Загрузка документа и ресурсов

DOMContentLoaded – браузер полностью загрузил **HTML**, было построено DOM-дерево, но внешние ресурсы, такие как картинки <img> и стили, могут быть ещё не загружены. Н-р:

```
document.addEventListener("DOMContentLoaded", ready);
```

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.

Есть два исключения из этого правила:

- Скрипты с атрибутом async, который мы рассмотрим немного позже, не блокируют DOMContentLoaded.
- Скрипты, сгенерированные динамически при помощи document.createElement('script') и затем добавленные на страницу, также не блокируют это событие.

Внешние таблицы стилей не затрагивают DOM, поэтому DOMContentLoaded их не ждёт.
Но здесь есть подводный камень. Если после стилей у нас есть скрипт, то этот скрипт должен дождаться, пока загрузятся стили:

```
<link type="text/css" rel="stylesheet" href="style.css">
<script>
 // скрипт не выполняется, пока не загрузятся стили
 alert(getComputedStyle(document.body).marginTop);
</script>
```

Причина в том, что скрипту может понадобиться получить координаты или другие свойства элементов, зависящих от стилей, как в примере выше. Естественно, он должен дождаться, пока стили загрузятся.
Так как DOMContentLoaded дожидается скриптов, то теперь он так же дожидается и стилей перед ними.

Firefox, Chrome и Opera **автоматически заполняют поля** при наступлении DOMContentLoaded. Например, если на странице есть форма логина и пароля и браузер запомнил значения, то при наступлении DOMContentLoaded он попытается заполнить их (если получил разрешение от пользователя). Так что, если DOMContentLoaded откладывается из-за долгой загрузки скриптов, в свою очередь – откладывается **автозаполнение**.

load – браузер загрузил **HTML и внешние ресурсы** (картинки, стили и т.д.). Н-р, размеры картинок будут показаны правильно:

```
<script>
 window.onload = function() {
  alert('Страница загружена');
  // к этому моменту страница загружена
  alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
 };
</script>
<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

beforeunload/unload – пользователь покидает страницу. Когда посетитель покидает страницу, на объекте window генерируется событие **unload**. В этот момент стоит совершать простые действия, не требующие много времени, вроде закрытия связанных всплывающих окон. Обычно здесь отсылают статистику. Для этого существует специальный метод **navigator.sendBeacon(url, data)**, описанный в спецификации https://w3c.github.io/beacon/. Он посылает данные в фоне. Его можно использовать вот так:

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

Если посетитель собирается уйти со страницы или закрыть окно, обработчик **beforeunload** попросит дополнительное подтверждение. Если мы отменим это событие, то браузер спросит посетителя, уверен ли он.
Вы можете попробовать это, запустив следующий код и затем перезагрузив страницу:

```
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. Поведение было изменено, потому что некоторые веб-разработчики злоупотребляли этим обработчиком события, показывая вводящие в заблуждение и надоедливые сообщения.

readyState

Что произойдёт, если мы установим обработчик DOMContentLoaded после того, как документ загрузился? Естественно, он никогда не запустится.
Есть случаи, когда мы не уверены, готов документ или нет. Мы бы хотели, чтобы наша функция исполнилась, когда DOM загрузился, будь то сейчас или позже.

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.

Так что мы можем проверить document.readyState и, либо установить обработчик, либо, если документ готов, выполнить код сразу же. Например, вот так:

```
function work() { /*...*/ }
if (document.readyState == 'loading') {
  // ещё загружается, ждём события
  document.addEventListener('DOMContentLoaded', work);
} else {
  // DOM готов!
  work();
}
```

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));
```

Событие readystatechange – альтернативный вариант отслеживания состояния загрузки документа, который появился очень давно. На сегодняшний день он используется редко.

Скрипты: async, defer

	Порядок	`DOMContentLoaded`
`async`	*Порядок загрузки* (кто загрузится первым, тот и сработает).	Не имеет значения. Может загрузиться и выполниться до того, как страница полностью загрузится. Такое случается, если скрипты маленькие или хранятся кеше, а документ достаточно большой.
`defer`	*Порядок документа* (как расположены в документе).	Выполняется после того, как документ загружен и обработан (ждёт), непосредственно перед **`DOMContentLoaded`.**

Атрибут **defer** сообщает браузеру, что он должен продолжать обрабатывать страницу и загружать скрипт в фоновом режиме, а затем запустить этот скрипт, когда он загрузится. На практике defer используется для скриптов, которым требуется доступ ко всему DOM и/или важен их относительный порядок выполнения. Атрибут defer предназначен только для ***внешних скриптов***

Атрибут **async** означает, что скрипт абсолютно независим. 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);
```

onload и onerror

Изображения <img>, внешние стили, скрипты и другие ресурсы (у которых есть src) предоставляют события load и error для отслеживания загрузки:
load срабатывает при успешной загрузке,
error срабатывает при ошибке загрузки.
Единственное исключение – это **<iframe>:** по историческим причинам срабатывает всегда load вне зависимости от того, как завершилась загрузка, даже если страница не была найдена.
Например:

```
let script = document.createElement('script');
// мы можем загрузить любой скрипт с любого домена
script.src = "https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"
document.head.append(script);
script.onload = function() {
  // в скрипте создаётся вспомогательная функция с именем "_"
  alert(_); // функция доступна
};
script.onerror = function() {
  alert("Error loading " + this.src); // Ошибка загрузки https://example.com/404.js
};
```

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.
Или, если быть более точным, один источник (домен/порт/протокол) не может получить доступ к содержимому с другого источника. Даже поддомен или просто другой порт будут считаться разными источниками, не имеющими доступа друг к другу. Чтобы разрешить кросс-доменный доступ, нам нужно поставить тегу <script> атрибут crossorigin, и, кроме того, удалённый сервер должен поставить специальные заголовки.