завершения генерируется другое событие, а именно transitioncancel.

Добавление обработчиков к этим событиям осуществляется как обычно:

JavaScriptКопировать

el.addEventListener('transitionend', () => {

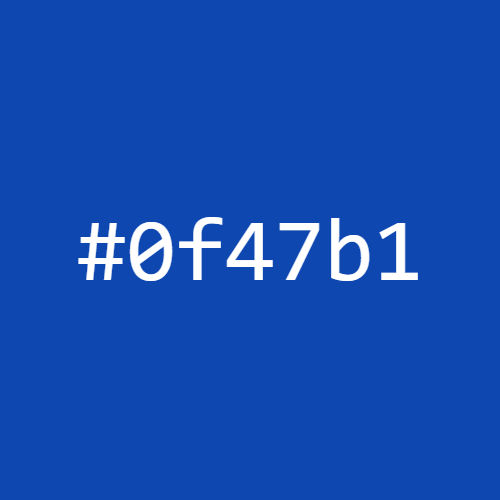
// действия, которые нужно выполнить при возникновении события transitionend на элементе el

});

Здесь мы добавили обработчик события transitionend с помощью метода addEventListener.

**Пример 1. Плавное изменение цвета фона элемента**

В этом примере создадим HTML-элемент .box. При нажатии на который мы будем с помощью JavaScript менять цвет его фона на случайный. Но изменение цвета будем производить не мгновенно, а плавно с помощью CSS-перехода:



Демо

HTML-код элемента:

HTMLКопировать

<div class="box">#f2c2b4</div>

Стили, с помощью которых мы зададим для .box определенные размеры, цвет фона, CSS-переход, а также некоторые другие вещи:

CSSКопировать

.box {

display: flex;

justify-content: center;

align-items: center;

width: 100px;

height: 100px;

margin: 20px;

color: black;

font-size: 18px;

font-family: monospace;

background-color: #f2c2b4;

cursor: pointer;

transition: background-color 0.5s ease-in-out 0.1s;

}

Код JavaScript, написанный с использованием ES6 классов:

JavaScriptКопировать

class ColorBox {

#el;

#isTransition = false;

constructor(selector) {

this.#el = document.querySelector(selector);

this.#addEventListeners();

}

#addEventListeners() {

this.#el.addEventListener('click', (e) => {

if (this.#isTransition) {

console.log('CSS-переход ещë не закончился', new Date().toLocaleTimeString());

return;

}

this.#isTransition = true;

const hex = `#${Math.floor(Math.random() \* 16777215).toString(16).padStart(6, '0')}`;

e.target.style.backgroundColor = hex;

const [r, g, b] = e.target.style.backgroundColor.replace(/[^\d,]/g, '').split(',');

const brightness = Math.floor((r \* 299 + g \* 587 + b \* 114) / 1000);

e.target.style.color = brightness < 128 ? 'white' : 'black';

e.target.textContent = hex;

});

this.#el.addEventListener('transitionrun', () => {

console.log('CSS-переход создан', new Date().toLocaleTimeString());

});

this.#el.addEventListener('transitionstart', () => {

console.log('CSS-переход начался', new Date().toLocaleTimeString());

});

this.#el.addEventListener('transitionend', () => {

this.#isTransition = false;

console.log('CSS-переход закончился', new Date().toLocaleTimeString());

});

}

}

const colorBox = new ColorBox('.box');

В коде мы обрабатываем следующие события, возникающие на элементе .box:

click – при клике;

transitionrun – при создании CSS-перехода;

transitionstart – при начале CSS-перехода;

transitionend – при окончании CSS-перехода.

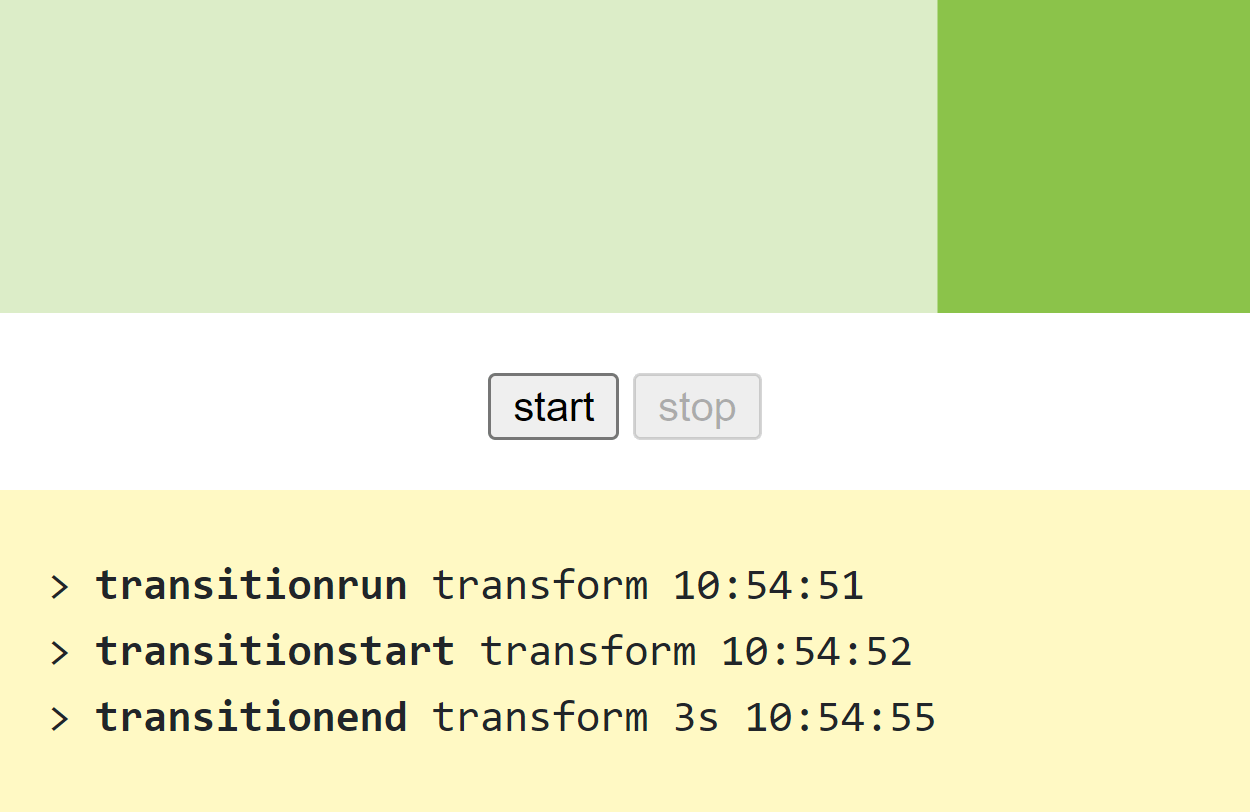
При клике мы проверяем, выполняется ли CSS-переход, то есть истинно ли значение свойства this.#isTransition. Если так, то выводим сообщение в консоль и прекращаем дальнейшее выполнение функции. В противном случае мы устанавливаем приватному свойствуthis.#isTransition значение true.

Далее мы генерируем случайный цвет и сохраняем его в переменную hex. После этого присваиваем CSS-свойству background-color значение hex. Затем мы объявляем переменные r, g и b и сохраняем в них компоненты соответственно красного, зеленого и синего цветов. Они нам понадобятся для определения яркости цвета и установки в зависимости от этого для текста черного или белого цвета. В конце с помощью textContent мы выведем текст цвета в формате HEX в качестве содержимого .box.

В обработчиках событий transitionrun, transitionstart и transitionend мы будем просто выводить соответствующее сообщение в консоль. Кроме этого в transitionend будем ещё дополнительно устанавливать приватному свойству this.#isTransition значение false, которое будет говорить о том, что CSS-переход завершился. И можно создавать новый при клике на .box.

**Пример 2. Плавное перемещение элемента**

В этом примере напишем код для плавного перемещения элемента с использованием translateX и transition:



Демо

Для этого нам понадобится вот такой HTML-код:

HTMLКопировать

<div class="box-wrapper container">

<div class="box"></div>

</div>

<div class="controls container">

<button type="button" class="start">start</button>

<button type="button" class="stop" disabled>stop</button>

</div>

<div class="logs container"></div>

Здесь .box-wrapper – это контейнер внутри которого мы будем плавно двигать элемент .box с помощью transform: translateX(300px) в течение 3 секунд. Запускать это действие будем при клике по кнопке «start», а в JavaScript в обработчике этого события будем добавлять к .box класс translating. Прервать анимацию, можно будет при клике по кнопке «stop».

В элемент .logs мы будем помещать события, связанные с CSS-переходом в порядке их возникновения.

Для оформления, а также для установки transform и transition напишем следующий CSS-код:

CSSКопировать

.container {

width: 400px;

margin-right: auto;

margin-bottom: 1rem;

margin-left: auto;

}

.box-wrapper {

background-color: #dcedc8;

}

.box {

width: 100px;

height: 100px;

background-color: #8bc34a;

}

.translating {

transform: translateX(300px);

transition: transform 3s ease-in-out 0.5s;

}

.controls {

margin-bottom: 1rem;

text-align: center;

}

.logs {

min-height: 100px;

padding: 20px 15px;

font-size: 14px;

font-family: monospace;

background-color: #fff9c4;

}

JavaScript-код:

JavaScriptКопировать

const elBox = document.querySelector('.box');

const elLogs = document.querySelector('.logs');

const btnStart = document.querySelector('.start');

const btnStop = document.querySelector('.stop');

btnStart.onclick = () => elBox.classList.add('translating');

btnStop.onclick = () => elBox.classList.remove('translating');

elBox.addEventListener('transitionrun', (e) => {

btnStart.disabled = true;

const now = new Date().toLocaleTimeString();

const html = `<div>> <b>${e.type}</b> ${e.propertyName} ${now}</div>`;

elLogs.innerHTML = html;

});

elBox.addEventListener('transitionstart', (e) => {

btnStop.disabled = false;

const now = new Date().toLocaleTimeString();

const html = `<div>> <b>${e.type}</b> ${e.propertyName} ${now}</div>`;

elLogs.insertAdjacentHTML('beforeend', html);

});

elBox.addEventListener('transitionend', (e) => {

btnStart.disabled = false;

btnStop.disabled = true;

const now = new Date().toLocaleTimeString();

const elapsedTime = Math.round(e.elapsedTime \* 100) / 100;

const html = `<div>> <b>${e.type}</b> ${e.propertyName} ${elapsedTime}s ${now}</div>`;

elLogs.insertAdjacentHTML('beforeend', html);

});

elBox.addEventListener('transitioncancel', (e) => {

btnStart.disabled = false; btnStop.disabled = true;

const now = new Date().toLocaleTimeString();

const elapsedTime = Math.round(e.elapsedTime \* 100) / 100;

const html = `<div>> <b>${e.type}</b> ${e.propertyName} ${elapsedTime}s ${now}</div>`;

elLogs.insertAdjacentHTML('beforeend', html);

});

**Что делает этот код?** При клике по кнопке .start мы будем добавлять к элементу .box класс translating и тем самым создавать CSS-переход. При клике по .stop убираем у элемента .box класс translating и тем самым удаляем CSS-переход.

Остальные обработчики событий просто выводят сообщения в элемент .logs со следующей информацией:

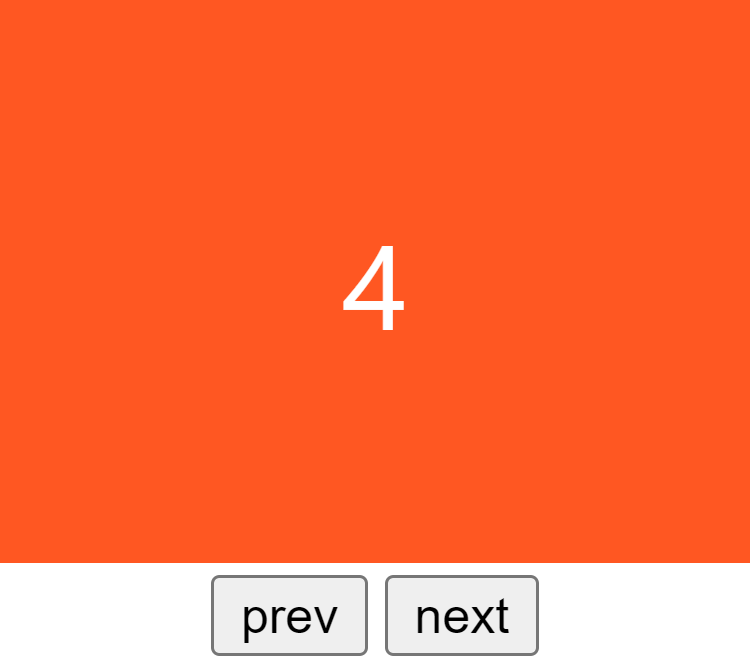
type – тип события;

propertyName – имя CSS-свойства для перехода;

elapsedTime – количество времени в секундах, в течение которого выполнялся переход.

**Пример 3. Слайдер с плавными переходами**

В этом примере мы создадим простой слайдер с зацикливанием или иными словами с бесконечной прокруткой.



Демо

Начнём с написания HTML-кода:

HTMLКопировать

<div class="slider-wrapper">

<div class="slider">

<div class="slider-item slider-item-active">1</div>

<div class="slider-item">2</div>

<div class="slider-item">3</div>

<div class="slider-item">4</div>

</div>

</div>

<div class="slider-controls">

<button type="button" class="slider-prev">prev</button>

<button type="button" class="slider-next">next</button>

</div>

Здесь .slider-item – это слайд. В этом слайдере их 4. Для смены слайдов будем использовать кнопки .slider-prev и .slider-next.

Для размещения элементов в слайдере будем использовать технологию CSS Flexbox. В качестве ширины установим .slider-item значение 200px:

CSSКопировать

.slider-wrapper {

width: 200px;

overflow-x: hidden;

}

.slider {

display: flex;

}

.slider-item {

flex: 0 0 200px;

height: 150px;

color: #fff;

display: flex;

align-items: center;

justify-content: center;

font-size: 2rem;

}

Смещать плавно слайды будем с помощью CSS-функции translateX и transition. Здесь мы будем их устанавливать посредством JavaScript:

JavaScriptКопировать

class Slider {

constructor(selector) {

this.slider = document.querySelector(selector);

this.elFirst = this.slider.firstElementChild;

this.elLast = this.slider.lastElementChild;

this.btnPrev = document.querySelector('.slider-prev');

this.btnNext = document.querySelector('.slider-next');

this.direction = 'next';

this.width = this.slider.parentElement.getBoundingClientRect().width;

this.count = this.slider.querySelectorAll('.slider-item').length;

this.step = this.width \* this.count;

this.elLast.style.transform = `translateX(${-this.step}px)`;

this.orders = [...this.slider.children].map((value, index) => index);

this.orders = [...this.orders.slice(-1), ...this.orders.slice(0, -1)];

this.addEventListener();

}

move() {

let value = new WebKitCSSMatrix(this.slider.style.transform).m41;

value += this.direction === 'next' ? -this.width : this.width;

this.slider.style.transform = `translateX(${value}px)`;

this.slider.style.transition = 'transform 1s ease';

const el = this.slider.querySelector('.slider-item-active');

let elNew = this.direction === 'next' ? el.nextElementSibling : el.previousElementSibling;

if (!elNew) {

elNew = this.direction === 'next' ? this.elFirst : this.elLast;

}

el.classList.remove('slider-item-active');

elNew.classList.add('slider-item-active');

}

addEventListener() {

this.btnPrev.onclick = () => {

this.direction = 'prev';

this.move();

}

this.btnNext.onclick = () => {

this.direction = 'next';

this.move();

}

this.slider.ontransitionrun = () => {

this.btnPrev.disabled = true;

this.btnNext.disabled = true;

const index = this.direction === 'next' ? this.orders[0] : this.orders[this.orders.length - 1];

const el = this.slider.querySelector(`.slider-item:nth-child(${index + 1})`);

let value = new WebKitCSSMatrix(el.style.transform).m41;

value += this.direction === 'next' ? this.step : -this.step;

el.style.transform = `translateX(${value}px)`;

if (this.direction === 'prev') {

this.orders = [...this.orders.slice(-1), ...this.orders.slice(0, -1)];

} else {

this.orders = [...this.orders.slice(1), ...this.orders.slice(0, 1)];

}

}

this.slider.ontransitionend = () => {

this.btnPrev.disabled = false;

this.btnNext.disabled = false;

}

}

}

const slider = new Slider('.slider');

Из интересного здесь то, что порядок следования элементов мы пишем в массив this.orders: [3, 0, 1, 2]. Который затем используем для того, чтобы организовать бесконечную прокрутку слайдера. При движении вправо, нам нужно крайний элемент слева переместить после последнего. А при движении влево, наоборот, крайний элемент справа переместить перед первым.

С помощью this.orders получить такие элементы очень легко:

this.orders[0] – крайний элемент слева;

this.orders[this.orders.length - 1] – крайний элемент справа.

Перемещение крайнего элемента для зацикливания мы делаем при создании CSS-перехода, то есть в обработчике события ontransitionrun. Здесь мы также переводим кнопки в неактивное состояние:

JavaScriptКопировать

this.btnPrev.disabled = true;

this.btnNext.disabled = true;

После завершения CSS-перехода делаем их опять активными:

JavaScriptКопировать

this.slider.ontransitionend = () => {

this.btnPrev.disabled = false;

this.btnNext.disabled = false;

}