Загрузка документа и ресурсов

DOMContentLoaded

Событие DOMContentLoaded срабатывает на объекте **document** (document.addEventListener("DOMContentLoaded", func)). DOMContentLoaded – браузер полностью загрузил HTML, было построено DOM-дерево, но внешние ресурсы, такие как картинки <img> и стили, могут быть ещё не загружены.

Когда браузер обрабатывает HTML-документ и встречает тег **<script>,** он должен выполнить его перед тем, как продолжить строить DOM. Это делается на случай, если скрипт захочет изменить DOM или даже дописать в него (document.write), так что DOMContentLoaded должен подождать. Исключения: скрипты с атрибутом **async** и скрипты, сгенерированные динамически при помощи **document.createElement('script')** и затем добавленные на страницу

Внешние таблицы стилей не затрагивают DOM, поэтому DOMContentLoaded их не ждёт. Но здесь есть подводный камень. Если после стилей есть скрипт, то этот скрипт должен дождаться, пока загрузятся стили. Так как DOMContentLoaded дожидается скриптов, то теперь он так же дожидается и стилей перед ними.

Firefox, Chrome и Opera делают **автозаполнение** полей (например, логин/пароль) при наступлении DOMContentLoaded.

load

load – браузер загрузил HTML и внешние ресурсы (картинки, стили и т.д.). Событие наступает на объекте **window** (window.onload). Размеры картинок верные.

beforeunload/unload

Оба события - на объекте **windows** (window.onunload, window.onbeforeunload
)
beforeunload/unload – пользователь покидает страницу. Мы можем проверить, сохранил ли он изменения и спросить, на самом ли деле он хочет уйти. Если мы отменим это событие, то браузер спросит посетителя, уверен ли он.

```
window.onbeforeunload = function() {
  return false;
};
```

По историческим причинам возврат непустой строки так же считается отменой события. Когда-то браузеры использовали её в качестве сообщения, но, как указывает современная спецификация, они не должны этого делать (потому что некоторые веб-разработчики злоупотребляли этим обработчиком события, показывая вводящие в заблуждение и надоедливые сообщения).

unload – пользователь почти ушёл, но мы всё ещё можем запустить некоторые операции, например, отправить статистику.

Предположим, мы собрали данные о том, как используется страница: клики, прокрутка, просмотры областей страницы и так далее и хотим сохранить эти данные. Для этого существует специальный метод navigator.sendBeacon(url, data), описанный в спецификации https://w3c.github.io/beacon/. Он посылает данные в фоне. Переход к другой странице не задерживается: браузер покидает страницу, но всё равно выполняет sendBeacon.

```
let analyticsData = { /* объект с собранными данными */ };
window.addEventListener("unload", function() {
  navigator.sendBeacon("/analytics", JSON.stringify(analyticsData));
});
```

Событие readystatechange, свойство document.readyState

Свойство document.readyState показывает нам текущее состояние загрузки.
Есть три возможных значения:

- "**loading**" – документ загружается.
- "**interactive**" – документ был полностью прочитан.
- "**complete**" – документ был полностью прочитан и все ресурсы (такие как изображения) были тоже загружены.

Также есть событие readystatechange, которое генерируется при изменении состояния, так что мы можем вывести все эти состояния таким образом:

```
// текущее состояние
console.log(document.readyState);
// вывести изменения состояния
document.addEventListener('readystatechange', () => console.log(document.readyState))
```

Скрипты: async, defer

В современных сайтах скрипты обычно «тяжелее», чем HTML: они весят больше, дольше обрабатываются.
Когда браузер загружает HTML и доходит до тега <script>...</script>, он не может продолжать строить DOM. Он должен сначала выполнить скрипт. То же самое происходит и с внешними скриптами <script src="..."></script>: браузер должен подождать, пока загрузится скрипт, выполнить его, и только затем обработать остальную страницу.

Это ведёт к двум важным проблемам:

- Скрипты не видят DOM-элементы ниже себя, поэтому к ним нельзя добавить обработчики и т.д.
- Если вверху страницы объёмный скрипт, он «блокирует» страницу. Пользователи не видят содержимое страницы, пока он не загрузится и не запустится

Конечно, есть пути, как это обойти. Например, мы можем поместить скрипт внизу страницы. Тогда он сможет видеть элементы над ним и не будет препятствовать отображению содержимого страницы. Но это решение далеко от идеального. Например, браузер замечает скрипт (и может начать загружать его) только после того, как он полностью загрузил HTML-документ. В случае с длинными HTML-страницами это может создать заметную задержку

К счастью, есть два атрибута тега <script>, которые решают нашу проблему: defer и async

defer

Атрибут defer сообщает браузеру, что он должен продолжать обрабатывать страницу и загружать скрипт в фоновом режиме, а затем запустить этот скрипт, когда DOM дерево будет полностью построено. Скрипты с defer всегда выполняются, когда дерево DOM готово, но до события **DOMContentLoaded**.

Отложенные с помощью defer скрипты **сохраняют порядок** относительно друг друга, как и обычные скрипты. Поэтому, если сначала загружается большой скрипт, а затем меньшего размера, то последний будет ждать.

Атрибут defer будет проигнорирован, если в теге <script> **нет src**.

На практике defer используется для скриптов, которым требуется доступ ко всему DOM и/или важен их относительный порядок выполнения.

async

Атрибут async означает, что скрипт абсолютно независим:

- Страница не ждёт асинхронных скриптов, содержимое обрабатывается и отображается.
- Событие **DOMContentLoaded** и асинхронные скрипты не ждут друг друга:
- DOMContentLoaded может произойти как до асинхронного скрипта (если асинхронный скрипт завершит загрузку после того, как страница будет готова),…так и после асинхронного скрипта (если он короткий или уже содержится в HTTP-кеше)
- Остальные скрипты не ждут async, и скрипты с async не ждут другие скрипты.
- Содержимое страницы отображается сразу же : async его не блокирует.

async хорош для независимых скриптов, например счётчиков и рекламы, относительный порядок выполнения которых не играет роли.

Динамически загружаемые скрипты

Мы можем также добавить скрипт и динамически, с помощью JavaScript:

```
let script = document.createElement('script');
script.src = "/article/script-async-defer/long.js";
document.body.append(script); // (*)
```

Скрипт начнёт загружаться, как только он будет добавлен в документ (*).
Динамически загружаемые скрипты по умолчанию ведут себя как «**async**».То есть:

- Они никого не ждут, и их никто не ждёт.
- Скрипт, который загружается первым – запускается первым (в порядке загрузки).

Мы можем изменить **относительный порядок скриптов** с «первый загрузился – первый выполнился» на порядок, в котором они идут в документе (как в обычных скриптах) с помощью явной установки свойства **async в false**:

```
let script = document.createElement('script');
script.src = "/article/script-async-defer/long.js";
script.async = false;
document.body.append(script);
```

Загрузка ресурсов: onload и onerror

Браузер позволяет отслеживать загрузку сторонних ресурсов: скриптов, ифреймов, изображений и др. Для этого существуют два события:
load – успешная загрузка,
error – во время загрузки произошла ошибка.

script.onload - срабатывает после того, как скрипт был загружен и выполнен.
script.onerror - ошибки, которые возникают во время загрузки скрипта.
Обработчики onload/onerror отслеживают только сам процесс загрузки. Ошибки обработки и выполнения загруженного скрипта ими не отслеживаются. Чтобы «поймать» ошибки в скрипте, нужно воспользоваться глобальным обработчиком window.onerror

События load и error также срабатывают и для других ресурсов, а вообще, для любых ресурсов, у которых есть внешний **src**.

Большинство ресурсов начинают загружаться после их добавления в документ. За исключением тега <**img**>. Изображения начинают загружаться, когда получают src (*).
Для <**iframe**> событие **load** срабатывает по окончании загрузки **как в случае успеха, так и в случае ошибки**.

Есть правило: **скрипты с одного сайта не могут получить доступ к содержимому другого сайта**. Например, скрипт с https://facebook.com не может прочитать почту пользователя на https://gmail.com. Чтобы разрешить **кросс-доменный доступ**, нам нужно поставить тегу <script> атрибут crossorigin, и, кроме того, удалённый сервер должен поставить **специальные заголовки**.