Front-end

JavaScript

JS. Map, Set

- Set
- Map
- Регулярные выражения
- Паттерны

JS. Map, Set

Set (по-русски говорят множество) — коллекция для хранения уникальных значений любого типа. Одно и то же значение нельзя добавить в Set больше одного раза.

Set — это неиндексированная коллекция, положить элемент в коллекцию можно, но достать нельзя. По элементам коллекции можно итерироваться.

Основные методы для работы с коллекцией:

add() — добавить элемент.

delete() — удалить элемент.

has() — проверить, есть ли элемент в коллекции.

clear() — очистить коллекцию.

forEach() — выполнить функцию для каждого элемента в коллекции, аналогично одноимённому методу массива.

Содержит свойство **size** для получения количества элементов в коллекции.

JS. Map, Set

```
const uniqueIds = new Set()

uniqueIds.add(123)
uniqueIds.add(456)
uniqueIds.add(111)
uniqueIds.add(123)

console.log(uniqueIds.size) // => 3
console.log(uniqueIds.has(111)) // => true

uniqueIds.delete(111)
console.log(uniqueIds.size) // => 2

uniqueIds.clear()
console.log(uniqueIds.size)
```

JS. Map, Set

Коллекция Set может хранить произвольный набор значений — нет разницы, хранить в коллекции примитивные типы, объекты или массивы. Set гарантирует, что одно и то же значение не может попасть в коллекцию больше одного раза. Если нам нужно получить коллекцию уникальных значений из массива неуникальных — Set один из способов этого достичь.

JS. Map, Set

Создание коллекции

Коллекция создаётся при помощи конструктора.

Можно создать пустой Set:

```
const set = new Set()

console.log(set.size) // => 0
```

JS. Map, Set

Или сразу добавить в него элементы. Для этого нужно передать в конструктор итерируемый список значений. Обычно это массив:

```
const filled = new Set([1, 2, 3, 3, 3, 'hello'])

console.log(filled.size) // => 4
```

JS. Map, Set

Работа с коллекцией

Set предоставляет небольшой набор методов, но их достаточно в большинстве случаев.

Добавляют элемент в Set с помощью метода add(), а удаляют с помощью delete(). В оба метода передаётся элемент, который нужно добавить или удалить:

```
const filled = new Set([1, 2, 3, '3', 3, 'hello'])

filled.add(100)
filled.delete(1)
```

JS. Map, Set

Количество элементов в множестве хранится в свойстве size. Проверить количество элементов в множестве filled:

```
console.log(filled.size) // => 5
```

JS. Map, Set

Set позволяет проверить, был ли элемент уже добавлен. За это отвечает метод has():

```
const filled = new Set([1, 2, 3, '3', 3, 'hello'])

console.log(filled.has(3)) // => true
console.log(filled.has('3')) // => true

console.log(filled.has('My name')) // => false
```

JS. Map, Set

Полностью очистить Set можно методом clear(). Технически это то же самое, что и создать новый Set:

```
const filled = new Set([1, 2, 3, 3, 3, 'hello'])

filled.clear()


console.log(filled.size) // => 0
```

JS. Map, Set

Обход

Set — это неиндексированная коллекция. В этой структуре данных нет понятия индекса элемента, поэтому нельзя получить произвольный элемент коллекции. В коллекцию можно только положить значение, а получить отдельное значение нельзя.

Основной инструмент работы с Set — обход коллекции. При обходе коллекции нам гарантируется, что мы будем получать элементы в порядке их добавления в Set, то есть первыми обойдём элементы добавленные раньше всего.

JS. Map, Set

Обход

Обход можно организовать двумя способами:

Использовать метод **forEach()**, который работает аналогично одноимённому методу массива:

```
const filled = new Set([1, 2, 3, 3, 3, 'hello'])

filled.forEach(function(value) {
  console.log(value)
})

// => 1
// => 2
// => 3
// => 'hello'
```

JS. Map, Set

Обход

Воспользоваться for...of:

```
const filled = new Set([1, 2, 3, 3, 3, 'hello'])

for (let n of filled) {
  console.log(n)
}

// => 1
// => 2
// => 3
// => 'hello'
```

JS. Map, Set

Set использует строгое сравнение для проверки, есть ли элемент в коллекции или нет. Добавление примитивных значений разных типов будет работать как ожидается, приведения типов нет. При добавлении числа и строки с этим числом оба добавятся в коллекцию:

```
const set = new Set()

set.add(1)
set.add('1')

console.log(set.size) // => 2
```

JS. Map, Set

Не примитивные типы хранятся по ссылке, поэтому Set будет проверять что мы действительно пытаемся добавить тот же самый объект в коллекцию или нет. Это может казаться нелогичным, потому что объекты могут выглядеть одинаково, но не быть одним и тем же объектом (то есть у них разные адреса в памяти)

JS. Map, Set

```
const cheapShirt = { size: 'L', color: 'white' }
const fancyShirt = { size: 'L', color: 'white' }

console.log(cheapShirt === fancyShirt) // => false
console.log(cheapShirt === cheapShirt) // => true
console.log(fancyShirt === fancyShirt) // => true
```

JS. Map, Set

Мы создали два разных объекта (фигурные скобки создают новый объект), которые выглядят одинаково, но по факту это разные объекты. Они не равны друг другу — если в один добавить новое свойство, то второй не изменится.

Попробуем добавить эти объекты в Set:

```
const closet = new Set()
closet.add(cheapShirt)
closet.add(fancyShirt)

console.log(closet.size) // => 2
```

JS. Map, Set

Так как это разные объекты, то оба добавились в коллекцию. Если же попробовать добавить их второй раз, то эта операция будет проигнорирована

JS. Map, Set

С помощью Set можно легко получить массив уникальных элементов из массива неуникальных с помощью конструктора и спред-синтаксиса:

```
const nonUnique = [1, 2, 3, 4, 5, 4, 5, 1, 1]
const uniqueValuesArr = [...new Set(nonUnique)]

console.log(uniqueValuesArr) // => [1, 2, 3, 4, 5]
```

JS. Map, Set

Map — коллекция для хранения данных любого типа в виде пар [ключ, значение], то есть каждое значение сохраняется по уникальному ключу, который потом используется для доступа к этому значению. Причём в качестве ключей тоже принимаются значения любого типа.

JS. Map, Set

Основные методы для работы с коллекцией Map:

set(ключ, значение) — устанавливает значение;
get(ключ) — возвращает значение;
has(ключ) — проверяет наличие переданного ключа;
values() — возвращает итератор всех значений коллекции;
keys() — возвращает итератор всех ключей коллекции;
entries() — возвращает итератор пар [ключ, значение];
delete(ключ) — удаляет конкретное значение;
clear() — полностью очищает коллекцию;
forEach(колбэк) — перебирает ключи и значения коллекции.
Содержит свойство **size** для получения количества значений в коллекции.

JS. Map, Set

```
const someData = new Map()

someData.set('1', 'Значение под строковым ключом 1')
someData.set(1, 'Значение под числовым ключом 1')
someData.set(true, 'Значение под булевым ключом true')

console.log(someData.size)// 3
console.log(someData.get(1))// Значение под числовым ключом 1
console.log(someData.get('1'))// Значение под строковым ключом 1
console.log(someData.has(true))// true

someData.clear()

console.log(someData.size)// 0
```

JS. Map, Set

Создание коллекции

Коллекция создаётся при помощи конструктора. Можно создать пустой Map:

```
const map = new Map()


console.log(map.size) // 0
```

А можно сразу передать начальные значения. Для этого в конструктор нужно передать массив, состоящий из других массивов. Эти массивы должны состоять из двух элементов: первый элемент — ключ, а второй — значение

JS. Map, Set

```
const map = new Map([['js', 'JavaScript'], ['css', 'Cascading Style Sheets']])

console.log(map.size) // 2

console.log(map.get('js')) // JavaScript
```

JS. Map, Set

Работа с коллекцией

Map предоставляет небольшой набор удобных методов для работы с данными.

Чтобы сохранить значение в коллекции, нужно использовать метод set(). Первым аргументом передаём ключ, а вторым - значение:

```
const map = new Map()

map.set('js', 'JavaScript')
```

JS. Map, Set

Получить значение можно при помощи метода get() единственным аргументом которого передаём ключ, данные которого хотим получить. Если в коллекции нет значения для переданного ключа, get() вернёт undefined.

```
const map = new Map()
map.set('js', 'JavaScript')

console.log(map.get('js'))
// JavaScript
```

JS. Map, Set

Узнать, есть ли в коллекции значение с конкретным ключом, можно с помощью метода has():

```
const map = new Map()
map.set('js', 'JavaScript')

console.log(map.has('js'))
// true

console.log(map.has('css'))
// false
```

JS. Map, Set

Обход значений

Map предоставляет встроенный итератор для обхода значений:

```
const map = new Map()

map.set('html', 'HTML')
map.set('css', 'CSS')
map.set('js', 'JavaScript')

for (let [key, value] of map) {
  console.log(`${key} - ${value}`)
}
// html - HTML
// css - CSS
// js - JavaScript
```

JS. Map, Set

А ещё можно сделать то же самое при помощи метода forEach():

```
const map = new Map()

map.set('html', 'HTML')
map.set('css', 'CSS')
map.set('js', 'JavaScript')

map.forEach((value, key) => {
  console.log(`${key} - ${value}`)
})
// html - HTML
// css - CSS
// js - JavaScript
```

JS. Map, Set

Обратите внимание: когда вызывается метод forEach(), в колбэк передаются текущий ключ и соответствующее ему значение — индексов в Map нет.

При обходе значений Map всегда выводит их в том порядке, в котором они были добавлены.

Отличия от объектов Скопировать ссылку "Отличия от объектов"

Обычные объекты тоже подходят для хранения данных. Однако ключи в них могут быть только строками или символами

JS. Map, Set

Map же позволяет использовать в качестве ключа любое значение: объект, функцию, примитивные значения и даже null, undefined и NaN

```
const func = (name) => `Hello, ${name}`
const obj = { foo: 'bar' }

const map = new Map()
map.set(func, 'func value')
map.set(obj, 'object value')
map.set(undefined, 'undefined value')
map.set(NaN, 'NaN value')
map.set(null, 'null value')

console.log(map.get(func))// func value
console.log(map.get(obj))// object value
console.log(map.get(undefined))// undefined value
console.log(map.get(NaN))// NaN value
console.log(map.get(null))// null value
```

JS. Map, Set

При использовании непримитивных типов в качестве ключей стоит помнить, что они хранятся по ссылке, поэтому для доступа к заданному с помощью объекта ключу, необходимо передавать тот же самый объект.

Если мы возьмём два отдельных объекта с одинаковым содержимым, то мы получим два разных ключа

JS. Map, Set

Что такое регулярные выражения

Регулярные выражения (ещё их называют Regular Expressions, сокращённо regex или regexp, регулярки) — специальные шаблоны, которые используют для поиска и обработки текста. Для поиска в них можно задавать дополнительные команды, например игнорирование регистра.

В отличие от обычных функций поиска и замены, которые встречаются во всех текстовых редакторах, регулярные выражения могут сочетаться с другим кодом. С их помощью можно выполнять сложные операции с текстом, менять порядок строк, извлекать отдельные слова, валидировать и передавать информацию на сервер.

А ещё регулярки экономят очень много кода. Но есть один недостаток — они часто выглядят пугающе, запутанно и странно, особенно если шаблон сложный.

JS. Map, Set

Например, этот шаблон ищет email-адреса:

/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/

Почти во всех языках программирования есть регулярные выражения — реализация может отличаться, но основные моменты похожи. Их можно сравнить с отдельным мини-языком, который функционирует сам по себе.

Интересный момент: регулярные выражения действительно сложно писать, сложно читать и сложно поддерживать или изменять, но часто именно с их помощью разумнее выполнять работу над строками.

JS. Map, Set

Как создать регулярное выражение в JavaScript

Есть два способа создания регулярок — разберём каждый.

1. Конструктор RegExp()

Конструктор RegExp() позволяет создавать регулярные выражения на основе строки, передаваемой ему в качестве аргумента. Формат конструктора RegExp() выглядит следующим образом:

```
const regex = new RegExp("pattern", "flags");
```

Где «pattern» — это строка, содержащая регулярное выражение, а «flags» — дополнительные флаги, определяющие, как будет работать регулярное выражение. Например, флаг i указывает, что регулярное выражение должно игнорировать регистр символов.

Вот пример использования конструктора RegExp() для создания регулярного выражения, которое ищет все цифры в строке:

```
const regex = new RegExp("\\d", "g");
```

JS. Map, Set

2. Литералы

Литерал регулярного выражения, так же как и конструктор, состоит из двух частей. Первая часть — шаблон, который необходимо описать, он заключается в слеши (//). Вторая часть, после закрывающего слеша, — флаги, но их использование также необязательно.

Вот как он выглядит:

```
const regex = /pattern/flags;
```

Разберём на примере использования литерала для создания того же регулярного выражения, которое ищет все цифры в строке:

```
const regex = /\d/g;
```

JS. Map, Set

Написание шаблона регулярного выражения

Регулярное выражение объявляется в коде программы с помощью шаблона. С помощью шаблонов программа сравнивает строки на совпадение.

Шаблон заключается в символы /../и состоит из букв и цифр: /hello/. Или содержит специальные символы: *, + и другие.

JS. Map, Set

Написание шаблона регулярного выражения

Общие специальные символы:

^ – начало ввода.

$ – конец ввода строки.

\ – для указания, что специальный символ является частью строки, а не шаблона. Для поиска знака звездочка * – шаблон будет выглядеть: /*/, или для экранирования самого символа косой черты \.

. – один любой символ.

JS. Map, Set

Соответствие набору символов

Следующие символы указывают, к какому набору символов относится шаблон:

\w- соответствие буквам, цифрам и символам подчеркивания.

\W- обратный эффект: соответствие всему, что не относится к буквам, цифрам или символам подчеркивания.

\d – соответствие цифрам 0-9.

\D – соответствие всему, но не цифрам 0-9.

\s – соответствие символам пробела.

\S – соответствие не символам пробела.

[...] – соответствие любому из символов, указанному в скобках.

[^...]- соответствие всем символам, не указанных в скобках.

JS. Map, Set

Указание количества символов для соответствия

+ – повторение символа минимум от одного раза.

* – повторение символа от нуля.

? – появление символа в указанной позиции нуль или один раз.

| – соответствует логическому ИЛИ. /hello|bye/- или hello, или bye.

{n} – n вхождений символа в строке.

{m, n} – вхождения от m до n раз символа в строке.

JS. Map, Set

Использование флагов в регулярных выражениях

Флаги используются как дополнительные сведения о шаблоне. Можно использовать как по одному, так и вместе в произвольном порядке.

В JS доступно 5 флагов:

g – поиск всех совпадений в строке.

i – регистронезависимый поиск подстроки.

m - определяет текст как множество строк и ищет совпадения маркеров ^, $ для каждого начала и конца строки в тексте.

u – запись в регулярное выражение Unicode.

y - поиск с символа на позиции lastindex в строке.

Указать флаги можно или в конце литерала, или добавить в конструктор RegExp.

```
let reg1 = /hello/i;
let reg2 = new RegExp('hello','i');
```

JS. Map, Set

Для экранирования символа используется обратный слеш \.

JS. Map, Set

Методы для работы с регулярными выражениями

Методы обработки и поиска совпадений шаблона регулярного выражения со строкой текста:

Метод test()

Метод test() проверяет, соответствует ли регулярное выражение заданной строке. Метод возвращает true, если строка соответствует регулярному выражению, и false, если не соответствует.

Пример использования метода test():

```
let str = "Hello, world!";
const pattern = /Hello/;

if (pattern.test(str)) {
  console.log("String contains 'Hello'");
} else {
  console.log("String does not contain 'Hello'");
```

JS. Map, Set

Методы для работы с регулярными выражениями

Метод exec()

Метод exec() используется для поиска совпадений регулярного выражения в заданной строке. Метод возвращает массив, содержащий найденное совпадение и дополнительную информацию о нём.

Пример использования метода exec():

```
let str = "Hello, world!";
const pattern = /Hello/;

let result = pattern.exec(str);
```

JS. Map, Set

Методы для работы с регулярными выражениями

Метод match()

Метод match() используется для поиска всех совпадений регулярного выражения в заданной строке. Метод возвращает массив, содержащий все найденные совпадения.

Пример использования метода match():

```
let str = "The quick brown fox jumps over the lazy dog.";
const pattern = /the/gi;

let result = str.match(pattern);
```

JS. Map, Set

Методы для работы с регулярными выражениями

Метод replace()

Метод replace() принимает два аргумента: регулярное выражение и строку, на которую нужно заменить найденное совпадение. Этот метод ищет все совпадения с заданным регулярным выражением в исходной строке и заменяет их на указанную строку.

Пример использования метода replace():

```
let str = "Привет, мир!";
const pattern = /мир/;

let newstr = str.replace(pattern, "земля");
```

JS. Map, Set

Методы для работы с регулярными выражениями

Метод **search()** принимает один аргумент — регулярное выражение. Он ищет первое совпадение с заданным регулярным выражением в исходной строке и возвращает индекс первого символа совпадения. Если совпадение не найдено, метод возвращает -1.

Например:

```
let str = "Это текст для примера";
const pattern = /текст/;

let index = str.search(pattern);
```

JS. Map, Set

Методы для работы с регулярными выражениями

Метод split()

Метод split() принимает один аргумент — регулярное выражение. Он разбивает исходную строку на массив подстрок, используя заданное регулярное выражение как разделитель.

Пример использования метода split():

```
let str = "Это текст для примера";
const pattern = / /;

let words = str.split(pattern);
```

JS. Map, Set

Методы для работы с регулярными выражениями

Метод split()

Метод split() принимает один аргумент — регулярное выражение. Он разбивает исходную строку на массив подстрок, используя заданное регулярное выражение как разделитель.

Пример использования метода split():

```
let str = "Это текст для примера";
const pattern = / /;

let words = str.split(pattern);
```

JS. Map, Set

Что такое Паттерн?

Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

В отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Паттерны часто путают с алгоритмами, ведь оба понятия описывают типовые решения каких-то известных проблем. Но если алгоритм — это чёткий набор действий, то паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах.

Если привести аналогии, то алгоритм — это кулинарный рецепт с чёткими шагами, а паттерн — инженерный чертёж, на котором нарисовано решение, но не конкретные шаги его реализации.

JS. Map, Set

Порождающие

Отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.

Структурные

Отвечают за построение удобных в поддержке иерархий классов.

Поведенческие

Решают задачи эффективного и безопасного взаимодействия между объектами программы.

JS. Map, Set

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

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

Синглтон может нарушать принцип открытости и закрытости, увеличивая зацепление объектов. Из-за этого иногда его считают антипаттерном.

Пример

Допустим, мы пишем приложение для описания Солнечной системы. Солнце у нас может быть только одно, поэтому создать его тоже можно лишь один раз.

Если по каким-то причинам в приложении есть код, пытающийся создать Солнце заново, то стоит использовать уже существующий объект, а не создавать ещё один.

JS. Map, Set

```
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
  }
}
```

JS. Map, Set

Фабричный метод

Также известен как: Виртуальный конструктор, Factory Method

Фабрика (англ. factory) создаёт объект, избавляя нас от необходимости знать детали создания.

Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

JS. Map, Set

Представьте, что вы создаёте программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях. Поэтому весь ваш код работает с объектами класса Грузовик.

В какой-то момент ваша программа становится настолько известной, что морские перевозчики выстраиваются в очередь и просят добавить поддержку морской логистики в программу.

JS. Map, Set

Отличные новости, правда?! Но как насчёт кода? Большая часть существующего кода жёстко привязана к классам Грузовиков. Чтобы добавить в программу классы морских Судов, понадобится перелопатить всю программу. Более того, если вы потом решите добавить в программу ещё один вид транспорта, то всю эту работу придётся повторить.

В итоге вы получите ужасающий код, наполненный условными операторами, которые выполняют то или иное действие, в зависимости от класса транспорта.

Решение

Паттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор new, а через вызов особого фабричного метода. Не пугайтесь, объекты всё равно будут создаваться при помощи new, но делать это будет фабричный метод.

JS. Map, Set

```
class Transport {
   constructor (type, width, height) {
      this.type = type;
      this.width = width;
      this.height = height;
   }

   transport (to) {
      console.log(`You delivery goods to ${to} by ${this.type}`);
   }
}
```

JS. Map, Set

```
// супер класс
class Logistics {
   constructTransport (type) {
      if (type === 'sea') {
         return new Transport(type, 300, 200);
      } else {
         return new Transport(type, 100, 50);
      }
   }
}

const fabricLogistics = new Logistics();

const fura = fabricLogistics.constructTransport('ground');
fura.transport('Gomel');

const korablik = fabricLogistics.constructTransport('sea');
korablik.transport('Gomel');
```

JS. Map, Set

Данный шаблон полезен в случае когда нужно создавать объекты одного класса с разными входными данными. Например, создание автомобилей одной марки с разными характеристиками.

Допустим, имеем следующий класс автомобиля:

```
class BMW {
  constructor(model, propA, propB) {
    this.model = model
    this.propA = propA
    this.propB = propB
  }
}
```

JS. Map, Set

Простой фабричный класс для данного класса может выглядеть следующим образом:

```
class BMWFactory {
  create(model) {
    switch (model) {
      case 'X1':
        return new BMW(model, 1, 2);
      case 'X2':
        return new BMW(model, 3, 4);
      case 'X3':
        return new BMW(model, 4, 5);
    }
  }
}
```

JS. Map, Set

Теперь чтобы создать X2 с помощью Factory достаточно вызвать метод create:

```
const factory = new BMWFactory()

const x2 = factory.create('X2')
```

Данный пример является более простой демонстрацией принципа. способ create может быть расширен в согласовании с требованиями проекта либо отдельной задачки.

JS. Map, Set

Теперь чтобы создать X2 с помощью Factory достаточно вызвать метод create:

```
const factory = new BMWFactory()

const x2 = factory.create('X2')
```

Данный пример является более простой демонстрацией принципа. способ create может быть расширен в согласовании с требованиями проекта либо отдельной задачки.

JS. Map, Set

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.

JS. Map, Set

Шаблон проектирования Abstract Factory используется в случаях, когда нужно добавить дополнительный слой абстракции над паттерном Factory.

Допустим есть следующие реализации классов различных моделей автомобилей:

```
class BMW_1_Series{}
class BMW_M_Series{}
```

А также, пусть существуют фабрики для этих классов:

```
function bmwFamilyfactory() {
  return new BMW_1_Series()
}
function bmwSportFactory() {
  return new BMW_M_Series()
}
```

JS. Map, Set

Пример реализации функции абстрактной фабрики:

```
// Abstract Factory
function bmwProducer(type) {
  switch (type) {
    case 'sport': return bmwSportFactory;
    case 'family': return bmwFamilyfactory;
    default: return null;
  }
}
```

JS. Map, Set

Теперь, чтобы использовать абстрактную фабрику нужно сначала инициализировать фабрику нужного типа, а затем создать экземпляр возвращаемого ей класса:

```
// Abstract Factory
const produceSport = bmwProducer('sport');
const sportCar = produceSport();

const produceFamily = bmwProducer('family');
const familyCar = produceFamily();
```

JS. Map, Set

Прокси (заместитель)

Прокси (англ. proxy) — это промежуточный модуль, предоставляет интерфейс к какому-либо другому модулю.

Он похож на декоратор, но в отличие от него не меняет поведение оригинального объекта. Вместо этого он «вмешивается» в общение с оригинальным объектом.

Заместитель — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

JS. Map, Set

Прокси (заместитель)

Паттерн Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.

Вы могли бы поместить в класс заместителя какую-то промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте. А благодаря одинаковому интерфейсу, объект-заместитель можно передать в любой код, ожидающий сервисный объект.

JS. Map, Set

Прокси (заместитель)

```
class Person {
  constructor (name) {
    this.name = name;
  }

  walk (to) {
    console.log(`${this.name} walks to ${to}`);
  }

  sleep (hours) {
    console.log(`${this.name} sleeps ${hours} hours`);
  }
}
```

JS. Map, Set

```
class HumanProxy {
  constructor (human) {
    this.human = human;
  }
  walk (to) {
    if (to === 'bar') {
      console.log('he does not go anywhere!!!')
    } else {
      this.human.walk(to);
    }
  }
  sleep (hours) {}
}
const proxiedHuman = new HumanProxy(new Person('Oleg'));
proxiedHuman.walk('bar');
proxiedHuman.walk('home');
```