



JS
AJAX
CGI

В прошлых сериях...

- JS – скриптовый язык. Скрипты могут встраиваться в HTML и выполняться браузером.
- Типы данных: `number`, `BigInt`, `Boolean`, `string`, `symbol`, `null`, `undefined`, `object`
- Стрелочные функции имеют вид:
`let func = (arg1, arg2, ...argN) => expression`
- Объект – составной тип данных. Объект может быть создан с помощью фигурных скобок `{...}` с необязательным списком свойств.
- «Символ» представляет собой уникальный идентификатор
- Значение `this` вычисляется во время выполнения кода и зависит от контекста.
Привязка контекста: `let bound = func.bind(context, [arg1], [arg2], ...);`
Вызов функции с определенным контекстом: `func.call(context, ...args)`
- У массивов есть такие методы как `.forEach()`, `.find()`, `.filter()`, `.map()`, `.reduce()`

Прототипное наследование

- В JavaScript объекты имеют специальное скрытое свойство `[[Prototype]]` (так оно названо в спецификации), которое либо равно `null`, либо ссылается на другой объект. Этот объект называется «прототип»
- При попытке прочитать отсутствующее свойство из object JavaScript автоматически берёт его из прототипа.
- Свойство `[[Prototype]]` является внутренним и скрытым, но есть много способов задать его. Одним из них является использование `__proto__`, например так:

```
let animal = {  
    eats: true  
};  
  
let rabbit = {  
    jumps: true  
};  
  
rabbit.__proto__ = animal;
```
- Цикл `for..in` проходит не только по собственным, но и по унаследованным свойствам объекта. Если унаследованные свойства не нужны, то можно отфильтровать их при помощи встроенного метода `obj.hasOwnProperty(key)`



F.prototype

- Если в F.prototype содержится объект, оператор new устанавливает его в качестве [[Prototype]] для нового объекта.
- Обратите внимание, что F.prototype означает обычное свойство с именем "prototype" для F. Это ещё не «прототип объекта», а обычное свойство F с таким именем.
- У каждой функции по умолчанию уже есть свойство "prototype".
- По умолчанию "prototype" – объект с единственным свойством constructor, которое ссылается на функцию-конструктор.

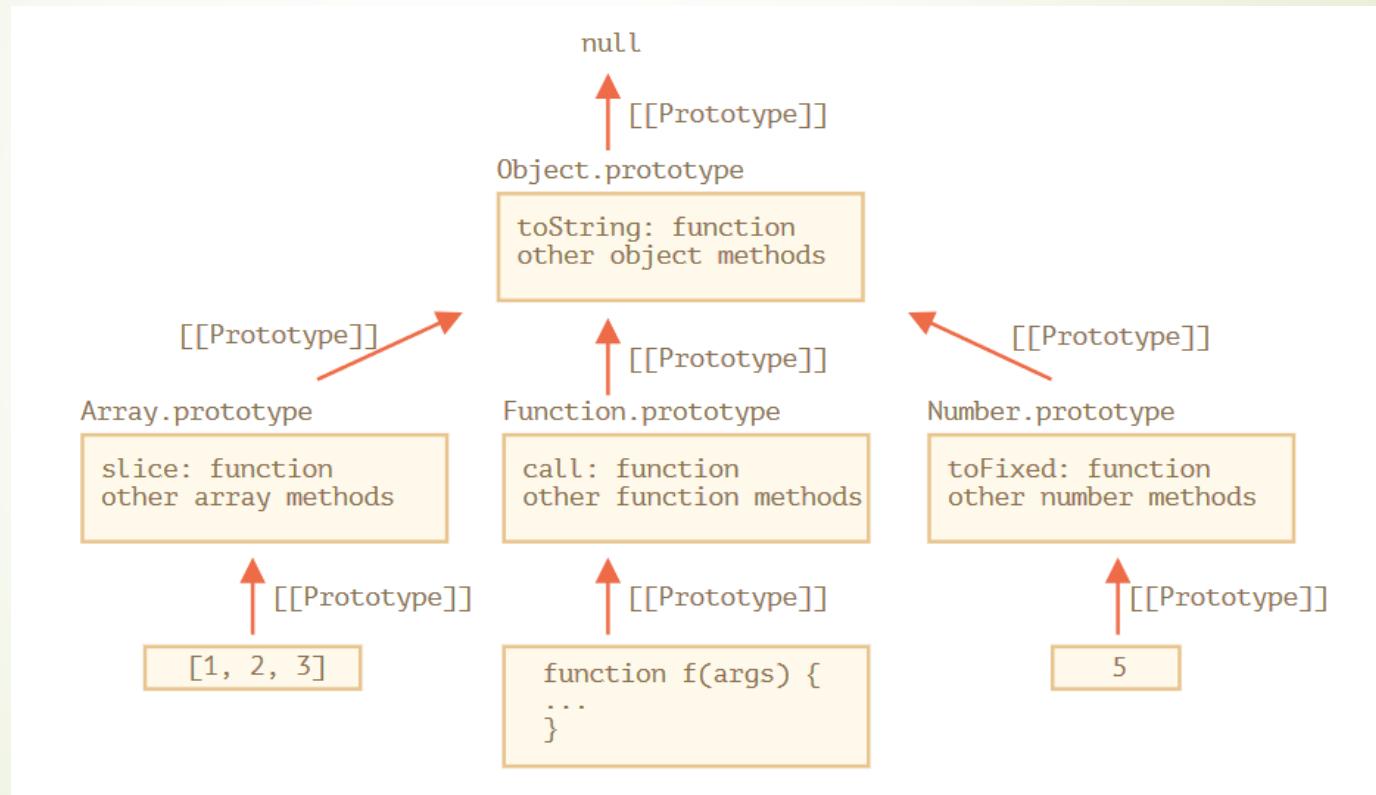
- 
- Можно использовать свойство `constructor` существующего объекта для создания нового.

- ```
function Rabbit(name) {
 this.name = name;
 alert(name);
}
```

```
let rabbit = new Rabbit("White Rabbit");
```

```
let rabbit2 = new rabbit.constructor("Black Rabbit");
```

# Схема прототипного наследования



- 
- Все встроенные объекты следуют одному шаблону:

Методы хранятся в прототипах (`Array.prototype`, `Object.prototype`, `Date.prototype` и т.д.).

Сами объекты хранят только данные (элементы массивов, свойства объектов, даты).

- Примитивы также хранят свои методы в прототипах объектов-обёрток: `Number.prototype`, `String.prototype`, `Boolean.prototype`. Только у значений `undefined` и `null` нет объектов-обёрток.
- Встроенные прототипы могут быть изменены или дополнены новыми методами. Но не рекомендуется менять их. Единственная допустимая причина – это добавление нового метода из стандарта, который ещё не поддерживается движком JavaScript (полифилы).



# Классы

- Для создания множества объектов одного типа используются классы. В JS классы – разновидности функций
- Приватные свойства и методы должны начинаться с #. Они доступны только внутри класса
- ```
class MyClass {  
    // методы класса  
    constructor() { ... }  
    method1() { ... }  
    method2() { ... }  
    method3() { ... }  
    ...  
}
```


Наследование классов

- ```
class Rabbit extends Animal {
 hide() {
 alert(`${this.name} прячется!`);
 }
}
```
- Ключевое слово `extends` работает, используя прототипы. Оно устанавливает `Rabbit.prototype.__proto__` в `Animal.prototype`. Так что если метод не найден в `Rabbit.prototype`, JavaScript берёт его из `Animal.prototype`
- Для обращения к методам родителя используется ключевое слово `super` (у стрелочных функций нет `super`)

# Переопределение конструктора

- Согласно спецификации, если класс расширяет другой класс и не имеет конструктора, то автоматически создаётся «пустой» конструктор, вызывающий конструктор класса:

```
constructor(...args) {
 super(...args);
}
```

- В классах-потомках конструктор обязан вызывать `super(...)`, и делать это перед использованием `this`
- Разница в следующем:

Когда выполняется обычный конструктор, он создаёт пустой объект и присваивает его `this`.

Когда запускается конструктор унаследованного класса, он этого не делает. Вместо этого он ждёт, что это сделает конструктор родительского класса.

# Статические методы и свойства

- Возможно присвоить метод самой функции-классу, а не её "prototype". Такие методы называются статическими. В классе такие методы обозначаются ключевым словом `static`.
- Статические свойства и методы наследуются

```
class User {
 static staticMethod() {
 alert(this === User);
 }
}
```

```
User.staticMethod(); // true
```

# Instanceof

- Оператор instanceof позволяет проверить, к какому классу принадлежит объект, с учётом наследования.
- ```
let arr = [1, 2, 3];  
alert( arr instanceof Array ); // true  
alert( arr instanceof Object ); // true
```
- Обычно оператор instanceof просматривает для проверки цепочку прототипов. Но это поведение может быть изменено при помощи статического метода `Symbol.hasInstance`

Алгоритм работы

- Если имеется статический метод `Symbol.hasInstance`, тогда вызвать его

```
class Animal {  
    static [Symbol.hasInstance](obj) {  
        if (obj.canEat) return true;  
    }  
}
```

- Большая часть классов не имеет метода `Symbol.hasInstance`. В этом случае используется стандартная логика: проверяется, равен ли `Class.prototype` одному из прототипов в прототипной цепочке `obj`

Примеси (mixin)

- Примесь - это класс, методы которого предназначены для использования в других классах, причём без наследования от примеси

```
let sayHiMixin = {  
  sayHi() {  
    alert(`Привет, ${this.name}`);  
  },  
  sayBye() {  
    alert(`Пока, ${this.name}`);  
  }  
};  
  
class User {  
  constructor(name) {  
    this.name = name;  
  }  
}  
  
Object.assign(User.prototype, sayHiMixin);
```



Обработка ошибок

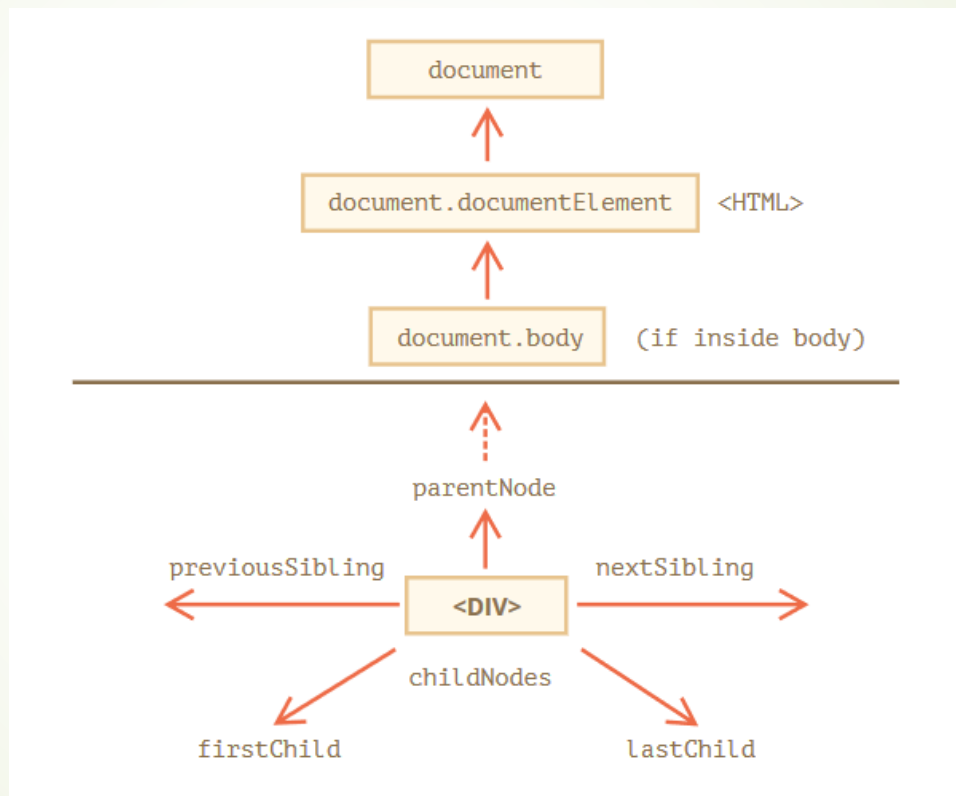
- Используются блоки try/catch/finally
- Для генерации ошибок используется оператор throw
- Можно создавать собственные классы ошибок, наследуясь от класса Exception



Работа с DOM

- HTML/XML документы представлены в браузере в виде DOM-дерева.
- Теги становятся узлами-элементами и формируют структуру документа.
- Текст становится текстовыми узлами.
- Для доступа к DOM используется глобальный объект `document`, например `document.body` – объект для тега `<body>`

Навигация



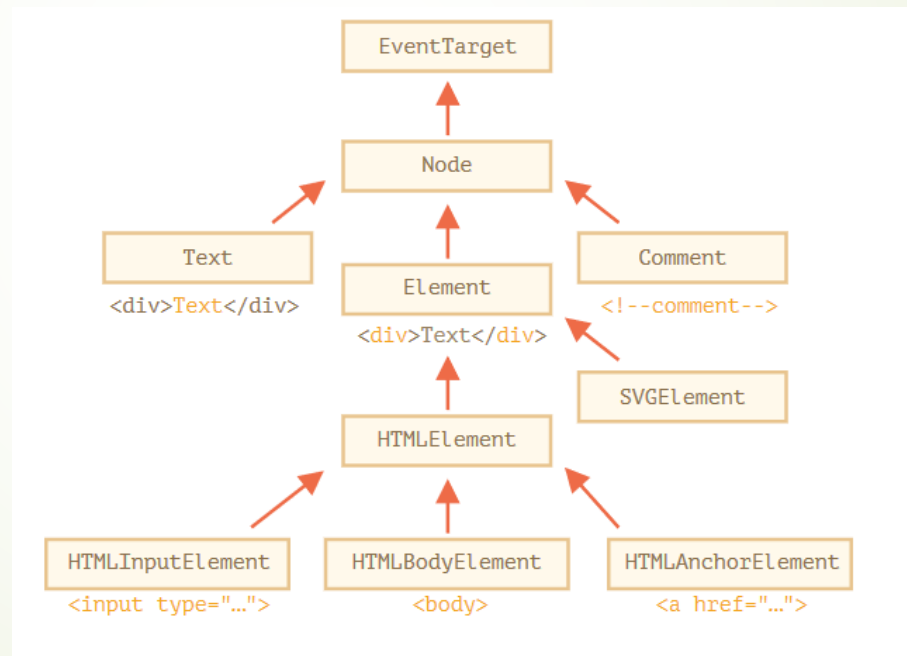
Поиск

Метод	Ищет по...	Ищет внутри элемента?	Возвращает живую коллекцию?
<code>querySelector</code>	CSS-selector	✓	-
<code>querySelectorAll</code>	CSS-selector	✓	-
<code>getElementById</code>	id	-	-
<code>getElementsByName</code>	name	-	✓
<code>getElementsByTagName</code>	tag or '*'	✓	✓
<code>getElementsByClassName</code>	class	✓	✓

- `elem.matches(css)` - проверяет, удовлетворяет ли `elem` CSS-селектору, и возвращает `true` или `false`
- `elem.closest(css)` - ищет ближайшего предка, который соответствует CSS-селектору. Сам элемент также включается в поиск

Свойства узлов

- У разных DOM-узлов могут быть разные свойства. Например, у узла, соответствующего тегу `<a>`, есть свойства, связанные со ссылками, а у соответствующего тегу `<input>` – свойства, связанные с полем ввода и т.д. Текстовые узлы отличаются от узлов-элементов. Но у них есть общие свойства и методы, потому что все классы DOM-узлов образуют единую иерархию.





Содержимое элементов

- `innerHTML` - внутреннее HTML-содержимое узла-элемента. Можно изменять.
- `outerHTML` - полный HTML узла-элемента.
- `textContent` - текст внутри элемента: HTML за вычетом всех <тегов>. Запись в него помещает текст в элемент, при этом все специальные символы и теги интерпретируются как текст. Можно использовать для защиты от вставки произвольного HTML кода.



Атрибуты элементов

- `elem.hasAttribute(name)` – проверяет наличие атрибута.
 - `elem.getAttribute(name)` – получает значение атрибута.
 - `elem.setAttribute(name, value)` – устанавливает значение атрибута.
 - `elem.removeAttribute(name)` – удаляет атрибут.
-
- Атрибуты – это то, что написано в HTML.
 - Свойства – это то, что находится в DOM-объектах.



Изменение документа

Создание элемента

- `document.createElement(tag)` - создаёт новый элемент с заданным тегом
- `document.createTextNode(text)` - создаёт новый текстовый узел с заданным текстом

Добавление информации

- `node.append(...nodes or strings)` – добавляет узлы или строки в конец `node`,
- `node.prepend(...nodes or strings)` – вставляет узлы или строки в начало `node`,
- `node.before(...nodes or strings)` — вставляет узлы или строки до `node`,
- `node.after(...nodes or strings)` — вставляет узлы или строки после `node`,
- `node.replaceWith(...nodes or strings)` — заменяет `node` заданными узлами или строками.



Вставка html

➤ `elem.insertAdjacentHTML(where, html)`

Удаление

➤ `node.remove()`

Клонирование

➤ `elem.cloneNode(true)`

События

- Событие – это сигнал от браузера о том, что что-то произошло
- Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло.

```
<input value="Нажми меня" onclick="alert('Клик!')" type="button">
```

```
<input id="elem" type="button" value="Нажми меня!">
```

```
<script>
```

```
  elem.onclick = function() {
```

```
    alert('Спасибо');
```

```
  };
```

```
</script>
```

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

Вложенные события работают синхронно!

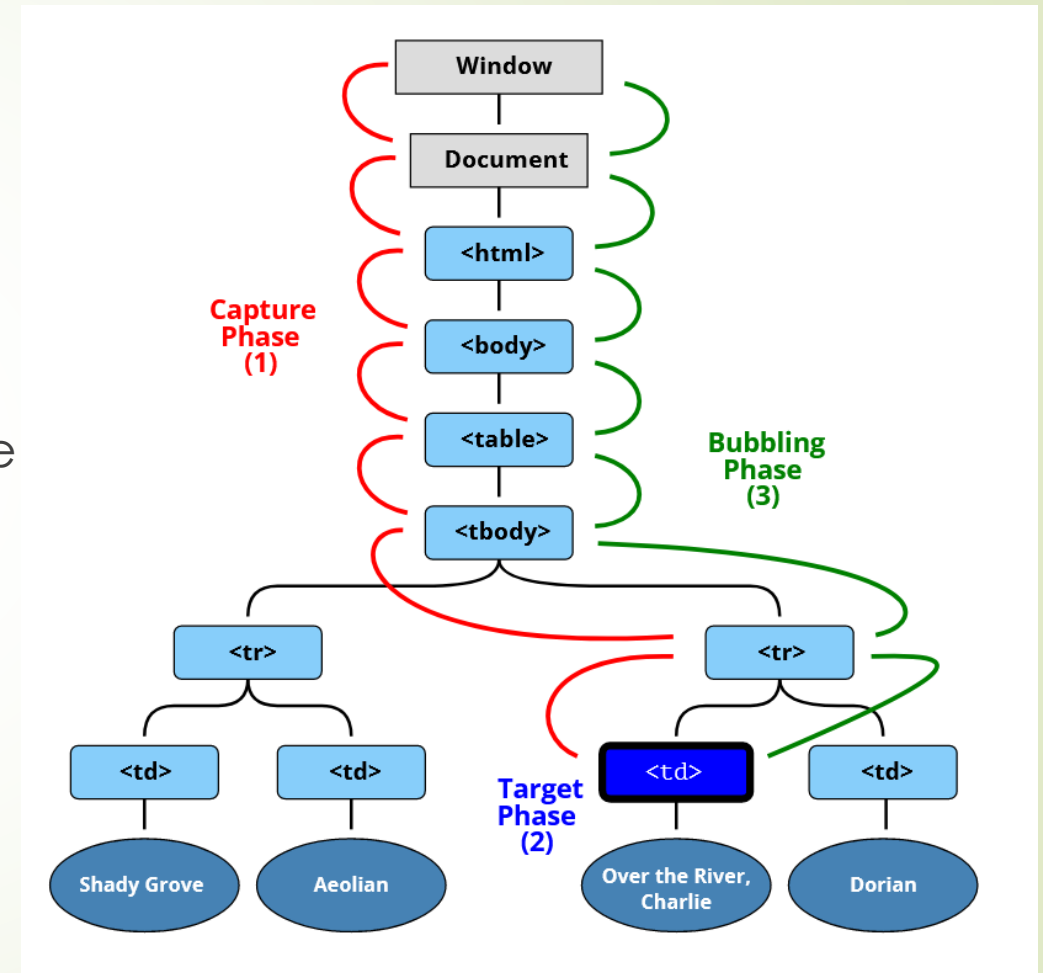
Всплытие


```
<form onclick="alert('form')">FORM  
  <div onclick="alert('div')">DIV  
    <p onclick="alert('p')">P</p>  
  </div>  
</form>
```

- `event.target` – это «целевой» элемент, на котором произошло событие, в процессе всплытия он неизменен.
- `this (event.currentTarget)` – это «текущий» элемент, до которого дошло всплытие, на нём сейчас выполняется обработчик.

Для остановки передачи события далее нужно вызвать метод `event.stopPropagation()`

- `elem.addEventListener(..., {capture: true})`
- `elem.addEventListener(..., true)`
- `event.eventPhase` – на какой фазе сработал обработчик (погружение=1, фаза цели=2, всплытие=3).





Отмена действий браузера по умолчанию

- `event.preventDefault()`.
- Если обработчик назначен через `on<событие>` (не через `addEventListener`), то также можно вернуть `false` из обработчика.

Создание события

- ▶ `let event = new Event(type[, options]);`
- ▶ `type` – тип события, строка, например "click" или же любой придуманный нами – "my-event".
- ▶ `options` – объект с необязательными свойствами:
 - `bubbles: true/false` – если `true`, тогда событие всплывает.
 - `cancelable: true/false` – если `true`, тогда можно отменить действие по умолчанию

Запуск: `elem.dispatchEvent(event)`



Создание пользовательского события

```
<h1 id="elem">Hello, Vasya!</h1>
```

```
<script>
```

```
    elem.addEventListener("hello", function(event) {  
        alert(event.detail.name);  
    });
```

```
    elem.dispatchEvent(new CustomEvent("hello", {  
        detail: { name: "Vasya" }  
    }));
```

```
</script>
```


Жизненный цикл HTML-страницы

- DOMContentLoaded – браузер полностью загрузил HTML, было построено DOM-дерево, но внешние ресурсы, такие как картинки `` и стили, могут быть ещё не загружены.
- load – браузер загрузил HTML и внешние ресурсы (картинки, стили и т.д.).
- beforeunload/unload – пользователь покидает страницу.

Атрибуты скриптов

- Атрибут `defer` сообщает браузеру, что он должен продолжать обрабатывать страницу и загружать скрипт в фоновом режиме, а затем запустить этот скрипт, когда он загрузится. Отложенные с помощью `defer` скрипты сохраняют порядок относительно друг друга, как и обычные скрипты.
- Атрибут `async` означает, что скрипт абсолютно независим
- Возможно также добавить скрипт и динамически, с помощью JavaScript:

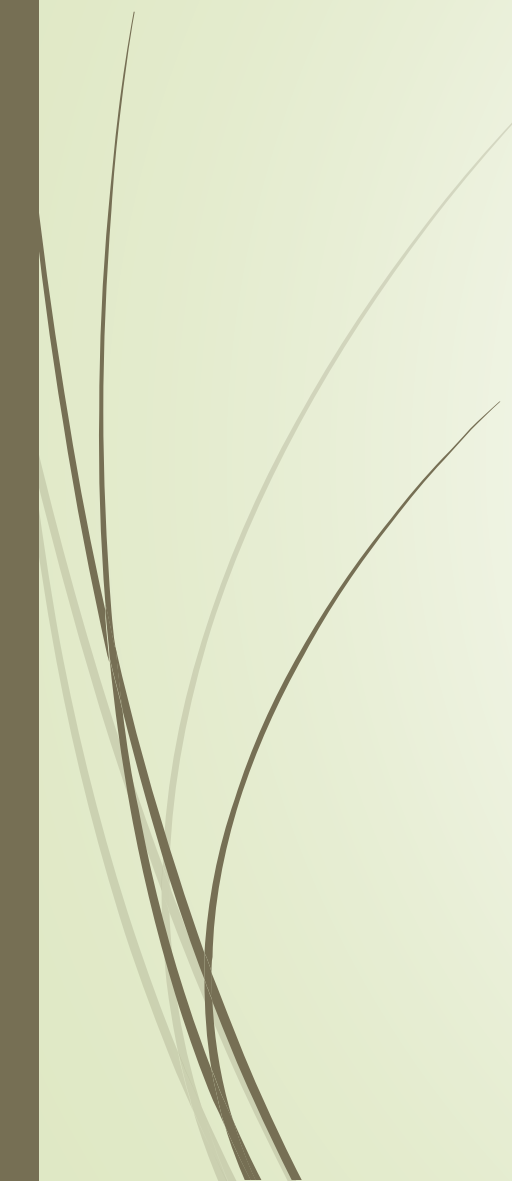
```
let script = document.createElement('script');  
script.src = "/article/script-async-defer/long.js";  
document.body.append(script); // (*)
```

Динамически загружаемые скрипты по умолчанию ведут себя как «`async`»




КОЛЛБЭКИ

```
function loadScript(src, callback) {  
    let script = document.createElement('script');  
    script.src = src;  
  
    script.onload = () => callback(script);  
    document.head.append(script);  
}
```



Адская пирамида вызовов

```
loadScript('1.js', function(error, script) {  
  if (error) {  
    handleError(error);  
  } else {  
    // ...  
    loadScript('2.js', function(error, script) {  
      if (error) {  
        handleError(error);  
      } else {  
        // ...  
        loadScript('3.js', function(error, script) {  
          if (error) {  
            handleError(error);  
          } else {  
            // ...и так далее, пока все скрипты не будут загружены (*)  
          }  
        });  
      }  
    });  
  }  
});
```



```
loadScript('1.js', step1);  
function step1 (error, script) {  
    if (error) {  
        handleError(error);  
    } else {  
        // ...  
        loadScript('2.js', step2);  
    }  
}  
function step2(error, script) {  
    if (error) {  
        handleError(error);  
    } else {  
        // ...  
    }  
}
```

Промисы (promise)

```
let promise = new Promise(function(resolve, reject) {  
    // функция-исполнитель (executor)  
});
```

```
let promise = new Promise(function(resolve, reject) {  
    setTimeout(() => reject(new Error("Whoops!")), 1000);  
});
```





Потребители

```
promise.then(  
    function(result) { /* обработает успешное выполнение */ },  
    function(error) { /* обработает ошибку */ }  
);
```

`promise.catch(f)` – это сокращённый, «укороченный» вариант `.then(null, f)`.

`promise.finally(f)` похож на `.then(f, f)`, в том смысле, что `f` выполнится в любом случае, когда промис завершится: успешно или с ошибкой.



Цепочка промисов

```
new Promise(function(resolve, reject) {  
    setTimeout(() => resolve(1), 1000);  
}).then(function(result) {  
    alert(result); // 1  
    return new Promise((resolve, reject) => { // (*)  
        setTimeout(() => resolve(result * 2), 1000);  
    });  
})
```



Promise API

- ▶ `let promise = Promise.all([...промисы...]);`
- ▶ `let promise = Promise.allSettled([...промисы...]);`
- ▶ `let promise = Promise.race([...промисы...]);`
- ▶ `Promise.resolve(value)` - устаревший
- ▶ `Promise.reject(value)` - устаревший

Асинхронные функции

- У слова `async` один простой смысл: эта функция всегда возвращает промис. Значения других типов оборачиваются в завершившийся успешно промис автоматически.
- Ключевое слово `await` заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от `await` не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится.

```
async function f() {  
    let promise = Promise.resolve(1);  
    let result = await promise;    }
```



Генераторы

```
function* generateSequence() {  
    yield 1;  
    yield 2;  
    return 3;  
}
```

```
let one = generator.next();
```



Возвращение значения в генератор

```
function* gen() {  
    let ask1 = yield "2 + 2 = ?";  
    alert(ask1); // 4  
    let ask2 = yield "3 * 3 = ?"  
    alert(ask2); // 9  
}
```

```
let generator = gen();  
alert( generator.next().value ); // "2 + 2 = ?"  
alert( generator.next(4).value ); // "3 * 3 = ?"  
alert( generator.next(9).done ); // true
```

Асинхронные генераторы

```
➤ async function* generateSequence(start, end) {  
  for (let i = start; i <= end; i++) {  
    await new Promise(resolve => setTimeout(resolve, 1000));  
    yield i;  
  }  
}  
  
(async () => {  
  let generator = generateSequence(1, 5);  
  for await (let value of generator) {  
    alert(value); // 1, потом 2, потом 3, потом 4, потом 5  
  }  
})();
```



AJAX



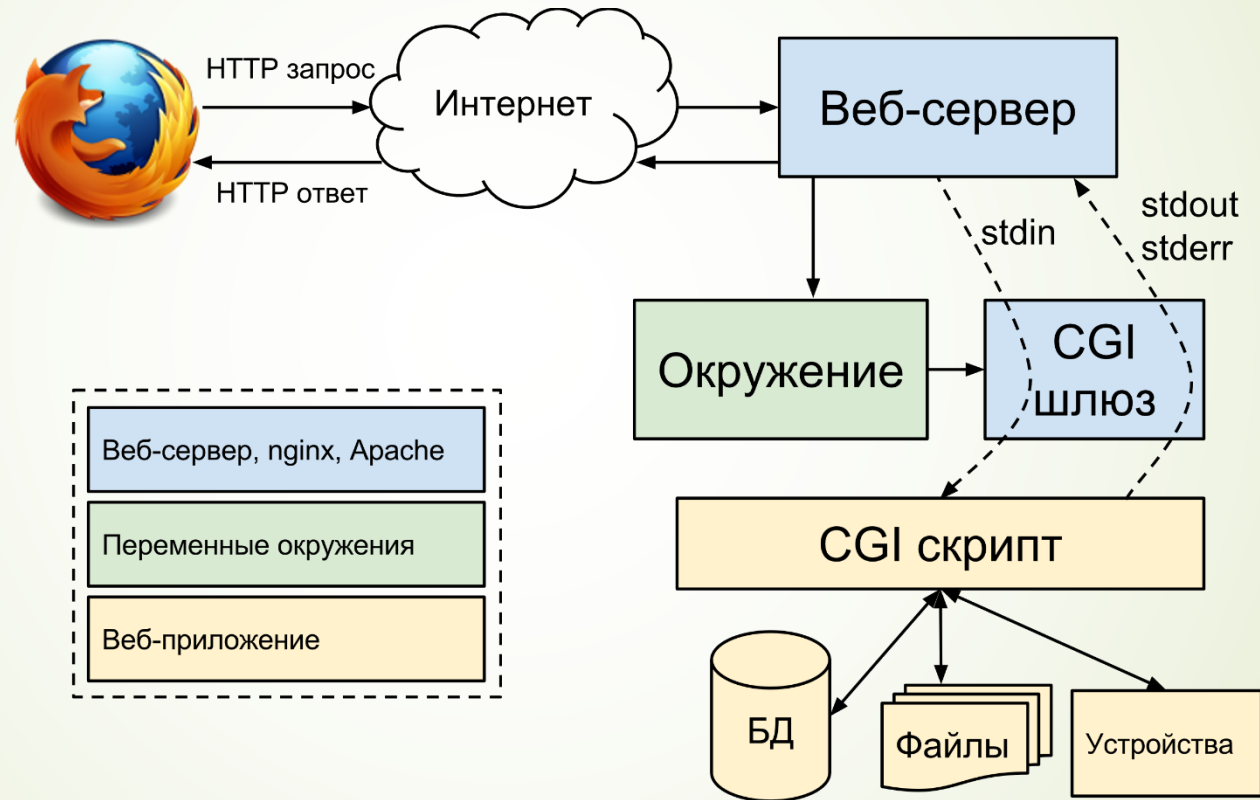
- AJAX — это аббревиатура, которая означает Asynchronous Javascript and XML
- В классической модели веб-приложения:
 1. Пользователь заходит на веб-страницу и нажимает на какой-нибудь её элемент.
 2. Браузер формирует и отправляет запрос серверу.
 3. В ответ сервер генерирует совершенно новую веб-страницу и отправляет её браузеру и т. д., после чего браузер полностью перезагружает всю страницу.
- При использовании AJAX:
 1. Пользователь заходит на веб-страницу и нажимает на какой-нибудь её элемент.
 2. Скрипт (на языке JavaScript) определяет, какая информация необходима для обновления страницы.
 3. Браузер отправляет соответствующий запрос на сервер.
 4. Сервер возвращает только ту часть документа, на которую пришёл запрос.
 5. Скрипт вносит изменения с учётом полученной информации (без полной перезагрузки страницы).



Fetch

- `let promise = fetch(url, [options])`
- `const response = await fetch(url, {
 method: 'POST', // *GET, POST, PUT, DELETE, etc.
 mode: 'cors', // no-cors, *cors, same-origin
 cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-cached
 credentials: 'same-origin', // include, *same-origin, omit
 headers: {
 'Content-Type': 'application/json'
 },
 redirect: 'follow', // manual, *follow, error
 referrerPolicy: 'no-referrer', // no-referrer, *client
 body: JSON.stringify(data) // body data type must match "Content-Type" header
});`

CGI (Common Gateway Interface)





CGI в Python

```
from http.server import HTTPServer, CGIHTTPRequestHandler

server_address = ("", 8000)
httpd = HTTPServer(server_address, CGIHTTPRequestHandler)
httpd.serve_forever()
```

Скрипт

```
print("Content-type: text/html")
print()
print("<h1>Hello world!</h1>")
```