**JS**

1. **Типы JS**

JavaScript определяет 7 типов данных:

* null
* undefined
* boolean
* string
* number
* object
* symbol (добавлен в ES6)

Все типы данных кроме объектов (objects) — примитивные.

Все типы данных в JavaScript, кроме объектов, являются иммутабельными (значения не могут быть модифицированы, а только перезаписаны новым полным значением). Например, в отличии от C, где строку можно посимвольно корректировать, в JavaScript строки пересоздаются только полностью. Значения таких типов называются «примитивными значениями».

1. **Что такое DOM?**

DOM или Document Object Model (объектная модель документа) — это прикладной программный интерфейс (API) для работы с HTML и XML документами. Когда браузер первый раз читает («парсит») HTML документ, он формирует большой объект, действительно большой объект, основанный на документе — DOM. DOM представляет собой древовидную структуру (дерево документа). DOM используется для взаимодействия и изменения самой структуры DOM или его отдельных элементов и узлов.

#### **3. Что такое поднятие (Hoisting)?**

Поднятие — это термин, описывающий подъем переменной или функции в глобальную или функциональную области видимости.  
  
Для того, чтобы понять, что такое Hoisting, необходимо разобраться с тем, что представляет собой контекст выполнения.  
  
Контекст выполнения — это среда, в которой выполняется код. Контекст выполнения имеет две фазы — компиляция и собственно выполнение.  
  
Компиляция. В этой фазе функциональные выражения и переменные, объявленные с помощью ключевого слова «var», со значением undefined поднимаются в самый верх глобальной (или функциональной) области видимости (как бы перемещаются в начало нашего кода. Это объясняет, почему мы можем вызывать функции до их объявления — прим. пер.).  
  
Выполнение. В этой фазе переменным присваиваются значения, а функции (или методы объектов) вызываются или выполняются.  
  
Запомните: поднимаются только функциональные выражения и переменные, объявленные с помощью ключевого слова «var». Обычные функции и стрелочные функции, а также переменные, объявленные с помощью ключевых слов «let» и «const» не поднимаются.

#### **4. Что такое область видимости (Scope)?**

Область видимости — это место, где (или откуда) мы имеем доступ к переменным или функциям. JS имеем три типа областей видимости: глобальная, функциональная (локальная) и блочная (ES6).  
  
Глобальная область видимости — переменные и функции, объявленные в глобальном пространстве имен, имеют глобальную область видимости и доступны из любого места в коде.

**5. Компоненты высшего порядка**

Компонент высшего порядка (Higher-Order Component, HOC) — это один из продвинутых способов для повторного использования логики. HOC не являются частью API React, но часто применяются из-за композиционной природы компонентов.

шаблон проектирования

Говоря просто, компонент высшего порядка — это функция, которая принимает компонент и возвращает новый компонент.

Если обычный компонент преобразует пропсы в UI, то компонент высшего порядка преобразует компонент в другой компонент.

HOC часто встречаются в сторонних библиотеках, например connect в Redux и createFragmentContainer в Relay.

**6. Отличие for...of от for...in в javascript**

Оба оператора, и for...in и for...of производят обход объектов. Разница лишь в том, как они это делают.

Оператор for...of выполняет обход по элементам коллекций (иначе говоря, итерируемых объектов, например, Array, Map, Set, String, arguments, DOM коллекций и т.д.), вызывая на каждом шаге итерации **значение**, а не ключ.

## **for...in**

Цикл for...in проходит через перечисляемые свойства объекта. Он пройдёт по каждому отдельному элементу, вызывая на каждом шаге **ключ**. Цикл for...in проходит по свойствам в произвольном порядке, поэтому его не следует использовать для Array.

Метод **map()** создаёт новый массив с результатом вызова указанной функции для каждого элемента массива.

**7. ECMAScript**

Проще говоря, ECMAScript — стандарт, а JavaScript — самая популярная реализация этого стандарта.

Версия ES6/ES2015 вышла в июне 2015 года.

**8. Замыкания**

**Замыкание** (**closure**) представляют собой конструкцию, когда функция, созданная в одной области видимости, запоминает свое лексическое окружение даже в том случае, когда она выполняет вне своей области видимости.

Замыкание технически включает три компонента:

* внешняя функция, которая определяет некоторую область видимости и в которой определены некоторые переменные - лексическое окружение
* переменные (лексическое окружение), которые определены во внешней функции
* вложенная функция, которая использует эти переменные

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function outer(){       // внешняя функция      var n;              // некоторая переменная      return inner(){     // вложенная функция          // действия с переменной n      }  } |

Рассмотрим замыкания на простейшем примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | function outer(){      let x = 5;      function inner(){          x++;          console.log(x);      };      return inner;  }  let fn = outer();   // fn = inner, так как функция outer возвращает функцию inner  // вызываем внутреннюю функцию inner  fn();   // 6  fn();   // 7  fn();   // 8 |

Здесь функция outer задает область видимости, в которой определены внутренняя функция inner и переменная x. Переменная x представляет лексическое окружение для функции inner. В самой функции inner инкрементируем переменную x и выводим ее значение на консоль. В конце функция outer возвращает функцию inner.

Далее вызываем функцию outer:

|  |  |
| --- | --- |
| 1 | let fn = outer(); |

Поскольку функция outer возвращает функцию inner, то переменная fn будет хранить ссылку на функцию inner. При этом эта функция запомнила свое окружение - то есть внешнюю переменную x.

Далее мы фактически три раза вызываем функцию Inner, и мы видим, что переменная x, которая определена вне функции inner, инкрементируется:

|  |  |
| --- | --- |
| 1  2  3 | fn();   // 6  fn();   // 7  fn();   // 8 |

То есть несмотря на то, что переменная x определена вне функции inner, эта функция запомнила свое окружение и может его использовать, несомотря на то, что она вызывается вне функции outer, в которой была определена. В этом и суть замыканий.

**Самовызывающиеся функции**

Обычно определение функции отделяется от ее вызова: сначала мы определяем функцию, а потом вызываем. Но это необязательно. Мы также можем создать такие функции, которые будут вызываться сразу при определении. Такие функции еще называют Immediately Invoked Function Expression (IIFE).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | (function(){      console.log("Привет мир");  }());    (function (n){        var result = 1;      for(var i=1; i<=n; i++)          result \*=i;      console.log("Факториал числа " + n + " равен " + result);  }(4)); |

Подобные функции заключаются в скобки, и после определения функции идет в скобках передача параметров.

## [Итого: замыкания](https://learn.javascript.ru/closures" \l "itogo-zamykaniya)

[Замыкание](http://en.wikipedia.org/wiki/Closure_(computer_science)) – это функция вместе со всеми внешними переменными, которые ей доступны.

Таково стандартное определение, которое есть в Wikipedia и большинстве серьёзных источников по программированию. То есть, замыкание – это функция + внешние переменные.

Тем не менее, в JavaScript есть небольшая терминологическая особенность.

**Обычно, говоря «замыкание функции», подразумевают не саму эту функцию, а именно внешние переменные.**

Иногда говорят «переменная берётся из замыкания». Это означает – из внешнего объекта переменных.

**Что это такое – «понимать замыкания?»**

Иногда говорят «Вася молодец, понимает замыкания!». Что это такое – «понимать замыкания», какой смысл обычно вкладывают в эти слова?

«Понимать замыкания» в JavaScript означает понимать следующие вещи:

1. Все переменные и параметры функций являются свойствами объекта переменных LexicalEnvironment. Каждый запуск функции создаёт новый такой объект. На верхнем уровне им является «глобальный объект», в браузере – window.
2. При создании функция получает системное свойство [[Scope]], которое ссылается на LexicalEnvironment, в котором она была создана.
3. При вызове функции, куда бы её ни передали в коде – она будет искать переменные сначала у себя, а затем во внешних LexicalEnvironment с места своего «рождения».

9. Eventloop

Поток выполнения в браузере, равно как и в Node.js, основан на событийном цикле.

Понимание работы событийного цикла важно для оптимизаций, иногда для правильной архитектуры.

В этой главе мы сначала разберём теорию, а затем рассмотрим её практическое применение.

## [Событийный цикл](https://learn.javascript.ru/event-loop" \l "sobytiynyy-tsikl)

Идея событийного цикла очень проста. Есть бесконечный цикл, в котором движок JavaScript ожидает задачи, исполняет их и снова ожидает появления новых.

Общий алгоритм движка:

1. Пока есть задачи:
   * выполнить их, начиная с самой старой
2. Бездействовать до появления новой задачи, а затем перейти к пункту 1

## [Макрозадачи и Микрозадачи](https://learn.javascript.ru/event-loop" \l "makrozadachi-i-mikrozadachi)

Помимо макрозадач, описанных в этой части, существуют микрозадачи, упомянутые в главе [Микрозадачи](https://learn.javascript.ru/microtask-queue).

Микрозадачи приходят только из кода. Обычно они создаются промисами: выполнение обработчика .then/catch/finally становится микрозадачей. Микрозадачи также используются «под капотом» await, т.к. это форма обработки промиса.

Также есть специальная функция queueMicrotask(func), которая помещает func в очередь микрозадач.

**Сразу после каждой макрозадачи движок исполняет все задачи из очереди микрозадач перед тем, как выполнить следующую макрозадачу или отобразить изменения на странице, или сделать что-то ещё.**

Например:

setTimeout(() => alert("timeout"));

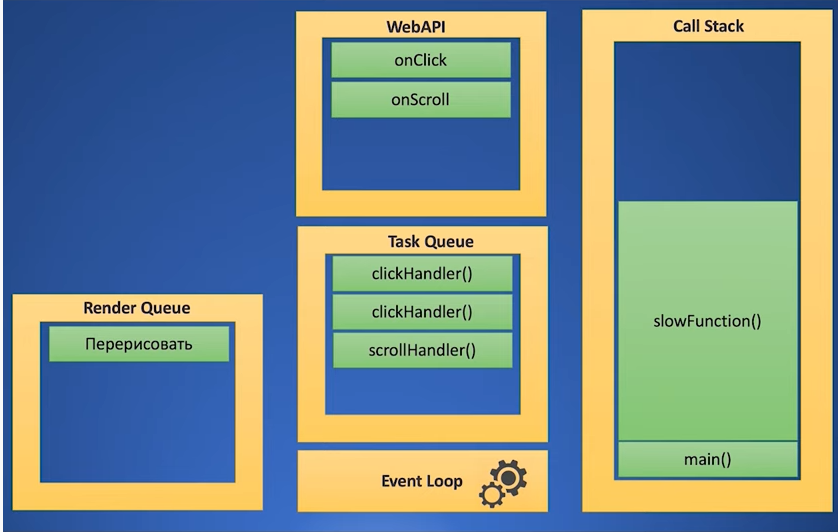
Promise.resolve()

.then(() => alert("promise"));

alert("code");

Какой здесь будет порядок?

1. code появляется первым, т.к. это обычный синхронный вызов.
2. promise появляется вторым, потому что .then проходит через очередь микрозадач и выполняется после текущего синхронного кода.
3. timeout появляется последним, потому что это макрозадача.



10. Стэк

**Стек** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *stack* — стопка; читается *стэк*) — [абстрактный тип данных](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), представляющий собой [список элементов](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), организованных по принципу [*LIFO*](https://ru.wikipedia.org/wiki/LIFO) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *last in — first out*, «последним пришёл — первым вышел»).

11. Прототип

prototype -это свойство объекта функции. Это прототип объектов, построенных с помощью этой функции.

\_\_proto\_\_ -внутреннее свойство объекта, указывающее на его прототип. Существующие стандарты предоставляют эквивалентный метод Object.getPrototypeOf(O) , хотя де-факто стандарт \_\_proto\_\_ быстрее.

\_\_proto\_\_ это ссылка на прототип родителя

## [Прототип](https://learn.javascript.ru/prototype" \l "prototip-proto)**[proto](https://learn.javascript.ru/prototype" \l "prototip-proto)**

Если один объект имеет специальную ссылку \_\_proto\_\_ на другой объект, то при чтении свойства из него, если свойство отсутствует в самом объекте, оно ищется в объекте \_\_proto\_\_.

Свойство \_\_proto\_\_ доступно во всех браузерах, кроме IE10-, а в более старых IE оно, конечно же, тоже есть, но напрямую к нему не обратиться, требуются чуть более сложные способы, которые мы рассмотрим позднее.

Пример кода (кроме IE10-):

var animal = {

eats: true

};

var rabbit = {

jumps: true

};

rabbit.\_\_proto\_\_ = animal;

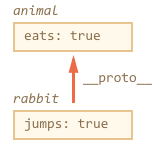
// в rabbit можно найти оба свойства

alert( rabbit.jumps ); // true

alert( rabbit.eats ); // true

1. Первый alert здесь работает очевидным образом – он выводит свойство jumps объекта rabbit.
2. Второй alert хочет вывести rabbit.eats, ищет его в самом объекте rabbit, не находит – и продолжает поиск в объекте rabbit.\_\_proto\_\_, то есть, в данном случае, в animal.

Иллюстрация происходящего при чтении rabbit.eats (поиск идёт снизу вверх):



**Объект, на который указывает ссылка \_\_proto\_\_, называется «прототипом». В данном случае получилось, что animal является прототипом для rabbit.**

**Также говорят, что объект rabbit «прототипно наследует» от animal.**

Обратим внимание – прототип используется исключительно при чтении. Запись значения, например, rabbit.eats = value или удаление delete rabbit.eats – работает напрямую с объектом.

В примере ниже мы записываем свойство в сам rabbit, после чего alert перестаёт брать его у прототипа, а берёт уже из самого объекта:

Прототипы - это механизм, с помощью которого объекты JavaScript наследуют свойства друг от друга.

12. Call/apply/bind

Call/apply просто вызывают фунцию

В JavaScript функции это объекты, как вы должны были бы уже знать. И как объекты, функции имеют свои методы, включая такие действенные, как apply(), call() и bind(). Можно сказать, что Apply() и Call() буквально идентичны друг другу и зачастую используются в JavaScript для того, чтобы заимствовать методы и выставлять значения this. Также мы используем Apply() для функций с большим количеством переменных и аргументов, но об этом вы узнаете дальше в статье.

Bind() же мы используем для выставления значения this в методах и для [каррирования функций](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%80%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5).

Мы рассмотрим каждый сценарий, в котором будет использоваться каждый из трех методов. Давайте начнем с метода bind().

# **Метод bind()**

В основном, мы используем метод bind(), чтобы вызывать функцию с указанием значения this. А другими словами, bind() позволяет нам легко выставлять какой именно объект будет привязан к this в момент вызова функции или метода.

13. Стрелочные функции:

Не имеют this.

* Не имеют this.
* Не имеют arguments.
* Не могут быть вызваны с new.
* У них также нет super.

14. Копирование в JS

Примитивные типы копируются по значению.

Объекты и массивы копируют лишь ссылки.

Скопировать объект, если нет других вложенных объектов

let clone = Object.assign({}, user);

Мы можем реализовать глубокое клонирование, используя рекурсию. Или, чтобы не изобретать велосипед, использовать готовую реализацию — метод [\_.cloneDeep(obj)](https://lodash.com/docs#cloneDeep) из JavaScript-библиотеки [lodash](https://lodash.com/).

Копия массива без глубокой копии

var oldArray = ["a", "b", "c"];

var newArray = oldArray.slice();

15. Null vs undefined

Summary. **null** is an assigned value. It means nothing. **undefined** typically means a variable has been declared but not defined yet.

16. Регулярные выражения - это шаблоны, используемые для сопоставления последовательностей символов в строках. В JavaScript регулярные выражения также являются объектами. Эти шаблоны используются в методах exec и test объекта RegExp, а также match, replace, search и split объекта String. Данная глава описывает регулярные выражения в JavaScript.

# 17. **Лексическое окружение (LexicalEnvironment)**

Каждый раз, когда в программе вызывается функция, внутри интерпретатора создается специальный словарь LexicalEnvironment (лексическое окружение), привязанный к этому вызову. Все определения констант, переменных и прочего внутри функции автоматически записываются в словарь. Имя определения (идентификатор, то есть имя константы, переменной и так далее) становится ключом, а значение определения становится значением в словаре. К таким определениям относятся аргументы, константы, функции, переменные и т.д. Лексическое окружение — это хранилище для данных в памяти и механизм для извлечения этих данных при обращении.

В примере ниже в комментариях показано состояние словаря перед выполнением каждой строчки кода. Не забывайте, что наполнение словаря происходит **при вызове** функции, а не при определении.

**const** showWarning **=** (field) **=>** {

*// LexicalEnvironment = { field: 'email' }*

**const** warning **=** `verify your ${field}, please`;

*// LexicalEnvironment = { warning: 'verify your email, please', field: 'email' }*

console.log(warning);

}

showWarning('email'); *// => verify your email, please*

Код console.log(warning) активизирует поиск значения идентификатора warning в лексическом окружении.

В процессе выполнения функции значения переменных могут меняться, что сразу же отражается в лексическом окружении. После выполнения функции её лексическое окружение уничтожается, а занятая им память освобождается.

Из этого поведения есть исключение — возврат функции. В следующем уроке мы рассмотрим связанный с ним механизм так называемых «замыканий». Ранее мы разбирали его в «[Введении в программирование](https://ru.hexlet.io/courses/introduction_to_programming/lessons/env/theory_unit)».

# **18. Map и Set**

Сейчас мы знаем о следующих сложных структурах данных:

* Объекты для хранения именованных коллекций.
* Массивы для хранения упорядоченных коллекций.

Но этого не всегда достаточно для решения повседневных задач. Поэтому также существуют Map и Set.

## [Map](https://learn.javascript.ru/map-set" \l "map)

[Map](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Map) – это коллекция ключ/значение, как и Object. Но основное отличие в том, что Map позволяет использовать ключи любого типа.

Методы и свойства:

* new Map() – создаёт коллекцию.
* map.set(key, value) – записывает по ключу key значение value.
* map.get(key) – возвращает значение по ключу или undefined, если ключ key отсутствует.
* map.has(key) – возвращает true, если ключ key присутствует в коллекции, иначе false.
* map.delete(key) – удаляет элемент по ключу key.
* map.clear() – очищает коллекцию от всех элементов.
* map.size – возвращает текущее количество элементов.

Например:

let map = new Map();

map.set("1", "str1"); // строка в качестве ключа

map.set(1, "num1"); // цифра как ключ

map.set(true, "bool1"); // булево значение как ключ

// помните, обычный объект Object приводит ключи к строкам?

// Map сохраняет тип ключей, так что в этом случае сохранится 2 разных значения:

alert(map.get(1)); // "num1"

alert(map.get("1")); // "str1"

alert(map.size); // 3

Как мы видим, в отличие от объектов, ключи не были приведены к строкам. Можно использовать любые типы данных для ключей.

**Map может использовать объекты в качестве ключей.**

1

**React**

# **1. В чем разница между элементом и компонентом React.JS?**

1. Если просто: элемент – это то, что вы хотите видеть на экране.
2. Если не так просто: элемент – объектное представление некоторого пользовательского интерфейса.

С компонентом все иначе. Это класс или функция, что может принимать данные и возвращать элементы

Общие вопросы

1. Agile, scrum, Kanban
2. Git, git flow

Git распределённая система управления версиями

 Система контроля версий — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Для контроля версий файлов в этой книге в качестве примера будет использоваться исходный код программного обеспечения, хотя на самом деле вы можете использовать контроль версий практически для любых типов файлов.

3.Redux

4. **SOLID** (сокр. от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) **single responsibility**, **open–closed**, **Liskov substitution**, **interface segregation** и **dependency inversion**) в программировании — [мнемонический](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%B5%D0%BC%D0%BE%D0%BD%D0%B8%D0%BA%D0%B0) [акроним](https://ru.wikipedia.org/wiki/%D0%90%D0%BA%D1%80%D0%BE%D0%BD%D0%B8%D0%BC), введённый Майклом Фэзерсом (*Michael Feathers*) для первых пяти принципов, названных [Робертом Мартином](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%80%D1%82%D0%B8%D0%BD,_%D0%A0%D0%BE%D0%B1%D0%B5%D1%80%D1%82_(%D0%B8%D0%BD%D0%B6%D0%B5%D0%BD%D0%B5%D1%80))[[1]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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)#cite_note-ub-old-web-solid-1)[[2]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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)#cite_note-ub-solid-2) в начале [2000-х](https://ru.wikipedia.org/wiki/2000-%D0%B5_%D0%B3%D0%BE%D0%B4%D1%8B)[[3]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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)#cite_note-metz-presentation-2009-3), которые означали 5 основных принципов [объектно-ориентированного программирования](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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) и [проектирования](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5).

## **Принципы**

Избавиться от «признаков плохого проекта»[[4]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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)#cite_note-:0-4) помогают следующие 5 принципов SOLID:

|  |  |  |
| --- | --- | --- |
| **Инициал** | **Представляет**[[1]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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)#cite_note-ub-old-web-solid-1) | **Название**[[4]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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)#cite_note-:0-4)**, понятие** |
| **S** | SRP[[5]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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)#cite_note-5) | [**Принцип единственной ответственности**](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%B5%D0%B4%D0%B8%D0%BD%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9_%D0%BE%D1%82%D0%B2%D0%B5%D1%82%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8)**(single responsibility principle)**  Для каждого класса должно быть определено единственное назначение. Все ресурсы, необходимые для его осуществления, должны быть [инкапсулированы](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%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)) в этот класс и подчинены только этой задаче. |
| **O** | OCP[[6]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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)#cite_note-6) | [**Принцип открытости/закрытости**](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%BE%D1%82%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D1%81%D1%82%D0%B8/%D0%B7%D0%B0%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D1%81%D1%82%D0%B8)**(open–closed principle)**  «программные сущности … должны быть открыты для расширения, но закрыты для модификации». |
| **L** | LSP[[7]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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)#cite_note-7) | [**Принцип подстановки Лисков**](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%BF%D0%BE%D0%B4%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B8_%D0%9B%D0%B8%D1%81%D0%BA%D0%BE%D0%B2)**(Liskov substitution principle)**  «объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы». Производный класс должен быть взаимозаменяем с родительским классом. |
| **I** | ISP[[8]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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)#cite_note-8) | [**Принцип разделения интерфейса**](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D1%80%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D0%B0)**(interface segregation principle)**  «много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения»[[9]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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)#cite_note-martin-design-principles-9). |
| **D** | DIP[[10]](https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%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)#cite_note-10) | [**Принцип инверсии зависимостей**](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%B8%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B8_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B5%D0%B9)**(dependency inversion principle)**  «Зависимость на Абстракциях. Нет зависимости на что-то конкретное». |

1. ООП Основные [принципы](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF) структурирования в случае ООП связаны с различными аспектами базового понимания предметной задачи, которое требуется для оптимального управления соответствующей моделью:

* [абстракция](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%86%D0%B8%D1%8F_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) для выделения в моделируемом предмете важного для решения конкретной задачи по предмету, в конечном счёте — контекстное понимание предмета, формализуемое в виде класса;
* [инкапсуляция](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%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)) для быстрой и безопасной организации собственно иерархической управляемости: чтобы было достаточно простой команды «что делать», без одновременного уточнения как именно делать, так как это уже другой уровень управления;
* [наследование](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%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)) для быстрой и безопасной организации родственных понятий: чтобы было достаточно на каждом иерархическом шаге учитывать только изменения, не дублируя всё остальное, учтённое на предыдущих шагах;
* [полиморфизм](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) для определения точки, в которой единое управление лучше распараллелить или наоборот — собрать воедино.

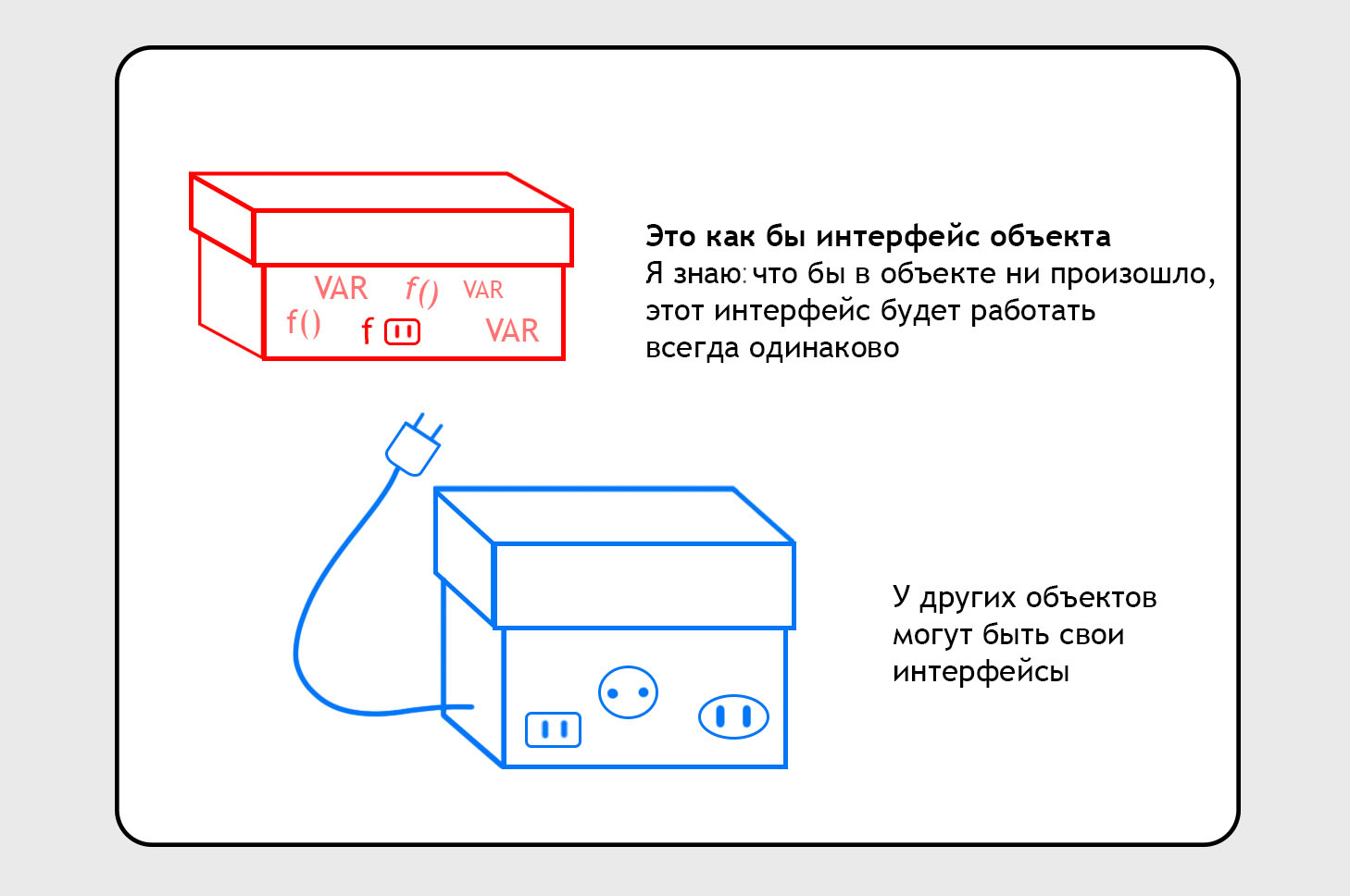
**Инкапсуляция, абстракция, наследование, полиморфизм**

**Инкапсуляция — объект независим:** каждый объект устроен так, что нужные для него данные живут внутри этого объекта, а не где-то снаружи в программе. Например, если у меня есть объект «Пользователь», то у меня в нём будут все данные о пользователе: и имя, и адрес, и всё остальное. И в нём же будут методы «Проверить адрес» или «Подписать на рассылку».

**Абстракция — у объекта есть «интерфейс»:** у объекта есть методы и свойства, к которым мы можем обратиться извне этого объекта. Так же, как мы можем нажать кнопку на блендере. У блендера есть много всего внутри, что заставляет его работать, но на главной панели есть только кнопка. Вот эта кнопка и есть абстрактный интерфейс.

В программе мы можем сказать: «Удалить пользователя». На языке ООП это будет «пользователь.удалить()» — то есть мы обращаемся к объекту «пользователь» и вызываем метод «удалить». Кайф в том, что нам не так важно, как именно будет происходить удаление: ООП позволяет нам не думать об этом в момент обращения.

Например, над магазином работают два программиста: один пишет модуль заказа, а второй — модуль доставки. У первого в объекте «заказ» есть метод «отменить». И вот второму нужно из-за доставки отменить заказ. И он спокойно пишет: «заказ.отменить()». Ему неважно, как другой программист будет реализовывать отмену: какие он отправит письма, что запишет в базу данных, какие выведет предупреждения.



**Наследование — способность к копированию.** ООП позволяет создавать много объектов по образу и подобию другого объекта. Это позволяет не копипастить код по двести раз, а один раз нормально написать и потом много раз использовать.

Например, у вас может быть некий идеальный объект «Пользователь»: в нём вы прописываете всё, что может происходить с пользователем. У вас могут быть свойства: имя, возраст, адрес, номер карты. И могут быть методы «Дать скидку», «Проверить заказ», «Найти заказы», «Позвонить».

На основе этого идеального пользователя вы можете создать реального «Покупателя Ивана». У него при создании будут все свойства и методы, которые вы задали у идеального покупателя, плюс могут быть какие-то свои, если захотите.

Идеальные объекты программисты называют классами.

**Полиморфизм — единый язык общения.** В ООП важно, чтобы все объекты общались друг с другом на понятном им языке. И если у разных объектов есть метод «Удалить», то он должен делать именно это и писаться везде одинаково. Нельзя, чтобы у одного объекта это было «Удалить», а у другого «Стереть».

При этом внутри объекта методы могут быть реализованы по-разному. Например, удалить товар — это выдать предупреждение, а потом пометить товар в базе данных как удалённый. А удалить пользователя — это отменить его покупки, отписать от рассылки и заархивировать историю его покупок. События разные, но для программиста это неважно. У него просто есть метод «Удалить()», и он ему доверяет.

