Урок 6

Обработка событий в JavaScript.

Понятие браузерного события. Асинхронное программирование.

Введение

На предыдущих уроках мы писали код, выполняющийся постепенно, зачастую – сверху вниз. Несмотря на то, что мы уже применяли функции, объекты и методы, ход программы был достаточно линеен, предопределён. Пользователь в свою очередь часто делает шаги по наитию, их невозможно предсказать. Т.е. действия пользователя для браузера являются некими событиями – пользователь может кликнуть по ссылке, загрузить данные и т.д. Помимо этого и сам браузер является генератором неявных событий. И в этой лекции мы рассмотрим как писать код, ориентированный на события.

Что такое «событие»

Событие – это сигнал от браузера о том, что что-то произошло. Каждый раз, когда происходит нажатие кнопки, перемещение мыши, поступление данных, изменение размеров окон, генерируются события. И при новом подходе Ваш код получает возможность это событие обработать, т.е. выполнить некое действие при определённом действии со стороны пользователя. Совершенно не обязательно реагировать на все события, но у Вас точно появится необходимость реагировать на те или иные действия.
Есть возможность указать обработчик любому событию. Как правило, он представляет собой некий фрагмент кода, который мы определяем к выполнению при возникновении события. С точки зрения программного кода обработчик – это обычная функция. В данном контексте её называют функция-обработчик. Для того, чтобы Ваш обработчик мог вызываться при возникновении события, его нужно зарегистрировать. Способов регистрации обработчика несколько – они зависят от типа события. Давайте попробуем создать обработчик на событие, которое возникает при загрузке события.
В прошлых лекциях мы сталкивались с событиями, но не разбирали их. Например, это событие загрузки страницы из прошлой лекции. Оно генерируется в момент, когда браузер полностью загрузил и отобразил всё содержимое страницы, построил модель DOM.

1. Перед тем, как присваивать какое-то действие, надо его определить. Предположим, что по завершению загрузки мы будем писать в консоль сообщение «Загрузка завершена».

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

2. Теперь мы должны 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 просто подключить скрипт к странице и открыть консоль, чтобы убедиться, что все работает корректно.

Обработка нажатий

Теперь давайте напишем более интересный скрипт. Мы загружаем фотогалерею с миниатюрами, которые располагаются в некой директории /img/gallery/small/. По клику по картинке пользователю будет показываться соответствующая большая картинка, расположенная в директории /img/gallery/big.
Большая картинка будет помещаться в специальную область вверху экрана, когда изображения будут находиться в нижней части.

Разметка будет выглядеть вот 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 перейдём к скрипту, который реализует нашу 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 её в div big_picture.

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

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

Изображений у нас много. Разумеется, можно создать отдельный обработчик для каждого из них, но это неэффективное решение, а изменять общую логику придётся в каждом из обработчиков. Таким образом, мы назначим нашу функцию changeBigPicture обработчиком для всех изображений.
Для изменения принципа работы нам понадобится выборка не при помощи getElementById, а при помощи getElementByTagName. Этот метод объекта document позволяет нам получать набор подходящих элементов по тегу. При этом надо будет перебрать полученную коллекцию в цикле, чтобы добавить обработчик события каждому элементу.
Метод возвращает нам объект, с которым мы будем работать как с массивом. Но это не массив, а также объект типа NodeList. Такой объект представляет собой набор элементов (узлов) дерева DOM. Для выполнения перебора этих элементов мы должны узнать длину коллекции (она хранится в свойстве length), после чего в цикле последовательно обратиться 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 может быть назначен прямо в разметке. Он указывается в атрибуте, который называется on<событие>. В нашем случае, чтобы прикрепить click-событие к картинке, нужно присвоить обработчик onclick, вот так:

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

При клике мышкой на кнопке выполнится код, указанный в атрибуте onclick. Но в силу того, что надо вручную указывать этот атрибут для каждого тега, такой способ не очень удобен.
При наступлении события обработчик получает сам объект события. Вообще, объект передаётся для большинства событий, связанных с моделью DOM. Объект события содержит общую информацию о событии: что за элемент породил событие, в какое время оно наступило и т. д. Также передаётся информация, связанная с конкретным событием: например, для щелчка мышью вы получите координаты точки щелчка.
Давайте посмотрим, как будет выглядеть наш обработчик.

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

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

Реализация скрипта

Теперь объединим весь код в единый скрипт. Также мы 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 мы можем протестировать наш код и убедиться, что все работает корректно.
Также существуют и более современные способы установки обработчиков событий – это методы addEventListener и removeEventListener. Назначение обработчика осуществляется 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. Этот аргумент редко нужен, мы его рассмотрим позже.

Удаление обработчика осуществляется вызовом removeEventListener:

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

При разработке обработчиков событий нужно помнить, что в браузере события имеют свою очередь, поэтому он обрабатывает события последовательно. И именно поэтому Ваши обработчики должны быть максимально ёмкими и эффективными. Иначе браузерная очередь событий может переполниться, а сам браузер столкнётся с необходимостью обработать их все. Это приведёт к замедлению работы Вашей страницы, а значит, действия пользователя будут обрабатываться очень медленно.
Есть и ещё одна интересная особенность в работе событий. К примеру, у нас есть структура:

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

Странно, не правда ли? Кликаем мы по code, а срабатывает обработчик на . Дело тут в эффекте всплытия. Он заключается в том, что при наступлении некоего события JS-обработчики сначала срабатывают на элементе, породившем событие, затем на его родителе, затем выше и выше вверх по цепочке вложенности. Т.е. события как будто всплывают от внутреннего элемента до родителей, подобно пузырьку воздуха в воде.

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

Вполне вероятно, что у Вас возникнет ситуация, при которой Вы будете пользоваться эффектом всплытия, но Вам необходимо будет знать, кто из вложенных элементов породил событие.
В JS, где бы мы ни поймали событие, мы всегда можем узнать, кто именно его породил. 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

Также на любой промежуточной стадии обработки мы можем решить, что событие полностью обработано, и остановить всплытие. Для этого необходимо вызвать метод event.stopPropagation(). Однако делать это не рекомендуется, т.к. это портит прозрачность архитектуры.

Другие браузерные события

На данный момент мы видели два типа событий:

1. событие загрузки (load), происходящее при завершении загрузки страницы браузером;
2. событие щелчка (click), которое происходит, когда пользователь щёлкает на элементе страницы;

При работе с JS в будущем Вы гарантированно столкнётесь со множеством других событий. Это могут быть события загрузки данных из сети, геопозиционирования, таймеров. Для всех событий обработчик всегда назначался некоторому свойству — onload, onclick. Однако не все события работают также.
Рассмотрим хронометражные события. Для задания их обработки разработчик всего лишь вызывает функцию setTimeout, в которой передаёт ей нужную функцию-обработчик.

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

Приведённый выше код реализует таймер, который будет срабатывать каждые 5 минут, выводя в консоль указанное сообщение.
При загрузке страницы мы выполняем две операции: назначаем обработчик с именем myChronoHandler и вызываем функцию setTimeout для создания события таймера, которое будет сгенерировано через 5000 миллисекунд. Когда сработает событие таймера, обработчик будет выполнен. Во время ожидания таймера браузер продолжает работать как обычно. Когда отсчёт времени доходит до нуля, браузер вызывает обработчик.
Также обратите внимание, что здесь мы передаём функции другую функцию в качестве аргумента. Функция setTimeout во время работы создаёт таймер обратного отсчета, связывает с ним указанный в аргументах обработчик, который будет вызываться при обнулении таймера. Для того, чтобы сообщить setTimeout, кого следует вызывать, нам нужно передать ей ссылку на функцию-обработчик. Сам setTimeout сохраняет ссылку, чтобы использовать ее позднее, когда сработает таймер.

«Но ведь setTimeout у нас нигде не определён. «Как же мы к нему так просто обращаемся?» - спросите Вы. Формально мы могли бы использовать запись window.setTimeout, но поскольку объект window object считается глобальным, то мы можем опустить имя window и применять сокращённую форму setTimeout, которая часто встречается на практике. Хотя с тем же onload так обычно не делают, т.к. это имя является достаточно распространённым именем свойства у других элементов.

Как видите, в этом коде одни обработчики создают другие. Этот стиль построения кода называется асинхронное программирование. Т.е., в отличие от наших прошлых программ, мы не пытаемся писать строгий алгоритм, который выполняется в заданной последовательности действий. Мы подключаем обработчики событий, которые управляют процессом выполнения по мере возникновения событий.

Очень много событий относится к группе событий DOM (это щелчок на элементе, наведение курсора) или событий таймеров (создаются вызовами setTimeout). Но наравне с ними существуют события некоторых API (JavaScript API, Geolocation, LocalStorage, Web Workers). Также существует целая категория событий, относящихся к вводу/выводу, например, запрос данных от веб-служб с использованием XmlHttpRequest или Web Sockets (Изучаются на курсе JavaScript Level 2).

Действия браузера по умолчанию

Многие события автоматически влекут за собой действие браузера. Например:

- Клик по ссылке инициирует переход на новый URL.
- Нажатие на кнопку «отправить» в форме – отсылку ее на сервер.
- Двойной клик на тексте – инициирует его выделение.

Если мы обрабатываем событие в JavaScript, то, зачастую, такое действие браузера нам не нужно. К счастью, его можно отменить.
Есть два способа отменить действие браузера. Основной способ – это воспользоваться объектом события. Для отмены действия браузера существует стандартный метод event.preventDefault().
Если же обработчик назначен через on-событие (не через addEventListener), то можно просто вернуть false из обработчика.
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.
Вот некоторые примеры событий, которые вызывают действие браузера:

- mousedown – нажатие кнопкой мыши в то время как курсор находится на тексте начинает его выделение.
- click на <input type="checkbox"> – ставит или убирает галочку.
- submit – при нажатии на <input type="submit"> в форме данные отправляются на сервер.
- wheel – движение колёсика 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 мы хотим обработать событие исключительно при помощи JavaScript.

Практикум. Крестики-нолики

Всем хорошо известна эта игра. Мы реализуем её в браузере. По клику на ячейку в ней будет проставляться символ того игрока, который в данный момент ходит. Т.е. игра идёт по схеме “Hot seat”. Самым интересным будет определение победителя. Оно должно происходить после каждого хода. Т.е. оно зависит от события.

Домашнее задание

1. Продолжаем реализовывать модуль корзины.
    a. Теперь нужно добавлять в объект корзины выбранные товары по клику по кнопке “Купить” у товара без перезагрузки страницы.
    b. Привязать к событию покупки товара пересчёт корзины и обновление её 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. https://habrahabr.ru/post/229189/ - свои собственные события в JS

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

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