1. **КЛАССЫ**

класс – это разновидность функции. представляют собой способ упростить синтаксис при использовании прототипов и наследования

class User {

  constructor(name) { вызывается метод **constructor(),** в нём мы можем инициализировать объект.

    this.name = name;

  }

  sayHi() { Сохраняет все методы, такие как*sayHi*, в **User.prototype.** объекты new User имеют доступ к

    alert(this.name); к методам класса.

  }

}

let user = new User("Иван"); Когда вызывается **new User("Иван"):** Создаётся новый объект,

1. user.sayHi();  **constructor** запускается с заданным аргументом и сохраняет его в **this.name**.

**Наследование классов** – это способ расширения одного класса другим классом.

Синтаксис -- class Child **extends** Parent (class Rabbit extends Animal {})

Внутри ключевое слово extends работает по старой доброй механике прототипов. Оно устанавливает Rabbit.prototype.[[Prototype]] в Animal.prototype. Таким образом, если метода не оказалось в Rabbit.prototype, JS берет его из Animal.prototype.

[**Переопределение методов**](https://learn.javascript.ru/class-inheritance#pereopredelenie-metodov)**.**

* super(...) для вызова родительского конструктора (работает только внутри нашего конструктора).
* class Rabbit extends Animal {
* stop() {…}
* }

class Rabbit extends Animal {

hide () {}

stop() {

super.stop(); // вызываем родительский метод stop

this.hide(); // и затем hide

} }

[**Переопределение конструктора**](https://learn.javascript.ru/class-inheritance#pereopredelenie-metodov)**.** если класс расширяет другой класс и не имеет конструктора, то автоматически создаётся такой «пустой» конструктор. он просто вызывает конструктор родительского класса. Так будет происходить, пока мы не создадим собственный конструктор.Конструкторы в наследуемых классах должны обязательно вызывать super(...), и (!) делать это перед использованием**this.**.

class Rabbit extends Animal {

constructor(name, earLength) {

super(name);

this.earLength = earLength; } }

**Статические свойства и методы.** Статические методы используются для функциональности, принадлежат классу «в целом», а не относятся к конкретному объекту класса.

class User {

static staticMethod() {

alert(this === User); } }

User.staticMethod(); // true Значением this при вызове User.staticMethod() является сам конструктор класса User

2. **Область видимости (лексическая область видимости). scope**  
 Областью видимости является коллекция переменных и правила доступа к этим переменным по имени. Только код внутри функции имеет доступ к пер.-м, определенным в ее области видимости. Имя пер. должно быть уникальным в рамках одной и той же области видимости: не может быть двух различных переменных a, расположенных рядом друг с другом. Но одно и тоже имя переменной a может появляться в разных областях видимости.  
function outer() {  
       var a = 1;  
  function inner() {  
                   var b = 2; // здесь у нас есть доступ и к a, и к b  
   console.log( a + b ); // 3  
               }  
  inner(); // здесь у нас есть доступ только к a  
  console.log( a ); // 1  
}  
outer();  
код внутри функции inner() имеет доступ к обеим переменным a и b, но у кода в outer() есть доступ только к a — у него нет доступа к b, потому что эта переменная внутри inner().

**Var / let / const**  
let и const ведут себя одинаково по отношению к лексическому окружению, области видимости.  
Для «var» не существует блочной области видимости(доступнa за пределами блока.). Используя var, можно переобъявлять переменную сколько угодно раз. Повторные var игнорируются. var считаются объявленными с самого начала исполнения функции вне зависимости от того, в каком месте функции реально находятся их объявления - Это поведение называется **«hoisting»** (всплытие, поднятие), потому что все объявления переменных var «всплывают» в самый верх функции.

Объект, объявленный через **const**, может быть изменён. объявление const защищает от изменений только саму переменную user, а не её содержимое.

**2. Функции** (ф это выражение)  
**Локальные** пер.-е => Пер.-e, объявленные внутри функции, видны только внутри этой функции.   
Функция обладает полным доступом к внешним переменным и может изменять их значение. Внешняя **(глобальная)** переменная используется, только если внутри функции нет такой локальной. Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю. Например, в коде ниже функция использует локальную переменную userName. Внешняя будет проигнорирована: let userName = 'Вася';  
function showMessage() {   
let userName = "Петя"; // объявляем локальную переменную  
let message = 'Привет, ' + userName; // Петя  
alert(message);  
}// функция создаст и будет использовать свою собственную локальную переменную userName  
showMessage();  
alert( userName ); // Вася, не изменилась, функция не трогала внешнюю переменную  
Function Declaration (код выше)  
 Function Expression (Функциональное Выражение)   
let sayHi = function() {  
  alert( "Привет" );  
};

**FunExp** создаётся, когда выполнение доходит до него, и затем уже может использоваться.  
**Function Declaration** может быть вызвана раньше, чем она объявлена.

**Функции-колбэки**. Ключевая идея в том, что мы передаём функцию и ожидаем, что она вызовется обратно (от англ. «call back» – обратный вызов) когда-нибудь позже, если это будет необходимо  
function ask(question, yes, no) {  
if (confirm(question)) yes()  
else no();  
}  
function showOk() {  
 alert( "Вы согласны." );  
}  
function showCancel() {  
alert( "Вы отменили выполнение." );  
} // использование: функции showOk, showCancel передаются в качестве аргументов ask  
ask("Вы согласны?", showOk, showCancel);

# **Промисы**

 это специальный объект в JavaScript, который связывает «создающий» и «потребляющий» коды вместе

let promise = new Promise(function(resolve, reject) {

// функция-исполнитель (executor)

});

Когда он получает результат, сейчас или позже – не важно, он должен вызвать один из этих колбэков:

* resolve(value) — если работа завершилась успешно, с результатом value.

reject(error) — если произошла ошибка, error – объект ошибки.

У объекта promise, возвращаемого конструктором new Promise, есть внутренние свойства:

* state («состояние») — вначале "pending" («ожидание»), потом меняется на "fulfilled" («выполнено успешно») при вызове resolve или на "rejected" («выполнено с ошибкой») при вызове reject.
* result («результат») — вначале undefined, далее изменяется на value при вызове resolve(value) или на error при вызове reject(error).

Исполнитель должен вызвать что-то одно: resolve или reject. Состояние промиса может быть изменено только один раз.Все последующие вызовы resolve и reject будут проигнорированы

**Then, catch**

Функции-потребители могут быть зарегистрированы (подписаны) с помощью методов **.then и .catch.**

promise.then(

function(result) { /\* обработает успешное выполнение \*/ },

function(error) { /\* обработает ошибку \*/ }

);

let promise = new Promise(function(resolve, reject) {

setTimeout(() => resolve("done!"), 1000); // resolve запустит первую функцию, переданную в .then

});

promise.then(

result => alert(result), // выведет "done!" через одну секунду

error => alert(error) // не будет запущена

);

**Catch** Вызов .catch(f) – это сокращённый, «укороченный» вариант .then(null, f).

let promise = new Promise((resolve, reject) => {

setTimeout(() => reject(new Error("Ошибка!")), 1000);

});

// .catch(f) это то же самое, что promise.then(null, f)

promise.catch(alert); // выведет "Error: Ошибка!" спустя одну секунду

**Очистка: Finally**

Идея finally состоит в том, чтобы настроить обработчик для выполнения очистки/доведения после завершения предыдущих операций. Например, остановка индикаторов загрузки, закрытие больше не нужных соединений и т.д.

new Promise((resolve, reject) => {

setTimeout(() => resolve("value"), 2000);

})

.finally(() => alert("Промис завершён")) // срабатывает первым

.then(result => alert(result)); // <-- .then показывает "value"

* Обработчик finally не получает результат предыдущего обработчика (у него нет аргументов). Вместо этого этот результат передается следующему подходящему обработчику. Обработчик finally также не должен ничего возвращать. Если это так, то возвращаемое значение молча игнорируется.

# **Async/await**

У слова **async** один простой смысл: эта функция всегда возвращает промис.

async function f() {

return 1; либо return Promise.resolve(1);

}

f().then(alert); // 1

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. работает только внутри async–функций. По сути, это просто «синтаксический сахар» для получения результата промиса, более наглядный, чем promise.then.

**3. Замыкание**

В JS у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый **лексическим окружением** (LexicalEnvironment).

**Когда код хочет получить доступ к переменной – сначала происходит поиск во внутреннем лексическом окружении, затем во внешнем, затем в следующем и так далее, до глобального. Поиск в области видимости прекращается как только он находит первое совпадение**

**Функция получает текущее значение внешних переменных, то есть их последнее значение**

let name = "John"; //В глобальном лексическом окружении есть name: "John"

function sayHi() {

alert("Hi, " + name);

}

name = "Pete"; //глобальная переменная изменяется, теперь name: "Pete".

sayHi(); // Pete

[**Замыкание**](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BC%D1%8B%D0%BA%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))– это функция, которая запоминает свои внешние переменные и может получить к ним доступ. **Все функции «при рождении» получают скрытое свойство [[Environment]], которое ссылается на лексическое окружение места, где они были созданы.**

**[IIFE](https://learn.javascript.ru/closure" \l "iife)**

«immediately-invoked function expressions» означает функцию, запускаемую сразу после объявления.

(function() {

let message = "Hello";

alert(message); // Hello

})();

скобки вокруг функции – это трюк, который позволяет показать JavaScript, что функция была создана в контексте другого выражения, и, таким образом, это функциональное выражение: ей не нужно имя и её можно вызвать немедленно.

**Сборка мусора**

Обычно лексическое окружение очищается и удаляется после того, как функция выполнилась.

Основной концепцией управления памятью в JavaScript является принцип достижимости. Любое другое значение считается достижимым, если оно доступно из корня по ссылке или по цепочке ссылок.

Основной алгоритм сборки мусора называется «алгоритм пометок» (от англ. «mark-and-sweep»).Согласно этому алгоритму, сборщик мусора регулярно выполняет следующие шаги:

* Сборщик мусора «помечает» (запоминает) все корневые объекты.
* Затем он идёт по ним и «помечает» все ссылки из них.
* Затем он идёт по отмеченным объектам и отмечает их ссылки. Все посещённые объекты запоминаются, чтобы в будущем не посещать один и тот же объект дважды.
* …И так далее, пока не будут посещены все достижимые (из корней) ссылки.
* Все непомеченные объекты удаляются.

**Объекты**

let user = {

name: "John",

age: 30,

isAdmin: true

};

for (let key in user) {

// ключи

alert( key ); // name, age, isAdmin

// значения ключей

alert( user[key] ); // John, 30, true

}

let user = {

    name: "John",

    age: 30,

    "likes birds": ‘yees’   // имя свойства из нескольких слов должно быть в кавычках

  };

Erroe - user.likes birds

Correct - user["likes birds"]

**this**

**Для доступа к информации внутри объекта метод может использовать ключевое слово this.(**  не ссылается на саму функцию**)**

В строгом режиме this === undefined. В нестрогом режиме значением this в таком случае будет глобальный объект (window). Если внутри функции используется this, тогда она ожидает, что будет вызвана в контексте какого-либо объекта.

**Глобальный объект (window)**

Глобальный объект предоставляет переменные и функции, доступные в любом месте программы.

В браузере глобальные функции и переменные, объявленные с помощью var (не let/const!), становятся свойствами глобального объекта:

var gVar = 5;

alert(window.gVar) // 5 (становится свойством глобального объекта)

**Загрузка ресурсов: onload и onerror**

* load – успешная загрузка,
* error – во время загрузки произошла ошибка.

DOM

объектной моделью документа («Document Object Model»).  Это представление HTML-документа в виде дерева тегов. DOM обеспечивает представление документа в виде структурированной группы узлов и объектов, которые имеют свойства и методы. По сути, она связывает веб -страницы со скриптами или языками программирования. операции в DOM начинается с объекта doc