JavaScript

Типы днных в js

В JavaScript есть 8 основных типов.

- `number` для любых чисел: целочисленных или чисел с плавающей точкой; целочисленные значения ограничены диапазоном ±(253-1).
- `bigint` для целых чисел произвольной длины.
- `string` для строк. Строка может содержать ноль или больше символов, нет отдельного символьного типа.
- `boolean` для true/false.
- `null` для неизвестных значений – отдельный тип, имеющий одно значение null.
- `undefined` для неприсвоенных значений – отдельный тип, имеющий одно значение undefined.
- `object` для более сложных структур данных.
- `symbol` для уникальных идентификаторов.Символы используются для создания скрытых свойств объектов. В отличие от свойств, ключом которых является строка, символьные свойства может читать только владелец символа. Скрытые свойства не видны при его обходе с помощью

Оператор typeof позволяет нам увидеть, какой тип данных сохранён в переменной.

Имеет две формы: `typeof x` или `typeof(x).` Возвращает строку с именем типа. Например, "string". Для null возвращается "object" – это ошибка в языке, на самом деле это не объект.

Разница между == и ===

`==` сранивает с приведением типов 0 == false // true

`===` стравнивает строго 0 === false // false

Как можно объявить переменную в js

`a = 5` ; // аналогичен var

`var b = 5` ; // глобальная или функциональная область видимости

`let c = 5` ; // let, const имеют блочную область видимости (например в if, try и тд) `const d = 5` ; // константа, но может изменяться если присвоенное значение объект или массив

В чем разница между null и undefined

- `null` для неизвестных значений – отдельный тип, имеющий одно значение null. При обнуление значения используется присвоение ей null. Если мы хотим показать что в переменной значение пустое, так же должны присвоить ей null.
- `undefined` для неприсвоенных значений – отдельный тип, имеющий одно значение undefined.

Шаблонные литералы

- Можно вставлять js выражения

```
const message = `You can ${age < 21 ? 'not' : ''} view this page`
```

- разрешен перенос строки

```
`
  <img>
      С новой
      строки
  </img>
`
```

Объекты

Объекты – ссылочный тип данных. То есть переменные и константы хранят не сами объекты (их данные), а ссылку на них.

spread оператор

Поверхностное копирование (clone) и слияние (merge) можно объединить в одну операцию. Это позволяет "обновлять" объекты в функциональном стиле, другими словами мы создаем новые объекты на основе старых, вместо их обновления.

```
// Поверхностное копирование
const user = { name: 'Vasya', married: true, age: 25 };
const user2 = { name: 'Irina', surname: 'Petrova' };

const mergedObject = { ...user, ...user2 };
// Object.assign({}, user, user2);
```

rest оператор

С его помощью во время деструктуризации можно собрать все "оставшиеся" свойства в один объект

```
const user = { name: 'Tirion', email: 'support@hexlet.io', age: 44 };
const { name, ...rest } = user;
console.log(rest);
// => { email: 'support@hexlet.io', age: 44 }
```

Деструктуризация

```
const person = { firstName: 'Rasmus', lastName: 'Lerdorf', manager: true };
const { firstName, manager } = person;
console.log(firstName); // => 'Rasmus'
console.log(manager); // => true

//При деструктуризации можно переименовывать имена.
// Такое бывает нужно, если подобная
```

```
// константа уже определена выше.
const person = { firstName: 'Rasmus', lastName: 'Lerdorf', manager: true };
const { manager: isManager } = person;
console.log(isManager); // => true
// В случае отсутствия свойств в объекте,
// деструктуризация позволяет задавать
// значения по умолчанию для таких свойств:
const person = { firstName: 'Rasmus', lastName: 'Lerdorf' };
console.log(person.manager); // undefined
const { manager = false } = person;
console.log(manager); // => false
//Деструктуризация может быть вложенной.
const { links, attributes: user, relationships: { author } } = response.data;
```

Ключи и значения

```
// получаем массив ключей
const course = { name: 'JS: React', slug: 'js-react' };
const keys = Object.keys(course); // [ 'name', 'slug' ]
// получаем массив значений
const course = { name: 'JS: React', slug: 'js-react' };
const values = Object.values(course); // [ 'JS: React', 'js-react' ]
// Ну, и последний вариант, метод, который возвращает сразу ключи и значения объекта
const course = { name: 'JS: React', slug: 'js-react' };
const entries = Object.entries(course); // [[ 'name', 'JS: React' ], [ 'slug', 'js-
react' ]]
```

Деструктуризация и обход

```
for (const [key, value] of entries) {
  console.log(key);
  console.log(value);
}
```

Массивы

Массив внутри – это тоже объект:

```
typeof []; // 'object'
```

Проектируя функции, работающие с массивами, есть два пути: менять исходный массив или формировать внутри новый и возвращать его наружу. Какой лучше? В подавляющем большинстве стоит предпочитать второй.

Агрегацией называются любые вычисления, которые, как правило, строятся на основ е всего набора данных, например, поиск максимального, среднего, суммы и так далее.

rest оператор

Rest-оператор позволяет "свернуть" часть элементов во время деструктуризации. Например с его помощью можно разложить массив на первый, второй элементы и все остальные:

```
const [first, second, ...rest] = 'some string';
console.log(first); // => 's'
console.log(second); // => 'o'
console.log(rest); // => [ 'm', 'e', ' ', 's', 't', 'r', 'i', 'n', 'g' ]
```

spread оператор

С его помощью обычно копируют или сливают массивы. Spread-оператор нередко используется для копирования массива. Копирование предотвращает изменение исходного массива, в том случае, когда необходимо менять его копию

```
const russianCities = ['moscow', 'kazan'];
const copy = [...russianCities];
```

деструктуризация

```
const [firstElement,
  secondElement,
  thirdElement = 3] = [1, 2];

console.log(firstElement);  // => 1
console.log(secondElement); // => 2
console.log(thirdElement);  // => 3
```

Что такое set/map?

Set

В отличие от массивов, объекты типа Set (мы будем называть их «коллекциями») представляют собой коллекции, содержащие данные в формате ключ/значение. Значение элемента в Set может присутствовать только в одном экземпляре, что обеспечивает его уникальность в коллекции Set.

Его основные методы это:

- `new Set(iterable)` – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.
- `set.add(value)` – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.
- `set.delete(value)` – удаляет значение, возвращает true, если value было в множестве на момент вызова, иначе false.
- `set.has(value)` – возвращает true, если значение присутствует в множестве, иначе false.
- `set.clear()` – удаляет все имеющиеся значения.
- `set.size` – возвращает количество элементов в множестве.

Map

`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.

Map, filter, forEach, every/some/find, reduce

Каждый из этих методов итерируется по массиву.

map

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

```
const numbers = [1, 4, 9];
const doubles = numbers.map(function(num) {
  return num * 2;
});
// теперь doubles равен [2, 8, 18], а numbers всё ещё равен [1, 4, 9]
```

filter

Метод `filter()` создаёт новый массив со всеми элементами, прошедшими проверку, задаваемую в передаваемой функции.

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter(word => word.length > 6);

console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

forEach

Метод `forEach()` используется для перебора массива.

```
var arr = ["Яблоко", "Апельсин", "Груша"];

arr.forEach(function(item, i, arr) {
  alert( i + ": " + item + " (массив:" + arr + ")" );
});
```

every/some/find /]

Метод `every()` проверяет, удовлетворяют ли все элементы массива условию, заданному в передаваемой функции.

```
function isBigEnough(element, index, array) {
  return element >= 10;
}
[12, 5, 8, 130, 44].every(isBigEnough);   // false
[12, 54, 18, 130, 44].every(isBigEnough); // true
```

Метод `some()` проверяет, удовлетворяет ли какой-либо элемент массива условию, заданному в передаваемой функции.

```
const array = [1, 2, 3, 4, 5];
// checks whether an element is even
const even = (element) => element % 2 === 0;
console.log(array.some(even));
// expected output: true
```

Метод `find()` возвращает значение первого найденного в массиве элемента, которое удовлетворяет условию переданному в callback функции. В противном случае возвращается undefined.

```
function isPrime(element, index, array) {
  var start = 2;
  while (start <= Math.sqrt(element)) {
    if (element % start++ < 1) {
      return false;
    }
  }
  return element > 1;
}

console.log([4, 6, 8, 12].find(isPrime)); // undefined, не найдено
console.log([4, 5, 8, 12].find(isPrime)); //5
```

reduce

Метод `reduce()` применяет функцию reducer к каждому элементу массива (слева-направо), возвращая одно результирующее значение.

```
[0, 1, 2, 3, 4].reduce(function(accumulator, currentValue, index, array) {
  return accumulator + currentValue;
}, 10);
// 10 + 0 = 10;
// 10 + 1 = 11; ....
```

this: контекст выполнения функций

Грубо говоря, this — это ссылка на некий объект, к свойствам которого можно достучаться внутри вызова функции. Этот this — и есть контекст выполнения.

Если мы запускаем JS-код в браузере, то глобальным объектом будет window. Если мы запускаем код в Node-окружении, то global. Когда мы вызываем функцию, значением this может быть лишь глобальный объект или undefined при использовании 'use strict'.

В нестрогом режиме при выполнении в браузере this при вызове функции будет равен window:

```
function whatsThis() {
  console.log(this === window)
}

whatsThis() // true
```

То же — если функция объявлена внутри функции:

```
function whatsThis() {
  function whatInside() {
    console.log(this === window)
  }

  whatInside()
}

whatsThis() // true
```

И то же — если функция будет анонимной и, например, вызвана немедленно:

```
;(function () {
  console.log(this === window)
})() // true
```

```
"use strict"

function whatsThis() {
  console.log(this === undefined)
}

whatsThis() // true
```

В этом случае значение this — этот объект.

```
const user = {
  name: "Alex",
  greet() {
    console.log(`Hello, my name is ${this.name}`)
  },
}

user.greet() // Hello, my name is Alex
```

Обратим внимание, что this определяется в момент вызова функции. Если записать метод объекта в переменную и вызвать её, значение this изменится. При вызове через точку user.greet значение this равняется объекту до точки (user)

```
const user = {
  name: "Alex",
  greet() {
    console.log(`Hello, my name is ${this.name}`)
  },
}

const greet = user.greet
greet()
// Hello, my name is
```

Непрямой вызов

Непрямым вызовом называют вызов функций через .call() или .apply().

Оба первым аргументом принимают this. То есть они позволяют настроить контекст снаружи, к тому же — явно. Разница между .call() и .apply() — в том, как они принимают аргументы для самой функции после this.

```
function greet(greetWord, emoticon) {
  console.log(`${greetWord} ${this.name} ${emoticon}`)
}

const user1 = { name: "Alex" }
const user2 = { name: "Ivan" }

// .call() принимает аргументы списком через запятую:
greet.call(user1, "Hello,", ":-)") // Hello, Alex :-)
greet.call(user2, "Good morning,", ":-D") // Good morning, Ivan :-D

// .apply() же — принимает массив аргументов:
greet.apply(user1, ["Hello,", ":-)"]) // Hello, Alex :-)
greet.apply(user2, ["Good morning,", ":-D"]) // Good morning, Ivan :-D

// В остальном они идентичны.
```

##bind .bind() в отличие от .call() и .apply() не вызывает функцию сразу. Вместо этого он возвращает другую функцию — связанную с указанным контекстом навсегда. Контекст у этой функции изменить невозможно.

```
function getAge() {
  console.log(this.age);
}

const howOldAmI = getAge.bind({age: 20}).bind({age: 30})
```

```
howOldAmI(); //20
```

Разница между стрелочной и обычной функцией

Отличие стрелочных функций от обычных в том, что у них нет this. Также стрелочные функции в силу своего синтаксиса анонимны, если не присвоить их переменной.

```
const arrowFunction = () => {}
// Стрелочная функция записывается сильно короче, чем обычная.
// Ключевое слово function не требуется, так как сама нотация
// "() =>" подразумевает функцию.

// Если функция ничему не присвоена, то она анонимна:
;(() => {
  console.log("Hello world")
})()
```

Так как у них нет this, то внутри нельзя получить доступ в arguments:

```
const arrow = () => {
  console.log(arguments)
}

arrow()
// ReferenceError: arguments is not defined.
```

Также из-за отсутствия this их нельзя использовать с new. Стрелочные функции не могут быть функциями-конструкторами.

```
const Factory = () => {
  return {
    name: "Arthur",
  }
}

const person = new Factory()
// TypeError: Factory is not a constructor.

// С обычной функцией — порядок.
function Factory() {
  return {
    name: "Arthur",
  }
}

const person = new Factory()
```

Изоляция модулей с помощью IIFE

Immediately Invoked Function Expression, IIFE — это функция, которая выполняется сразу же после того, как была определена.

```
;(function () {
  // ...Тело функции
})()
```

При помощи IIFE мы можем использовать одинаковые названия переменных, не боясь, что они случайно перезапишут значения переменных из чужих модулей, если мы не контролируем кодовую базу полностью сами.

```
;(function module1() {
  const a = 42
  console.log(a)
})()
;(function module2() {
  const a = '43!'
  alert(a)
})()
```

«Поднятие» переменных (hoisting)

```
var hi = 'Hello world!'
console.log(window.hi)
// Hello world!
```

```
function scope() {
  a = 42
  var b = 43
}

scope()

console.log(a)
// 42
console.log(b)
// Reference error
```

Так как переменная a не была объявлена, то JavaScript сам решил, где объявлять переменную, и «поднял» объявление наверх.

Область видимости

Функциональная область видимости — это область видимости в пределах тела функции. Можно сказать, что она ограничена { и } функции.

```
function outer() {
  function inner() {
    const a = 42
  }

  console.log(a)
  // Reference error
}
```

Блочная область видимости ограничена программным блоком, обозначенным при помощи { и }. Простейший пример такой области — это выражение внутри скобок:

```
const a = 42
console.log(a)
// 42

if (true) {
  const b = 43
  console.log(a)
  // 42
  console.log(b)
  // 43
}

console.log(b)
// ReferenceError: Can't find variable: b
```

Что такое замыкание?

Замыкание это функция у которой есть доступ к своей внешней функции по области видимости, даже после того, как внешняя функция прекратилась. Это говорит о том, что замыкание может запоминать и получать доступ к переменным, и аргументам своей внешней функции, даже после того, как та прекратит выполнение.

```
function getCounter() {
  let counter = 0;
  return function() {
    return counter++;
  }
}
let count = getCounter();
console.log(count());  // 0
console.log(count());  // 1
console.log(count());  // 2
```

И снова, мы храним анонимную внутреннюю функцию, возвращенную функцией getCounter в переменной count. Так как функция count теперь замыкание, она может получать доступ к переменной counter в функции getCounter, даже после того, как та завершится. Но обратите внимание, что значение counter не сбрасывается до 0 при каждом вызове count, как вроде бы она должна делать. Так

происходит, потому что при каждом вызове count(), создаётся новая область видимости, но есть только одна область видимости, созданная для getCounter, так как переменная counter объявлена в области видимости getCounter(), она увеличится при каждом вызове функции count, вместо того, чтобы сброситься до 0.

Браузерное окружение, BOM

Современный JavaScript используется не только в браузерах. Среда, в которой он запускается, будь то браузер, сервер или что-то ещё, называется окружением.

У разных окружений разные возможности и функциональность. В этой статье рассмотрим браузерное окружение и браузерную модель документа.

Browser Object Model предоставляет доступ к navigator, location, fetch и другим объектам.

navigator - Объект navigator содержит информацию о браузере: название, версия, платформа, доступные плагины, доступ к буферу обмена и прочее. Это один из самых больших объектов в окружении. С помощью этого объекта можно узнать, разрешён ли доступ к кукам, получить доступ к буферу обмена, геолокации, узнать, с какого браузера пользователь смотрит на страницу через userAgent.

```
if (`bluetooth` in navigator) {
  // Есть доступ к Bluetooth API.
}

if (`serviceWorker` in navigator) {
  // Есть доступ к Service Worker API.
}
```

Забавный факт: поле userAgent объекта navigator часто используется, чтобы определять, в каком именно браузере пользователь смотрит страницу сайта. Но читать его глазами достаточно трудно, поэтому лучше это дело оставить какому-нибудь парсеру.

screen - Объект screen содержит информацию об экране браузера.

```
// Без учёта полосы:
const screenWidth = screen.width

// С учётом полосы прокрутки:
const withoutScrollBar = screen.availWidth
```

location - Объект location даёт возможность узнать, на какой странице мы находимся (какой у неё URL) и перейти на другую страницу программно.

```
location.href = "/another-page"
// Так браузер перейдёт на страницу
// по адресу another-page на текущем сайте.

location.href = "https://google.com"
// так браузер перейдёт на другой сайт.
```

fetch - Fetch предоставляет возможность работы с сетью, с его помощью можно отправлять запросы на сервер.

history - history даёт доступ к истории браузера, которая ограничена текущей вкладкой. То есть с её помощью можно перейти на страницу назад, только если мы пришли с неё.

```
history.pushState(null, null, "/new/page/url")
```

localStorage, sessionStorage - Локальные хранилища используются, чтобы хранить какие-то данные в браузере пользователя.

```
function saveToStorage(key, data) {
  try {
    // Если браузер не поддерживает localStorage,
    // блок try обезопасит код от неожиданной ошибки.
    window.localStorage.setItem(key', JSON.stringify(data));
  }
  catch {
    alert('Failed to save data to local storage.');
  }
}

function loadFromStorage(key) {
  try {
    return JSON.parse(window.localStorage.getItem(key));
  }
  catch {
    alert('Failed to load data from local storage.');
  }
}

saveToStorage('user', {name: 'Alex', age: 26});
loadFromStorage('user');
```

DOM

DOM (Document Object Model) — это специальная древовидная структура, которая позволяет управлять HTML-разметкой из JavaScript-кода. Управление обычно состоит из добавления и удаления элементов, изменения их стилей и содержимого.

Браузер создаёт DOM при загрузке страницы, складывает его в переменную document и сообщает, что DOM создан, с помощью события DOMContentLoaded. С переменной document начинается любая работа с HTML-разметкой в JavaScript.

Как браузер рисует страницы

Чтобы нарисовать на экране результат работы нашего кода, браузеру нужно выполнить несколько этапов:

- Сперва ему нужно скачать исходники.
- Затем их нужно прочитать и распарсить.

- После этого браузер приступает к рендерингу — отрисовке.

DOM

Браузер работает не с текстом разметки, а с абстракциями над ним. Одна из таких абстракций, результат парсинга HTML-кода, называется DOM. DOM (Document Object Model) — абстрактное представление HTML-документа, с помощью которого браузер может получать доступ к его элементам, изменять его структуру и оформление. DOM — это дерево. Корень этого дерева — это элемент HTML, все остальные элементы — это дочерние узлы.

CSSOM

Когда браузер находит элемент link, который указывает на файл стилей, браузер скачивает и парсит его. Результат парсинга CSS-кода — CSSOM. CSSOM (CSS Object Model) — по аналогии с DOM, представление стилевых правил в виде дерева.

Чтение стилей приостанавливает чтение кода страницы. Поэтому рекомендуется в самом начале отдавать только критичные стили — которые есть на всех страницах и конкретно на этой. Так мы уменьшаем время ожидания, пока «страница загрузится». После того, как браузер составил DOM и CSSOM, он объединяет их в общее дерево рендеринга — Render Tree.

Render Tree

После того, как браузер составил DOM и CSSOM, он объединяет их в общее дерево рендеринга — Render Tree.

Вычисление позиции и размеров, Layout

После того как у браузера появилось дерево рендеринга (Render Tree), он начинает «расставлять» элементы на странице. Этот процесс называется Layout. Чтобы понимать, где какой элемент должен находиться и как он влияет на расположение других элементов, браузер рассчитывает размеры и положение каждого рекурсивно

Именно поэтому при вёрстке макетов рекомендуется «находиться в потоке» — чтобы браузеру не приходилось несколько раз пересчитывать один и тот же элемент, так страница отрисовывается быстрее.

Перерисовка, Reflow (relayout) и Repaint

Процесс отрисовки — циклический. Браузер перерисовывает экран каждый раз, когда на странице происходят какие-то изменения.

Если, например, в DOM-дереве добавился новый узел, или изменился текст, то браузер построит новое дерево рендеринга и запустит вычисление позиции и отрисовку заново.

Один цикл обновления — это animation frame.

Зная «расписание отрисовки» браузера, мы можем «предупредить» его, что хотим запустить какую-то анимацию на каждый новый фрейм. Это можно сделать с помощью requestAnimationFrame.

React и VirtualDOM

Virtual DOM, вероятно, является одной из самых известных функций React.js и была одним из ключей к его успеху. Концепция предполагает, что добавление HTML в браузер является самой дорогой частью. Вместо того, чтобы делать это напрямую, приложение создает модель, которая представляет HTML, а React переводит части, которые изменились, в настоящий HTML.

При первоначальном рендеринге все заканчивается отрисовкой HTML-кода. Но при повторном рендеринге обновляются только те части, которые были изменены. Это минимизирует одну из самых дорогих частей веб-приложения.

Асинхронность в JS

При вызове какой-то функции она попадает в так называемый стек вызовов.

Стек — это структура данных, в которой элементы упорядочены так, что последний элемент, который попадает в стек, выходит из него первым (LIFO: last in, first out).

Дополнительная функциональность (Web API) берёт на себя работу с таймерами, интервалами, обработчиками событий. То есть, когда мы регистрируем обработчик клика на кнопку — он попадает в окружение Web API. Именно оно знает, когда обработчик нужно вызвать.

Управление тем, как должны вызываться функции Web API, берёт на себя цикл событий (Event loop). Работает он по приницпу «первый зашёл, первый вышел» (FIFO: first in, first out).

Callback — (колбэк, функция обратного вызова) — функция, которая вызывается в ответ на совершение некоторого события.

Promise — это объект-обёртка для асинхронного кода. Он содержит в себе состояние: вначале pending («ожидание»), затем — одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).

В понятиях цикла событий промис работает так же, как колбэк: функция, которая должна выполниться (resolve или reject), находится в окружении Web API, а при наступлении события — попадает в очередь задач, откуда потом — в стек вызова.

Веб-воркер

Веб-воркер — штука, которая умеет выполняться в браузере параллельно с основным скриптом. А значит — выполняться, не блокируя отрисовку страницы. Проще говоря, если запустить что-то тяжёлое в основном скрипте, то страница будет тормозить, а если в веб-воркере — то не будет.

Веб-воркер не имеет доступа ни к DOM-дереву, ни к объекту window. Напрямую к локальному хранилищу обратиться внутри него тоже нельзя. Всё потому, что работа с DOM-деревом и локальным хранилищем — последовательная, а веб-воркер работает параллельно.

Архитектура и паттерны проектирования

Порождающие паттерны проектирования

Архитектура приложения — это набор решений о том, как модули приложения будут общаться друг с другом и с внешним миром. MVC (сокращение от Model—View—Controller) — это архитектурный паттерн, который делит модули на три группы:

модель (model), представление (view), контроллер (controller).

Паттерн проектирования — шаблонное решение частой архитектурной проблемы.

Фабрика Фабрика (англ. factory) создаёт объект, избавляя нас от необходимости знать детали создания. Фабрика в программировании принимает от нас сигнал, что надо создать объект, и создаёт его, инкапсулируя логику создания внутри себя. Используйте фабрику, если создание объекта сложнее, чем 1–2 строки кода.

```
function createGuitar(stringsCount = 6) {
  return {
    strings: stringsCount,
    frets: 24,
    fretBoardMaterial: "cedar",
    boardMaterial: "maple",
  };
}
```

Абстрактная фабрика Абстрактная фабрика (англ. abstract factory) — это фабрика фабрик 😃

Этот шаблон группирует связанные или похожие фабрики объектов вместе, позволяя выбирать нужную в зависимости от ситуации.

Абстрактная фабрика не возвращает конкретный объект, вместо этого она описывает тип объекта, который будет создан.

Если в приложении есть общая логика создания связанных или похожих, но не одинаковых объектов, абстрактная фабрика поможет избавиться от дублирования и инкапсулировать правила создания в себе.

```
// Общий интерфейс:
interface ReservationFactory {
  reserveInstrument(): Instrument;
  notifyPlayer(): Musician;
}

// Реализации под разные инструменты:
class ViolinReservation implements ReservationFactory {
  reserveInstrument = () => new Violin();
  notifyPlayer = () => new Violinist();
}

class CelloReservation implements ReservationFactory {
  reserveInstrument = () => new Cello();
  notifyPlayer = () => new Cellist();
}
```

Билдер Билдер, или строитель, (англ. builder) позволяет создавать объекты, добавляя им свойства по заданным правилам. Он полезен, когда при создании объекта нужно выполнить много шагов, часть из которых могут быть необязательными. Если в приложении требуется создавать объекты с разными особенностями, или процесс создания объекта делится на отдельные шаги, то билдер помогает не засорять код условиями и проверками.

```
class DrinkBuilder {
  settings = {
    base: "espresso",
  };

  addMilk = () => {
    this.settings.milk = true;
```

```
    return this;
  };

  addSugar = () => {
    this.settings.sugar = true;
    return this;
  };

  addCream = () => {
    this.settings.cream = true;
    return this;
  };

  addSyrup = () => {
    this.settings.syrup = true;
    return this;
  };

  build = () => new Drink(this.settings);
}

const latte = new DrinkBuilder().addMilk().build();
const withSugarAndCream = new DrinkBuilder().addSugar().addCream().build();
```

Синглтон Синглтон, или одиночка, (англ. singleton) — это шаблон, который позволяет создать лишь один объект, а при попытке создать новый возвращает уже созданный.

```
class Sun {
  // Держим ссылку на созданный объект:
  static instance = null;

  // Делаем конструктор приватным:
  #constructor() {}

  static get instance() {
    // Если объект был создан ранее, возвращаем его:
    if (this.instance) return this.instance;

    // Иначе создаём новый экземпляр:
    this.instance = new this();
    return this.instance;
  }
}
// При первом вызове создастся новый объект:
const sun = Sun.instance;

// В дальнейшем instance будет возвращать
// ранее созданный объект:
const sun1 = Sun.instance;
const sun2 = Sun.instance;
```

Структурные паттерны проектирования

Адаптер - (англ. adapter) помогает сделать не совместимое с нашим модулем API совместимым и использовать его. Адаптер (англ. adapter) помогает сделать не совместимое с нашим модулем API совместимым и использовать его.

```
function fakeAPI() {
  return {
    entries: [
      {
        user_name: "Alex",
        email_address: "some@site.com",
        ID: "some-unique-id",
      },
      {
        user_name: "Alice",
        email_address: "some@other-site.com",
        ID: "another-unique-id",
      },
    ],
  };
}

const wantedResponse = [{
  userName: "Alex",
  email: "some@site.com",
  id: 'some-unique-id'
}, {
  userName: "Alice",
  email: "some@other-site.com",
  id: "another-unique-id"
}]

function responseToWantedAdapter(response) {
  return response.entries.map((entry) => ({
    userName: entry.user_name,
    email: entry.email_address,
    id: entry.ID,
  }));
}
```

Фасад - прячет за собой сложную логику других модулей, предоставляя более простые методы или функции.

Допустим, мы пишем мобильное приложение — пульт для кофеварки.

Мы хотим добавить кнопку «Нагреть воду» или «Помолоть зерно», но кофеварка предлагает нам более атомарное API: она может по отдельности включить машину, узнать, сколько воды набрано, включить набор воды, отключить набор воды и т. д.

```
class CoffeeMachine {
  turnOn() {}
  getWaterLevel() {}
  getWater() {}
  turnOnHeater() {}
  turnOffHeater() {}
  getTemperature() {}
  // ...
}

const machine = new CoffeeMachine();

function heatWater() {
  machine.turnOn();

  while (machine.getWaterLevel() <= 1000) {
    machine.getWater();
  }

  machine.turnOnHeater();

  if (machine.getTemperature() <= 90) {
    machine.turnOffHeater();
  }
}

heatWater();
```

Декоратор - (англ. decorator) позволяет динамически менять поведение объекта в рантайме. Допустим, нам надо логировать каждый вызов функции update():

```
function loggingDecorator(fn) {
  return function wrapped(...args) {
    console.log(`Logging... ${args.join(",")}`);
    return fn(...args);
  };
}
```

SOLID

Принцип единой ответственности (SRP) — S Секция статьи "Принцип единой ответственности (SRP) — S"

Принцип предлагает нам проводить границы между модулями так, чтобы изменение в бизнес-логике затрагивало как можно меньше модулей, в идеале — один.

Принцип открытости и закрытости (OCP) — O Секция статьи "Принцип открытости и закрытости (OCP) — O"

Модули, которые удовлетворяют этому принципу:

- открыты для расширения — их функциональность может быть дополнена с помощью других модулей, если изменятся требования;
- закрыты для изменения — их код менять нельзя (можно лишь исправлять ошибки).

Принцип подстановки Барбары Лисков (LSP) — L

Реализующие классы не должны противоречить базовому типу или интерфейсу. Поведение таких классов должно быть ожидаемым для функций, которые используют базовый тип.

Принцип разделения интерфейса (ISP) — I

Принцип разделения интерфейса (Interface Segregation Principle) содержит правила и ограничения для того, как следует проектировать интерфейсы.

В частности он предлагает группировать их по темам или фичам. Можно сказать, что это такой SRP для интерфейсов.

Принципе инверсии зависимостей (DIP) — D

- Высокоуровневые модули не должны зависеть от низкоуровневых; оба типа должны зависеть от абстракций (интерфейсов).
- Абстракции не должны зависеть от деталей, детали должны зависеть от абстракций.