Урок 6

Обработка событий в JavaScript

Понятие браузерного события. Асинхронное программирование.

Введение

Понятие события

Обработка нажатий

Реализация скрипта

Другие браузерные события

Действия браузера по умолчанию

Практикум. «Крестики-нолики»

Практическое задание

Дополнительные материалы

Используемая литература

Введение

На предыдущих уроках мы писали код, выполняющийся постепенно, зачастую — сверху вниз. И хотя мы уже применяли функции, объекты и методы, ход программы был достаточно линейным, предопределенным. А пользователь часто делает шаги по наитию, их невозможно предсказать. Его действия для браузера — это события: он может кликнуть по ссылке, загрузить данные. Но и сам браузер генерирует неявные события. На этом уроке научимся писать код, ориентированный на события.

Понятие события

Событие — это сигнал от браузера о том, что что-то произошло. При каждом нажатии кнопки, перемещении мыши, поступлении данных, изменении размеров окон генерируются события. И при новом подходе код получает возможность их обработать: выполнить действие в ответ на пользовательское. Не обязательно реагировать на все события, но всегда есть требующие этого.

Можно указать обработчик любому событию. Как правило, он представляет собой фрагмент кода, который мы определяем к выполнению, когда возникает событие. Обработчик — с точки зрения программного кода — это обычная функция. В данном контексте ее называют функцией-обработчиком. Чтобы обработчик можно было вызывать при возникновении события, его нужно зарегистрировать. Есть несколько способов — они зависят от типа события.

Рассмотрим событие загрузки страницы, которое упоминали на прошлой лекции. Оно генерируется в момент, когда браузер полностью загрузил и отобразил все содержимое страницы, построил модель DOM.

1. Прежде чем присваивать действие, надо его определить. По завершении загрузки будем писать в консоль сообщение «Загрузка завершена»:

```
function myProcessor(){
   console.log("Загрузка завершена");
}
```

2. Теперь надо создать связь, оповещая браузер о функции, которая будет вызываться при каждом возникновении события загрузки страницы. За это отвечает свойство **onload** у объекта **window**:

```
window.onload = myProcessor;
```

3. Теперь надо подключить скрипт к странице и открыть консоль, чтобы убедиться, что все работает корректно.

Обработка нажатий

Напишем более интересный скрипт, где загрузим фотогалерею с миниатюрами, которые располагаются в директории **/img/gallery/small/**. По клику по картинке будем показывать пользователю соответствующую большую картинку, расположенную в директории **/img/gallery/big**.

Большое изображение будет помещаться в специальную область вверху экрана, когда картинки будут находиться в нижней части.

Разметка будет выглядеть так:

```
<!doctype html>
<html lang="en">
<head>
      <meta charset="utf-8">
      <title> Image Gallery </title>
      <link rel="stylesheet" href="css/style.css" />
      <script></script>
</head>
<body>
      <div id="big_picture"></div>
      <div id="gallery">
            <img id="image_1" src="img/gallery/small/1.jpg">
            <img id="image_2" src="img/gallery/small/2.jpg">
            <img id="image_3" src="img/gallery/small/3.jpg">
      </div>
</body>
</html>
```

Теперь перейдем к скрипту, который реализует логику. Он будет состоять из следующих шагов:

1. Очистить **div big_picture** от содержимого.
2. Прочесть имя картинки, по которой произошел клик.
3. Найти соответствующую большую картинку.
4. Поместить ее в **div big_picture**.

Функцию по обновлению большой картинки мы поместим в отдельную функцию **changeBigPicture**. Чтобы добавить обработчик для событий щелчка на изображении, назначим созданную функцию свойству **onclick** элемента картинки:

```
var image = document.getElementById("image_1");
image.onclick = changeBigPicture;
```

Изображений много, и можно создать отдельный обработчик для каждого из них, но это неэффективное решение — изменять общую логику придется в каждом из обработчиков. Назначим функцию **changeBigPicture** обработчиком для всех изображений.

Чтобы изменить принцип работы, понадобится выборка не с **getElementById**, а при помощи **getElementByTagName**. Этот метод объекта **document** позволяет получать набор подходящих элементов по тегу. При этом надо будет перебрать полученную коллекцию в цикле, чтобы добавить обработчик события каждому элементу.

Метод возвращает объект, с которым будем работать как с массивом, хотя это объект типа **NodeList**. Он представляет собой набор элементов (узлов) дерева **DOM**. Чтобы перебрать эти элементы, надо узнать длину коллекции — она хранится в свойстве **length**. После этого в цикле последовательно обратимся к каждому элементу по индексу, совпадающему с номером итерации. В остальных аспектах с объектом **NodeList** нужно работать как с объектом, а не как с массивом.

```
var images = document.getElementsByTagName("img");
for (var i = 0; i < images.length; i++) {
   images[i].onclick = changeBigPicture;
}
```

Есть и другой способ: обработчик можно назначить прямо в разметке. Он указывается в атрибуте **on<событие>**. В нашем случае, чтобы прикрепить click-событие к картинке, нужно присвоить обработчик **onclick**:

```
<img onclick="changeBigPicture()" src="img/gallery/small/1.jpg">
```

При клике мышкой на кнопке выполнится код, указанный в атрибуте **onclick**. Но такой способ не очень удобен, так как надо вручную указывать этот атрибут для каждого тега.

При наступлении события обработчик получает сам его объект. Он передается для большинства событий, связанных с моделью DOM. Объект события содержит общую информацию о нем: что за элемент его породил, в какое время оно наступило и так далее. Передается и информация, связанная с конкретным событием: например, координаты точки щелчка мышью.

Так будет выглядеть обработчик:

```
function changeBigPicture(eventObj){
      console.log(eventObj);
}
```

Пока мы просто выводим в консоль информацию об объекте события. В свойстве **target** этого объекта сможем найти ссылку на объект-родитель **img**.

Реализация скрипта

Объединим весь код в единый скрипт и поместим весь вызов в функцию **init**, которая будет срабатывать только после полной загрузки страницы.

```
function init(){
      var images = document.getElementsByTagName("img");
      for (var i = 0; i < images.length; i++) {
            images[i].onclick = changeBigPicture;
      }
}
function changeBigPicture(eventObj){
      var appDiv = document.getElementById("big_picture");
      appDiv.innerHTML = "";
      var eventElement = eventObj.target;
      var imageNameParts = eventElement.id.split("_");
      var src = "img/gallery/big/" + imageNameParts[1] + ".jpg";
      var imageDomElement = document.createElement("img");
      imageDomElement.src = src;
      appDiv.appendChild(imageDomElement);
}
window.onload = init;
```

Теперь можем протестировать код и убедиться, что все работает корректно.

Есть и более современные способы установки обработчиков событий — это методы **addEventListener** и **removeEventListener**. Обработчик назначается вызовом **addEventListener** с тремя аргументами:

```
element.addEventListener(event, handler[, phase]);
```

event — имя события (например, **click**).

handler — ссылка на функцию, которую надо поставить обработчиком.

phase — необязательный аргумент, «фаза», на которой обработчик должен сработать. Он редко нужен, и мы его рассмотрим позже.

Обработчик удаляют вызовом **removeEventListener**:

```
element.removeEventListener(event, handler[, phase]);
```

При создании обработчиков нужно помнить, что в браузере у событий есть своя очередь, поэтому он обрабатывает их последовательно. И именно поэтому обработчики должны быть максимально емкими и эффективными. Иначе браузерная очередь событий переполнится, а самому браузеру придется обработать их все. Это затормозит работу страницы, а значит действия пользователя будут обрабатываться очень медленно.

Рассмотрим еще одну интересную особенность в работе событий. Есть структура:

```
<div onclick="alert('Обработчик для Div сработал!')">
  <p>Кликните сюда <code>CLICK</code>, а сработает обработчик для DIV</p>
</div>
```

Здесь есть странности: кликаем по **code**, а срабатывает обработчик на ** **. Дело в эффекте всплытия. Он заключается в том, что при наступлении события JS-обработчики сначала срабатывают на элементе, породившем его, затем на его родителе, а далее — выше и выше по цепочке вложенности. События как будто всплывают от внутреннего элемента до родителей.

Но тут есть и исключения. К примеру, событие **focus** не обладает эффектом всплытия.

Бывают ситуации, при которых можно использовать эффект всплытия, но необходимо будет знать, который из вложенных элементов породил событие.

В JS, где бы мы ни поймали событие, всегда можем узнать, кто именно его родитель. Порождающий элемент в данном случае называется целевым или исходным и доступен через **event.target**. При этом **this** внутри события все равно будет указывать на родительский элемент. В примере выше:

- **event.target** укажет на элемент **code**;
- **this** укажет на родительский **div**, на котором включился обработчик события.

На любой промежуточной стадии обработки можно решить, что событие полностью обработано, и остановить всплытие. Для этого необходимо вызвать метод **event.stopPropagation()**. Но делать это не рекомендуется, так как страдает прозрачность архитектуры.

Другие браузерные события

Нам знакомы два типа событий:

1. Событие загрузки (**load**), происходящее при завершении загрузки страницы браузером.
2. Событие щелчка (**click**), которое происходит, когда пользователь щелкает на элементе страницы.

При работе с JS происходит еще множество других событий: загрузка данных из сети, геопозиционирование, таймеры. Для событий обработчик зачастую назначался свойству — **onload**, **onclick**. Но не все события работают так же.

Рассмотрим хронометражные события. Чтобы задать их обработку, разработчик вызывает функцию **setTimeout**, в которой передает нужную функцию-обработчик:

```
function myChronoHandler() {
   console.log("Ты еще тут?");
}
setInterval(myChronoHandler, 5000);
```

Этот код реализует таймер, который будет срабатывать каждые пять минут, выводя в консоль указанное сообщение.

При загрузке страницы выполняем две операции: назначаем обработчик с именем **myChronoHandler** и вызываем функцию **setTimeout** для создания события таймера, которое будет сгенерировано через 5000 миллисекунд. Когда оно сработает, обработчик будет выполнен. Ожидая таймер, браузер продолжает работать в обычном режиме. Когда отсчет времени достигает нуля, браузер вызывает обработчик.

Обратите внимание, что здесь мы передаем одной функции другую в качестве аргумента. Функция **setTimeout** во время работы создает таймер обратного отсчета и связывает с ним указанный в аргументах обработчик, который будет вызываться при обнулении таймера. Чтобы сообщить **setTimeout**, кого следует вызывать, нужно передать ей ссылку на функцию-обработчик. Сама **setTimeout** сохраняет ссылку, чтобы использовать ее позднее, когда сработает таймер.

Вопрос: если **setTimeout** нигде не определена, как мы к ней так просто обращаемся? Формально можно было бы использовать запись **window.setTimeout**, но поскольку объект **window object** считается глобальным, то можно опустить имя **window** и применить сокращенную форму **setTimeout**, которая часто встречается на практике. Хотя с **onload** так обычно не делают, так как это имя достаточно распространено среди свойств у других элементов.

В этом коде одни обработчики создают другие. Такой стиль построения кода называется асинхронным программированием. В отличие от прошлых программ, здесь мы не пытаемся писать строгий алгоритм, который выполняется в заданной последовательности. Мы подключаем обработчики, которые управляют процессом выполнения программы по мере того, как возникают события.

Многие события относятся к группе событий **DOM** (щелчок на элементе, наведение курсора) или событий таймеров (создаются вызовами **setTimeout**). Но наравне с ними существуют события API (JavaScript API, Geolocation, LocalStorage, Web Workers). Еще есть категория событий, относящихся к вводу/выводу — например, запрос данных от веб-служб с использованием **XmlHttpRequest** или **Web Sockets** (изучаются на продвинутом курсе JavaScript).

Действия браузера по умолчанию

Многие события автоматически влекут за собой действия браузера:

- клик по ссылке инициирует переход на новый URL;
- нажатие на кнопку «Отправить» в форме — отсылку ее на сервер;
- двойной клик на тексте выделяет его.

При обработке события в JavaScript зачастую такое действие браузера не нужно — и его можно отменить.

Есть два способа. Основной — воспользоваться объектом события. Для отмены действия браузера существует стандартный метод **event.preventDefault()**.

Если же обработчик назначен через on-событие, а не через **addEventListener**, то можно просто вернуть из него **false**.

В следующем примере при клике по ссылке переход не произойдет:

```
<a href="/" onclick="return false">Нажми здесь</a>
<a href="/" onclick="event.preventDefault()">здесь</a>
```

Действий браузера по умолчанию достаточно много. Примеры событий, которые их вызывают:

- **mousedown** — нажатие кнопкой мыши, когда курсор находится на тексте, начинает его выделять;
- **click** на **<input type="checkbox">** ставит или убирает галочку;
- **submit** — при нажатии на **<input type="submit">** в форме данные отправляются на сервер;
- **wheel** — движение колесика мыши инициирует прокрутку;
- **keydown** — при нажатии клавиши в поле ввода появляется символ;
- **contextmenu** — при правом клике показывается контекстное меню браузера.

Все эти действия можно отменить, если хотим обработать событие исключительно при помощи JavaScript.

Практикум. «Крестики-нолики»

Реализуем эту известную игру в браузере. По клику на ячейку в ней будет проставляться символ игрока, который в данный момент ходит. Игра идет по схеме ***Hot seat***.

Самым интересным будет определять победителя. Это должно происходить после каждого хода — то есть в зависимости от события.

Практическое задание

1. Продолжаем реализовывать модуль корзины:
    a. Добавлять в объект корзины выбранные товары по клику на кнопке «Купить» без перезагрузки страницы;
    b. Привязать к событию покупки товара пересчет корзины и обновление ее внешнего вида.
2. * У товара может быть несколько изображений. Нужно:
    a. Реализовать функционал показа полноразмерных картинок товара в модальном окне;
    b. Реализовать функционал перехода между картинками внутри модального окна.

Дополнительные материалы

1. Кастомные события в JS.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Дэвид Флэнаган. JavaScript. Подробное руководство.
2. Эрик Фримен, Элизабет Робсон. Изучаем программирование на JavaScript.