РОСЖЕЛДОР
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Ростовский государственный университет путей сообщения»
(ФГБОУ ВО РГУПС)

В.С. Якуничев, М.И. Муконина

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ СТУДЕНТОВ ПО ДИСЦИПЛИНЕ

МДК.03.02 «Фронтенд-разработка (клиентская часть)»

для специальности
09.02.09 Веб-разработка

Ростов-на-Дону
РГУПС
2025

СОДЕРЖАНИЕ

Лабораторная работа № 1

Введение в JavaScript. Среда выполнения и базовый синтаксис

Цель лабораторной работы: освоить среду разработки, базовый синтаксис JavaScript и операции с примитивными типами данных, научиться выводить и проверять результаты.

Методические указания

Введение в JavaScript

JavaScript — мультипарадигменный язык программирования, поддерживает объектно-ориентированный, императивный и функциональный стили.

JavaScript обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности web-страницам.

JavaScript является *интерпретируемым* языком, это означает, что код на языке JavaScript выполняется с помощью интерпретатора. Он получает инструкции языка JavaScript, которые определены на web-странице, выполняет их (или интерпретирует).

Основы синтаксиса

Код JavaScript состоит из инструкций, каждая из которых завершается точкой запятой:

```
alert("Вычисление выражения"); var a = 5 + 8; alert(a);
```

Однако современные браузеры вполне могут различать отдельные инструкции, если они просто располагаются на отдельных строках без точки запятой:

```
alert("Вычисление выражения")
var a = 5 + 8
alert(a)
```

Но чтобы улучшить читабельность кода и снизить число возможных ошибок, рекомендуется определять каждую инструкцию JavaScript на отдельной строчке и завершать ее точкой с запятой.

```
alert("Вычисление выражения");
var a = 5 + 8;
alert(a);
```

В коде JavaScript могут использоваться комментарии, они не обрабатываются интерпретатором JavaScript и никак не учитываются в работе программы. Комментарии предназначен для ориентации по коду, чтобы указать, что делает тот или иной код.

Комментарии могут быть однострочными, для которых используется двойной слэш //:

```
// вывод сообщения
alert("Вычисление выражения");
// арифметическая операция
var a = 5 + 8;
alert(a);
```

Кроме однострочных комментариев могут использоваться и многострочные комментарии. Такие комментарии заключаются между символами /* текст комментария */. Например:

```
// вывод сообщения
/* вывод сообщения и
арифметическая операция */
alert("Вычисление выражения");
var a = 5 + 8;
alert(a);
```

Переменные

Для хранения данных в программе используются переменные, они предназначены для хранения каких-нибудь временных данных или таких данных, которые в процессе работы могут менять свое значение.

Для создания переменных применяются ключевые слова **var** и **let**. Например, объявим переменную **myIncome:**

```
var myIncome;
// другой вариант
let myIncome2;
```

Каждая переменная имеет имя, оно представляет собой произвольный набор алфавитно-цифровых символов, знака подчеркивания _ или знака доллара $, причем названия не должны начинаться с цифровых символов. То есть можно использовать в названии буквы, цифры, подчеркивание. Однако все остальные

символы запрещены.

Также нельзя давать переменным такие имена, которые совпадают с зарезервированными ключевыми словами. В JavaScript не так много ключевых слов, поэтому данное правило несложно соблюдать. Например, следующее название будет некорректным, так как **for** - ключевое слово в JavaScript:

```
var for;
```

При названии переменных надо учитывать, что JavaScript является **регистрозависимым** языком. Процесс присвоения переменной начального значения называется **инициализацией.** Отличительной чертой переменных является то, что можно изменить их значение:

```
var income = 300;
income = 400;
console.log(income);

let price = 76;
price = 54;
console.log(price);
```

Константы

С помощью ключевого слова **const** можно определить константу, которая, как и переменная, хранит значение, однако это значение не может быть изменено.

Если попробовать изменить ее значение, то возникнет ошибка:

```
const rate = 10;
rate = 23;  // ошибка, rate - константа, поэтому нельзя изменить ее значение
```

Также стоит отметить, что поскольку нельзя изменить значение константы, то константа должна быть инициализирована, то есть при ее определении необходимо предоставить ей начальное значение. Если этого не сделать, возникнет ошибка:

```
const rate; // ошибка, rate не инициализирована
```

Типы данных

Все используемые данные в JavaScript имеют определенный тип. В JavaScript имеется пять примитивных типов данных:

String: представляет строку

Number: представляет числовое значение

Boolean: представляет логическое значение true или false

undefined: указывает, что значение не установлено

null: указывает на неопределенное значение

Все данные, которые не попадают под вышеперечисленные пять типов, относятся к типу **object**.

1. Числовые данные

Числа в JavaScript могут иметь две формы:

- Целые числа, например, 35. Можно использовать как положительные, так и отрицательные числа. Диапазон используемых чисел: от -2^53 до 2^53.
- Дробные числа (числа с плавающей точкой), например, 3.5575. Опять же можно использовать как положительные, так и отрицательные числа. Для чисел с плавающей точкой используется тот же диапазон: от -2^53 до 2^53.

```
var x = 45;
var y = 23.897;
```

В качестве разделителя между целой и дробной частями, как и в других языках программирования, используется точка.

2. Строки

Тип **string** представляет строки, то есть такие данные, которые заключены в кавычки. Причем можно использовать как двойные, так и одинарные кавычки.

```
var helloWorld = "Привет мир";
var helloWorld2 = 'Привет мир';
```

Единственно ограничение: тип закрывающей кавычки должен быть тот же, что и тип открывающей, то есть либо обе двойные, либо обе одинарные кавычки.

Если внутри строки встречаются кавычки, то их нужно экранировать слэшем \.

```
var companyName = "Бюро \"Рога и копыта\"";
```

Также можно внутри стоки использовать другой тип кавычек:

```
var companyName1 = "Бюро 'Рога и копыта'";
var companyName2 = 'Бюро "Рога и копыта"';
```

3. Тип Boolean

Тип **Boolean** представляет булевы или логические значения true и false (то

есть да или нет):

```
var isAlive = true;
var isDead = false;
```

4. null и undefined

Нередко возникает путаница между **null** и **undefined**. Итак, когда только определяется переменная без присвоения ей начального значения, она представляет тип **undefined**: Присвоение значение **null** означает, что переменная имеет некоторое неопределенное значение (не число, не строка, не логическое значение), но все-таки имеет значение. **undefined** означает, что переменная не имеет значения.

```
var isAlive;
console.log(isAlive); // undefined
isAlive = null;
console.log(isAlive); // null
isAlive = undefined;  // снова установим тип undefined
console.log(isAlive); // undefined
```

5. object

Тип **object** представляет сложный объект. Простейшее определение объекта представляют фигурные скобки:

```
var user = {};
```

Объект может иметь различные свойства и методы:

```
var user = {name: "Tom", age:24};
console.log(user.name); //Tom
```

В данном случае объект называется **user**, и он имеет два свойства: **name** и **age**. Это краткое описание объектов.

Слабая типизация

JavaScript является языком со слабой типизацией. Это значит, что переменные могут динамически менять тип.

```
var xNumber; // тип `undefined`
console.log(xNumber); //undefined

xNumber = 45; // тип `number`
console.log(xNumber); //45
```

```
xNumber = "45"; // тип `string`
console.log(xNumber); //45
```

Несмотря на то, что во втором и третьем случае консоль выведет число 45, но во втором случае переменная **xNumber** будет представлять число, а в третьем случае - строку. Это важный момент, который надо учитывать. От этого зависит поведение переменной в программе:

```
var xNumber = 45; // тип `number`
var yNumber = xNumber + 5;
console.log(yNumber); //50

xNumber = "45"; // тип `string`
var zNumber = xNumber + 5
console.log(zNumber); //455
```

Выше в обоих случая к переменной **xNumber** применяется операция сложения +. Но в первом случае **xNumber** представляет число, поэтому результатом операции **xNumber + 5** будет число 50. Во втором случае **xNumber** представляет строку. Но операция сложения между строкой и числом 5 невозможна. Поэтому число 5 будет преобразовываться к строке, и будет происходить операция объединения строк. И результатом выражения **xNumber + 5** будет стока "455".

Операторы

1. Оператор typeof

С помощью оператора **typeof** можно получить тип переменной:

```
var name = "Tom";
console.log(typeof name); // string
var income = 45.8;
console.log(typeof income); // number
var isEnabled = true;
console.log(typeof isEnabled); // boolean
var undefVariable;
console.log(typeof undefVariable); // undefined
```

2. Математические операторы

JavaScript поддерживает все базовые математические операции:

Сложение:

```
var x = 10;
var y = x + 50;
```

Вычитание:

```
var x = 100;
var y = x - 50;
```

Умножение:

```
var x = 4;
var y = 5;
var z = x * y;
```

Деление:

```
var x = 40;
var y = 5;
var z = x / y;
```

Деление по модулю (оператор %) возвращает остаток от деления:

```
var x = 40;
var y = 7;
var z = x % y;
console.log(z); //5
```

3. Операции присваивания

Приравнивает переменной определенное значение: `var x = 5;`

Сложение с последующим присвоением результата.

```
var a = 23;
a += 5; // аналогично a = a + 5
console.log(a); //28
```

Вычитание с последующим присвоением результата.

```
var a = 28;
a -= 10; // аналогично a = a - 10
console.log(a); //18
```

Умножение с последующим присвоением результата:

```
var x = 20;
x *= 2; // аналогично x = x * 2
console.log(x); //40
```

Деление с последующим присвоением результата:

```
var x = 40;
x /= 4; // аналогично x = x / 4
console.log(x); //10
```

Получение остатка от деления с последующим присвоением результата:

```
var x = 10;
x %= 3; // аналогично x = x % 3
console.log(x); //1
```

4. Операторы сравнения

Как правило, для проверки условия используются **операторы сравнения**. Операторы сравнения сравнивают два значения и возвращают значение true или false:

Оператор равенства сравнивает два значения, и если они равны, возвращает true, иначе возвращает false: **`x == 5`**

Оператор тождественности также сравнивает два значения и их тип, и если они равны, возвращает true, иначе возвращает false: **`x === 5`**

Сравнивает два значения, и если они не равны, возвращает true, иначе возвращает false: **`x != 5`**

Сравнивает два значения и их типы, и если они не равны, возвращает true, иначе возвращает false: **`x !== 5`**

Сравнивает два значения, и если первое больше второго, то возвращает true, иначе возвращает false: **`x > 5`**

Сравнивает два значения, и если первое меньше второго, то возвращает true, иначе возвращает false: **`x < 5`**

Сравнивает два значения, и если первое больше или равно второму, то возвращает true, иначе возвращает false: **`x >= 5`**

Сравнивает два значения, и если первое меньше или равно второму, то возвращает true, иначе возвращает false: **`x <= 5`**

5. Логические операции

Логические операции применяются для объединения результатов двух операций сравнения. В JavaScript есть следующие логические операции:

- **&&**

Возвращает true, если обе операции сравнения возвращают true, иначе возвращает false:

```
var income = 100;
var percent = 10;
var result = income > 50 && percent < 12;
console.log(result); //true
```

- ||

Возвращает true, если хотя бы одна операция сравнения возвращают true, иначе возвращает false:

```
var income = 100;
var isDeposit = true;
var result = income > 50 || isDeposit == true;
console.log(result); //true
```

- !

Возвращает true, если операция сравнения возвращает false:

```
var income = 100;
var result1 = !(income > 50);
console.log(result1); // false

var isDeposit = false;
var result2 = !isDeposit;
console.log(result2); // true
```

6. Операции со строками

Строки могут использовать оператор + для объединения.

```
var name = "Том";
var surname = "Сойер"
var fullname = name + " " + surname;
console.log(fullname); //Том Сойер
```

Если одно из выражений представляет строку, а другое - число, то число преобразуется к строке и выполняется операция объединения строк:

```
var name = "Том";
var fullname = name + 256;
console.log(fullname); //Том256
```

Пример программы, которая демонстрирует работу с операциями над

переменными:

```
<!DOCTYPE html>
<html>
    <head>
    <meta charset="utf-8" />
    <title>JavaScript</title>
</head>
<body>
    <script>
        var sum = 500; // сумма вклада
        var percent = 10;  // процент по вкладу
        var income = sum * percent / 100;  // доход по вкладу
        sum = sum + income; // определяем новую сумму
        console.log("Доход по вкладу: " + income);
        console.log("Сумма вклада после первого года: " + sum);
    </script>
</body>
</html>
```

В скрипте объявляются три переменных: **sum**, **percent** и **income**. Переменная **income** вычисляется по остальным двум переменным с помощью операций умножения и деления. И в конце ее значение суммируется со значением переменной **sum**.

И консоль браузера выведет:

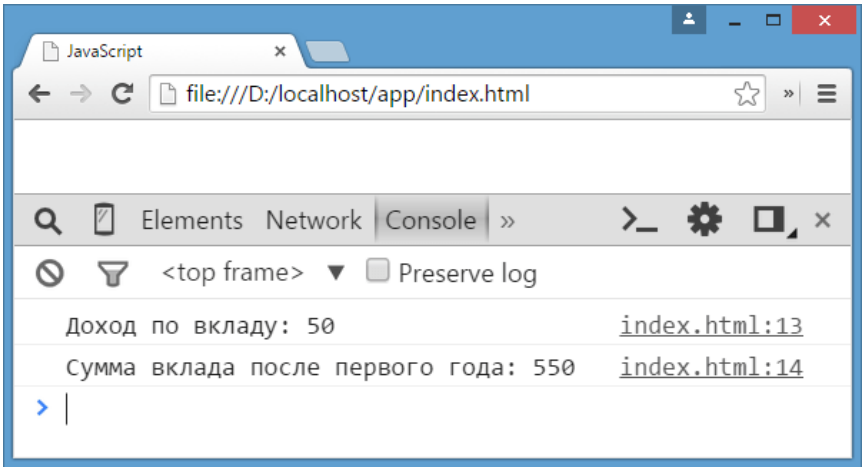

Рисунок 1.1 – Вывод результата в консоли

Преобразование данных

Нередко возникает необходимость преобразовать одни данные в другие. Например:

```
var number1 = "46";
var number2 = "4";
var result = number1 + number2;
console.log(result); //464
```

Обе переменных представляют строки, а точнее строковые представления чисел. И в итоге получим не число 50, а строку 464. Но было бы неплохо, если бы их тоже можно было бы складывать, вычитать, в общем работать как с обычными числами.

В этом случае можно использовать **операции преобразования**. Для преобразования строки в число применяется функция **parseInt():**

```
var number1 = "46";
var number2 = "4";
var result = parseInt(number1) + parseInt(number2);
console.log(result); //50
```

Для преобразования строк в дробные числа применяется функция **parseFloat():**

```
var number1 = "46.07";
var number2 = "4.98";
var result = parseFloat(number1) + parseFloat(number2);
console.log(result); //51.05
```

Задания для выполнения лабораторной работы

1. Подготовка среды разработки

– Откройте браузер. Нажмите F12 или Ctrl+Shift+I для открытия *Инструментов разработчика*. Перейдите на вкладку Console — это среда выполнения JavaScript-кода.

– Для подключения внешних файлов с расширением .js к html-документу необходимо добавить тег **<script src='путь к файлу'> </script>** перед закрывающимся тегом **</body>**. Также можно записывать скрипт в самом html-документу внутри парного тега **<script> </script>.**

2. Базовые команды

- Вывод данных:

используйте **`console.log(), alert()`** или **`document.write().`**

- Объявление переменных: **`let, const, var`** (устаревший).
- Типы данных: число (**`number`**), строка (**`string`**), булевый (**`boolean`**), **`undefined, null`**.
- Операторы: арифметические (+, -, *, /, %), сравнения (==, ===, !=, !==, >, <), логические (&&, ||, !).

Вариант 1

1. Объявите две переменные: **`name`** (строка с вашим именем) и **`birthYear`** (число — год рождения).
2. Вычислите ваш текущий возраст и сохраните в переменную **`age`**.
3. Используя шаблонные строки (обратные кавычки), выведите в консоль фразу: "Меня зовут [**`name`**], мне [**`age`**] лет."
4. Проверьте тип данных переменной **`birthYear`** с помощью **`typeof`**.

Вариант 2

1. Создайте переменные для сторон прямоугольника: **`a = 5, b = 10`**.
2. Вычислите периметр и площадь, сохраните в переменные **`perimeter`** и **`area`**.
3. Выведите результаты в консоль в формате: "Периметр: [**`perimeter`**], Площадь: [**`area`**]."
4. Преобразуйте переменную a в строку и проверьте её тип до и после преобразования.

Вариант 3

1. Объявите три переменные с числами: **`x = 15, y = 4, z = 7.`**
2. Найдите среднее арифметическое этих чисел и выведите результат.
3. Определите, является ли x чётным числом (используйте оператор остатка %), и выведите true или false.
4. Выведите результат сравнения: z больше, чем y, но меньше, чем x.

Вариант 4

1. Создайте переменные: **`price`** = 1000 (цена), **`discount`** = 15 (скидка в процентах).
2. Рассчитайте сумму скидки и итоговую цену, сохраните в переменные.
3. Выведите: "Цена со скидкой: [итог] руб. (скидка [сумма_скидки] руб.)".
4. Используйте **`alert()`** для показа итоговой цены.

Вариант 5

1. Объявите строку: **`sentence`** = "JavaScript — это интересно!".
2. Выведите её длину (свойство **`.length`**).
3. Преобразуйте строку к верхнему регистру, затем к нижнему. Выведите оба результата.
4. Проверьте, содержит ли строка слово "интересно" (метод **`.includes()`**).

Вариант 6

1. Задайте переменную **`temperature`** = -5.
2. Напишите код, который определяет, положительная температура или отрицательная, и выводит "Температура выше нуля" или "Температура ниже нуля".
3. Добавьте проверку на ноль: если **`temperature`** == 0, выводите "Температура равна нулю".
4. Выполните все проверки, используя **`if, else if, else.`**

Вариант 7

1. Создайте переменную **`number`** = 42.
2. Преобразуйте её в строку, затем обратно в число (используйте **`String()`** и **`Number()`**).
3. Проверьте типы после каждого преобразования через **`typeof`**.
4. Выведите в консоль "Результат: [число] — это [тип]" для каждого шага.

Вариант 8

1. Объявите две логические переменные: **`hasTicket`** =

true, **hasPassport** = false.

2. Используя логические операторы (&&, ||, !), определите:
   - Может ли человек пройти (**hasTicket** && **hasPassport**).
   - Доступен ли хоть один документ (**hasTicket** || **hasPassport**).
3. Выведите оба результата.

Вариант 9

1. Задайте строку: **str** = "25.5px".
2. Преобразуйте её в число (используйте **parseFloat() и parseInt()**).
3. Сравните результаты преобразования и исходную строку.
4. Выведите в консоль все три значения и их типы.

Вариант 10

1. Объявите константу **PI** = 3.14159 и переменную **radius** = 7.
2. Вычислите длину окружности и площадь круга.
3. Округлите результаты до двух знаков после запятой (.**toFixed**(2)).
4. Выведите: "Окружность: [длина], Площадь: [площадь]".

Вариант 11

1. Создайте переменные: **num1** = "100", **num2** = 50.
2. Выполните сложение (+) и вычитание (-) с этими переменными. Объясните разницу в результатах.
3. Преобразуйте **num1** в число явно и повторите операции.
4. Запишите вывод в консоль с пояснениями.

Вариант 12

1. Объявите переменную **userName** без значения.
2. Проверьте её тип через **typeof**.
3. Присвойте ей ваше имя и снова проверьте тип.
4. Выведите разницу между **null** и **undefined**, создав обе переменные и сравнив их через == и ===.

Вариант 13

1. Задайте число **n** = 123.

2. Разбейте его на цифры (используя математические операции: деление и остаток от деления) и сохраните в отдельные переменные.

3. Найдите сумму цифр числа.

4. Выведите: "Число: [**n**], Сумма цифр: [сумма]".

Вариант 14

1. Создайте две строки: **str1** = "Hello", **str2** = "World".

2. Соедините их через пробел тремя способами: через +, через **.concat**(), через шаблонную строку.

3. Выведите результаты и сравните.

4. Проверьте, равны ли строки **str1** и **str2** (строгое и нестрогое сравнение).

Вариант 15

1. Объявите переменные**: a = 10, b = "10", c = 15.**

2. Сравните **a** и **b**, **c** помощью == и ===. Объясните разницу.

3. Проверьте, является ли **a** больше **b** И **a** меньше **c** за одно выражение.

4. Выведите все результаты сравнений в табличном виде.

Лабораторная работа № 2

Разработка алгоритмов с использованием условных операторов

Цель лабораторной работы: научиться применять операторы ветвления (if, else, else if) для решения типовых задач.

Методические указания

Условные конструкции позволяют выполнить те или иные действия в зависимости от определенных условий.

1. Выражение if

Конструкция **if** проверяет некоторое условие и если это условие верно, то выполняет некоторые действия. Общая форма конструкции **if**:

```
if (условие) действия;
```

Например:

```
var income = 100;
if (income > 50) alert("доход больше 50");
```

Здесь в конструкции **if** используется следующее условие: **income > 50**. Если это условие возвращает **true**, то есть переменная **income** имеет значение больше 50, то браузер отображает сообщение. Если же значение **income** меньше 50, то никакого сообщения не отображается.

Если необходимо выполнить по условию набор инструкций, то они помещаются в блок из фигурных скобок:

```
var income = 100;
if (income > 50) {
    var message = "доход больше 50";
    alert(message);
}
```

Причем условия могут быть сложными:

```
var income = 100;
var age = 19;
if (income < 150 && age > 18) {
    var message = "доход больше 50";
    alert(message);
}
```

Конструкция **if** позволяет проверить наличие значения.

```
var myVar = 89;
if (myVar) {
    // действия
}
```

Если переменная **myVar** имеет значение, то в условной конструкции она возвратит значение **true**.

Но нередко для проверки значения переменной используют альтернативный вариант - проверяют на значение **undefined**:

```
if (typeof myVar != "undefined") {
    // действия
}
```

В конструкции **if** также можно использовать блок **else**. Данный блок содержит инструкции, которые выполняются, если условие после **if** ложно, то есть равно **false**:

```
var age = 17;
if (age >= 18) {
    alert("Вы допущены к программе кредитования");
} else {
    alert("Вы не можете участвовать в программе, так как возраст меньше 18");
}
```

С помощью конструкции **else if** можно добавить альтернативное условие к блоку **if**:

```
var income = 300;
if (income < 200) {
    alert("Доход ниже среднего");
} else if (income >= 200 && income <= 400) {
    alert("Средний доход");
} else {
    alert("Доход выше среднего");
}
```

В данном случае выполнится блок **else if.** При необходимости можно использовать несколько блоков **else if** с разными условиями:

```
if (income < 200) {
```

```
    alert("Доход ниже среднего");
} else if (income >= 200 && income < 300) {
    alert("Чуть ниже среднего");
} else if (income >= 300 && income < 400) {
    alert("Средний доход");
} else {
    alert("Доход выше среднего");
}
```

2. true или false

В JavaScript любая переменная может применяться в условных выражениях, но не любая переменная представляет тип **boolean**. Поэтому возникает вопрос, что возвратит та или иная переменная - **true** или **false**? Много зависит от типа данных, который представляет переменная:

Undefined //Возвращает **false**

Null //Возвращает **false**

Boolean //Если переменная равна **false**, то возвращается **false**. Соответственно, если переменная равна **true**, то возвращается **true**

Number //Возвращает **false**, если число равно 0 или **NaN (Not a Number)**, в остальных случаях возвращается **true**

```
var x = NaN;
if (x) {  // false
}
```

String //Возвращает **false**, если переменная равна пустой строке, то есть ее длина равна 0, в остальных случаях возвращается **true**

```
var y = ""; // false - так как пустая строка
var z = "javascript"; // true - строка не пустая
```

Object

Всегда возвращает true

```
var user = {name:"Tom"}; // true
var isEnabled = new Boolean(false); // true
var car = {}; // true
```

3. Конструкция switch..case

Конструкция **switch..case** является альтернативой использованию

конструкции **if..else if..else** и также позволяет обработать сразу несколько условий:

```
var income = 300;
switch(income) {
    case 100 :
        console.log("Доход равен 100");
        break;
    case 200 :
        console.log("Доход равен 200");
        break;
    case 300 :
        console.log("Доход равен 300");
        break;
}
```

После ключевого слова **switch** в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора **case**. И если совпадение будет найдено, то будет выполняться определенный блок **case.**

В конце каждого блока **case** ставится оператор **break**, чтобы избежать выполнения других блоков.

Если есть необходимость обработать ситуацию, когда совпадения не будет найдено, то можно добавить блок **default**:

```
var income = 300;
switch(income) {
    case 100 :
        console.log("Доход равен 100");
        break;
    case 200 :
        console.log("Доход равен 200");
        break;
    case 300 :
        console.log("Доход равен 300");
        break;
    default:
        console.log("Доход неизвестной величины");
```

```
        break;
}
```

4. Тернарная операция

Тернарная операция состоит из трех операндов и имеет следующее определение:

[первый операнд - условие] ? [второй операнд] : [третий операнд]

В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно **true**, то возвращается второй операнд; если условие равно **false**, то третий. Например:

```
var a = 1;
var b = 2;
var result = a < b ? a + b : a - b;
console.log(result); //3
```

Если значение переменной **a** меньше значения переменной **b**, то переменная **result** будет равняться **a + b**. Иначе значение **result** будет равняться **a - b**.

Задания для выполнения лабораторной работы

Вариант 1: Калькулятор оценок

Создайте программу, которая запрашивает у пользователя балл (от 0 до 100) и выводит оценку по системе:

- 90–100: "Отлично (A)"
- 75–89: "Хорошо (B)"
- 60–74: "Удовлетворительно (C)"
- 0–59: "Неудовлетворительно (F)"

При вводе числа вне диапазона выводить: "Некорректный ввод".

Вариант 2: Проверка года на високосность

Напишите программу, которая определяет, является ли введённый год високосным. Учитывайте правило: год високосный, если он кратен 4, но не кратен 100, либо кратен 400. Вывести "Високосный год" или "Не високосный год".

Вариант 3: Калькулятор ИМТ (Индекса массы тела)

Запросите у пользователя рост (в метрах) и вес (в кг). Рассчитайте ИМТ по

формуле вес / (рост * рост) и определите категорию:

- Менее 18.5: "Недостаточный вес"
- 18.5–24.9: "Нормальный вес"
- 25–29.9: "Избыточный вес"
- 30 и более: "Ожирение"

Вариант 4: Определение времени суток

Запросите у пользователя текущий час (0–23). Определите и выведите время суток:

- 5–11: "Утро"
- 12–16: "День"
- 17–22: "Вечер"
- 23–4: "Ночь"

Если число вне диапазона, вывести "Некорректное время".

Вариант 5: Простейший калькулятор операций

Запросите два числа и знак операции (+, -, *, /). Выполните соответствующую операцию и выведите результат. Для деления добавьте проверку деления на ноль. При вводе неизвестной операции выводить "Неизвестная операция".

Вариант 6: Проверка треугольника

Запросите длины трёх сторон. Определите, можно ли из них построить треугольник (сумма любых двух сторон больше третьей). Если можно, определите его тип:

- Равносторонний (все стороны равны)
- Равнобедренный (две стороны равны)
- Разносторонний (все стороны разные)

Вариант 7: Система скидок в магазине

Запросите сумму покупки. Примените скидку в зависимости от суммы:

- До 500: без скидки
- 500-1000: скидка 5%
- 1001-2000: скидка 10%
- Свыше 2000: скидка 15%

Вывести итоговую сумму к оплате.

Вариант 8: Конвертер единиц измерения

Запросите число и выберите тип конвертации:

1. Километры в мили (1 км = 0.621371 миль)
2. Килограммы в фунты (1 кг = 2.20462 фунта)
3. Литы в галлоны (1 литр = 0.264172 галлона)

Вывести результат с округлением до 2 знаков.

Вариант 9: Проверка логина и пароля

Задайте в коде правильный логин (admin) и пароль (qwerty123). Запросите у пользователя логин и пароль. Проверьте:

- Если оба верны: "Доступ разрешён"
- Если логин верен, пароль нет: "Неверный пароль"
- Если логин неверен: "Пользователь не найден"

Вариант 10: Определение типа символа

Запросите у пользователя один символ. Определите его тип:

- Цифра (0–9)
- Строчная буква латинского алфавита (a-z)
- Заглавная буква латинского алфавита (A-Z)
- Специальный символ (все остальные)

Используйте методы строк или сравнение по кодам ASCII.

Вариант 11: Калькулятор налога на доход

Запросите годовой доход. Рассчитайте налог по прогрессивной шкале:

- До 15000: 10%
- 15001–50000: 15%
- 50001–100000: 20%
- Свыше 100000: 25%

Вывести сумму налога и чистый доход.

Вариант 12: Проверка на чётность и кратность

Запросите целое число. Проверьте:

- Является ли число чётным
- Кратно ли число 3, 5 или 7 одновременно
- Положительное ли число

Вывести все проверки в формате булевых значений.

Вариант 13: Система уровней игрока

Запросите количество опыта игрока. Определите его уровень:

- 0–999: Новичок
- 1000–2999: Ученик
- 3000–6999: Адепт
- 7000–14999: Мастер
- 15000+: Грандмастер

Добавьте проверку на отрицательный опыт.

Вариант 14: Калькулятор стоимости доставки

Запросите вес посылки (кг) и расстояние (км). Рассчитайте стоимость:

- Базовая ставка: 100 единиц
- За каждый кг свыше 5: +20 единиц за кг
- За каждые 100 км: +50 единиц

Вывести итоговую стоимость.

Вариант 15: Определение типа кредита

Запросите у пользователя: возраст, ежемесячный доход, сумму кредита, срок (лет). Одобрить кредит, если:

- Возраст 21–65 лет
- Ежемесячный платеж не превышает 40% дохода
- Сумма кредита от 10000 до 5000000

Вывести решение (одобрено/отклонено) и причину в случае отказа.

Лабораторная работа № 3

Разработка алгоритмов с использованием циклов

Цель лабораторной работы: освоить принципы циклических алгоритмов (for, while) для обработки числовых последовательностей.

Методические указания

Циклы — это конструкции, которые помогают повторять заданный участок кода столько раз, сколько нужно. Они экономят время, делают программу короче и позволяют автоматизировать рутинные действия.

Всё сводится к простой идее: пока условие цикла истинно, код внутри него выполняется. Когда условие перестаёт быть истинным, цикл завершается и выполнение скрипта продолжается.

Пример:

```
for (let i = 0; i < 5; i++) { // Условие цикла
  console.log(i); // Вывод в консоль
}
```

Что делает этот код:

- создаёт переменную i со значением 0;
- проверяет: если i меньше 5, выводит переменную в консоль;
- увеличивает i на 1 и повторяет;
- как только i станет 5, цикл остановится.

Результат:

```
0
1
2
3
4
```

В JavaScript есть несколько видов циклов. Они подходят для разных задач, но цель у них общая — повторять действия до тех пор, пока выполняется заданное условие. Пять основных видов циклов:

Цикл while

Циклы **`while`** и **`for`** очень похожи друг на друга — оба повторяют участок

Название	Когда используется
for	Когда вы знаете, сколько повторов нужно
while	Когда нужно повторять до тех пор, пока условие верно
do...while	Когда нужно хотя бы один раз выполнить код
for...of	Когда нужно пройтись по значениям массива или по символам строки
for...in	Когда нужно пройтись по свойствам объекта

кода несколько раз. Цикл **while** считается более базовым: он состоит из исполняемого кода и условия. А цикл **for** — это удобная надстройка над **while**, которая позволяет перебрать что-то определённое количество раз.

Цикл **while** удобен, когда заранее неизвестно точное количество повторений, но известно условие, при котором цикл должен продолжать работу.

```
while (условие) {
    // Код, который будет выполняться
}
```

Алгоритм работы

1. Проверяется условие. Если оно истинно, цикл запускается.
2. Выполняется код внутри цикла.
3. После выполнения снова проверяется условие:

- Если оно истинно, код запускается снова.
- Если оно ложно, цикл останавливается, а управление передаётся следующей части программы.

Важно следить за условием: если оно никогда не становится ложным, цикл станет бесконечным.

Предположим, вы делаете простой to-do-список задач. У вас есть массив с задачами, и нужно вывести все задачи на страницу.

```
const tasks = ['Купить продукты', 'Позвонить маме', 'Подготовить
```

```
проект']; // Список задач в массиве
    let index = 0; // Начальный индекс элемента массива
    while (index < tasks.length) { // Условие: пока не дойдём до конца массива
        console.log(`Задача №${index + 1}: ${tasks[index]}`);
        index++;
    }
    // Вывод в консоли:
    // Задача №1: Купить продукты
    // Задача №2: Позвонить маме
    // Задача №3: Подготовить проект
```

В этом примере вы перебираете список задач из массива и выводите каждую задачу с её номером. Подобный подход используется при выводе элементов списка на веб-странице или в меню приложений.

Цикл do…while

Цикл **do…while** всегда выполняется минимум один раз, даже если условие с самого начала ложное. Обычный **while** сначала проверяет условие, а потом решает, выполнять код или нет. А **do…while** сначала выполняет код один раз, а уже потом проверяет условие и решает, повторять ли ещё.

```
do {
    // Код, который выполнится хотя бы один раз
} while (условие);
```

Используйте **do…while**, например, при получении пользовательского ввода или при начальной инициализации переменных перед проверкой.

Представим, что вы делаете форму регистрации для вашего списка задач. Нужно убедиться, что пользователь ввёл email, который выглядит как email, — и если нет, то запрашивать ввод снова.

```
    let email; // Объявляем переменную, в которую пользователь будет вводить email
    const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    // Создаём регулярное выражение — это шаблон, по которому можно проверить, похожа ли строка на email
    // ^ — начало строки
    // [^\s@]+ — одна или несколько букв или цифр, кроме пробелов и
```

```
символа @
    // @ — обязательно должен быть символ @
    // [^\s@]+ — после @ снова одна или несколько букв или цифр, кроме пробелов и @
    // \. — обязательно точка
    // [^\s@]+ — после точки снова одна или несколько букв или цифр
    // $ — конец строки
    do {
        email = prompt("Введите свой email:");
        // Показываем пользователю окно ввода. Что бы он ни ввёл, сохраняем в переменную email
    } while (!emailPattern.test(email));
    // Повторяем цикл, пока введённый email НЕ проходит проверку по шаблону
    // emailPattern.test(email) возвращает true, если строка подходит под шаблон
    // ! — означает «не»: если проверка не прошла, продолжаем спрашивать
    console.log(`Введён корректный email: ${email}`);
    // Когда пользователь ввёл правильный email, выводим его в консоль
```

Цикл for

Цикл **for** отлично подходит, когда известно, сколько раз нужно выполнить действие. Он объединяет в себе сразу три части: начальное значение, условие окончания и шаг итерации, поэтому с помощью **for** удобно работать с числами и массивами.

```
for (начало; условие; шаг) {
    // Код, который будет повторяться
}
```

Алгоритм работы

- Задаётся начальное значение переменной.
- Проверяется условие — если оно истинно, выполняется тело цикла.
- После каждой итерации переменная изменяется согласно шагу.
- Когда условие становится ложным, цикл прекращается.

Порядок действий всегда одинаковый: сначала счётчик, потом условие, потом шаг.

Сначала разберём, как бы выглядел код для вывода на экран элементов списка задач с помощью цикла **for**.

```
const tasks = ['Купить продукты', 'Позвонить маме', 'Подготовить проект'];
for (let i = 0; i < tasks.length; i++) {
    console.log(`Задача №${i + 1}: ${tasks[i]}`);
}
// Вывод:
// Задача №1: Купить продукты
// Задача №2: Позвонить маме
// Задача №3: Подготовить проект
```

Теперь представим, что вы хотите приложить фото заметок к некоторым задачам и у вас есть список ссылок на изображения. Разберём, как цикл **for** может подготовить данные для вывода.

```
const imageUrls = ['image1.jpg', 'image2.jpg', 'image3.jpg']; // Создаём массив строк — это пути к изображениям
const gallery = []; // Создаём пустой массив gallery, куда позже добавим объекты с данными об изображениях
for (let i = 0; i < imageUrls.length; i++) { // Запускаем цикл for, который перебирает все элементы массива imageUrls
    const img = {
        src: imageUrls[i], // Путь к изображению — берём его из imageUrls[i]
        alt: `Изображение ${i + 1}` // Альтернативный текст, например "Изображение 1", "Изображение 2" и так далее
    };
    gallery.push(img); // Добавляем объект img в массив gallery
}
```

В результаты мы получим массив из трёх объектов, который потом удобно использовать для вывода изображений на страницу или для передачи данных на сервер.

```
[
  { src: 'image1.jpg', alt: 'Изображение 1' },
```

```
  { src: 'image2.jpg', alt: 'Изображение 2' },
  { src: 'image3.jpg', alt: 'Изображение 3' }
]
```

Прерывание и перезапуск цикла

Иногда в процессе выполнения цикла нужно досрочно остановить его или, наоборот, пропустить текущую итерацию и перейти к следующей. В JavaScript для этого используется два ключевых слова: **break** и **continue**.

1. Оператор break

Если внутри цикла срабатывает **break**, выполнение цикла немедленно останавливается, и программа продолжает работать дальше — уже после цикла.

Предположим, у нас есть массив задач, и мы хотим найти первую задачу, где надо кому-то позвонить.

```
const tasks = ['Позвонить в авиакомпанию', 'Сделать домашнее задание', 'Позвонить другу'];
for (let i = 0; i < tasks.length; i++) {
  if (tasks[i].toLowerCase().includes('позвонить')) { // Переводим всё в нижний регистр для проверки и ищем слово «позвонить»
    console.log(`Нужно позвонить: ${tasks[i]}`);
    break; //Выходим из цикла после вывода 1-й задачи
  }
}
```

В этом примере цикл перебирает массив задач и останавливается при нахождении первой выполненной задачи, что ускоряет процесс.

2. Оператор continue

Если внутри цикла срабатывает **continue**, текущая итерация прерывается, но сам цикл продолжается — сразу переходит к следующему шагу.

Допустим, мы хотим вывести только задачи, связанные с покупками.

```
const tasks = ['Купить билеты в кино', 'Устранить ошибки в проекте', 'Купить хлеба'];

for (let i = 0; i < tasks.length; i++) {
  if (!tasks[i].toLowerCase().includes('купить')) {
    continue; // Пропускаем задачи, не связанные с покупками
  }
```

```
    console.log(`Задача на покупку: ${tasks[i]}`);
}
```

В этом случае цикл пропускает все задачи, которые уже выполнены, и выводит только те, что ещё предстоит выполнить.

Задания для выполнения лабораторной работы

Вариант 1: Таблица умножения

Выведите таблицу умножения для числа, введённого пользователем (от 1 до 10). Используйте цикл **`for`**. Формат вывода: 7 * 3 = 21.

Вариант 2: Сумма и произведение чисел

Запросите у пользователя число N. Вычислите:

1. Сумму всех чисел от 1 до N
2. Произведение всех чисел от 1 до N (факториал)
3. Среднее арифметическое чисел от 1 до N

Используйте цикл **`while`**.

Вариант 3: Обратный отсчёт

Запросите начальное число (положительное). Выведите обратный отсчёт от этого числа до 0, затем выведите "Старт!". Используйте цикл **`for`** с уменьшением счётчика.

Вариант 4: Поиск делителей

Запросите целое положительное число. Найдите и выведите все его делители (кроме 1 и самого числа). Если делителей нет, вывести "Простое число".

Вариант 5: Последовательность Фибоначчи

Запросите количество элементов последовательности N ($N \geq 2$). Сгенерируйте и выведите первые N чисел Фибоначчи: 0, 1, 1, 2, 3, 5, 8...

Вариант 6: Сумма чётных и нечётных чисел

Запросите 10 чисел у пользователя. Посчитайте и выведите:

1. Сумму всех чётных чисел
2. Сумму всех нечётных чисел

3. Количество введённых чисел

Используйте цикл **`for`**.

Вариант 7: Проверка пароля (с ограничением попыток)

Задайте правильный пароль в коде. Дайте пользователю 3 попытки ввести пароль. После каждой неудачи сообщайте об ошибке и оставшихся попытках. При успешном вводе выведите "Доступ разрешён", при исчерпании попыток — "Доступ заблокирован".

Вариант 8: Степени двойки

Запросите число N. Выведите все степени двойки от 2^0 до 2^N. Используйте цикл **`for`**. Формат: 2^3 = 8.

Вариант 9: Поиск минимального и максимального

Запросите у пользователя количество чисел, которые он хочет ввести, затем сами числа. Найдите и выведите минимальное и максимальное из введённых чисел. Используйте цикл **`for`**.

Вариант 10: Анализ последовательности чисел

Запросите у пользователя числа до тех пор, пока он не введёт 0. Посчитайте:

1. Количество введённых чисел (не считая 0)
2. Их сумму
3. Среднее арифметическое
4. Количество положительных и отрицательных чисел

Используйте цикл **`while`**.

Вариант 11: Числовая пирамида

Запросите целое число больше 1. Разложите его на простые множители и выведите в формате: 60 = 2 * 2 * 3 * 5. Используйте цикл **`while`**.

Вариант 12: Разложение числа на простые множители

Запросите высоту пирамиды N (от 1 до 9). Выведите числовую пирамиду:

text

1

22

333

4444

...

Используйте вложенные циклы f**or**.

Вариант 13: Поиск НОД (алгоритм Евклида)

Запросите два целых положительных числа. Найдите их наибольший общий делитель (НОД) с помощью алгоритма Евклида:

1. Пока оба числа не равны 0
2. Если a > b, то a = a % b, иначе b = b % a
3. НОД = a + b

Выведите результат.

Вариант 14: Симулятор банковского вклада

Запросите начальную сумму вклада, годовую процентную ставку и срок в годах. Рассчитайте и выведите по годам:

- Сумму на начало года
- Начисленные проценты
- Сумму на конец года

Используйте цикл **`for`** для каждого года.

Вариант 15: Генератор случайных чисел с анализом

Сгенерируйте 20 случайных чисел в диапазоне от 1 до 100. Используйте **`Math.random()`**. Найдите и выведите:

1. Все чётные числа
2. Все числа, кратные 5
3. Максимальное и минимальное сгенерированные числа
4. Среднее арифметическое всех чисел

Используйте цикл **`for`**.

Лабораторная работа № 4

Структурирование кода с помощью функций

Цель лабораторной работы: научиться создавать и использовать функции, передавать параметры и возвращать значения, применять принцип декомпозиции

для решения сложных задач.

Методические указания

Функция — это самостоятельный блок кода, предназначенный для выполнения конкретной задачи, который можно многократно вызывать из различных частей программы. Функции являются фундаментальными строительными блоками в JavaScript и позволяют структурировать код по логическим модулям.

Основные цели использования функций:

Декомпозиция — это принцип «разделяй и властвуй», когда сложная задача разбивается на более мелкие, понятные подзадачи. Вместо того чтобы писать один длинный скрипт, вы создаёте несколько функций, каждая из которых решает конкретную часть проблемы.

Повторное использование кода — принцип DRY (Don't Repeat Yourself). Если определённая логика или расчёт нужны в программе более одного раза, их следует оформить в функцию. Это избавляет от дублирования кода, а при необходимости изменения логики её нужно будет исправить только в одном месте.

Упрощение отладки и тестирования — маленькую, изолированную функцию, выполняющую одну чёткую задачу, проверить на ошибки гораздо проще, чем большой монолитный блок кода.

Улучшение читаемости и поддержки кода — хорошо названные функции делают программу похожей на книгу: их вызовы понятно описывают, что происходит в коде, не требуя каждый раз вникать в детали реализации.

Объявление и вызов функций

В JavaScript есть несколько способов объявить функцию, каждый со своими особенностями.

Это классический и самый распространённый способ:

```
function greetUser(name) {
    return `Привет, ${name}!`;
}
```

Особенность: такие функции «поднимаются» в начало своей области видимости. Это означает, что функцию, объявленную этим способом, можно

вызвать в коде до её фактического объявления.

Здесь функция создаётся как часть выражения, обычно через присваивание переменной:

```
const calculateSum = function(a, b) {
    return a + b;
};
```

Особенность: такие функции не поднимаются. Переменная **calculateSum** будет поднята, но её значение (сама функция) будет присвоено только когда выполнение кода дойдёт до этой строки. Вызвать такую функцию до её объявления нельзя.

Вызов функции осуществляется указанием её имени и круглых скобок:

```
const result = calculateSum(5, 3); // result = 8
greetUser('Иван'); // Выполнится код внутри функции
```

Параметры и аргументы функций

Параметры — это переменные, указанные в объявлении функции. Они являются локальными для этой функции.

Аргументы — это конкретные значения, которые передаются в функцию при её вызове.

```
// Параметры: a, b, c
function example(a, b, c) {
    // ...
}
// Аргументы: 1, 2, 3
example(1, 2, 3);
```

Если при вызове передано больше аргументов, чем объявлено параметров, «лишние» аргументы не вызывают ошибку, но получить к ним доступ через имена параметров нельзя.

Если передано меньше аргументов, недостающим параметрам присваивается значение **undefined.**

Возврат значений

Ключевое слово **return** выполняет две задачи:

1. Завершает выполнение функции.

2. Возвращает указанное значение в то место, откуда функция была вызвана.

```
function isAdult(age) {
    if (age >= 18) {
        return true; // Функция завершается здесь
    }
    return false; // Этот код выполнится, только если age < 18
}
const status = isAdult(20); // status = true
```

Если функция не имеет оператора **return** или он указан без значения, она возвращает **undefined**.

```
function doNothing() {
    // нет return
}
const result = doNothing(); // result = undefined
```

Область видимости переменных

Область видимости определяет, где в коде переменная доступна для использования.

Глобальная область видимости — переменные, объявленные вне любых функций или блоков. Они доступны везде в коде, что может приводить к непреднамеренным конфликтам и изменениям.

```
let globalVar = 'Я везде'; // Глобальная переменная

function someFunction() {
    console.log(globalVar); // Доступна
}
```

Локальная область видимости (функциональная) — переменные, объявленные внутри функции с помощью **var**, доступны только внутри этой функции.

```
function myFunc() {
    var localVar = 'Я только тут'; // Локальная переменная
    console.log(localVar); // Работает
}
console.log(localVar); // Ошибка! localVar не определена
```

Принципы декомпозиции и чистые функции

Декомпозиция — это процесс разбиения сложной системы на более простые, независимые компоненты (функции). Хорошая декомпозиция напоминает создание чертежа: сначала вы определяете основные модули, затем разбиваете каждый модуль на подфункции.

Чистая функция — это функция, которая:

1. Детерминирована: при одинаковых входных данных всегда возвращает одинаковый результат.

2.Не имеет побочных эффектов: не изменяет внешние переменные, глобальное состояние, не выполняет ввод/вывод.

```
// Чистая функция
function add(a, b) {
    return a + b; // Только возврат значения, нет побочных эффектов
}
// Нечистая функция (имеет побочный эффект)
let total = 0;
function addToTotal(x) {
    total += x; // Меняет внешнюю переменную
    return total;
}
```

Чистые функции предсказуемы, их легко тестировать и отлаживать. Старайтесь отделять чистую бизнес-логику (вычисления, преобразования данных) от функций с побочными эффектами (работа с DOM, запросы к серверу).

Функция обратного вызова

Callback-функция (функция обратного вызова) — это функция, которая передаётся в другую функцию в качестве аргумента и затем вызывается внутри этой внешней функции.

Это мощная концепция, позволяющая делать код асинхронным и гибким.

```
// Функция, которая выполняет операцию и затем вызывает callback
function processData(data, callback) {
    // ... какая-то обработка данных ...
    const result = data * 2;
```

```
    // Вызов переданной функции обратного вызова
    callback(result);
}

// Callback-функция, которую мы передаём
function showResult(value) {
    console.log(`Результат: ${value}`);
}

// Использование
processData(10, showResult); // Выведет: "Результат: 20"
```

Задания для выполнения лабораторной работы

Вариант 1: Простой калькулятор

Создайте набор функций:

1. add(a, b) - складывает два числа
2. subtract(a, b) - вычитает второе число из первого
3. multiply(a, b) - умножает числа
4. divide(a, b) - делит первое число на второе
5. calculator(operation, a, b) - выбирает нужную операцию

Вариант 2: Проверка данных

Создайте функции для проверки:

1. isEven(number) - проверяет, четное ли число
2. isPositive(number) - проверяет, положительное ли число
3. isValidEmail(email) - проверяет, есть ли @ в email
4. isAdult(age) - проверяет, совершеннолетний ли (возраст ≥ 18)
5. checkAll(data) - проверяет все условия сразу

Вариант 3: Конвертер температур

Создайте функции:

1. celsiusToFahrenheit(celsius) - переводит Цельсий в Фаренгейт
2. fahrenheitToCelsius(fahrenheit) - переводит Фаренгейт в Цельсий
3. celsiusToKelvin(celsius) - переводит Цельсий в Кельвин
4. convertTemperature(value, from, to) - общая функция конвертации

Вариант 4: Генератор случайных чисел

Создайте функции:

1. randomInt(min, max) - случайное целое число
2. randomFloat(min, max) - случайное дробное число
3. randomBoolean() - случайно true или false
4. randomFromArray(array) - случайный элемент массива

Вариант 5: Работа со строками

Создайте функции:

1. reverseString(text) - переворачивает строку
2. countVowels(text) - считает гласные буквы
3. isPalindrome(text) - проверяет, палиндром ли
4. capitalizeFirst(text) - делает первую букву заглавной

Вариант 6: Калькулятор скидок

Создайте функции для магазина:

1. calculatePrice(price, quantity) - цена × количество
2. applyDiscount(price, discountPercent) - применяет скидку
3. addTax(price, taxPercent) - добавляет налог
4. calculateTotal(items, discount, tax) - полный расчет

Вариант 7: Проверка пароля

Создайте функции:

1. checkLength(password, minLength) - проверяет длину
2. hasNumber(password) - проверяет, есть ли цифра
3. hasUpperCase(password) - проверяет, есть ли заглавная буква
4. validatePassword(password) - полная проверка пароля

Вариант 8: Геометрические расчеты

Создайте функции:

1. rectangleArea(width, height) - площадь прямоугольника
2. rectanglePerimeter(width, height) - периметр прямоугольника
3. circleArea(radius) - площадь круга
4. triangleArea(base, height) - площадь треугольника

Вариант 9: Работа с датами

Создайте функции:

1. getCurrentDate() - возвращает текущую дату
2. calculateAge(birthYear) - вычисляет возраст
3. isWeekend(date) - проверяет, выходной ли день
4. daysUntil(date) - сколько дней до указанной даты

Вариант 10: Простой шифратор

Создайте функции:

1. caesarEncode(text, shift) - шифрует текст (сдвиг букв)
2. caesarDecode(text, shift) - расшифровывает текст
3. reverseText(text) - переворачивает текст
4. encodeMessage(text, method) - выбирает метод шифрования

Вариант 11: Калькулятор ИМТ

Создайте функции:

1. calculateBMI(weight, height) - вычисляет ИМТ
2. interpretBMI(bmi) - интерпретирует результат
3. idealWeight(height) - вычисляет идеальный вес
4. healthReport(weight, height) - выдает полный отчет

Вариант 12: Конвертер валют

Создайте функции:

1. rubToUsd(rubles, rate) - рубли в доллары
2. usdToRub(dollars, rate) - доллары в рубли
3. rubToEur(rubles, rate) - рубли в евро
4. convertMoney(amount, from, to, rate) - общая функция

Вариант 13: Генератор поздравлений

Создайте функции:

1. greet(name) - приветствие по имени
2. birthdayGreeting(name, age) - поздравление с ДР
3. holidayGreeting(name, holiday) - поздравление с праздником
4. generateGreeting(type, name, extra) - генератор поздравлений

Вариант 14: Простая викторина

Создайте функции:

1. createQuestion(question, answer) - создает вопрос

2. checkAnswer(userAnswer, correctAnswer) - проверяет ответ
3. calculateScore(correct, total) - вычисляет процент
4. runQuiz(questions) - запускает викторину

Вариант 15: Учет задач

Создайте функции:

1. addTask(tasks, newTask) - добавляет задачу
2. removeTask(tasks, taskIndex) - удаляет задачу
3. completeTask(tasks, taskIndex) - отмечает как выполненную
4. listTasks(tasks, showCompleted) - показывает задачи

Лабораторная работа № 5

Обработка коллекций данных. Методы массивов

Цель лабораторной работы: освоить методы работы с массивами для эффективного перебора, фильтрации, поиска, преобразования и агрегации данных.

Методические указания

Для работы с наборами данных предназначены массивы. Для создания массива применяется литерал массива или конструкция **new Array():**

```
let array_name1 = [item1, item2, ...];
let array_name2 = new Array([item1, item2, ...]);
```

Для повышения производительности и читабельности программного кода рекомендуется использовать литерал массива.

Для получения доступа к элементам массива используется индекс. Индексация элементов начинается с нуля:

```
let cars = ["Saab", "Volvo", "BMW"]; console.log(cars[0]); // Saab
```

Индекс используется как для чтения, так и для записи элемента массива. Причем в отличие от других языков, таких как C# или Java, можно установить элемент, который изначально не установлен:

```
let cars = ["Saab", "Volvo", "BMW"]; cars[10] = "Toyota" console.log(cars[10]); // Toyota console.log(cars[3]); // undefined
```

Также стоит отметить, что в отличие от ряда языков программирования в JavaScript массивы не являются строго типизированными, один массив может хранить данные разных типов:

```
var objects = ["Tom", 12, true, 3.14, false]; console.log(objects.toString());
```

Массивы могут быть одномерными и многомерными. Каждый элемент в многомерном массиве может представлять собой отдельный массив.

```
const students = [ ["Иванов", 20, 5.5],
["Петров", 18, 8.2],
["Сидоров", 21, 7.8]
];
students[0][1] = 19; // присваиваем отдельное значение
console.log(students[0][1]); // 56
```

В языке JavaScript все свойства и методы обработки массивов собраны в глобальном объекте **Array.prototype**, от которого автоматически наследуются все создаваемые массивы.

Все массивы обладают свойством **length**, которой устанавливает или возвращает количество элементов в массиве:

```
let cars = ["Saab", "Volvo", "BMW"]; console.log(cars.length);
// 3
cars.length = 5; console.log(cars[4]); // undefined
```

Методы массивов

Для добавления и удаления элементов массива используются следующие методы:

- **push(...items)** – добавляет элементы в конец,
- **pop()** – извлекает элемент из конца,
- **shift()** – извлекает элемент из начала,
- **unshift(...items)** – добавляет элементы в начало.

Для удаления элемента массива можно использовать оператор **delete**. Однако этот оператор удаляет только значение элемента с заданным ключом без переиндексации:

```
let cars = ["Saab", "Volvo", "BMW"]; delete cars[1];
console.log(cars.length); // 3 console.log(cars[1]); // undefined
```

Универсальный метод **splice()** используется для добавления, удаления и замены элементов массива:

```
splice(index[, deleteCount, elem1, ..., elemN])
```

Он начинает с позиции **index** удалять **deleteCount** элементов и вставлять **elem1, ..., elemN** на их место. Возвращает массив из удалённых элементов.

Метод **slice()** возвращает новый массив, в который копирует элементы, начиная с индекса **start** и до **end** (не включая **end**). Оба ин- декса **start** и **end** могут быть отрицательными. В таком случае отсчёт будет осуществляться с конца массива:

```
slice([start], [end])
```

Метод **forEach()** позволяет запускать функцию для каждого элемента массива. Его синтаксис:

```
forEach(function(item, index, array) {
// ... делать что-то с item
});
```

Функция обратного вызова (callback) вызывается по очереди для каждого элемента массива и принимает следующие параметры:

– **item** – очередной элемент;

– **index** – его индекс;

– **array** – сам массив.

Для поиска элементов в массиве используются следующие методы:

– **indexOf(item, from)** ищет **item**, начиная с индекса **from**, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.

– **lastIndexOf(item, from)** – то же самое, но ищет справа налево.

– **includes(item, from)** – ищет **item**, начиная с индекса **from**, и возвращает **true**, если поиск успешен.

Методы **find**, **findIndex** и **filter** в качестве условия поиска используют функцию-предикат:

```
let result = arr.find(function(item, index, array) {
// если true - возвращается текущий элемент и перебор прерывается
// если все итерации оказались ложными, возвращается undefined
});

let result = arr.findIndex(function(item, index, array) {
// если true - возвращается индекс, на котором был найден элемент, и перебор прерывается
// если все итерации оказались ложными, возвращается -1
});
```

```
let results = arr.filter(function(item, index, array) {
// если true - элемент добавляется к результату, и перебор продолжается
// возвращается пустой массив в случае, если ничего не найдено
})
```

Метод **`map()`** является одним из наиболее полезных и часто используемых. Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции:

```
let result = arr.map(function(item, index, array) {
// возвращается новое значение вместо элемента
});
```

Метод **`sort(fn)`** сортирует массив на месте, меняя в нём порядок элементов. Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам массив.

Полный список методов есть в справочнике MDN.

Задания для выполнения лабораторной работы

Вариант 1: Анализ успеваемости студентов

Дан массив объектов студентов: [{name: "Ольга", grades: [4,5,3,4]}, {name: "Иван", grades: [5,5,4,5]}, {name: "Мария", grades: [3,4,3,4]}].

1. Рассчитайте средний балл каждого студента, добавив новое свойство averageGrade
2. Отфильтруйте студентов со средним баллом выше 4.0
3. Создайте массив имён отличников, отсортированный по алфавиту
4. Найдите общее количество всех оценок "5" среди всех студентов

Вариант 2: Обработка данных о продажах

Дан массив продаж: [{id: 1, product: "Ноутбук", price: 50000, quantity: 2}, {id: 2, product: "Мышь", price: 1500, quantity: 5}, ...].

1. Рассчитайте общую выручку от всех продаж
2. Найдите товары, проданные в количестве более 3 штук
3. Создайте новый массив с товарами и их общей стоимостью (price * quantity)
4. Отсортируйте товары по убыванию общей выручки от каждого

Вариант 3: Фильтрация и группировка книг

Дан массив книг: [{title: "JS для начинающих", author: "Иванов", year: 2020, pages: 300}, ...].

1. Найдите все книги, изданные после 2018 года
2. Сгруппируйте книги по авторам
3. Создайте массив названий книг с указанием автора: ["JS для начинающих (Иванов)", ...]
4. Найдите самую толстую книгу (максимальное количество страниц)

Вариант 4: Обработка данных о погоде

Дан массив температур за неделю: [{day: "Пн", temp: 15}, {day: "Вт", temp: 17}, ...].

1. Найдите среднюю температуру за неделю
2. Определите самый теплый и самый холодный дни
3. Создайте новый массив с температурами в градусах Фаренгейта (temp * 9/5 + 32)
4. Отфильтруйте дни с температурой выше средней

Вариант 5: Управление задачами (ToDo List)

Дан массив задач: [{id: 1, task: "Купить продукты", completed: true, priority: "high"}, ...].

1. Найдите все незавершенные задачи с высоким приоритетом
2. Посчитайте процент выполненных задач
3. Создайте новый массив только с названиями задач, отсортированный по приоритету
4. Обновите статус всех задач с низким приоритетом на "completed: true"

Вариант 6: Анализ социальных сетей

Дан массив постов: [{id: 1, user: "Иван", likes: 150, tags: ["путешествия", "фото"]}, ...].

1. Найдите самый популярный пост (максимальное количество лайков)
2. Соберите все уникальные теги из всех постов
3. Сгруппируйте посты по пользователям
4. Рассчитайте среднее количество лайков на одного пользователя

Вариант 7: Обработка данных о сотрудниках

Дан массив сотрудников: [{name: "Мария", department: "IT", salary: 80000, experience: 3}, ...].

1. Найдите среднюю зарплату по каждому отделу
2. Отфильтруйте сотрудников с опытом более 5 лет
3. Создайте массив сотрудников с повышением зарплаты на 10%
4. Посчитайте общий фонд зарплаты компании

Вариант 8: Анализ товаров интернет-магазина

Дан массив товаров: [{id: 1, name: "Телефон", category: "электроника", price: 30000, rating: 4.5}, ...].

1. Найдите все товары категории "электроника" с рейтингом выше 4.0
2. Рассчитайте среднюю цену по каждой категории
3. Создайте массив товаров со скидкой 15%
4. Определите самый дорогой и самый дешевый товар

Вариант 9: Обработка данных о фильмах

Дан массив фильмов: [{title: "Интерстеллар", genre: "фантастика", duration: 169, year: 2014}, ...].

1. Найдите все фильмы определенного жанра длительностью более 2 часов
2. Сгруппируйте фильмы по годам выпуска
3. Создайте массив строк в формате: "Интерстеллар (2014) - 169 мин"
4. Определите самый длинный фильм

Вариант 10: Анализ финансовых транзакций

Дан массив транзакций: [{id: 1, amount: 5000, type: "доход", category: "зарплата"}, ...].

1. Рассчитайте общий доход и общий расход
2. Сгруппируйте транзакции по категориям
3. Найдите категорию с наибольшими расходами
4. Создайте массив транзакций только с положительными суммами

Вариант 11: Обработка данных о спортивных соревнованиях

Дан массив участников: [{name: "Алексей", score: 85, country: "Россия"}, ...].

1. Найдите тройку лидеров (наибольший score)

2. Посчитайте средний балл по каждой стране
3. Создайте массив участников, отсортированный по стране и имени
4. Определите, есть ли участники с баллом выше 90

Вариант 12: Анализ данных о пользователях сайта

Дан массив пользователей: [{id: 1, email: "test@mail.ru", age: 25, active: true}, ...].

1. Найдите всех активных пользователей старше 18 лет
2. Проверьте, все ли пользователи имеют валидный email (содержит @)
3. Создайте массив email-ов, отсортированный по алфавиту
4. Посчитайте распределение пользователей по возрастным группам

Вариант 13: Обработка данных о заказах в ресторане

Дан массив заказов: [{id: 1, dishes: ["пицца", "салат"], total: 1500, tip: 150}, ...].

1. Рассчитайте средний чек (средний total)
2. Найдите все заказы с чаевыми более 10% от суммы
3. Создайте массив всех уникальных блюд из всех заказов
4. Определите самое популярное блюдо

Вариант 14: Анализ данных о транспорте

Дан массив транспортных средств: [{type: "автобус", capacity: 50, fuel: "дизель"}, ...].

1. Найдите все электромобили (fuel: "электричество")
2. Посчитайте общую вместимость всех транспортных средств
3. Сгруппируйте транспорт по типу топлива
4. Создайте массив объектов с дополнительным свойством isEco (true для электричества и газа)

Вариант 15: Обработка данных о проектах

Дан массив проектов: [{name: "Веб-сайт", team: ["Мария", "Иван"], deadline: "2024-12-01", progress: 75}, ...].

1. Найдите проекты с дедлайном в текущем году
2. Определите проекты, где прогресс менее 50%
3. Создайте массив всех уникальных участников проектов
4. Рассчитайте средний прогресс по всем проектам

Лабораторная работа № 6

Моделирование данных с помощью объектов

Цель лабораторной работы: научиться создавать и использовать объекты для описания сущностей, работать с массивами объектов, осуществлять поиск и фильтрацию данных в коллекциях.

Методические указания

Моделирование данных – это процесс создания абстрактного представления реальных сущностей и их взаимосвязей в программной системе. В JavaScript основным инструментом для моделирования данных являются **объекты** – структуры, которые позволяют объединять данные и поведение в единую сущность.

Объекты в JavaScript представляют собой коллекцию пар "ключ-значение", где ключи являются строками (или символами), а значения могут быть любого типа данных, включая другие объекты, массивы, функции и примитивные значения.

Самый простой способ создания объекта - использование объектного литерала:

```
// Создание объекта с помощью литерала
const user = {
  id: 1,
  name: 'Иван Петров',
  age: 28,
  email: 'ivan@example.com',
  isActive: true,

  // Метод объекта
  greet: function() {
    return `Привет, меня зовут ${this.name}`;
  }
};
console.log(user.name); // Иван Петров
console.log(user.greet()); // Привет, меня зовут Иван Петров
```

Конструкторы объектов

Для создания множества объектов с одинаковой структурой используют функции-конструкторы:

```
// Функция-конструктор
function Book(title, author, year, price) {
  this.id = Date.now(); // уникальный идентификатор
  this.title = title;
  this.author = author;
  this.year = year;
  this.price = price;
  this.isAvailable = true;

  // Метод
  this.getInfo = function() {
    return `${this.title} (${this.author}, ${this.year}) - ${this.price} руб.`;
  };
}

// Создание объектов с помощью конструктора
const book1 = new Book('Преступление и наказание', 'Ф. Достоевский', 1866, 450);
const book2 = new Book('Мастер и Маргарита', 'М. Булгаков', 1966, 520);
console.log(book1.getInfo());
console.log(book2.getInfo());
```

Классы

Современный синтаксис для создания объектов - классы:

```
class Product {
  constructor(id, name, category, price, stock) {
    this.id = id;
    this.name = name;
    this.category = category;
    this.price = price;
    this.stock = stock;
  }
```

```
    // Метод экземпляра
    applyDiscount(percent) {
      const discount = this.price * (percent / 100);
      this.price = Math.round((this.price - discount) * 100) /
100;
      return this.price;
    }

    // Геттер (вычисляемое свойство)
    get isInStock() {
      return this.stock > 0;
    }

    // Статический метод
    static comparePrice(productA, productB) {
      return productA.price - productB.price;
    }
  }

  // Создание объектов-продуктов
  const laptop = new Product(1, 'Ноутбук Dell', 'Электроника',
75000, 5);
  const phone = new Product(2, 'iPhone 14', 'Электроника', 89990,
3);

  console.log(laptop.isInStock); // true
  laptop.applyDiscount(10);
  console.log(laptop.price); // 67500
```

Работа с массивами объектов

Массивы объектов – это коллекции, где каждый элемент является объектом с одинаковой или схожей структурой. Это позволяет хранить и обрабатывать наборы связанных данных.

```
  // Массив объектов-студентов
  const students = [
    { id: 1, name: 'Алексей Иванов', age: 20, course:
```

```
'Информатика', grade: 4.5 },
      { id: 2, name: 'Мария Смирнова', age: 21, course:
'Математика', grade: 4.8 },
      { id: 3, name: 'Дмитрий Кузнецов', age: 19, course: 'Физика',
grade: 4.2 },
      { id: 4, name: 'Анна Попова', age: 22, course: 'Информатика',
grade: 4.9 },
      { id: 5, name: 'Сергей Васильев', age: 20, course:
'Математика', grade: 4.0 }
    ];

    // Доступ к свойствам объектов в массиве
    console.log(students[0].name); // Алексей Иванов
    console.log(students[2]['course']); // Физика
```

Поиск объектов

```
    // Поиск первого объекта, удовлетворяющего условию (find)
    const studentWithHighGrade = students.find(student =>
student.grade >= 4.8);
    console.log('Студент с высоким баллом:',
studentWithHighGrade?.name);

    // Поиск индекса объекта (findIndex)
    const studentIndex = students.findIndex(student => student.id
=== 3);
    console.log('Индекс студента с id=3:', studentIndex);

    // Проверка существования объекта (some)
    const hasInformaticsStudents = students.some(student =>
student.course === 'Информатика');
    console.log('Есть ли студенты информатики?',
hasInformaticsStudents);

    // Проверка всех объектов (every)
    const allAdults = students.every(student => student.age >= 18);
    console.log('Все студенты совершеннолетние?', allAdults);
```

Фильтрация объектов

```
// Фильтрация массива (filter)
const informaticsStudents = students.filter(student => student.course === 'Информатика');
console.log('Студенты информатики:', informaticsStudents);

const topStudents = students.filter(student => student.grade >= 4.5);
console.log('Отличники:', topStudents);

// Комбинированная фильтрация
const youngInformaticsStudents = students.filter(student =>
  student.course === 'Информатика' && student.age < 22
);
console.log('Молодые студенты информатики:', youngInformaticsStudents);
```

Сортировка объектов

```
// Сортировка по возрастанию оценки
const sortedByGradeAsc = [...students].sort((a, b) => a.grade - b.grade);
console.log('Сортировка по возрастанию оценки:', sortedByGradeAsc);

// Сортировка по убыванию оценки
const sortedByGradeDesc = [...students].sort((a, b) => b.grade - a.grade);
console.log('Сортировка по убыванию оценки:', sortedByGradeDesc);

// Сортировка по имени (алфавитная)
const sortedByName = [...students].sort((a, b) => a.name.localeCompare(b.name));
console.log('Сортировка по имени:', sortedByName);
```

Моделирование данных с помощью объектов в JavaScript позволяет создавать структурированные, читаемые и поддерживаемые приложения. Ключевые аспекты успешного моделирования:

1. **Абстракция** – выделение существенных характеристик сущности
2. **Инкапсуляция** – объединение данных и методов работы с ними
3. **Модульность** – разделение системы на независимые компоненты
4. **Гибкость** – возможность легко расширять и изменять структуры данных

Задания для выполнения лабораторной работы

Вариант 1: Система учета студентов

Создайте систему для учета студентов в университете:

1. Создать объект Student со свойствами: id, имя, фамилия, группа, средний балл
2. Создать массив из 5–7 объектов студентов
3. Написать функции для:
   - Поиска студента по id
   - Поиска всех студентов определенной группы
   - Вычисления среднего балла всей группы
   - Сортировки студентов по среднему баллу
4. Вывести результаты в консоль

Вариант 2: Каталог книг библиотеки

Создать электронный каталог книг.

1. Создать объект Book со свойствами: ISBN, название, автор, год издания, жанр, количество страниц
2. Создать массив из 6–8 объектов книг
3. Написать функции для:
   - Поиска книги по автору
   - Поиска книг определенного жанра
   - Вывода списка книг, изданных после указанного года
   - Подсчета средней длины книги по жанрам
4. Реализовать возможность добавления/удаления книг

Вариант 3: Система заказов в кафе

Создать систему учета заказов.

1. Создать объект MenuItem для блюд (название, цена, категория, время

приготовления)

2. Создать объект Order для заказов (номер, список блюд, статус, общая сумма)
3. Создать массив из 5–7 блюд и 4–6 заказов
4. Написать функции для:
   - Подсчета общей выручки за день
   - Поиска самых популярных блюд
   - Фильтрации заказов по статусу
   - Расчет времени приготовления заказа

Вариант 4: Учет сотрудников компании

Создать систему учета персонала.

1. Создать объект Employee со свойствами: табельный номер, ФИО, должность, отдел, зарплата, дата приема
2. Создать массив из 8–10 объектов сотрудников
3. Написать функции для:
   - Поиска сотрудников по отделу
   - Вычисления средней зарплаты по отделам
   - Поиска сотрудников с зарплатой выше средней
   - Сортировки по дате приема на работу
4. Добавить метод для расчета стажа работы

Вариант 5: Система бронирования отелей

Создать систему бронирования номеров.

1. Создать объект Room (номер, тип, цена за ночь, вместимость, свободна ли)
2. Создать объект Booking (ID брони, гость, номер, даты заезда/выезда, стоимость)
3. Создать массив номеров и бронирований
4. Написать функции для:
   - Поиска свободных номеров на определенные даты
   - Расчет стоимости проживания
   - Поиска бронирований по гостю
   - Вычисления загрузки отеля

Вариант 6: Каталог фильмов

Создать базу данных фильмов.

1. Создать объект Movie со свойствами: название, режиссер, год, жанр, рейтинг, длительность
2. Создать массив из 10–12 объектов фильмов
3. Написать функции для:
   - Поиска фильмов по режиссеру
   - Поиска фильмов определенного жанра
   - Сортировки по рейтингу
   - Вывода списка фильмов выпущенных в указанном году
4. Добавить возможность добавления отзывов к фильмам

Вариант 7: Система учета техники

Создать учет компьютерной техники в офисе.

1. Создать объект Device со свойствами: инвентарный номер, тип, модель, год покупки, ответственный, статус
2. Создать массив из 8–10 объектов техники
3. Написать функции для:
   - Поиска техники по ответственному сотруднику
   - Подсчета техники каждого типа
   - Поиска техники, требующей замены (старше 5 лет)
   - Изменения ответственного за устройство
4. Реализовать историю перемещений техники

Вариант 8: Система тренировок

Создать систему учета тренировок.

1. Создать объект Exercise (название, группа мышц, подходы, повторения)
2. Создать объект Workout (дата, список упражнений, общая длительность, калории)
3. Создать массив упражнений и тренировок
4. Написать функции для:
   - Подсчета общего количества сожженных калорий

- Поиска тренировок по дате
- Анализа прогресса по упражнениям
- Создания тренировочного плана

Вариант 9: Учет транспортных средств

Создать систему учета автомобилей.

1. Создать объект Vehicle (марка, модель, год выпуска, госномер, владелец, пробег)
2. Создать объект Service (дата, вид обслуживания, стоимость, пробег на момент обслуживания)
3. Создать массив транспортных средств и услуг
4. Написать функции для:
   - Поиска автомобилей по владельцу
   - Подсчета общей стоимости обслуживания
   - Определения необходимости ТО (каждые 15000 км)
   - Сортировки автомобилей по пробегу

Вариант 10: Система рецептов

Создать кулинарную книгу с рецептами.

1. Создать объект Ingredient (название, количество, единица измерения)
2. Создать объект Recipe (название, категория, время приготовления, сложность, ингредиенты, инструкция)
3. Создать массив из 6–8 рецептов
4. Написать функции для:
   - Поиска рецептов по ингредиентам
   - Фильтрации по времени приготовления
   - Подсчета стоимости рецепта
   - Конвертации порций (увеличение/уменьшение количества)

Вариант 11: Система учета проектов

Создать систему управления проектами.

1. Создать объект Task (название, исполнитель, срок, статус, приоритет)
2. Создать объект Project (название, руководитель, задачи, бюджет, сроки)

3. Создать массив проектов и задач
4. Написать функции для:
   - Поиска задач по исполнителю
   - Подсчета выполнения проекта в процентах
   - Поиска просроченных задач
   - Распределения бюджета по задачам

Вариант 12: Учет домашних животных

Создать систему учета питомцев в ветеринарной клинике.

1. Создать объект Pet (кличка, вид, порода, возраст, владелец, вес)
2. Создать объект Visit (дата, причина, диагноз, лечение, стоимость)
3. Создать массив питомцев и посещений
4. Написать функции для:
   - Поиска питомцев по владельцу
   - Подсчета средней стоимости лечения по видам животных
   - Напоминания о прививках (раз в год)
   - Ведения истории болезней

Вариант 13: Система учета финансов

Создать систему личного финансового учета.

1. Создать объект Transaction (дата, тип (доход/расход), категория, сумма, описание)
2. Создать объект Budget (категория, лимит на месяц, потрачено)
3. Создать массив транзакций за месяц
4. Написать функции для:
   - Подсчета доходов и расходов
   - Анализа трат по категориям
   - Поиска превышения бюджета
   - Прогнозирования расходов на следующий месяц

Вариант 14: Учет спортивных мероприятий

Создать систему учета спортивных соревнований.

1. Создать объект Athlete (имя, страна, вид спорта, возраст, рекорд)

2. Создать объект Competition (название, дата, место, участники, результаты)
3. Создать массив спортсменов и соревнований
4. Написать функции для:
    - Поиска спортсменов по виду спорта
    - Определения победителей соревнований
    - Сортировки спортсменов по рекордам
    - Подсчета медалей по странам

Вариант 15: Система учета товаров на складе

Создать систему складского учета.

1. Создать объект Product (код товара, название, категория, цена, количество, поставщик)
2. Создать объект Transaction (тип (приход/расход), товар, количество, дата, ответственный)
3. Создать массив товаров и транзакций
4. Написать функции для:
    - Поиска товаров по категории
    - Подсчета общей стоимости товаров на складе
    - Определения товаров, требующих пополнения (меньше минимального запаса)
    - Анализа продаж по периодам

Лабораторная работа № 7

Взаимодействие с DOM: чтение данных страницы

Цель лабораторной работы: освоить методы поиска элементов на HTML-странице и извлечения информации из них с помощью JavaScript.

Методические указания

Document Object Model, или DOM, представляет собой концептуальный мост между статической структурой HTML-документа и динамической природой JavaScript. Это не просто инструмент, а целая философия представления веб-документов как иерархических структур данных, доступных для программной манипуляции. Когда браузер получает HTML-документ, он выполняет нетривиальный процесс синтаксического анализа, преобразуя линейную последовательность тегов в сложное древовидное представление, где каждый элемент становится объектом с определенными свойствами и методами.

Сущность DOM заключается в его двойственной природе: с одной стороны, это точное отражение HTML-структуры, с другой — динамическая модель, которая может существовать независимо от исходного документа. Эта двойственность позволяет разработчикам работать с веб-страницами как с живыми сущностями, способными изменяться, реагировать и адаптироваться к действиям пользователя. DOM служит универсальным языком общения между различными компонентами веб-платформы, обеспечивая согласованность взаимодействия между HTML, CSS и JavaScript.

Архитектурные принципы построения DOM-дерева

В основе DOM лежит древовидная структура, где корневым элементом выступает объект документа, а каждая ветвь представляет собой определенный аспект HTML-разметки. Это дерево строится по принципам иерархии и вложенности, отражая семантические отношения между элементами. Каждый узел в этом дереве не является изолированной сущностью — он находится в сложной системе связей: родительские и дочерние отношения, соседство, наследование свойств.

Узлы DOM классифицируются по типам, каждый из которых обладает специфическими характеристиками. Элементные узлы соответствуют HTML-

тегам и составляют структурный каркас документа. Текстовые узлы содержат собственно содержание страницы — символы, слова, предложения. Узлы атрибутов хранят дополнительные характеристики элементов, влияющие на их поведение и отображение. Комментарии также становятся частью дерева, образуя отдельный тип узлов. Такая типизация обеспечивает строгость модели и позволяет применять к разным категориям узлов соответствующие методы работы.

Важным аспектом архитектуры DOM является его "живая" природа — дерево не является статичной конструкцией, раз и навсегда построенной при загрузке страницы.

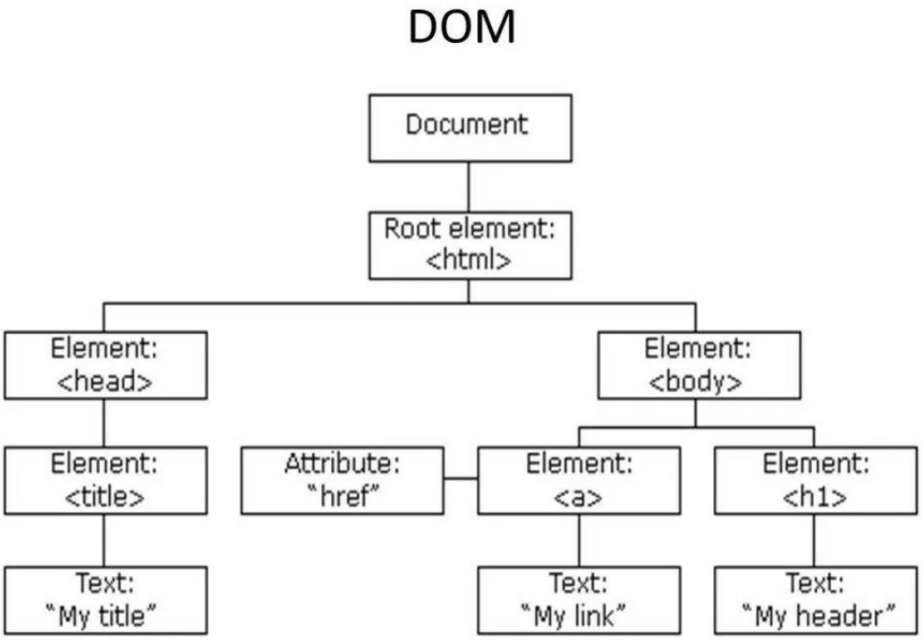

Рисунок 7.1 – Дерево DOM

Оно постоянно находится в состоянии потенциального изменения, готовое трансформироваться под воздействием скриптов и пользовательских взаимодействий. Эта динамичность достигается через механизм событий и реактивных обновлений, которые перестраивают связи между узлами без необходимости полной перезагрузки документа.

```
<div id="container">
    <h1>Заголовок документа</h1>
    <p class="content">Основной текст параграфа</p>
    <!-- Это комментарий в структуре -->
</div>
```

```
    <script>
    // Демонстрация древовидной структуры
    const container = document.getElementById('container');
    console.log('Тип узла контейнера:', container.nodeType); // 1 -
ELEMENT_NODE
    console.log('Имя тега:', container.nodeName); // DIV
    console.log('Количество дочерних узлов:',
container.childNodes.length);
    </script>
```

Методология поиска элементов в DOM-пространстве

Поиск элементов в DOM — это искусство навигации по сложной иерархической структуре. Разработчику доступны различные стратегии, каждая из которых оптимальна для конкретных сценариев. Исторически первыми появились методы, ориентированные на простые критерии поиска: идентификаторы, имена тегов, классы. Эти методы, такие как **`getElementById`** и **`getElementsByTagName`**, формировали первоначальную парадигму работы с DOM, основанную на прямом доступе к элементам через их базовые характеристики.

Эволюция подходов к поиску привела к появлению более выразительных средств — методов, использующих язык CSS-селекторов. **`querySelector`** и **`querySelectorAll`** представляют собой качественный скачок в возможностях навигации по DOM, позволяя описывать сложные условия выборки в компактной и интуитивно понятной форме. Эти методы не просто упрощают синтаксис — они изменяют сам способ мышления о поиске элементов, переводя его из плоскости последовательных проверок в область декларативных описаний.

Выбор между традиционными методами и селекторным API — это не просто вопрос синтаксического предпочтения, а стратегическое решение, влияющее на производительность, читаемость кода и его поддерживаемость. Традиционные методы часто демонстрируют лучшую производительность в простых сценариях благодаря специализированной оптимизации в браузерах. Селекторные методы, в свою очередь, предлагают беспрецедентную гибкость и

выразительность, позволяя в одном выражении описать сложные комбинации условий, для которых ранее требовались многострочные конструкции.

Производительность поиска зависит от множества факторов: глубины вложенности элемента, специфичности селектора, размера DOM-дерева, текущего состояния документа. Понимание этих зависимостей позволяет разработчику выбирать оптимальные стратегии поиска, минимизируя вычислительные затраты и обеспечивая отзывчивость интерфейса. Важным принципом является локализация поиска — ограничение области поиска конкретным поддеревом DOM, что значительно сокращает пространство перебора.

```
<section class="main-content">
    <article data-category="технологии">
        <h2>Новость о технологиях</h2>
        <p>Текст новости...</p>
    </article>
    <article data-category="политика">
        <h2>Политическая новость</h2>
        <p>Текст новости...</p>
    </article>
</section>

<script>
// Различные подходы к поиску элементов
const traditionalSearch =
document.getElementsByTagName('article');
console.log('Традиционный поиск:', traditionalSearch.length,
'элементов');

const selectorSearch = document.querySelectorAll('article[data-
category="технологии"]');
console.log('Селекторный поиск:', selectorSearch.length,
'элементов');

// Навигация по найденным элементам
selectorSearch.forEach((article, index) => {
```

```
        const title = article.querySelector('h2');
        console.log(`Заголовок статьи ${index + 1}:`,
title.textContent);
    });
    </script>
```

Семантика извлечения данных из DOM-элементов

Извлечение информации из элементов DOM — процесс, требующий понимания семантических различий между различными свойствами и методами доступа к содержимому. Каждое свойство, будь то **`textContent`**, **`innerText`** или **`innerHTML`**, несет определенную смысловую нагрузку и предназначено для решения конкретных задач.

`textContent` представляет собой наиболее фундаментальный способ получения текстового содержимого, игнорируя любую HTML-разметку и стилевое оформление. Это свойство возвращает "сырой" текст, включая содержимое всех дочерних элементов, без учета CSS-свойств видимости. Его поведение предсказуемо и детерминировано, что делает его идеальным выбором для сценариев, где требуется получить именно текстовое наполнение элемента без каких-либо интерпретаций.

`innerText`, в отличие от своего собрата, учитывает визуальное представление элемента. Это свойство возвращает только тот текст, который фактически отображается на экране, исключая содержимое скрытых элементов и учитывая CSS-свойства. Такое поведение делает **`innerText`** зависимым от стилей и текущего состояния рендеринга страницы, что может приводить к неожиданным результатам при динамическом изменении стилей.

`innerHTML` открывает доступ к HTML-разметке внутри элемента, позволяя работать с содержимым как со строкой, содержащей теги и атрибуты. Это свойство представляет собой мощный инструмент, требующий осторожного обращения из-за потенциальных уязвимостей безопасности, таких как XSS-атаки. Однако при правильном использовании **`innerHTML`** предоставляет уникальные возможности для анализа и манипуляции структурой содержимого.

Атрибуты элементов образуют отдельный пласт данных, доступ к которым

осуществляется через специализированные методы. Прямое обращение к свойствам объекта, соответствующим стандартным атрибутам HTML, предоставляет простой и интуитивный интерфейс. Методы **getAttribute**, **setAttribute** и **hasAttribute** предлагают более универсальный подход, работающий с любыми атрибутами, включая пользовательские data-атрибуты, которые стали неотъемлемой частью современной веб-разработки.

Навигация по DOM — это не только поиск элементов, но и умение перемещаться между уже найденными узлами. Свойства **parentNode**, **childNodes**, **nextSibling** и **previousSibling** образуют систему координат, позволяющую перемещаться по дереву в любом направлении. Понимание этих связей критически важно для написания эффективного кода, работающего с DOM, так как позволяет минимизировать количество поисковых операций.

```
<article id="news-article">
    <header>
        <h1 data-priority="high">Важная новость</h1>
        <div class="meta" style="display: none;">
            <span class="author">Автор статьи</span>
            <span class="date">2024-01-15</span>
        </div>
    </header>
    <div class="content">
        <p>Первый абзац с <strong>важной</strong>
информацией.</p>
        <p>Второй абзац текста.</p>
    </div>
</article>
<script>
const article = document.getElementById('news-article');
const header = article.querySelector('header');
// Сравнение различных способов получения содержимого
console.log('textContent заголовка:', header.textContent);
console.log('innerText заголовка:', header.innerText);
console.log('innerHTML заголовка:', header.innerHTML);
```

```
    // Работа с атрибутами
    const title = article.querySelector('h1');
    console.log('Стандартный атрибут:', title.id);
    console.log('Пользовательский атрибут:',
title.getAttribute('data-priority'));
    console.log('Все атрибуты:',
Array.from(title.attributes).map(attr => attr.name));

    // Навигация по узлам
    const firstParagraph = article.querySelector('p');
    console.log('Родительский элемент параграфа:',
firstParagraph.parentNode.className);
    console.log('Следующий элемент после параграфа:',
firstParagraph.nextElementSibling.tagName);
    </script>
```

Коллекции узлов и их поведенческие особенности

Коллекции, возвращаемые методами поиска DOM, представляют собой особый тип объектов с уникальными характеристиками поведения. Понимание природы этих коллекций — ключ к написанию корректного и эффективного кода. HTMLCollection и NodeList, две основные разновидности коллекций, отличаются не только названиями, но и фундаментальными аспектами своего функционирования.

HTMLCollection характеризуется своей "живой" природой — это динамическая коллекция, автоматически отражающая текущее состояние DOM. Любые изменения в структуре документа немедленно находят отражение в коллекции, что делает ее синхронизированной с актуальным состоянием страницы. Эта особенность создает как возможности, так и потенциальные проблемы: с одной стороны, разработчик всегда работает с актуальными данными, с другой — изменение DOM во время итерации по коллекции может привести к неожиданным результатам.

NodeList демонстрирует более сложное поведение, зависящее от способа получения коллекции. При использовании свойства childNodes или некоторых других методов NodeList ведет себя как "живая" коллекция, аналогично

HTMLCollection. Однако **querySelectorAll** возвращает статическую NodeList — моментальный снимок состояния DOM на момент выполнения запроса. Эта статичность обеспечивает предсказуемость, но требует явного обновления при изменении документа.

Итерация по коллекциям DOM требует особого внимания к выбору подходящего паттерна. Традиционные циклы for демонстрируют устойчивость при работе с "живыми" коллекциями, особенно когда структура документа может изменяться в процессе обхода. Современные методы итерации, такие как forEach (доступный для NodeList) или преобразование в массив с последующим использованием методов Array, предлагают более выразительный синтаксис, но могут требовать дополнительных мер предосторожности при динамических изменениях DOM.

Оптимизация работы с коллекциями включает несколько ключевых принципов: минимизация количества поисковых операций через кэширование результатов, использование наиболее специфичных селекторов для сокращения размера коллекций, предпочтение статических коллекций в сценариях, где не требуется реактивность к изменениям DOM. Эти принципы, применяемые осознанно, позволяют значительно повысить производительность кода, работающего с DOM.

```
<ul id="item-list">
    <li class="item">Элемент 1</li>
    <li class="item">Элемент 2</li>
    <li class="item">Элемент 3</li>
</ul>

<script>
const list = document.getElementById('item-list');

// Разные типы коллекций
const itemsByClass = list.getElementsByClassName('item'); // HTMLCollection
const itemsBySelector = list.querySelectorAll('.item'); // NodeList (статический)
```

```
    console.log('Тип HTMLCollection:',
itemsByClass.constructor.name);
    console.log('Тип NodeList:', itemsBySelector.constructor.name);

    // Демонстрация "живой" природы HTMLCollection
    console.log('Количество элементов до добавления:',
itemsByClass.length);

    const newItem = document.createElement('li');
    newItem.className = 'item';
    newItem.textContent = 'Новый элемент';
    list.appendChild(newItem);
    console.log('Количество элементов после добавления:',
itemsByClass.length);

    // Итерация с учетом особенностей коллекций
    // Для HTMLCollection безопаснее использовать цикл for
    for (let i = 0; i < itemsByClass.length; i++) {
        console.log('Элемент HTMLCollection:',
itemsByClass[i].textContent);
    }

    // Для NodeList можно использовать forEach
    itemsBySelector.forEach((item, index) => {
        console.log(`Элемент NodeList ${index}:`,
item.textContent);
    });
    </script>
```

Задания для выполнения лабораторной работы

Вариант 1: Анализатор новостной ленты

Создать страницу с несколькими новостными блоками (статьями). Написать скрипт, который анализирует все статьи и выводит статистику: количество статей, список заголовков, среднее количество абзацев в статье.

Вариант 2: Информационная панель пользователя

Создать профиль пользователя с различными полями (имя, возраст, email, навыки в виде списка). Написать скрипт, который собирает все данные из профиля и формирует сводный отчет в отдельном блоке.

Вариант 3: Инвентаризатор интернет-магазина

Создать каталог товаров в виде карточек. Написать скрипт, который анализирует все товары: подсчитывает общее количество, находит самый дорогой и самый дешевый товар, вычисляет среднюю цену.

Вариант 4: Анализатор расписания занятий

Создать таблицу расписания с днями недели и занятиями. Написать скрипт, который извлекает данные о количестве занятий в каждый день, находит день с максимальным количеством занятий.

Вариант 5: Парсер комментариев

Создать секцию с комментариями пользователей (имя, дата, текст, рейтинг). Написать скрипт, который анализирует комментарии: подсчитывает общее количество, находит самый популярный (с максимальным рейтингом), вычисляет средний рейтинг.

Вариант 6: Анализатор меню ресторана

Создать меню ресторана с категориями блюд и ценами. Написать скрипт, который извлекает все цены, подсчитывает количество блюд в каждой категории, находит самое дорогое блюдо.

Вариант 7: Инспектор галереи изображений

Создать галерею изображений с подписями и описаниями. Написать скрипт, который собирает информацию обо всех изображениях: количество, список подписей, размеры изображений.

Вариант 8: Анализатор списка задач (todo-лист)

Создать список задач с чекбоксами и приоритетами. Написать скрипт, который подсчитывает общее количество задач, количество выполненных, количество задач с высоким приоритетом.

Вариант 9: Парсер статистики спортивной команды

Создать таблицу с игроками команды и их статистикой (голы, передачи,

игры). Написать скрипт, который вычисляет лучшего бомбардира, игрока с наибольшим количеством передач, средние показатели команды.

Вариант 10: Анализатор книжной полки

Создать коллекцию книг с авторами, жанрами и годом издания. Написать скрипт, который находит самую старую книгу, подсчитывает книги по жанрам, составляет список всех авторов.

Вариант 11: Инспектор форм регистрации

Создать многостраничную форму регистрации с различными полями. Написать скрипт, который извлекает все названия полей, типы полей, обязательные поля.

Вариант 12: Анализатор финансовых операций

Создать таблицу с финансовыми операциями (дата, тип, сумма, категория). Написать скрипт, который подсчитывает общий доход и расход, находит самую крупную операцию, группирует операции по категориям.

Вариант 13: Парсер блога

Создать страницу блога с постами, тегами и датами публикации. Написать скрипт, который извлекает количество постов за каждый месяц, список всех использованных тегов, самый популярный тег.

Вариант 14: Анализатор маршрутов путешествий

Создать список маршрутов путешествий с городами, датами и стоимостью. Написать скрипт, который находит самый длинный маршрут (по количеству городов), самый дорогой маршрут, среднюю стоимость путешествия.

Вариант 15: Инспектор системы отзывов

Создать систему отзывов о продуктах с оценками от 1 до 5. Написать скрипт, который вычисляет средний рейтинг продукта, количество отзывов каждой оценки, находит самый позитивный и самый негативный отзыв.

Лабораторная работа № 8

Взаимодействие с DOM: изменение структуры и стилей

Цель лабораторной работы: научиться динамически изменять содержимое, атрибуты и CSS-свойства элементов в ответ на события для модификации интерфейса.

Методические указания

Динамическое изменение DOM — это фундаментальный механизм, который превращает веб-страницы из пассивных документов в активные приложения. Эта способность к трансформации лежит в основе всех современных интерактивных интерфейсов, от простых форм обратной связи до сложных одностраничных приложений. Манипуляция DOM-элементами позволяет создавать интерфейсы, которые не просто показывают информацию, а ведут диалог с пользователем, адаптируясь к его действиям и предпочтениям.

Три уровня воздействия на DOM-элементы

Изменение содержимого DOM-элементов представляет собой наиболее прямой способ модификации интерфейса. Это процесс, затрагивающий саму суть элементов — то, что пользователь видит и воспринимает как информационное наполнение страницы. Современный JavaScript предлагает несколько подходов к манипуляции содержимым, каждый из которых несет определенную семантическую нагрузку и предназначен для конкретных сценариев использования.

Свойство **`innerHTML`** предоставляет мощный, но требующий осторожности инструмент для работы с HTML-разметкой внутри элемента. Его сила заключается в способности одновременно изменять и структуру, и содержание, вставляя целые фрагменты разметки. Однако эта сила сопряжена с рисками безопасности, особенно при работе с пользовательским вводом. Современные практики рекомендуют использовать **`innerHTML`** только для доверенного контента, предпочитая более безопасные альтернативы для динамического контента.

Методы **`textContent`** и **`innerText`** предлагают более безопасный подход

к работе с текстовым содержимым. **textContent** изменяет исключительно текстовые данные, автоматически экранируя любые HTML-теги, что делает его идеальным выбором для ситуаций, где требуется предотвратить инъекцию произвольного HTML. **innerText**, в свою очередь, учитывает стили отображения и может вести себя непредсказуемо при изменении CSS-свойств, но иногда именно это поведение оказывается полезным.

```
// Базовые операции изменения содержимого
const element = document.getElementById('target');

// Безопасная установка текста
element.textContent = 'Новый безопасный текст';

// Установка HTML (только для доверенного контента)
const trustedHTML = '<span class="highlight">Выделенный текст</span>';
element.innerHTML = trustedHTML;

// Добавление контента без замены существующего
element.insertAdjacentHTML('beforeend', '<p>Дополнительный параграф</p>');
```

Атрибуты: свойства и метаданные элементов

Атрибуты DOM-элементов представляют собой систему метаданных, определяющих дополнительные характеристики и поведение элементов. Работа с атрибутами — это управление невидимой инфраструктурой, которая, тем не менее, оказывает существенное влияние на функционирование интерфейса. Атрибуты можно разделить на две категории: стандартные HTML-атрибуты, такие как **id**, **class**, **src**, и пользовательские data-атрибуты, которые стали неотъемлемой частью современной веб-разработки.

Методы **setAttribute** и **removeAttribute** предоставляют унифицированный интерфейс для работы со всеми типами атрибутов. Их преимущество заключается в последовательности и предсказуемости поведения независимо от типа атрибута. Однако для стандартных атрибутов часто более удобным оказывается прямое обращение к соответствующим свойствам DOM-

объекта, которое может обеспечивать дополнительную валидацию и преобразование значений.

Особого внимания заслуживает работа с атрибутом **class**. Классы CSS являются основным механизмом стилизации элементов, и их динамическое изменение позволяет создавать сложные переходы между состояниями интерфейса. Современный DOM API предлагает несколько подходов к работе с классами: от прямого манипулирования строкой **className** до использования специализированных методов **classList**, которые предоставляют более выразительный и безопасный интерфейс.

```
// Работа с атрибутами и классами
const image = document.querySelector('img.profile');

// Установка стандартного атрибута
image.setAttribute('alt', 'Фотография пользователя');

// Прямой доступ к свойствам (альтернативный способ)
image.src = 'new-image.jpg';

// Работа с классами через classList
const card = document.querySelector('.user-card');
card.classList.add('active', 'highlighted');  // Добавление классов
card.classList.remove('inactive');            // Удаление класса
card.classList.toggle('expanded');            // Переключение класса
card.classList.replace('old-theme', 'new-theme'); // Замена класса

// Проверка наличия класса
if (card.classList.contains('important')) {
    console.log('Элемент помечен как важный');
}
```

Стили: визуальная трансформация элементов

Манипуляция CSS-стилями через JavaScript представляет собой наиболее

тонкий и выразительный инструмент изменения внешнего вида интерфейса. В отличие от изменения классов, которые предопределяют набор стилей, прямое управление свойствами стиля позволяет осуществлять точные, инкрементальные изменения, необходимые для анимаций и динамических эффектов.

```
// Динамическое изменение стилей
const animatedElement = document.getElementById('animatable');

// Прямое изменение стилей через свойство style
animatedElement.style.backgroundColor = '#3498db';
animatedElement.style.transform = 'translateX(100px)
rotate(15deg)';
animatedElement.style.transition = 'all 0.3s ease-out';

// Получение вычисленных стилей (учитывая все источники CSS)
const computedStyles =
window.getComputedStyle(animatedElement);
console.log('Фактический цвет фона:',
computedStyles.backgroundColor);

// Работа с кастомными CSS-переменными
document.documentElement.style.setProperty('--primary-color',
'#e74c3c');
animatedElement.style.setProperty('--local-size', '200px');

// Анимация через изменение стилей
function animateElement(element, properties, duration) {
    element.style.transition = `all ${duration}ms`;
    Object.assign(element.style, properties);
}
```

Событийно-управляемая архитектура изменений

Динамическое изменение DOM почти всегда происходит в контексте событий — пользовательских действий, системных триггеров или временных интервалов. Событийная модель DOM представляет собой мощный механизм реагирования на изменения состояния, который позволяет создавать отзывчивые и интерактивные интерфейсы.

Привязка обработчиков событий к элементам DOM — это процесс установки связей между физическими действиями (клики, нажатия клавиш, движения мыши) и программной логикой изменения интерфейса. Современный DOM API предлагает несколько способов привязки событий, от традиционного присваивания функций свойствам **onclick** и **onchange** до использования метода **addEventListener**, который поддерживает множественные обработчики и предоставляет более гибкий контроль над поведением событий.

Архитектура событий в DOM построена на принципе всплытия и перехвата, который определяет порядок выполнения обработчиков при вложенности элементов. Понимание этой модели критически важно для создания сложных интерактивных компонентов, где события могут обрабатываться на разных уровнях иерархии. Механизм предотвращения стандартного поведения (**preventDefault**) и остановки распространения события (**stopPropagation**) предоставляет тонкий контроль над потоком событий в приложении.

```
// Работа с событиями для триггирования изменений DOM
const interactiveForm = document.getElementById('user-form');
const feedbackPanel = document.querySelector('.feedback');

// Добавление обработчика событий
interactiveForm.addEventListener('submit', function(event) {
    event.preventDefault(); // Предотвращаем стандартную
отправку формы

    // Изменяем DOM в ответ на событие
    feedbackPanel.textContent = 'Обработка данных...';
    feedbackPanel.style.display = 'block';
    feedbackPanel.classList.add('processing');

    // Динамическое изменение стилей
    this.style.opacity = '0.7';
    this.style.pointerEvents = 'none';

    // Имитация асинхронной операции
    setTimeout(() => {
```

```
            feedbackPanel.textContent = 'Данные успешно
отправлены!';
            feedbackPanel.classList.remove('processing');
            feedbackPanel.classList.add('success');

            this.style.opacity = '';
            this.style.pointerEvents = '';

            // Динамическое создание нового элемента
            const confirmation = document.createElement('div');
            confirmation.className = 'confirmation-message';
            confirmation.innerHTML = `
                <h3>Операция завершена</h3>
                <p>Ваши данные были обработаны системой.</p>
            `;

            this.parentNode.insertBefore(confirmation,
this.nextSibling);
        }, 2000);
    });

    // Обработка динамически созданных элементов через
делегирование
    document.addEventListener('click', function(event) {
        if (event.target.matches('.dynamic-button')) {
            const button = event.target;
            button.style.transform = 'scale(0.95)';

            setTimeout(() => {
                button.style.transform = '';
            }, 150);
        }
    });
```

Задания для выполнения лабораторной работы

Вариант 1: Интерактивная галерея с фильтрами

Создать галерею изображений. Добавить кнопки для фильтрации по категориям. При клике на кнопку должны скрываться/показываться соответствующие изображения, меняться стили активной кнопки.

Вариант 2: Динамический переключатель темы

Создать переключатель между светлой и темной темой. При переключении должны изменяться цвета фона, текста, кнопок на всей странице через изменение CSS-классов.

Вариант 3: Аккордеон FAQ

Создать секцию часто задаваемых вопросов. При клике на вопрос должен плавно раскрываться/скрываться ответ, меняться иконка (плюс/минус), выделяться активный вопрос.

Вариант 4: Валидация формы в реальном времени

Создать форму регистрации. Добавить динамическую проверку полей: подсвечивать поле красным при ошибке, зеленым при корректном заполнении, показывать подсказки под полями.

Вариант 5: Интерактивный рейтинг

Создать систему оценки от 1 до 5 звезд. При наведении на звезды они должны подсвечиваться, при клике - фиксироваться оценка, меняться отображение выбранного количества звезд.

Вариант 6: Анимация прогресс-бара

Создать прогресс-бар, который заполняется анимацией при нажатии кнопки "Запуск". Добавить возможность сброса, паузы, изменения скорости анимации.

Вариант 7: Динамическая таблица с сортировкой

Создать таблицу данных. Добавить возможность сортировки по столбцам при клике на заголовок - менять порядок строк, подсвечивать активный столбец.

Вариант 8: Модальное окно с анимацией

Создать кнопку для открытия модального окна. Реализовать плавное появление и исчезновение окна, изменение фона страницы, блокировку прокрутки при открытом окне.

Вариант 9: Переключатель вида списка/сетки

Создать каталог товаров. Добавить кнопки для переключения между

отображением в виде списка и в виде сетки. Менять CSS-классы контейнера, анимацию перехода.

Вариант 10: Динамический слайдер изображений

Создать слайдер с несколькими изображениями. Реализовать кнопки "вперед/назад", автоматическое переключение, индикаторы текущего слайда с анимацией перехода.

Вариант 11: Кастомные чекбоксы и радиокнопки

Заменить стандартные input-элементы на кастомные с изменением внешнего вида при изменении состояния (активный/неактивный).

Вариант 12: Сортировка списка

Создать список элементов, которые можно перетаскивать для изменения порядка. Реализовать визуальные подсказки при перетаскивании, анимацию перемещения.

Вариант 13: Интерактивная карта с маркерами

Создать карту с маркерами. При клике на маркер должно появляться описание места, маркер должен увеличиваться, другие маркеры - уменьшаться.

Вариант 14: Динамический фильтр цен

Создать фильтр товаров по цене с двумя ползунками (мин/макс). При изменении ползунков должны фильтроваться товары, обновляться значения полей, подсвечиваться подходящие товары.

Вариант 15: Анимированное меню "бургер"

Создать адаптивное меню, которое на мобильных устройствах превращается в "бургер". При клике на "бургер" должно разворачиваться меню с анимацией иконки и плавным появлением пунктов.

Лабораторная работа № 9

Динамическая генерация контента

Цель лабораторной работы: научиться создавать и вставлять новые DOM-элементы на страницу на основе структурирования данных.

Методические указания

В современной веб-разработке способность динамически создавать и управлять контентом представляет собой не просто технический навык, а фундаментальную парадигму мышления. Динамическая генерация контента — это искусство превращения данных в живые, интерактивные интерфейсы, которые не просто отображают информацию, но и адаптируются, трансформируются и эволюционируют в реальном времени. Этот процесс выходит за рамки простого добавления элементов в DOM; он становится мостом между абстрактными структурами данных и их конкретным визуальным воплощением.

Принципы конструирования DOM-элементов

Ключевым концептом здесь является идея "шаблонизации" — не в смысле использования шаблонных движков, а как общего подхода к структурированию процесса преобразования. Каждый тип данных требует своего способа визуализации, своей логики превращения в DOM-элементы. Это может быть простой текст, сложная карточка товара, интерактивная форма или целый компонент интерфейса.

```
// Базовый принцип: от данных к DOM
const userData = {
    id: 1,
    name: "Мария Иванова",
    role: "разработчик",
    skills: ["JavaScript", "React", "Node.js"]
};

// Функция-трансформер: превращает данные в DOM-элемент
function createUserCard(data) {
    const card = document.createElement('div');
    card.className = 'user-card';
```

```
        card.dataset.userId = data.id;

        card.innerHTML = `
            <h3 class="user-name">${escapeHTML(data.name)}</h3>
            <span class="user-role">${escapeHTML(data.role)}</span>
            <ul class="skills-list">
                ${data.skills.map(skill =>
                    `<li class="skill-
item">${escapeHTML(skill)}</li>`
                ).join('')}
            </ul>
        `;

        return card;
    }

    // Вспомогательная функция для безопасной вставки HTML
    function escapeHTML(text) {
        const div = document.createElement('div');
        div.textContent = text;
        return div.innerHTML;
    }
```

Методологии создания элементов

Современный DOM API предлагает несколько подходов к созданию элементов, каждый из которых обладает своей философией и оптимальными сценариями применения. Понимание этих различий критически важно для выбора правильного инструмента в конкретной ситуации.

Метод document.createElement() представляет собой фундаментальный, атомарный способ создания DOM-узлов. Его сила — в точности и контроле: каждый элемент создается явно, каждый атрибут устанавливается осознанно. Этот подход особенно ценен при создании сложных, многоуровневых структур, где требуется тонкое управление процессом конструирования.

Альтернативный подход — использование строковых шаблонов с последующим присваиванием innerHTML. Этот метод предлагает

декларативность и выразительность, позволяя описывать структуру элемента в виде HTML-строки. Однако он требует особой осторожности из-за потенциальных уязвимостей безопасности и менее явного контроля над процессом создания.

```
// Сравнение подходов к созданию элементов

// Императивный подход (через createElement)
function createCardImperative(title, description) {
    const card = document.createElement('div');
    card.className = 'card';

    const titleEl = document.createElement('h3');
    titleEl.className = 'card-title';
    titleEl.textContent = title;

    const descEl = document.createElement('p');
    descEl.className = 'card-description';
    descEl.textContent = description;

    card.appendChild(titleEl);
    card.appendChild(descEl);

    return card;
}

// Декларативный подход (через шаблонные строки)
function createCardDeclarative(title, description) {
    const card = document.createElement('div');
    card.className = 'card';

    card.innerHTML = `
        <h3 class="card-title">${escapeHTML(title)}</h3>
        <p class="card-
description">${escapeHTML(description)}</p>
    `;
```

```
        return card;
    }

    // Гибридный подход (DocumentFragment + шаблоны)
    function createCardHybrid(title, description) {
        const template = document.createElement('template');
        template.innerHTML = `
            <div class="card">
                <h3 class="card-title"></h3>
                <p class="card-description"></p>
            </div>
        `;

        const card =
template.content.firstElementChild.cloneNode(true);
        card.querySelector('.card-title').textContent = title;
        card.querySelector('.card-description').textContent =
description;

        return card;
    }
```

Архитектурные паттерны генерации контента

Фабричный подход: специализация создания

Одним из наиболее мощных паттернов в динамической генерации контента является фабричный метод. Суть этого подхода заключается в создании специализированных функций или классов, ответственных за производство определенных типов элементов. Каждая фабрика инкапсулирует знания о том, как создать конкретный вид контента, скрывая сложность его конструирования за простым интерфейсом.

Фабрики могут быть организованы в иерархии, где базовые фабрики определяют общие принципы создания элементов, а специализированные — добавляют специфические детали. Это позволяет создавать сложные системы генерации контента, которые легко расширяются и модифицируются.

```
    // Реализация фабричного паттерна для генерации контента
```

```
    class ContentFactory {
        createElement(type, config) {
            const methodName =
`create${type.charAt(0).toUpperCase() + type.slice(1)}`;

            if (typeof this[methodName] === 'function') {
                return this[methodName](config);
            }

            throw new Error(`Неизвестный тип элемента: ${type}`);
        }
    }

    class UIContentFactory extends ContentFactory {
        createButton(config) {
            const button = document.createElement('button');
            button.type = config.type || 'button';
            button.className = `btn ${config.variant ? 'btn-' +
config.variant : ''}`;
            button.textContent = config.text;

            if (config.onClick) {
                button.addEventListener('click', config.onClick);
            }

            return button;
        }

        createCard(config) {
            const card = document.createElement('div');
            card.className = 'card';

            if (config.title) {
                const title = document.createElement('h3');
                title.className = 'card-title';
                title.textContent = config.title;
```

```
            card.appendChild(title);
        }

        if (config.content) {
            const content = document.createElement('div');
            content.className = 'card-content';

            if (typeof config.content === 'string') {
                content.textContent = config.content;
            } else if (Array.isArray(config.content)) {
                config.content.forEach(item => {
                    const p = document.createElement('p');
                    p.textContent = item;
                    content.appendChild(p);
                });
            }

            card.appendChild(content);
        }

        if (config.actions && config.actions.length > 0) {
            const actions = document.createElement('div');
            actions.className = 'card-actions';

            config.actions.forEach(actionConfig => {
                const actionBtn =
this.createButton(actionConfig);
                actions.appendChild(actionBtn);
            });

            card.appendChild(actions);
        }

        return card;
    }
```

```
        createList(config) {
            const list = document.createElement(config.ordered ?
'ol' : 'ul');
            list.className = config.className || '';

            config.items.forEach(item => {
                const li = document.createElement('li');

                if (typeof item === 'string') {
                    li.textContent = item;
                } else {
                    // Поддержка сложных элементов в списке
                    const itemElement =
this.createElement(item.type, item.config);
                    li.appendChild(itemElement);
          }

                list.appendChild(li);
            });

            return list;
        }
    }

    // Использование фабрики
    const factory = new UIContentFactory();

    const userCard = factory.createElement('card', {
        title: 'Алексей Петров',
        content: 'Старший разработчик с 8-летним опытом',
        actions: [
            { text: 'Профиль', variant: 'primary', onClick: () =>
console.log('Профиль') },
            { text: 'Сообщение', variant: 'secondary' }
        ]
    });
```

```
const skillList = factory.createElement('list', {
    items: ['JavaScript', 'TypeScript', 'React', 'Node.js'],
    ordered: false,
    className: 'skills-list'
});
```

Композитный подход: сборка сложных структур

Еще один важный паттерн — композитный, основанный на идее сборки сложных структур из более простых компонентов. В этом подходе каждый элемент рассматривается как потенциальный строительный блок, который можно комбинировать с другими для создания более сложных конструкций.

Композитный подход особенно эффективен при работе с иерархическими данными, где каждый уровень вложенности соответствует определенному типу DOM-элемента. Это позволяет естественным образом отражать структуру данных в структуре интерфейса.

```
// Композитный подход к генерации контента

class Component {
    constructor(config) {
        this.config = config;
        this.children = [];
    }

    addChild(child) {
        this.children.push(child);
        return this;
    }

    render() {
        const element = this.createElement();

        // Рекурсивный рендеринг детей
        this.children.forEach(child => {
            const childElement = child.render();
            if (childElement) {
```

```
                element.appendChild(childElement);
            }
        });

        return element;
    }

    createElement() {
        // Базовый метод, переопределяется в дочерних классах
        return document.createElement('div');
    }
}

class ContainerComponent extends Component {
    createElement() {
        const container = document.createElement('div');
        container.className = this.config.className ||
'container';
        return container;
    }
}

class HeaderComponent extends Component {
    createElement() {
        const header = document.createElement('header');
        header.className = 'page-header';

        const title = document.createElement('h1');
        title.textContent = this.config.title;
        header.appendChild(title);

        if (this.config.subtitle) {
            const subtitle = document.createElement('p');
            subtitle.className = 'subtitle';
            subtitle.textContent = this.config.subtitle;
            header.appendChild(subtitle);
        }
```

```
            return header;
        }
    }

    class ListComponent extends Component {
        createElement() {
            const list = document.createElement('ul');
            list.className = this.config.className || 'item-list';

            // Если переданы элементы конфигурации, создаем их
            if (this.config.items) {
                this.config.items.forEach(itemConfig => {
                    const item = new ListItemComponent({ text:
itemConfig });
                    list.appendChild(item.render());
                });
            }

            return list;
        }
    }

    class ListItemComponent extends Component {
        createElement() {
            const li = document.createElement('li');
            li.textContent = this.config.text;
            return li;
        }
    }

    // Сборка сложной структуры
    const page = new ContainerComponent({ className: 'main-page'
});

    const header = new HeaderComponent({
```

```
        title: 'Мои проекты',
        subtitle: 'Список текущих и завершенных проектов'
    });

    const projectList = new ListComponent({ className: 'projects-
list' });
    projectList.addChild(new ListItemComponent({ text:
'Корпоративный портал' }));
    projectList.addChild(new ListItemComponent({ text: 'Мобильное
приложение' }));
    projectList.addChild(new ListItemComponent({ text: 'Система
аналитики' }));

    page.addChild(header);
    page.addChild(projectList);

    // Вставка в DOM
    document.body.appendChild(page.render());
```

Адаптивная генерация для различных контекстов

Современные интерфейсы должны адаптироваться к различным контекстам: размеру экрана, типу устройства, предпочтениям пользователя. Динамическая генерация контента позволяет создавать адаптивные интерфейсы, которые изменяют свою структуру в зависимости от контекста.

```
    class AdaptiveContentGenerator {
        constructor() {
            this.breakpoints = {
                mobile: 768,
                tablet: 1024,
                desktop: 1200
            };

            this.currentViewport = this.detectViewport();
            this.setupViewportListener();
        }

        detectViewport() {
```

```
            const width = window.innerWidth;

            if (width < this.breakpoints.mobile) return 'mobile';
            if (width < this.breakpoints.tablet) return 'tablet';
            if (width < this.breakpoints.desktop) return 'desktop';
            return 'large';
        }

        setupViewportListener() {
            let resizeTimeout;
            window.addEventListener('resize', () => {
                clearTimeout(resizeTimeout);
                resizeTimeout = setTimeout(() => {
                    const newViewport = this.detectViewport();
                    if (newViewport !== this.currentViewport) {
                        this.currentViewport = newViewport;
                        this.onViewportChange(newViewport);
                    }
                }, 250);
            });
        }

        onViewportChange(viewport) {
            // Вызываем кастомное событие для уведомления других
компонентов
            const event = new CustomEvent('viewportchange', {
                detail: { viewport }
            });
            document.dispatchEvent(event);
        }

        generateAdaptiveComponent(data, componentConfig) {
            const viewportConfig =
componentConfig[this.currentViewport] ||
                                componentConfig.default;
```

```
            const container = document.createElement('div');
            container.className = `adaptive-component
${this.currentViewport}`;

            // Генерация контента в зависимости от viewport
            switch (this.currentViewport) {
                case 'mobile':
                    return this.generateMobileView(data,
viewportConfig);
                case 'tablet':
                    return this.generateTableView(data,
viewportConfig);
                case 'desktop':
                    return this.generateDesktopView(data,
viewportConfig);
                default:
                    return this.generateDefaultView(data,
viewportConfig);
            }
        }

        generateMobileView(data, config) {
            const element = document.createElement('div');
            element.className = 'mobile-view';

            // Упрощенное представление для мобильных
            element.innerHTML = `
                <h4>${escapeHTML(data.title)}</h4>
                <button class="expand-btn">Подробнее</button>
            `;

            return element;
        }

        generateDesktopView(data, config) {
            const element = document.createElement('div');
```

```
            element.className = 'desktop-view';

            // Полное представление для десктопа
            element.innerHTML = `
                <div class="card">
                    <div class="card-header">
                        <h3>${escapeHTML(data.title)}</h3>
                        <span
class="badge">${escapeHTML(data.category)}</span>
                    </div>
                    <div class="card-body">
                        <p>${escapeHTML(data.description)}</p>
                        <div class="details">
                            <span>Автор:
${escapeHTML(data.author)}</span>
                            <span>Дата:
${formatDate(data.date)}</span>
                        </div>
                    </div>
                    <div class="card-actions">
                        <button class="btn">Читать</button>
                        <button class="btn">Сохранить</button>
                        <button class="btn">Поделиться</button>
                    </div>
                </div>
            `;

            return element;
        }
    }

    // Использование адаптивного генератора
    const adaptiveGenerator = new AdaptiveContentGenerator();

    const articleData = {
        title: 'Новые тенденции в веб-разработке',
```

```
        category: 'Технологии',
        description: 'Полный обзор современных подходов...',
        author: 'Алексей Смирнов',
        date: '2024-01-15'
    };

    const articleComponent =
adaptiveGenerator.generateAdaptiveComponent(articleData, {
        mobile: { simplified: true },
        desktop: { detailed: true, showActions: true },
        default: { standard: true }
    });

    document.getElementById('content-
area').appendChild(articleComponent);
```

Задания для выполнения лабораторной работы

Вариант 1: Генератор случайных пользователей

Создать форму для генерации случайных пользователей с полями: имя, возраст, профессия, аватар. При нажатии кнопки динамически создавать карточку пользователя и добавлять в список.

Вариант 2: Динамический календарь событий

Создать календарь, который генерирует дни месяца. Добавить возможность создания событий - при клике на день открывается форма, после заполнения событие добавляется в этот день.

Вариант 3: Генератор постов в социальной сети

Создать ленту социальной сети. Реализовать форму для создания нового поста (текст, изображение, теги). Динамически генерировать пост и добавлять в начало ленты.

Вариант 4: Конструктор опросов

Создать интерфейс для создания опросов с вопросами и вариантами ответов. Динамически генерировать форму опроса на основе введенных данных и отображать результаты.

Вариант 5: Генератор таблицы умножения

Создать форму для задания размеров таблицы умножения. Динамически генерировать HTML-таблицу с результатами умножения с чередованием цветов строк.

Вариант 6: Динамический список дел с категориями

Создать систему управления задачами с категориями. Добавлять новые задачи в соответствующие категории, создавать новые категории по мере необходимости.

Вариант 7: Генератор игрового инвентаря

Создать систему инвентаря RPG-игры. Динамически генерировать предметы с различными характеристиками (оружие, броня, зелья) и добавлять в инвентарь.

Вариант 8: Конструктор резюме

Создать форму для ввода данных резюме (личная информация, опыт работы, образование, навыки). Динамически генерировать красиво оформленное резюме в отдельной секции.

Вариант 9: Генератор меню ресторана

Создать форму для добавления блюд в меню (название, описание, цена, категория). Динамически генерировать меню с группировкой по категориям.

Вариант 10: Динамическая галерея из API

Получать данные об изображениях с внешнего API (например, Unsplash). Динамически генерировать галерею с карточками изображений, описаниями и информацией об авторе.

Вариант 11: Генератор игровой доски

Создать генератор игровых досок (шахматы, шашки, морской бой). Динамически создавать доску заданного размера с чередующимися цветами клеток.

Вариант 12: Конструктор интернет-магазина

Создать админ-панель для добавления товаров. Динамически генерировать карточки товаров в каталоге на основе введенных данных (фото, название, цена, описание).

Вариант 13: Генератор музыкального плейлиста

Создать форму для добавления треков в плейлист (название, исполнитель, альбом, длительность). Динамически генерировать список треков с возможностью сортировки.

Вариант 14: Динамическая таблица сотрудников

Создать систему учета сотрудников компании. Динамически генерировать таблицу с данными сотрудников (ФИО, должность, отдел, зарплата) с возможностью фильтрации.

Вариант 15: Генератор графика работ

Создать систему планирования смен сотрудников. Динамически генерировать график на неделю/месяц с распределением сотрудников по сменам и дням.

Лабораторная работа № 10

Обработка событий в браузере

Цель лабораторной работы: научиться корректно обрабатывать события мыши и клавиатуры, связывая пользовательский ввод с изменениями в интерфейсе.

Методические указания

Событийная модель браузера представляет собой сложную экосистему взаимодействий, где каждое действие пользователя — клик, нажатие клавиши, движение мыши, прокрутка — генерирует каскад событий, распространяющихся через DOM-дерево подобно волнам в воде. Понимание этой модели — не техническая необходимость, а ключ к созданию интерфейсов, которые чувствуют себя не набором статических элементов, а живыми, отзывчивыми системами.

Событие в браузере — это не просто триггер, а сложный объект, содержащий исчерпывающую информацию о произошедшем взаимодействии. Каждое событие несет в себе метаданные: тип события, целевой элемент, координаты курсора, состояние модификаторов клавиш, временную метку. Эта информация образует контекст события — полную картину обстоятельств, при которых оно произошло.

Объект события — это посланник, путешествующий через DOM-дерево и рассказывающий каждому встречному элементу о том, что произошло. Он содержит не только факты ("произошел клик"), но и намерения ("где именно", "с какими дополнительными условиями"). Понимание структуры этого объекта — первый шаг к осмысленной обработке событий, позволяющей различать нюансы взаимодействия: был ли это одиночный клик или двойной, нажата ли клавиша в сочетании с Shift, произошло ли касание на мобильном устройстве или клик мышью на десктопе.

Одним из наиболее концептуально важных аспектов событийной модели DOM является механизм распространения событий через фазы. Этот процесс напоминает бросание камня в воду: событие возникает в точке взаимодействия (целевой элемент) и расходится кругами через всё DOM-дерево.

Фаза погружения (**`capturing phase`**) представляет собой движение события от корня документа вглубь дерева, к целевому элементу. Это этап, когда

событие только начинает свой путь, и немногие обработчики настроены на его перехват на этой ранней стадии. Фаза цели (**target phase**) — момент, когда событие достигает элемента, с которым непосредственно взаимодействовал пользователь. Здесь происходит основная обработка. Наконец, фаза всплытия (**bubbling phase**) — восходящее движение события от целевого элемента обратно к корню документа, подобно пузырькам воздуха, всплывающим к поверхности воды.

Понимание этих фаз — не академическое знание, а практический инструмент для создания сложных интерактивных систем. Оно позволяет решать, где именно перехватывать события: на ранней стадии погружения для принудительной обработки (например, для блокировки определенных действий), на уровне цели для прямой реакции, или на стадии всплытия для делегированной обработки множества элементов.

```
// Демонстрация фаз событий
document.querySelector('.container').addEventListener('click', function(e) {
    console.log('Фаза всплытия: контейнер');
}, false); // false означает обработку на фазе всплытия

document.querySelector('.container').addEventListener('click', function(e) {
    console.log('Фаза погружения: контейнер');
    e.stopPropagation(); // Остановка распространения
}, true); // true означает обработку на фазе погружения
```

Классификация событий и их семантика

События мыши: от простого клика к сложным жестам

События мыши образуют богатый язык физических взаимодействий, каждый элемент которого несет свою семантическую нагрузку. Клик (**click**) — базовая единица интерактивности, подтверждение выбора. Но за кажущейся простотой скрывается сложность: двойной клик (**dblclick**) требует особой обработки, чтобы не конфликтовать с одиночными кликами; контекстное меню (**contextmenu**) открывает возможности для дополнительных действий; движение

мыши (**mousemove**) позволяет создавать интерфейсы, реагирующие на приближение и траекторию.

Отдельного внимания заслуживают события **drag-and-drop**, которые превращают статические элементы в перемещаемые объекты. Этот комплекс событий (**dragstart, drag, dragover, drop, dragend**) создает целую микромодель физического взаимодействия с виртуальными объектами, требующую точной координации между состоянием интерфейса и действиями пользователя.

События клавиатуры: диалог через текст и комбинации

Клавиатурные события представляют собой второй основной канал взаимодействия, особенно важный для доступности и эффективности работы. События **keydown, keypress и keyup** образуют полный цикл нажатия клавиши, каждый этап которого может использоваться для разных целей: **keydown** — для немедленной реакции, **keypress** — для обработки символов, **keyup** — для завершающих действий.

Особая мощность клавиатурных событий раскрывается в работе с модификаторами (Ctrl, Shift, Alt, Meta) и комбинациями клавиш. Эти механизмы позволяют создавать "горячие клавиши" — быстрые пути к функциональности, которые ускоряют работу опытных пользователей и делают интерфейс многослойным: простым для новичков (через видимые элементы управления) и мощным для экспертов (через клавиатурные комбинации).

События форм: валидация и взаимодействие в реальном времени

Элементы форм генерируют специализированные события, отражающие их уникальную семантику: изменение значения (**change**), ввод данных (**input**), получение и потерю фокуса (**focus**, **blur**), отправку формы (**submit**). Эти события образуют каркас для создания интеллектуальных форм, которые не просто собирают данные, а взаимодействуют с пользователем: проверяют ввод в реальном времени, предлагают подсказки, адаптируют следующие поля на основе уже введенных данных.

Особенность событий форм — их тесная связь с концепцией валидации. Правильная обработка этих событий позволяет создавать формы, которые

обучают пользователя правильному заполнению, предотвращают ошибки до отправки данных, обеспечивают плавный и интуитивный процесс ввода информации.

Делегирование событий

Делегирование событий — это не просто техника оптимизации, а философский подход к организации интерактивности. Вместо того чтобы привязывать обработчики к каждому элементу в отдельности, мы привязываем один обработчик к их общему родителю и определяем, какой именно дочерний элемент стал целью события. Это превращает статическую структуру обработчиков в динамическую систему, способную работать с элементами, которых еще не существует в момент привязки обработчика.

Красота делегирования заключается в его двойственной природе: с одной стороны, это мощный инструмент оптимизации, уменьшающий количество обработчиков и потребление памяти; с другой — это архитектурный паттерн, который делает код более модульным и устойчивым к изменениям. Элементы могут добавляться, удалять, изменяться, но логика их обработки остается централизованной и последовательной.

```
// Паттерн делегирования событий
document.querySelector('.task-list').addEventListener('click', function(event) {
    // Определяем, был ли клик по элементу с классом 'task-item'
    if (event.target.matches('.task-item')) {
        const taskId = event.target.dataset.taskId;
        console.log('Выбрана задача:', taskId);
        // Единая логика обработки для всех задач
        markTaskAsCompleted(taskId);
    }

    // Обработка кнопок внутри задач
    if (event.target.matches('.delete-btn')) {
        const taskElement = event.target.closest('.task-item');
        const taskId = taskElement.dataset.taskId;
```

```
            deleteTask(taskId);
        }
    });
```

Управление потоком событий

В сложных интерфейсах события редко остаются изолированными. Они взаимодействуют, накладываются, конкурируют. Методы **`stopPropagation()`**, **`stopImmediatePropagation()`** и **`preventDefault()`** предоставляют тонкий контроль над этим потоком, позволяя создавать иерархии обработчиков с четкими правилами приоритета и области действия.

`preventDefault()` — это инструмент переопределения стандартного поведения браузера. Он позволяет сказать: "Я знаю, что обычно при клике на ссылку происходит переход, но в этом конкретном контексте я хочу обработать это по-своему". Это мощное средство, требующее осторожности: отключая стандартное поведение, мы берем на себя ответственность за предоставление альтернативной функциональности.

`stopPropagation()` и **`stopImmediatePropagation()`** управляют горизонтальным распространением событий через DOM. Первый останавливает движение события к родительским элементам, но позволяет выполниться другим обработчикам на том же элементе. Второй — более радикальный — останавливает все дальнейшие обработчики, включая те, что привязаны к текущему элементу. Выбор между ними — это баланс между изоляцией и сотрудничеством обработчиков.

Задания для выполнения лабораторной работы

Вариант 1: Интерактивная палитра цветов

Создать палитру цветов, где при клике на цветной квадрат меняется фон страницы, а при наведении показывается информация о цвете.

Вариант 2: Drag-and-drop сортировка изображений

Реализовать галерею изображений, которые можно перетаскивать для изменения порядка. При двойном клике открывать увеличенное изображение.

Вариант 3: Игра "Реакция"

Создать игру на измерение реакции: при появлении мишени в случайном

месте нужно кликнуть по ней как можно быстрее. Записывать время реакции.

Вариант 4: Визуальный редактор текста

Реализовать текстовый редактор с изменением шрифта, цвета, размера через контекстное меню (правый клик) и горячие клавиши.

Вариант 5: Интерактивная карта помещений

Создать схему офиса/квартиры, где при клике на комнату появляется информация о ней, а при наведении подсвечиваются смежные помещения.

Вариант 6: Музыкальный синтезатор

Создать клавиатуру пианино, где клавиши активируются кликом мыши или нажатием клавиш на клавиатуре, с визуальной обратной связью.

Вариант 7: Симулятор рисования

Реализовать холст для рисования с разными кистями, изменением толщины линии по колесу мыши, очисткой области двойным кликом.

Вариант 8: Интерактивный календарь событий

Создать календарь, где можно добавлять события кликом на день, редактировать перетаскиванием, удалять долгим нажатием.

Вариант 9: Игра "Поймай шарик"

Разработать игру, где шарики появляются в случайных местах и исчезают через время. Нужно успеть кликнуть по ним.

Вариант 10: 3D просмотрщик моделей

Создать просмотрщик 3D моделей (или изображений), где вращение происходит при зажатии и движении мыши, масштаб изменяется колесом.

Вариант 11: Система аннотаций

Реализовать возможность добавлять комментарии к тексту: выделение текста создает аннотацию, клик по аннотации показывает детали.

Вариант 12: Игра "Лабиринт"

Создать лабиринт, где персонаж управляется клавишами стрелок, а мышь используется для подсказок и просмотра карты.

Вариант 13: Визуальный конструктор интерфейсов

Разработать конструктор, где элементы перетаскиваются из палитры, изменяют размер при захвате углов, удаляются клавишей Delete.

Вариант 14: Симулятор физики

Создать симулятор, где можно добавлять объекты кликом, задавать им скорость перетаскиванием, а пространственный клик создает силу притяжения/отталкивания.

Вариант 15: Интерактивный опросник

Реализовать опрос с разными типами вопросов: выбор ответа кликом, перетаскивание вариантов для ранжирования, ввод текста с валидацией.

Лабораторная работа № 11

Создание интерактивных интерфейсов (на примере компонента)

Цель лабораторной работы: научиться практически реализовать компонент с управляемым состоянием на основе пользовательских событий.

Методические указания

Современная веб-разработка преодолела эволюционный путь от монолитных страниц к модульным системам, где каждая функциональная единица интерфейса становится автономным миром со своими правилами, состоянием и поведением. Компонентный подход — это не просто техническая методология, а философия декомпозиции сложности, превращающая хаотичное взаимодействие элементов в упорядоченную экосистему независимых, но взаимодействующих сущностей.

Компонент в этом контексте — это не просто группа HTML-элементов, а концептуальная единица, объединяющая три фундаментальных аспекта: представление (как компонент выглядит), состояние (какие данные он хранит) и поведение (как реагирует на взаимодействия). Эта триада образует замкнутый цикл, где изменения состояния автоматически отражаются в представлении, а пользовательские взаимодействия модифицируют состояние.

Архитектура компонента: состояние, представление, события

В сердце каждого интерактивного компонента лежит его состояние — динамическая модель данных, определяющая, что компонент знает о себе в данный момент. Состояние — это не просто переменные, а единый источник истины, от которого производны все аспекты представления компонента. Изменение состояния должно быть единственным способом изменить то, что видит пользователь.

Концепция управляемого состояния предполагает четкое разделение между данными, которыми компонент управляет самостоятельно (внутреннее состояние), и данными, которые приходят извне (внешние свойства). Это разделение создает предсказуемость: зная состояние компонента, мы можем точно предсказать его отображение; изменяя состояние определенным образом, мы получаем конкретные изменения в интерфейсе.

```
// Базовый паттерн компонента с состоянием
class InteractiveComponent {
    constructor(container, initialState = {}) {
        this.container = container;
        this.state = initialState;
        this.init();
    }

    init() {
        this.render(); // Первоначальный рендеринг
        this.bindEvents(); // Привязка обработчиков
    }

    // Единый метод изменения состояния
    setState(newState) {
        // Слияние нового состояния со старым
        this.state = { ...this.state, ...newState };
        // Автоматическое обновление представления
        this.render();
        // Оповещение о изменении состояния
        this.onStateChange();
    }

    render() {
        // Генерация HTML на основе текущего состояния
        this.container.innerHTML = this.generateTemplate();
    }

    generateTemplate() {
        // Абстрактный метод - должен быть реализован в
дочерних классах
        throw new Error('Метод generateTemplate должен быть
реализован');
    }

    bindEvents() {
        // Привязка обработчиков событий
```

```
    }

    onStateChange() {
        // Хук для реакции на изменение состояния
    }
}
```

Представление как функция состояния

Представление компонента — это не статичная разметка, а динамическое выражение, вычисляемое на основе текущего состояния. Этот принцип, известный как "представление как функция состояния", превращает рендеринг из процедуры рисования в процесс вычисления: при заданном состоянии всегда получается одинаковое представление.

Такой подход создает предсказуемость и упрощает отладку: если что-то выглядит неправильно, мы можем проверить состояние; если нужно изменить внешний вид, мы модифицируем способ преобразования состояния в разметку, не трогая саму логику состояния. Это разделение ответственности — ключ к созданию поддерживаемых компонентов.

```
// Пример конкретного компонента
class CounterComponent extends InteractiveComponent {
    constructor(container) {
        super(container, {
            count: 0,
            max: 10,
            min: 0,
            step: 1
        });
    }

    generateTemplate() {
        const { count, max, min } = this.state;

        return `
            <div class="counter">
                <h3>Счетчик: ${count}</h3>
```

```
                    <div class="controls">
                        <button class="btn decrement" ${count <=
min ? 'disabled' : ''}>
                            -
                        </button>
                        <span class="display">${count}</span>
                        <button class="btn increment" ${count >=
max ? 'disabled' : ''}>
                            +
                        </button>
                    </div>
                    <div class="progress">
                        <div class="progress-bar" style="width:
${(count / max) * 100}%"></div>
                    </div>
                    <p class="hint">
                        ${count === max ? 'Достигнут максимум!' :
                          count === min ? 'Достигнут минимум!' :
                          `Можно увеличить до ${max}`}
                    </p>
                </div>
            `;
        }

        bindEvents() {
            this.container.addEventListener('click', (e) => {
                if (e.target.classList.contains('increment')) {
                    this.increment();
                } else if
(e.target.classList.contains('decrement')) {
                    this.decrement();
                }
            });
        }

        increment() {
            const { count, max, step } = this.state;
```

```
        if (count < max) {
            this.setState({ count: count + step });
        }
    }

    decrement() {
        const { count, min, step } = this.state;
        if (count > min) {
            this.setState({ count: count - step });
        }
    }

    onStateChange() {
        console.log('Состояние изменилось:', this.state);
        // Можно отправить событие для родительских компонентов
        const event = new CustomEvent('counterChange', {
            detail: this.state.count
        });
        this.container.dispatchEvent(event);
    }
}
```

События как интерфейс взаимодействия

События в компонентной архитектуре выполняют двойную роль: они являются механизмом внутренней реакции на пользовательские действия и каналом коммуникации с внешним миром. Внутренние события изменяют состояние компонента, внешние — сообщают о значимых изменениях родительским компонентам или другим частям приложения.

Дизайн событийной модели компонента — это искусство баланса между автономией и интеграцией. Компонент должен быть достаточно самостоятельным, чтобы обрабатывать типичные взаимодействия без внешнего вмешательства, но достаточно открытым, чтобы сообщать о важных изменениях и позволять внешнее управление в особых случаях.

Паттерны управления состоянием в компонентах

Конечный автомат

Для компонентов со сложным поведением эффективным паттерном является модель конечного автомата, где компонент всегда находится в одном из четко определенных состояний, а переходы между состояниями происходят в ответ на определенные события. Этот подход превращает хаотичное поведение в предсказуемый процесс с явными правилами.

Составные компоненты и управление состоянием

Сложные интерфейсы часто строятся как иерархии компонентов, где состояние распределяется между уровнями. Ключевой вопрос такой архитектуры — где хранить состояние: на каком уровне иерархии оно должно находиться, чтобы минимизировать сложность и максимизировать переиспользуемость.

Принцип "поднятия состояния" гласит, что, если два компонента должны синхронизировать свое состояние, это состояние должно быть поднято к их ближайшему общему предку. Это создает единый источник истины и предотвращает рассинхронизацию.

```
// Пример управления состоянием между компонентами
class FilterableProductList {
    constructor(container, products) {
        this.container = container;
        this.products = products;
        this.state = {
            filteredProducts: products,
            filters: {
                category: 'all',
                minPrice: 0,
                maxPrice: Infinity,
                inStock: false
            }
        };
        this.init();
    }

    init() {
```

```
            this.render();
            // Передача методов управления состоянием дочерним
компонентам
            this.filterComponent = new FilterComponent(
                this.container.querySelector('.filters'),
                this.onFilterChange.bind(this)
            );

            this.listComponent = new ProductListComponent(
                this.container.querySelector('.product-list'),
                this.state.filteredProducts
            );
        }

        render() {
            this.container.innerHTML = `
                <div class="filterable-list">
                    <div class="filters"></div>
                    <div class="product-list"></div>
                    <div class="stats">
                        Показано:
${this.state.filteredProducts.length} из ${this.products.length}
                    </div>
                </div>
            `;
        }

        onFilterChange(newFilters) {
            // Обновление состояния родительского компонента
            this.state.filters = { ...this.state.filters,
...newFilters };

            // Применение фильтров
            this.applyFilters();

            // Обновление дочернего компонента
```

```
this.listComponent.updateProducts(this.state.filteredProducts);
        }

        applyFilters() {
            const { category, minPrice, maxPrice, inStock } =
this.state.filters;

            this.state.filteredProducts =
this.products.filter(product => {
                const matchesCategory = category === 'all' ||
product.category === category;
                const matchesPrice = product.price >= minPrice &&
product.price <= maxPrice;
                const matchesStock = !inStock || (inStock &&
product.inStock);

                return matchesCategory && matchesPrice &&
matchesStock;
            });
        }
    }

    class FilterComponent {
        constructor(container, onFilterChange) {
            this.container = container;
            this.onFilterChange = onFilterChange;
            this.render();
            this.bindEvents();
        }

        render() {
            this.container.innerHTML = `
                <div class="filter-group">
                    <select class="category-filter">
                        <option value="all">Все категории</option>
                        <option
```

```
value="electronics">Электроника</option>
                        <option value="books">Книги</option>
                        <option value="clothing">Одежда</option>
                    </select>

                    <div class="price-range">
                        <label>
                            Мин. цена: <input type="range"
class="min-price" min="0" max="1000" value="0">
                            <span class="min-price-value">0</span>
                        </label>
                        <label>
                            Макс. цена: <input type="range"
class="max-price" min="100" max="2000" value="2000">
                            <span class="max-price-
value">2000</span>
                        </label>
                    </div>

                    <label>
                        <input type="checkbox" class="in-stock">
Только в наличии
                    </label>
                </div>
            `;
        }

        bindEvents() {
            // Делегирование событий изменения фильтров
            this.container.addEventListener('change', (e) => {
                const filters = {};

                if (e.target.classList.contains('category-filter'))
{
                    filters.category = e.target.value;
                } else if (e.target.classList.contains('min-
price')) {
```

```
                    filters.minPrice = parseInt(e.target.value);
                    this.container.querySelector('.min-price-
value').textContent = e.target.value;
                } else if (e.target.classList.contains('max-
price')) {
                    filters.maxPrice = parseInt(e.target.value);
                    this.container.querySelector('.max-price-
value').textContent = e.target.value;
                } else if (e.target.classList.contains('in-stock'))
{
                    filters.inStock = e.target.checked;
                }

                this.onFilterChange(filters);
            });
        }
    }
```

Задания для выполнения лабораторной работы

Вариант 1: Компонент "Виджет погоды"

Создать виджет погоды, отображающий текущую погоду и прогноз. Реализовать переключение между городами, единицами измерения.

Вариант 2: Компонент "Таблица с фильтрацией"

Разработать таблицу данных с сортировкой по столбцам, фильтрацией по значениям, пагинацией и сохранением состояния фильтров.

Вариант 3: Компонент "Дерево файловой системы"

Создать компонент отображения древовидной структуры с возможностью раскрытия/сворачивания веток, выделением элементов.

Вариант 4: Компонент "График/диаграмма"

Реализовать компонент для отображения графиков с настройкой типа диаграммы, масштабированием, отображением значений при наведении.

Вариант 5: Компонент "Модальное окно"

Создать переиспользуемый компонент модального окна с анимациями, разными типами контента, обработкой клавиши Escape.

Вариант 6: Компонент "Вкладки (табы)"

Разработать компонент вкладок с анимированным переключением, возможностью закрытия вкладок, переупорядочиванием перетаскиванием.

Вариант 7: Компонент "Слайдер изображений"

Реализовать слайдер с автоматическим переключением, ручным управлением, превью миниатюр, полноэкранным режимом.

Вариант 8: Компонент "Календарь бронирования"

Создать календарь для выбора дат с выделением диапазонов, блокировкой определенных дат, отображением занятых периодов.

Вариант 9: Компонент "Рейтинг звездами"

Разработать компонент оценки с возможностью выбора целых и половинных звезд, отображением среднего рейтинга, анимацией при выборе.

Вариант 10: Компонент "Поиск с автодополнением"

Реализовать поле поиска с подсказками при вводе, историей поиска, фильтрацией результатов, выбором с клавиатуры.

Вариант 11: Компонент "Прогресс выполнения"

Создать компонент отображения прогресса с разными типами индикаторов (линейный, круговой), анимацией, этапами.

Вариант 12: Компонент "Дерево зависимостей"

Разработать визуализацию зависимостей между элементами с возможностью выделения цепочек, фильтрации, масштабирования.

Вариант 13: Компонент "Списки с чекбоксами"

Реализовать компонент вложенных списков с чекбоксами, где выбор родительского элемента влияет на дочерние.

Вариант 14: Компонент "Хлебные крошки"

Создать компонент навигационной цепочки с возможностью редактирования, перехода к любому уровню, адаптивным отображением.

Вариант 15: Компонент "Превью загружаемых файлов"

Разработать компонент для предпросмотра загружаемых файлов с миниатюрами, информацией о файлах, возможностью удаления.

Лабораторная работа № 12

Валидация пользовательского ввода в формах

Цель лабораторной работы: научиться проверять данные, введенные в HTML-формы, на корректность и полноту, блокировать некорректную отправку данных.

Методические указания

Валидация пользовательского ввода — это не просто техническая проверка данных, а сложный процесс коммуникации между пользователем и системой. Это диалог, в ходе которого система обучает, направляет и поддерживает пользователя. Правильная валидация превращает заполнение формы из рутинной задачи в интуитивный процесс, где пользователь чувствует помощь системы, а не ее сопротивление.

Три уровня защиты данных

Нативная валидация HTML5

Современный HTML5 предоставляет базовые механизмы проверки: обязательные поля (**required**), шаблоны **(pattern**), минимальные и максимальные значения. Эти проверки работают без JavaScript и обеспечивают мгновенную обратную связь.

```
<!-- Примеры нативной валидации -->
<input type="email" required placeholder="email@example.com">
<input type="password" minlength="8" required>
<input type="number" min="18" max="100">
```

JavaScript-валидация

JavaScript дает полный контроль над процессом проверки. Можно определять сложные правила, создавать кастомные сообщения об ошибках и контролировать время проверки (во время ввода, при потере фокуса, при отправке).

```
// Базовая структура проверки формы
document.querySelector('form').addEventListener('submit',
function(e) {
    if (!validateForm()) {
```

```
            e.preventDefault(); // Блокируем отправку
            showErrors();       // Показываем ошибки
        }
    });

    // Проверка в реальном времени
    document.querySelector('#email').addEventListener('blur',
function() {
        validateEmail(this.value);
    });
```

Серверная валидация

Независимо от клиентских проверок, серверная валидация обязательна. Она защищает от обхода клиентских проверок, проверяет уникальность данных и выполняет сложные бизнес-правила.

Принципы эффективной валидации

Сообщения должны быть понятными и конкретными. Не "Неверный ввод", а "Email должен содержать символ @".

Ошибки лучше показывать сразу, но не прерывая ввод. Идеальный момент — при потере фокуса с поля. Не просто сообщать об ошибке, а подсказывать, как ее исправить.

Специальные виды проверок

Для телефонных номеров, кредитных карт, паролей со сложными требованиями нужны специальные алгоритмы.

Когда обязательность поля зависит от значения другого поля. Например, номер паспорта обязателен только для граждан определенной страны.

Проверка уникальности email или имени пользователя требует обращения к серверу.

Важно обеспечивать доступность для пользователей скринридеров через ARIA-атрибуты:

- **`aria-invalid="true"`** для неверных полей
- **`aria-describedby`** для связывания поля с сообщением об ошибке
- **`aria-live`** для объявления динамических сообщений

Валидация — это искусство баланса между защитой данных и созданием положительного пользовательского опыта. Хорошая валидация не просто предотвращает ошибки, а помогает пользователю успешно выполнить задачу.

Задания для выполнения лабораторной работы

Вариант 1: Форма регистрации пользователя

Создать форму с полями: имя, email, пароль, подтверждение пароля. Валидация формата email, сложности пароля, совпадения паролей.

Вариант 2: Форма оформления заказа

Реализовать форму заказа с валидацией номера телефона, адреса доставки, номера кредитной карты, даты доставки.

Вариант 3: Форма обратной связи

Создать форму с полями: тема, сообщение, приоритет, email для ответа. Валидация длины сообщения, формата email, обязательных полей.

Вариант 4: Форма бронирования столика

Разработать форму бронирования с валидацией даты и времени, количества гостей, номера телефона, специальных требований.

Вариант 5: Форма создания события

Реализовать форму создания события с валидацией даты начала и окончания, участников, бюджета, повторяемости.

Вариант 6: Форма поиска работы

Создать форму поиска вакансий с валидацией диапазона зарплаты, опыта работы, местоположения, ключевых навыков.

Вариант 7: Форма расчета кредита

Разработать форму с валидацией суммы кредита, срока, процентной ставки, первоначального взноса.

Вариант 8: Форма загрузки документа

Реализовать форму загрузки с валидацией типа файла, размера, разрешения изображений, наличия обязательных полей в документе.

Вариант 9: Форма настройки профиля

Создать форму редактирования профиля с валидацией даты рождения, пола, интересов, ссылок на социальные сети.

Вариант 10: Форма подписки на рассылку

Разработать форму подписки с валидацией email, категорий интересов, частоты рассылки, формата писем.

Вариант 11: Форма жалобы/предложения

Реализовать форму обращения с валидацией категории проблемы, описания, важности, контактных данных.

Вариант 12: Форма планирования путешествия

Создать форму планирования поездки с валидацией маршрута, дат, бюджета, участников, необходимых документов.

Вариант 13: Форма оценки товара/услуги

Разработать форму отзыва с валидацией рейтинга, текста отзыва, преимуществ/недостатков, рекомендуемого возраста.

Вариант 14: Форма смены пароля

Реализовать форму смены пароля с валидацией старого пароля, сложности нового, истории использованных паролей.

Вариант 15: Форма импорта данных

Создать форму импорта с валидацией формата файла, структуры данных, обязательных колонок, кодировки.

Лабораторная работа № 13

Работа с Web Storage API (localStorage)

Цель лабораторной работы: научиться сохранять и загружать состояние клиентского приложения, синхронизировать данные интерфейса с локальным хранилищем браузера.

Методические указания

Web Storage API, и в частности его компонент localStorage, представляет собой фундаментальную технологию, которая бросает вызов традиционному представлению о веб-приложениях как о статичных, не сохраняющих состояние сущностях. localStorage — это мост между миром HTTP, где каждое соединение начинается с чистого листа, и миром полноценных приложений, которые помнят пользователя, его предпочтения, его историю взаимодействий.

Суть localStorage заключается в концепции "сохранения состояния" — способности приложения запоминать себя между сессиями, подобно тому, как человек, закрывая книгу, оставляет закладку, чтобы продолжить чтение с того же места. Это превращает веб-приложение из последовательности независимых запросов-ответов в непрерывный опыт, в диалог с памятью.

С одной стороны, данные сохраняются после закрытия браузера, перезагрузки компьютера, даже после переустановки системы — до тех пор, пока пользователь явно не очистит хранилище. С другой стороны, эта память хрупка: она привязана к конкретному браузеру и устройству, ограничена по объему, может быть очищена автоматически при нехватке места. Эта двойственность делает работу с localStorage искусством баланса между надеждой на сохранение и готовностью к потере.

Архитектура Web Storage API: два уровня памяти

localStorage представляет собой хранилище данных без срока действия. Данные, сохраненные здесь, остаются доступными до тех пор, пока их явно не удалят — будь то программно через JavaScript или пользователь через настройки браузера. Эта перманентность делает localStorage идеальным инструментом для хранения пользовательских предпочтений, настроек интерфейса, токенов аутентификации, состояния сложных форм — всего того, что должно

"переживать" сессии.

Важнейшая характеристика localStorage — его привязка к источнику (origin). Это означает, что данные, сохраненные с одного домена, недоступны для другого. Такая изоляция обеспечивает безопасность, но также создает сложности для кросс-доменных сценариев. Каждый источник получает, как правило, 5-10 МБ пространства — достаточно для большинства клиентских нужд, но недостаточно для хранения больших медиафайлов или объемных баз данных.

```
// Базовые операции с localStorage
// Сохранение данных
localStorage.setItem('username', 'Алексей');
localStorage.setItem('theme', 'dark');
localStorage.setItem('lastVisit', new Date().toISOString());

// Получение данных
const username = localStorage.getItem('username'); // "Алексей"
const theme = localStorage.getItem('theme'); // "dark"

// Удаление данных
localStorage.removeItem('theme');

// Полная очистка
// localStorage.clear();

// Проверка наличия ключа
if (localStorage.getItem('username')) {
    console.log('Пользователь сохранен в localStorage');
}
```

Второй компонент Web Storage API — **sessionStorage** — представляет собой хранилище, данные которого живут только в течение одной сессии браузера. Закрытие вкладки или окна приводит к потере всех сохраненных значений. Эта эфемерность делает sessionStorage идеальным для временных данных: состояния многостраничных форм, информации о текущей сессии, промежуточных вычислений.

sessionStorage можно рассматривать как "рабочую память" приложения —

место для хранения информации, которая нужна здесь и сейчас, но не имеет ценности в долгосрочной перспективе. Это память процесса, а не пользователя; контекста, а не личности.

Основные методы API

Web Storage API намеренно спроектирован с минималистичным, интуитивно понятным интерфейсом. Четыре основных метода покрывают большинство сценариев использования:

- **`setItem(key, value)`** — сохранение значения
- **`getItem(key)`** — получение значения
- **`removeItem(key)`** — удаление значения
- **`clear()`** — полная очистка хранилища

Простота этого API — это не недостаток, а дизайнерское решение. Оно отражает философию "делай одну вещь и делай ее хорошо". Для более сложных операций (итерация, поиск, массовые операции) разработчик может использовать дополнительные методы или надстройки.

```
// Дополнительные операции
// Получение количества сохраненных элементов
const itemCount = localStorage.length;

// Получение ключа по индексу
const firstKey = localStorage.key(0);

// Итерация по всем элементам
for (let i = 0; i < localStorage.length; i++) {
    const key = localStorage.key(i);
    const value = localStorage.getItem(key);
    console.log(`${key}: ${value}`);
}

// Проверка поддержки Web Storage
function isWebStorageSupported() {
    try {
        const testKey = '__test__';
        localStorage.setItem(testKey, testKey);
```

```
            localStorage.removeItem(testKey);
            return true;
        } catch (e) {
            return false;
        }
    }
```

Данные в Web Storage хранятся исключительно в виде строк. Это простое, но важное ограничение формирует архитектуру работы с хранилищем. Любые сложные структуры данных (объекты, массивы) должны быть сериализованы в строки, обычно с помощью JSON.

Задания для выполнения лабораторной работы

Вариант 1: Сохранение настроек темы

Реализовать переключатель тем (светлая/темная) с сохранением выбора в localStorage и восстановлением при загрузке.

Вариант 2: Корзина покупок

Создать корзину интернет-магазина, сохраняющую товары в localStorage между сессиями.

Вариант 3: Черновик формы

Разработать форму, которая автоматически сохраняет введенные данные в localStorage при изменении и восстанавливает при перезагрузке.

Вариант 4: Журнал посещений

Создать систему учета посещений страниц с сохранением истории в localStorage, статистикой, очисткой старых записей.

Вариант 5: Локальное кэширование данных API

Реализовать кэширование результатов API запросов в localStorage с проверкой актуальности по времени.

Вариант 6: Сохранение прогресса игры

Создать игру с сохранением прогресса, рекордов, настроек в localStorage.

Вариант 7: Оффлайн-заметки

Разработать приложение для заметок, работающее полностью оффлайн с хранением данных в localStorage.

Вариант 8: Персонализированные настройки интерфейса

Реализовать систему настроек интерфейса (размер шрифта, расположение элементов) с сохранением в localStorage.

Вариант 9: Локальная база контактов

Создать приложение для хранения контактов с поиском, категориями, хранением в localStorage.

Вариант 10: Сохранение сессии пользователя

Разработать систему запоминания пользователя с хранением токена, профиля, предпочтений в localStorage.

Вариант 11: Кэширование изображений

Реализовать систему кэширования изображений в localStorage с ограничением по размеру, LRU-алгоритмом удаления.

Вариант 12: Локальный планировщик задач

Создать планировщик задач с хранением в localStorage, напоминаниями, приоритетами.

Вариант 13: Сохранение состояния многостраничной формы

Разработать форму из нескольких шагов с сохранением прогресса в localStorage и возможностью продолжить позже.

Вариант 14: Локальная аналитика поведения

Реализовать сбор статистики поведения пользователя на странице с хранением в localStorage и периодической отправкой.

Вариант 15: Автосохранение текстового редактора

Создать текстовый редактор с автосохранением в localStorage, историей версий, восстановлением при сбое.

Лабораторная работа № 14

Управление временем: таймеры и интервалы

Цель лабораторной работы: освоить использование setInterval/setTimeout для создания динамических элементов интерфейса.

Методические указания

Управление временем в JavaScript представляет собой одну из наиболее фундаментальных и вместе с тем парадоксальных концепций языка. В среде, которая изначально проектировалась как синхронная и однопоточная, механизмы **`setTimeout`** и **`setInterval`** вводят измерение времени, создавая иллюзию параллелизма в последовательном потоке исполнения. Это не просто технические инструменты — это окна в асинхронную природу современного веба, мосты между мгновенным выполнением и протяженным во времени взаимодействием.

Философская суть таймеров заключается в концепции "отложенного действия". В мире, где большинство операций происходит немедленно, способность сказать "сделай это позже" представляет собой качественный скачок в проектировании взаимодействий. Таймеры позволяют создавать интерфейсы, которые живут в своем временном ритме: анимации, которые разворачиваются во времени, уведомления, которые появляются и исчезают, процессы, которые протекают с контролируемой скоростью.

setTimeout: единичное отложенное выполнение

`setTimeout` — это обещание, данное среде выполнения выполнить определенный код в будущем. Его философия — "сделай это один раз, но позже". Этот механизм фундаментально меняет поток управления программой, вводя концепцию временной задержки между причиной (установка таймера) и следствием.

Внутренняя механика **`setTimeout`** тесно связана с концепцией Event Loop. Когда вызывается **`setTimeout`**, его колбэк не помещается в стек вызовов немедленно. Вместо этого он регистрируется в окружении браузера (или Node.js) с временной меткой. Event Loop постоянно проверяет, не настало ли время выполнения какого-либо из зарегистрированных таймеров, и когда время приходит — помещает соответствующий колбэк в очередь задач, откуда он в

итоге попадает в стек вызовов.

```
// Базовый синтаксис setTimeout
const timerId = setTimeout(() => {
    console.log('Это сообщение появится через 2 секунды');
}, 2000); // 2000 миллисекунд = 2 секунды

// setTimeout с параметрами
const showMessage = (message, user) => {
    console.log(`${user}: ${message}`);
};

setTimeout(showMessage, 1500, 'Привет, мир!', 'Система');

// Отмена таймера
const delayedTask = setTimeout(() => {
    console.log('Этого сообщения вы не увидите');
}, 3000);

// Отменяем выполнение
clearTimeout(delayedTask);
```

setInterval: циклическое повторение во времени

Если **setTimeout** — это единичное обещание, то **setInterval** — это ритм, пульс, регулярное повторение действия через равные промежутки времени. Его философия — "делай это снова и снова, с заданным интервалом". Этот механизм создает временные циклы внутри линейного потока выполнения, позволяя реализовывать процессы, которые должны происходить периодически: обновление данных, анимация, проверка состояния.

Важнейшее концептуальное различие между **setInterval** и серией последовательных setTimeo**ut** заключается в гарантиях временных интервалов. **setInterval** пытается запускать колбэк через равные промежутки времени, но не гарантирует точность из-за загруженности Event Loop. Если выполнение колбэка занимает больше времени, чем интервал, следующие вызовы будут "накладываться" или пропускаться.

```
// Базовый синтаксис setInterval
```

```
let counter = 0;
const intervalId = setInterval(() => {
    counter++;
    console.log(`Прошло ${counter} секунд`);

    if (counter >= 5) {
        clearInterval(intervalId);  //  Останавливаем  через  5
секунд
        console.log('Таймер остановлен');
    }
}, 1000); // Каждую секунду

// Точный интервал с учетом времени выполнения
function preciseInterval(callback, interval) {
    let expected = Date.now() + interval;

    const timeout = setTimeout(tick, interval);

    function tick() {
        const drift = Date.now() - expected;

        callback();

        expected += interval;
        setTimeout(tick, Math.max(0, interval - drift));
    }

    return {
        clear: () => clearTimeout(timeout)
    };
}

// Использование точного интервала
const preciseTimer = preciseInterval(() => {
    console.log('Точное время: ' + new Date().toISOString());
}, 1000);
```

Event Loop: сцена времени выполнения

Понимание работы таймеров невозможно без осознания архитектуры Event Loop. JavaScript имеет один поток выполнения, но благодаря Event Loop может обрабатывать асинхронные операции. Таймеры **(setTimeout, setInterval)** относятся к макротаскам (macrotasks), которые выполняются после всех микротасков (microtasks) — промисов, queueMicrotask, mutation observers.

Эта иерархия выполнения создает важные следствия для временного планирования. Если в микротаске происходит бесконечный цикл или блокирующая операция, макротаски (включая таймеры) никогда не получат управления. Это нужно учитывать при проектировании систем с таймерами.

requestAnimationFrame: время анимаций

Для анимаций и визуальных обновлений существует специализированный API — **requestAnimationFrame**. В отличие от **setInterval,** который работает с фиксированными временными интервалами, **requestAnimationFrame** синхронизируется с частотой обновления экрана (обычно 60 FPS), что делает его идеальным для плавных анимаций.

```
// Анимация с requestAnimationFrame
function animateElement(element, duration = 1000) {
    const startTime = performance.now();

    function animate(currentTime) {
        const elapsed = currentTime - startTime;
        const progress = Math.min(elapsed / duration, 1);

        // Вычисляем позицию (например, от 0 до 500px)
        const position = 500 * progress;
        element.style.transform = `translateX(${position}px)`;

        if (progress < 1) {
            requestAnimationFrame(animate);
        }
    }
```

```
    requestAnimationFrame(animate);
}
```

Дебаунсинг (Debouncing): контроль частоты вызовов

Дебаунсинг — это техника, которая ограничивает частоту вызова функции, гарантируя, что она будет выполнена только после определенного периода бездействия. Это особенно полезно для обработки событий, которые происходят часто (ввод текста, изменение размера окна).

```
function debounce(func, delay) {
    let timeoutId;

    return function(...args) {
        clearTimeout(timeoutId);

        timeoutId = setTimeout(() => {
            func.apply(this, args);
        }, delay);
    };
}

// Использование
const searchInput = document.getElementById('search');
const performSearch = debounce((query) => {
    console.log('Ищем:', query);
    // API запрос или другая тяжелая операция
}, 300);

searchInput.addEventListener('input', (e) => {
    performSearch(e.target.value);
});
```

Таймеры и интервалы в JavaScript — это не просто утилитарные инструменты для отложенного выполнения кода. Это фундаментальные механизмы, которые позволяют создавать интерфейсы, живущие во времени. Они превращают статичные страницы в динамичные приложения, добавляя измерение времени к двухмерному пространству экрана.

Задания для выполнения лабораторной работы

Вариант 1: Таймер обратного отсчета

Создать таймер обратного отсчета до конкретной даты с отображением дней, часов, минут, секунд.

Вариант 2: Анимированный слайдер

Реализовать слайдер изображений с автоматическим переключением через заданные интервалы, паузой при наведении.

Вариант 3: Игра на время

Разработать игру, где нужно выполнить задание за ограниченное время с визуализацией оставшегося времени.

Вариант 4: Анимация прогресс-бара

Создать прогресс-бар с анимированным заполнением, паузой, возобновлением, сбросом.

Вариант 5: Симулятор роста

Реализовать визуализацию роста растения/популяции с изменением состояния через фиксированные интервалы.

Вариант 6: Таймер Pomodoro

Разработать таймер техники Pomodoro с рабочими интервалами, перерывами, уведомлениями.

Вариант 7: Анимация появления элементов

Создать систему последовательного появления элементов с задержками, эффектами, циклическим повторением.

Вариант 8: Таймер кухонный

Реализовать кухонный таймер с несколькими одновременно работающими таймерами, звуковыми сигналами.

Вариант 9: Визуализация алгоритма

Разработать анимированную визуализацию алгоритма сортировки/поиска с пошаговым выполнением через интервалы.

Вариант 10: Игра "Реакция на время"

Создать игру, где нужно нажимать кнопку в определенные моменты времени под музыку/ритм.

Вариант 11: Анимированные уведомления

Реализовать систему уведомлений с автоматическим исчезновением через заданное время, очередью показа.

Вариант 12: Таймер для презентации

Разработать таймер для презентаций с отображением оставшегося времени, предупреждениями, управлением с клавиатуры.

Вариант 13: Симулятор суточных циклов

Создать визуализацию суточного цикла (день/ночь) с плавными переходами, изменяющейся скоростью.

Вариант 14: Анимация загрузки

Реализовать различные анимированные индикаторы загрузки с циклической анимацией, изменением состояния.

Вариант 15: Таймер медитации

Разработать таймер для медитации с этапами, плавными переходами, звуковыми сигналами начала/окончания.

Лабораторная работа № 15

Разработка автономного виджета

Цель лабораторной работы: научиться создавать независимый компонент, работающего с локальным набором данных. Реализовать логику обновления его состояния.

Методические указания

Автономный виджет в веб-разработке представляет собой не просто изолированный компонент интерфейса, а целостную микро-экосистему, существующую в симбиозе с основным приложением, но сохраняющую свою независимость и самодостаточность. Это концепция, которая бросает вызов традиционному монолитному подходу к разработке интерфейсов, предлагая вместо этого модульную архитектуру, где каждый компонент — это самостоятельная вселенная со своими правилами, состоянием и жизненным циклом.

Виджет в этом контексте — это не просто UI-компонент, а концептуальная единица, объединяющая три фундаментальных аспекта: данные (что он знает), поведение (как реагирует) и представление (как выглядит). Автономность означает, что виджет управляет всеми тремя аспектами самостоятельно, не полагаясь на внешние системы для своего базового функционирования, но при этом способен к взаимодействию через четко определенные интерфейсы.

Архитектурные принципы автономного виджета

Идеальный автономный виджет функционирует как черный ящик в системной инженерии. Внешний мир видит только его интерфейс — входные параметры (props) и выходные события (events), но не имеет доступа к его внутренней реализации. Эта инкапсуляция обеспечивает несколько критически важных преимуществ:

1. Изолированность изменений — модификации внутренней реализации виджета не затрагивают окружающий код

2. Переиспользуемость — виджет можно использовать в разных контекстах без адаптации

3. Тестируемость — виджет можно тестировать изолированно от остальной системы

4. Надежность — внутренние ошибки виджета локализуются и не распространяются на приложение

Инкапсуляция достигается через четкое разделение ответственности. Виджет полностью владеет своим DOM-поддеревом, управляет своими стилями, обрабатывает свои события, хранит свое состояние. Любое взаимодействие с внешним миром происходит через явно определенные каналы.

Сердце автономного виджета — его состояние. В отличие от компонентов, которые получают состояние извне (через props или контекст), автономный виджет владеет своим состоянием полностью. Он сам определяет структуру данных, которые ему нужны, сам управляет их жизненным циклом, сам обеспечивает их сохранение и восстановление.

Это состояние может быть двух типов: внутреннее (private state) — данные, используемые исключительно для внутренней работы виджета, и публичное (public state) — данные, которые виджет готов демонстрировать внешнему миру или получать из него. Граница между этими типами состояния четко определена и контролируется виджетом.

```
// Базовый каркас автономного виджета
class AutonomousWidget {
    constructor(container, initialConfig = {}) {
        // Внешний контейнер для встраивания
        this.container = container;

        // Конфигурация виджета
        this.config = {
            theme: 'light',
            locale: 'ru',
            ...initialConfig
        };

        // Внутреннее состояние (приватное)
        this._state = {
```

```
            initialized: false,
            data: null,
            loading: false,
            error: null
        };

        // DOM элементы виджета
        this.elements = {};

        // Инициализация
        this.init();
    }

    init() {
        this.createDOMStructure();
        this.applyConfig();
        this.loadInitialState();
        this.bindEvents();
        this.render();

        this._state.initialized = true;
        this.emit('initialized', this.getPublicState());
    }

    // Абстрактные методы для переопределения
    createDOMStructure() {
        throw new Error('Метод createDOMStructure должен быть
реализован');
    }

    render() {
        throw new Error('Метод render должен быть реализован');
    }

    // Управление состоянием
    setState(newState) {
        const prevState = { ...this._state };
```

```
            this._state = { ...this._state, ...newState };

            // Автоматический рендер при изменении состояния
            if (this.shouldRerender(prevState, this._state)) {
                this.render();
            }

            // Уведомление об изменении состояния
            this.emit('stateChanged', {
                prevState:
this.getPublicStateFromInternal(prevState),
                newState: this.getPublicState()
            });
        }

        getPublicState() {
            // Экспорт только публичной части состояния
            return {
                data: this._state.data,
                loading: this._state.loading,
                error: this._state.error
            };
        }

        // Жизненный цикл
        destroy() {
            this.unbindEvents();
            this.elements.root?.remove();
            this.emit('destroyed');
        }
    }
```

Локальность данных

Автономный виджет стремится к максимальной независимости от внешних источников данных. Он либо работает с данными, которые получает при инициализации, либо способен генерировать их самостоятельно, либо хранит их

локально (в localStorage, IndexedDB). Эта локальность обеспечивает несколько ключевых преимуществ:

1. Работа оффлайн — виджет функционирует без сетевого соединения
2. Производительность — отсутствие сетевых задержек
3. Надежность — независимость от доступности внешних сервисов
4. Конфиденциальность — данные не покидают устройство пользователя

При этом виджет не должен быть полностью изолированным от мира. Он может получать обновления данных через периодические запросы, WebSocket соединения или реакцию на внешние события. Но ключевой принцип: виджет должен оставаться функциональным даже при отсутствии этих каналов связи.

```
// Виджет с локальным хранилищем данных
class LocalStorageWidget extends AutonomousWidget {
    constructor(container, config) {
        super(container, config);

        // Ключ для хранения в localStorage
        this.storageKey = `widget_${config.id || 'default'}`;
    }

    loadInitialState() {
        try {
            const saved = localStorage.getItem(this.storageKey);
            if (saved) {
                const parsed = JSON.parse(saved);
                this.setState({ data: parsed.data });
            } else {
                // Инициализация начальными данными
                this.setState({ data: this.getDefaultData() });
            }
        } catch (error) {
            console.error('Ошибка загрузки состояния:', error);
            this.setState({
                data: this.getDefaultData(),
                error: 'Не удалось загрузить сохраненные данные'
```

```
            });
        }
    }

    saveState() {
        try {
            const stateToSave = {
                data: this._state.data,
                timestamp: Date.now(),
                version: this.config.version || '1.0'
            };

            localStorage.setItem(this.storageKey,
JSON.stringify(stateToSave));
            this.emit('saved', stateToSave);
        } catch (error) {
            console.error('Ошибка    сохранения    состояния:',
error);
            this.emit('saveError', error);
        }
    }

    getDefaultData() {
        // Возвращает данные по умолчанию
        return {
            items: [],
            lastUpdated: null,
            metadata: {}
        };
    }

    // Автосохранение при изменении данных
    setState(newState) {
        super.setState(newState);

        // Автосохранение, если изменились данные
        if     (newState.data     &&     !newState.loading     &&
```

```
!newState.error) {
                this.debouncedSave();
            }
        }

        debouncedSave = this.debounce(() => {
            this.saveState();
        }, 1000);

        debounce(func, delay) {
            let timeout;
            return function(...args) {
                clearTimeout(timeout);
                timeout = setTimeout(() => func.apply(this, args),
delay);
            };
        }
    }
```

Паттерны проектирования автономных виджетов

Автономный виджет можно рассматривать как клиентский аналог микросервиса. Он имеет четко определенный API (методы и события), выполняет конкретную бизнес-функцию, развертывается независимо (может быть загружен динамически), и имеет собственный жизненный цикл.

Сложные виджеты с множеством состояний и переходов между ними эффективно реализуются через паттерн конечного автомата (state machine). Каждое состояние виджета явно определено, переходы между состояниями происходят только в ответ на конкретные события.

Фазы жизни виджета

Автономный виджет имеет четко определенный жизненный цикл, аналогичный компонентам в современных фреймворках:

1. Инициализация — создание DOM, установка начального состояния
2. Монтирование — вставка в DOM страницы
3. Обновление — реакции на изменения состояния
4. Размонтирование — очистка ресурсов при удалении

Интерфейс взаимодействия

Несмотря на автономность, виджет должен уметь общаться с окружающим миром через четко определенные интерфейсы:

1. Конфигурация при создании — начальные параметры
2. Методы API — публичные методы для управления
3. События — уведомления о внутренних изменениях
4. Двусторонняя связь данных — через props и события

```
// Виджет с полным интерфейсом взаимодействия
class CommunicativeWidget extends AutonomousWidget {
    constructor(container, config) {
        super(container, config);

        // Внешние обработчики событий
        this.externalHandlers = new Map();

        // Публичное API
        this.publicAPI = this.createPublicAPI();
    }

    createPublicAPI() {
        return {
            // Чтение состояния
            getState: () => this.getPublicState(),
            getConfig: () => ({ ...this.config }),

            // Управление
            update: (updates) => this.updateWidget(updates),
            reset: () => this.resetWidget(),
            refresh: () => this.refreshData(),

            // Подписка на события
            on: (event, handler) => this.onExternal(event,
handler),
            off: (event, handler) => this.offExternal(event,
handler),
```

```
            // Жизненный цикл
            mount: () => this.mount(),
            unmount: () => this.unmount(),
            destroy: () => this.destroy()
        };
    }

    // Метод для внешнего обновления
    updateWidget(updates) {
        if (updates.config) {
            this.updateConfig(updates.config);
        }

        if (updates.data) {
            this.setState({ data: updates.data });
        }

        if (updates.method && typeof this[updates.method] ===
'function') {
            return this[updates.method](...updates.args || []);
        }
    }

    // Внешние подписки на события
    onExternal(event, handler) {
        if (!this.externalHandlers.has(event)) {
            this.externalHandlers.set(event, new Set());
        }
        this.externalHandlers.get(event).add(handler);
    }

    offExternal(event, handler) {
        if (this.externalHandlers.has(event)) {
            this.externalHandlers.get(event).delete(handler);
        }
    }
```

```
        // Внутренняя эмиссия событий с уведомлением внешних подписчиков
        emit(event, data) {
            super.emit(event, data);

            // Уведомление внешних подписчиков
            if (this.externalHandlers.has(event)) {
                this.externalHandlers.get(event).forEach(handler => {
                    try {
                        handler(data);
                    } catch (error) {
                        console.error(`Ошибка во внешнем обработчике ${event}:`, error);
                    }
                });
            }
        }
    }
```

Автономный виджет, правильно спроектированный и реализованный, становится не просто компонентом интерфейса, а надежным партнером пользователя — инструментом, который работает предсказуемо, сохраняет состояние между сессиями, функционирует без сетевого соединения, и при этом гармонично интегрируется в общую экосистему приложения. Это высшая форма уважения к пользователю: предоставить ему инструмент, который уважает его время, его данные, его контекст.

Задания для выполнения лабораторной работы

Вариант 1: Виджет "Погода"

Создать автономный виджет погоды с локальным кэшированием данных, обновлением по расписанию, оффлайн1работой.

Вариант 2: Виджет "Курсы валют"

Разработать виджет отображения курсов валют с автономной работой на кэшированных данных, настройками частоты обновления.

Вариант 3: Виджет "Список задач"

Реализовать автономный todo-лист с хранением задач в localStorage, сортировкой, фильтрацией, метками.

Вариант 4: Виджет "Календарь событий"

Создать мини-календарь с локальным хранением событий, напоминаниями, отображением на месяц.

Вариант 5: Виджет "Плеер"

Разработать аудиоплеер с плейлистом, сохраненным локально, управлением, прогрессом воспроизведения.

Вариант 6: Виджет "Новостная лента"

Реализовать новостной агрегатор с кэшированием статей, оффлайн-чтением, настройками источников.

Вариант 7: Виджет "Конвертер величин"

Создать конвертер единиц измерения с историей конвертаций, избранными конвертациями, локальным хранением.

Вариант 8: Виджет "Калькулятор"

Разработать продвинутый калькулятор с историей вычислений, сохранением состояния, несколькими режимами.

Вариант 9: Виджет "Счетчик привычек"

Реализовать трекер привычек с локальным хранением прогресса, статистикой, напоминаниями.

Вариант 10: Виджет "Генератор паролей"

Создать генератор паролей с сохранением настроек, историей сгенерированных паролей, оценкой сложности.

Вариант 11: Виджет "Бюджет"

Разработать мини-приложение для учета расходов с локальным хранением операций, категориями, отчетами.

Вариант 12: Виджет "Таймер воды"

Реализовать виджет для напоминания о питье воды с настройками расписания, отслеживанием прогресса.

Вариант 13: Виджет "Цитата дня"

Создать виджет с ежедневной цитатой, локальной базой цитат, возможностью сохранения понравившихся.

Вариант 14: Виджет "Погода для растений"

Разработать виджет рекомендаций по уходу за растениями на основе локально хранимых данных о растениях.

Вариант 15: Виджет "Коллекция рецептов"

Реализовать мини-приложение с коллекцией рецептов, локальным хранением, поиском, фильтрацией по ингредиентам.

Лабораторная работа № 16

Загрузка и отображение внешних данных (AJAX/Fetch)

Цель лабораторной работы: научиться использовать fetch() для получения данных из локального файла (JSON) и их динамического отображения на странице.

Методические указания

Загрузка и отображение внешних данных представляет собой фундаментальный сдвиг парадигмы в веб-разработке — переход от статичных документов к динамическим приложениям, которые живут и дышат через непрерывный обмен данными с внешним миром. Это эволюция от веб-страниц как завершенных произведений к веб-приложениям как открытым системам, постоянно находящимся в диалоге с различными источниками информации.

Суть AJAX (Asynchronous JavaScript and XML) и его современного преемника Fetch API заключается в концепции "непрерывного обновления". Вместо того чтобы рассматривать веб-страницу как конечный продукт, который пользователь получает полностью сформированным, современный подход видит страницу как живую сущность, способную изменяться, расти, обновляться в реальном времени без полной перезагрузки. Это превращает пользовательский опыт из последовательности дискретных "переходов" между страницами в плавный, непрерывный поток взаимодействия.

Fetch API, в частности, представляет собой не просто технический инструмент, а философию работы с сетевыми запросами, основанную на промисах и современной асинхронной модели JavaScript. Это API, которое говорит на языке современного JavaScript, интегрируется с async/await, и отражает эволюцию веба от XML к JSON как lingua franca обмена данными.

Fetch API, представленный в 2015 году, стал ответом на недостатки XMLHttpRequest. Основанный на промисах, он предлагает чистый, цепочечный API, который естественным образом интегрируется с современным JavaScript. Fetch не просто техническое улучшение — это концептуальный сдвиг, который рассматривает сетевые запросы как операции, возвращающие промисы, а не как события, которые нужно слушать.

Ключевые философские принципы Fetch:

1. Промисы как основа — все операции возвращают промисы

2. Потоковое чтение — поддержка чтения больших ответов по частям

3. Сервис-воркеры — интеграция с современными веб-возможностями

4. Гибкость — возможность тонкой настройки запросов через объект конфигурации

Архитектура Fetch API

Fetch API построен вокруг простой, но мощной концепции: функция fetch() принимает URL и опциональный объект конфигурации, и возвращает Promise, который разрешается в Response объект. Эта простота скрывает сложность, позволяя разработчику сосредоточиться на логике приложения, а не на деталях реализации сетевых запросов.

```
// Базовый пример использования fetch
fetch('https://api.example.com/data')
    .then(response => {
        if (!response.ok) {
            throw new Error(`HTTP error! status:
${response.status}`);
        }
        return response.json(); // Парсинг JSON
    })
    .then(data => {
        console.log('Полученные данные:', data);
        // Обработка данных
    })
    .catch(error => {
        console.error('Ошибка загрузки:', error);
        // Обработка ошибок
    });
```

Объект Response

Объект Response, возвращаемый fetch, предоставляет богатый набор методов для работы с ответом сервера:

- response.json() — парсинг JSON

- response.text() — получение текста

- response.blob() — работа с бинарными данными

- response.formData() — работа с формами

- response.arrayBuffer() — работа с бинарными буферами

Каждый из этих методов возвращает Promise, что позволяет строить цепочки обработки. Важное свойство Response — одноразовость тела ответа. После вызова одного из методов чтения тела, тело становится "потребленным" и не может быть прочитано снова. Это требует тщательного проектирования цепочек обработки.

Объект Request

Второй параметр fetch — объект конфигурации — позволяет тонко настраивать HTTP-запрос:

- method — HTTP метод (GET, POST, PUT, DELETE)

- headers — заголовки запроса

- body — тело запроса

- mode — режим CORS

- credentials — отправка кук

- cache — стратегия кэширования

```
// Пример сложного запроса с конфигурацией
const requestOptions = {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + token
    },
    body: JSON.stringify({
        name: 'Иван',
        email: 'ivan@example.com'
    }),
    mode: 'cors',
    credentials: 'include'
};

fetch('https://api.example.com/users', requestOptions)
    .then(response => response.json())
```

```
.then(data => console.log('Ответ сервера:', data));
```

Работа с локальными файлами JSON

Работа с локальными JSON-файлами имеет свои особенности по сравнению с запросами к удаленным API. Браузерные ограничения безопасности (CORS) обычно не применяются при загрузке файлов с того же origin, что делает этот процесс более простым для локальной разработки и тестирования.

Важно понимать структуру проекта: JSON-файлы должны располагаться в директориях, доступных для веб-сервера. При локальной разработке без сервера (открытие HTML-файла напрямую через file://) могут возникать ограничения безопасности, которые можно обойти с помощью локального сервера разработки.

Динамическое отображение данных

Процесс отображения данных, полученных через fetch, можно представить как конвейер трансформаций, где данные проходят через несколько этапов:

1. Загрузка — получение сырых данных из источника
2. Валидация — проверка структуры и целостности данных
3. Нормализация — приведение к единому формату
4. Обогащение — добавление вычисляемых полей
5. Рендеринг — преобразование в DOM элементы
6. Интеграция — вставка в существующий интерфейс

Каждый этап должен быть изолирован и тестируем независимо. Это позволяет создавать гибкие системы, которые легко адаптировать к изменениям в структуре данных или требованиях к отображению.

```
// Паттерн трансформации данных для отображения
class DataRenderer {
    constructor(containerSelector, options = {}) {
        this.container =
document.querySelector(containerSelector);
        this.options = {
            emptyMessage: 'Нет данных для отображения',
            loadingMessage: 'Загрузка данных...',
            errorMessage: 'Ошибка загрузки данных',
            autoRefresh: false,
            refreshInterval: 30000,
```

```
            ...options
        };

        this.data = null;
        this.isLoading = false;
        this.renderQueue = Promise.resolve();
    }

    // Основной метод загрузки и отображения
    async loadAndRender(dataSource) {
        this.setLoadingState(true);

        try {
            const rawData = await dataSource.load();
            const validatedData = this.validateData(rawData);
            const normalizedData =
this.normalizeData(validatedData);
            const enrichedData =
this.enrichData(normalizedData);

            this.data = enrichedData;
            await this.render();

        } catch (error) {
            console.error('Ошибка загрузки/отображения:',
error);
            this.showError(error);

        } finally {
            this.setLoadingState(false);
        }
    }

    // Валидация структуры данных
    validateData(data) {
        // Проверяем, что данные - массив или объект
        if (!data || (typeof data !== 'object')) {
```

```
                throw new Error('Неверный формат данных');
            }

            // Дополнительные проверки в зависимости от ожидаемой
структуры
            if (Array.isArray(data)) {
                // Проверка массива
                if (data.length === 0) {
                    console.warn('Получен пустой массив данных');
                }
            } else {
                // Проверка объекта
                if (!data.data && !data.items) {
                    console.warn('Данные не содержат ожидаемых
полей');
                }
            }

            return data;
        }

        // Нормализация данных к единому формату
        normalizeData(data) {
            if (Array.isArray(data)) {
                return { items: data, type: 'array' };
            } else if (data.items && Array.isArray(data.items)) {
                return { ...data, type: 'items' };
            } else if (data.data && Array.isArray(data.data)) {
                return { items: data.data, metadata: data.metadata,
type: 'data' };
            } else {
                // Преобразуем объект в массив ключ-значение
                const items = Object.entries(data).map(([key,
value]) => ({
                    key,
                    value,
                    type: typeof value
```

```
            }));
            return { items, type: 'object' };
        }
    }

    // Обогащение данных вычисляемыми полями
    enrichData(data) {
        if (!data.items || !Array.isArray(data.items)) {
            return data;
        }

        const enrichedItems = data.items.map((item, index) => {
            // Добавляем индекс
            const enriched = { ...item, _index: index };

            // Добавляем вычисляемые поля
            if (item.createdAt) {
                enriched.timeAgo =
this.getTimeAgo(item.createdAt);
            }

            if (item.price && item.discount) {
                enriched.finalPrice = item.price * (1 -
item.discount / 100);
                enriched.hasDiscount = item.discount > 0;
            }

            return enriched;
```

Задания для выполнения лабораторной работы

Вариант 1: Каталог книг библиотеки

Создать страницу для отображения каталога книг из JSON-файла. Реализовать поиск по названию/автору, фильтрацию по жанру, сортировку по году издания. Отображать обложки, описание, доступность книг.

Вариант 2: Галерея фотографий

Загрузить JSON с данными фотографий (URL, название, автор, дата, теги).

Реализовать сетку изображений с ленивой загрузкой, полноэкранный просмотр, фильтрацию по тегам, слайдер для просмотра.

Вариант 3: Список сотрудников компании

Отобразить данные о сотрудниках из JSON (ФИО, должность, отдел, email, фото). Реализовать организационную диаграмму, поиск по отделу/должности, карточку сотрудника с детальной информацией.

Вариант 4: Расписание занятий

Загрузить JSON с расписанием (день, время, предмет, преподаватель, аудитория). Создать табличное представление с цветовым кодированием предметов, фильтрацией по дню/преподавателю.

Вариант 5: Меню ресторана

Отобразить блюда из JSON (название, описание, цена, категория, время приготовления). Реализовать группировку по категориям, фильтр по цене, калькулятор стоимости заказа, отметку вегетарианских блюд.

Вариант 6: Карта городов с достопримечательностями

Загрузить JSON с городами и их достопримечательностями. Создать интерактивную карту/список, отображение информации при выборе города, фильтрацию по типу достопримечательности.

Вариант 7: Календарь событий с API

Использовать JSON с событиями (дата, название, описание, участники). Реализовать месячный/недельный календарь, просмотр деталей события, фильтрацию по участникам.

Вариант 8: Система заказов интернет-магазина

Отобразить заказы из JSON (номер, дата, товары, статус, сумма). Реализовать таблицу с сортировкой по дате/сумме, фильтрацию по статусу, детальный просмотр заказа.

Вариант 9: Портфолио проектов

Загрузить JSON с проектами (название, описание, технологии, ссылки, скриншоты). Создать карточки проектов, фильтр по технологиям, модальное окно с деталями проекта.

Вариант 10: База знаний/FAQ

Отобразить статьи из JSON (категория, вопрос, ответ, теги). Реализовать древовидную структуру категорий, поиск по тексту, фильтрацию по тегам, подсветку найденного.

Вариант 11: Список фильмов/сериалов

Загрузить JSON с фильмами (название, год, режиссер, жанр, рейтинг, постер). Создать галерею постеров, детальную страницу фильма, фильтры по жанру/году, сортировку по рейтингу.

Вариант 12: Каталог товаров с отзывами

Отобразить товары из JSON (название, цена, описание, характеристики, отзывы). Реализовать сравнение товаров, фильтрацию по характеристикам, отображение рейтинга на основе отзывов.

Вариант 13: Расписание транспорта

Загрузить JSON с маршрутами транспорта (номер, тип, остановки, время). Создать поиск маршрута между остановками, отображение расписания для выбранной остановки.

Вариант 14: База данных рецептов

Отобразить рецепты из JSON (название, ингредиенты, шаги приготовления, время, сложность). Реализовать поиск по ингредиентам, фильтрацию по времени приготовления, список покупок для выбранных рецептов.

Вариант 15: Система учета студентов и оценок

Загрузить JSON с данными студентов (ФИО, группа, оценки по предметам). Создать ведомость успеваемости, расчет среднего балла, фильтрацию по группе, сортировку по успеваемости.

Лабораторная работа № 17

Практическая работа: разработка компонента «Калькулятор» (этап 1 – верстка)

Цель лабораторной работы: создать статический пользовательский интерфейс компонента с использованием HTML и CSS.

Методические указания

Разработка компонента калькулятора начинается с создания его статического представления, что составляет фундаментальную основу всего проекта. На этом этапе акцент делается исключительно на визуальной составляющей и структурной организации элементов, без внедрения какой-либо интерактивной логики. Важно понимать, что качественно выполненная верстка — это не просто набор HTML-тегов и CSS-правил, а продуманная система визуальной коммуникации, где каждый элемент занимает свое законное место и выполняет конкретную семантическую роль.

Процесс верстки калькулятора следует рассматривать как искусство баланса между эстетикой и функциональностью. Внешний вид должен не только радовать глаз, но и интуитивно подсказывать пользователю, как взаимодействовать с интерфейсом. Расположение кнопок, размер дисплея, цветовые акценты — все это элементы языка, на котором система разговаривает с человеком. Калькулятор, несмотря на кажущуюся простоту, представляет собой сложную композицию, где важна каждая деталь: от расстояния между кнопками до толщины границ.

Семантическая корректность разметки играет первостепенную роль. Выбор HTML-элементов должен отражать их назначение: кнопки оформляются тегом **`<button>`**, область вывода может использовать **`<output>`**, а группирующие контейнеры — **`<div>`** с соответствующими классами. Правильная семантика облегчает дальнейшую разработку, улучшает доступность для пользователей с ограниченными возможностями и способствует лучшей индексации поисковыми системами.

Архитектура CSS для калькулятора требует применения современных подходов, таких как CSS Grid для создания точных сеток расположения кнопок и Flexbox для выравнивания содержимого внутри контейнеров. Модульная

организация стилей позволяет легко вносить изменения и адаптировать внешний вид под разные устройства. Особое внимание уделяется состоянию элементов: кнопки должны визуально реагировать на наведение курсора, получение фокуса и нажатие, что создает ощущение отзывчивости интерфейса даже на статическом этапе.

Доступность компонента — этический и практический императив современной веб-разработки. Калькулятор должен быть удобен для всех пользователей.

Отзывчивость дизайна обеспечивает корректное отображение калькулятора на экранах различных размеров. Принцип mobile-first заставляет задуматься о том, как компонент будет выглядеть на мобильных устройствах, где пространство ограничено, а затем адаптировать его для планшетов и десктопов. Медиа-запросы становятся инструментом тонкой настройки, позволяя менять размеры шрифтов, отступы и расположение элементов в зависимости от контекста использования.

Типографика калькулятора заслуживает отдельного внимания. Шрифт дисплея должен быть моноширинным для четкого отображения чисел, достаточно крупным для комфортного чтения и иметь хорошую разборчивость цифр. История вычислений, если она предусмотрена дизайном, оформляется меньшим кеглем и контрастным цветом, чтобы визуально отделить ее от основного результата.

Визуальные детали — тени, градиенты, скругления углов — создают глубину и делают интерфейс приятным для восприятия. Однако важно соблюдать умеренность: избыток эффектов может отвлекать от основной функции калькулятора. Каждая визуальная характеристика должна служить цели улучшения пользовательского опыта, а не просто украшать интерфейс.

На этом этапе также закладывается основа для будущей JavaScript-логики. Структура классов и идентификаторов должна быть продумана таким образом, чтобы позже можно было легко добавить обработчики событий и манипулировать элементами. Хорошо организованная верстка значительно упрощает интеграцию функциональности и снижает вероятность возникновения ошибок на последующих этапах разработки.

В итоге верстка калькулятора — это процесс трансформации абстрактной

идеи в конкретный визуальный образ. Качество выполнения этой работы определяет не только внешний вид компонента, но и удобство его использования, легкость дальнейшей разработки и доступность для разнообразной аудитории. Правильно созданная статическая основа становится прочным фундаментом, на котором строится вся последующая функциональность интерактивного компонента.

Задания для выполнения лабораторной работы

Вариант 1

Создать минималистичный калькулятор только с основными функциями (цифры, 4 операции, равно, очистить).

Вариант 2

Сверстать научный калькулятор с тремя рядами кнопок: тригонометрия, логарифмы, степени, константы π и e.

Вариант 3

Дизайн банковского калькулятора с кнопками памяти (M+, M-, MR, MC), %, +/-. Строгий, деловой стиль.

Вариант 4

Калькулятор с HEX/DEC/OCT/BIN переключателями, кнопками AND/OR/XOR/NOT, битовыми сдвигами.

Вариант 5

Ретро-калькулятор 70-х с оранжевым дисплеем, металлическими кнопками.

Вариант 6

Интерфейс с двумя большими полями ввода, выпадающими списками валют, кнопкой обмена валют местами.

Вариант 7

Форма с полями: сумма кредита, срок, процент, и результатами: платеж, переплата, график.

Вариант 8

Калькулятор с кнопками скобок, расширенным дисплеем для отображения всего выражения.

Вариант 9

Неоновый калькулятор в темных тонах с эффектами на кнопках памяти.

Вариант 10

Яркий калькулятор с крупными кнопками, иконками вместо цифр, анимациями.

Вариант 11

Калькулятор для кухни с иконками продуктов, мерными стаканами, таймером приготовления.

Вариант 12

Голографический калькулятор с областью для графика.

Вариант 13

Калькулятор с областью для ввода набора чисел, кнопками статистических функций.

Вариант 14

Калькулятор как игровой контроллер, панелями кнопок, элементами прогресса и системой достижений в игровой цветовой гамме.

Вариант 15

Калькулятор как арт-объект в стиле поп-арта, конструктивизма, ар-деко или японского минимализма. Кнопки — уникальные графические элементы, дисплей как холст, акцент на эстетике.

Лабораторная работа № 18

Практическая работа: разработка компонента «Калькулятор» (этап 2 – логика)

Цель лабораторной работы: реализовать интерактивную логику компонента: обработка событий, управление состоянием, вычисления.

Методические указания

Переход от статической верстки к интерактивной логике представляет собой качественный скачок в разработке калькулятора. На этом этапе безжизненный интерфейс оживает, обретая способность реагировать на действия пользователя и выполнять математические операции. Логика калькулятора — это его "мозг", преобразующий нажатия кнопок в осмысленные вычисления.

Основой интерактивности становится обработка событий. Каждое нажатие кнопки должно быть корректно перехвачено, идентифицировано и преобразовано в конкретное действие. Событийная модель JavaScript позволяет связать физическое взаимодействие пользователя с интерфейсом и программную логику обработки этих взаимодействий. Ключевым моментом является правильное делегирование событий — вместо привязки отдельных обработчиков к каждой из десятков кнопок, создается единый обработчик, анализирующий цель события.

Управление состоянием калькулятора — центральная архитектурная задача. Состояние включает текущее значение на дисплее, историю операций, выбранный оператор, флаги ошибок и другую служебную информацию. Это состояние должно быть единым источником истины: любые изменения интерфейса происходят только в результате изменения состояния, а любые действия пользователя в первую очередь модифицируют это состояние. Такой подход обеспечивает предсказуемость и облегчает отладку.

Математическая логика требует тщательной обработки крайних случаев и ошибок. Деление на ноль, переполнение дисплея, последовательность операций — все эти сценарии должны быть предусмотрены и корректно обработаны. Особое внимание уделяется работе с плавающей точкой, где из-за особенностей представления чисел в компьютере могут возникать неточности, требующие специальной обработки.

Логика калькулятора строится вокруг конечного автомата, где система находится в одном из четко определенных состояний: ожидание первого числа, ввод первого числа, выбор оператора, ввод второго числа, вычисление результата. Переходы между состояниями происходят в ответ на конкретные действия пользователя, что делает поведение калькулятора предсказуемым и устойчивым к ошибочным вводам.

Обратная связь с пользователем становится важным аспектом интерактивности. Визуальное выделение активного оператора, анимация нажатия кнопок, звуковое подтверждение ввода — все это создает ощущение отзывчивости интерфейса. Особое внимание уделяется обработке ошибок: сообщения должны быть понятными и предлагать пути решения проблемы.

Тестирование логики — критически важный этап. Каждая операция, каждый крайний случай должны быть проверены на корректность работы. Автоматизированные тесты помогают убедиться, что изменения в одной части логики не нарушают работу других частей. Модульная архитектура позволяет тестировать отдельные функции изолированно, что повышает надежность всего компонента.

Интеграция логики с ранее созданной верстkой требует аккуратного подхода. JavaScript должен манипулировать существующими DOM-элементами, не нарушая их структуру и стили. Динамическое обновление дисплея, изменение состояний кнопок, отображение истории вычислений — все это происходит через управление уже созданными элементами интерфейса.

Производительность логики, хотя и не является первостепенной задачей для калькулятора, все же заслуживает внимания. Эффективная обработка событий, оптимизированные математические операции, минимальное количество перерисовок DOM — все это влияет на отзывчивость интерфейса, особенно на мобильных устройствах.

Создание логики калькулятора — это искусство баланса между функциональной полнотой и простотой реализации, между обработкой всех возможных сценариев и читаемостью кода, между мгновенным откликом и стабильностью работы. Качественно реализованная логика превращает набор

кнопок и дисплей в надежный инструмент для вычислений, готовый к использованию в реальных условиях.

Задания для выполнения лабораторной работы

Вариант 1

Реализовать идеальную базовую арифметику — цепочки вычислений, обработка деления на ноль, но без процентов и памяти.

Вариант 2

Реализовать полный набор научных функций с обработкой углов в градусах/радианах, сложными формулами.

Вариант 3

Реализовать финансовые функции: сложные проценты, аннуитетные платежи, расчет скидок/наценок через %.

Вариант 4

Реализовать конвертацию между системами счисления, битовые операции, работу с дополнением до двух.

Вариант 5

Реализовать логику как у оригинальных калькуляторов того времени — последовательные вычисления без приоритета операций.

Вариант 6

Реализовать конвертацию любых единиц измерения: валюта, длина, вес, температура с сохранением курсов.

Вариант 7

Реализовать финансовые формулы: аннуитетные/дифференцированные платежи, досрочное погашение.

Вариант 8

Реализовать парсер математических выражений с приоритетом операций, вложенными скобками.

Вариант 9

Реализовать расширенную систему памяти: несколько ячеек памяти, именованные переменные.

Вариант 10

Реализовать сохранение всего состояния в localStorage: история, настройки, последние расчеты.

Вариант 11

Реализовать конвертацию кулинарных мер: граммы/стаканы/ложки, пересчет рецептов.

Вариант 12

Реализовать построение графиков функций, возможность нажимать на график для получения координат.

Вариант 13

Реализовать статистические расчеты: среднее, дисперсия, корреляция, регрессия.

Вариант 14

Функции разблокируются постепенно: сложение/вычитание с начала, умножение после 10 операций, деление после 20, проценты после решения задач, скобки на высшем уровне.

Вариант 15

Каждое вычисление сопровождается стилизованной анимацией: цифры выплывают, операции визуализируются, результат плавно проявляется.

Лабораторная работа № 19

Практическая работа: разработка модуля «Корзина» (этап 1 – модель и представление)

Цель лабораторной работы: спроектировать структуру данных и создать статический интерфейс для отображения списка товаров и корзины.

Методические указания

Разработка модуля корзины начинается с проектирования фундаментальных структур данных, которые станут основой всей системы. Этот этап можно сравнить с архитектурным проектированием здания: прежде чем возводить стены, необходимо создать чертежи, рассчитать нагрузки, определить назначение каждого помещения. Модель данных корзины — это скелет, на который будет наращиваться функциональность.

Структура данных для корзины должна отражать все аспекты реального процесса покупок. Каждый товар представляет собой сложный объект с идентификатором, названием, ценой, количеством, возможными скидками и другими атрибутами. Корзина в целом — это не просто список товаров, а комплексная система, включающая промежуточные итоги, налоги, стоимость доставки, примененные промокоды. Важно продумать связи между этими сущностями: как товары связаны с категориями, как скидки применяются к конкретным позициям, как изменение количества влияет на общую стоимость.

Проектирование модели требует баланса между достаточностью и избыточностью. Модель должна содержать все необходимые данные для отображения и вычислений, но при этом оставаться понятной и управляемой. Часто применяется нормализация данных — разделение информации на логические группы: основные данные товаров, данные о наличии, цены, скидки, пользовательские предпочтения. От качества проектирования модели зависит не только текущая функциональность, но и возможность будущего расширения системы.

Создание статического интерфейса — это визуализация проектируемой модели. Интерфейс корзины должен демонстрировать все элементы системы: список товаров с их характеристиками, количество каждой позиции, стоимость,

итоговые суммы. Важным аспектом является визуальная иерархия: самые важные элементы (итоговая сумма, кнопка оформления заказа) должны быть наиболее заметны, второстепенные детали (скидки по каждой позиции) — доступны, но не доминируют.

При верстке интерфейса корзины особое внимание уделяется табличным представлениям данных. Таблицы позволяют компактно отображать множество атрибутов товаров, сохраняя при этом читаемость. Однако современный подход часто сочетает табличное представление с карточным, где каждый товар отображается как самостоятельный блок с изображением и основными характеристиками. Это особенно важно для мобильных устройств, где горизонтальное пространство ограничено.

Доступность интерфейса корзины критически важна, так как это финальный этап перед совершением покупки. Все элементы должны быть доступны с клавиатуры, иметь четкие текстовые описания, обеспечивать достаточный контраст для слабовидящих пользователей. Интерактивные элементы (кнопки изменения количества, удаления товаров) должны быть визуально различимы и иметь понятные состояния при наведении и фокусе.

Отзывчивость дизайна корзины обеспечивает комфортное использование на любых устройствах. На десктопах можно позволить себе расширенные таблицы с множеством колонок, на планшетах — компактное представление с основными данными, на мобильных — вертикальный список карточек товаров с акцентом на ключевую информацию. Адаптация происходит не только через изменение размеров, но и через перегруппировку элементов, изменение порядка отображения, показ или скрытие второстепенных деталей.

Визуальная обратная связь в статическом интерфейсе закладывается через состояние элементов. Даже без функциональности кнопки должны выглядеть кликабельными, поля ввода — доступными для редактирования, элементы выбора — изменяемыми. Цветовая схема помогает различать типы информации: цены выделяются одним цветом, количество — другим, предупреждения (например, о низком остатке товара) — контрастным акцентом.

На этом этапе также закладываются основы для анимаций и переходов,

которые оживят интерфейс на следующем этапе. Позиционирование элементов, размеры отступов — все это должно учитывать будущие анимации добавления и удаления товаров, изменения количества, пересчета итогов.

Ключевой принцип первого этапа — создание прочного, продуманного фундамента. Хорошо спроектированная модель и тщательно сверстанный интерфейс становятся основой, на которой будет построена вся логика работы корзины. Это инвестиция в будущее: время, потраченное на качественное проектирование, многократно окупится на этапе реализации функциональности и последующей поддержки системы.

Задания для выполнения лабораторной работы

Вариант 1

Создать интерфейс корзины для магазина электроники с товарами-карточками: изображение гаджета, название, характеристики (память, цвет, процессор), цена, выбор количества, иконки быстрой доставки и гарантии. Дизайн — минимализм.

Вариант 2

Создать корзину продуктового магазина с группировкой по категориям (овощи, молочка, мясо). Для каждого товара: вес/количество, цена за единицу/кг, срок годности, иконка акции. Дизайн яркий с изображениями продуктов.

Вариант 3

Создать корзину для интернет-магазина одежды с отображением товаров в виде "гардероба": фотомодели в одежде, выбор размера и цвета через выпадающие списки, иконки наличия размеров. Элегантный дизайн с фоновыми текстурами тканей.

Вариант 4

Создать корзину для книжного магазина с отображением книг: обложка, автор, жанр, краткое описание, рейтинг. Группировка по жанрам или сериям. Дизайн в стиле библиотеки — темное дерево, кремовый фон, шрифты с засечками.

Вариант 5

Создать корзину для цветочного магазина с товарами-букетами: большое фото букета, состав цветов, возможность добавления открытки, выбор даты доставки.

Романтичный дизайн с пастельными тонами.

Вариант 6

Создать корзину для магазина спортпита: товары с детальной информацией (белки/жиры/углеводы на порцию), выбор вкуса, фасовки. Динамичный дизайн с энергичными цветами, иконками спортивных достижений.

Вариант 7

Создать корзину для детского магазина с товарами: игрушки, одежда, питание. Яркий, "мультяшный" дизайн с закругленными формами, крупными кнопками, анимационными элементами. Указание возраста ребенка для рекомендаций.

Вариант 8

Создать корзину для строительного магазина с товарами: материалы, инструменты, сантехника. Утилитарный дизайн с "технической" эстетикой: схемы, чертежи, промышленные цвета. Указание единиц измерения (м², кг, шт).

Вариант 9

Создать корзину для ювелирного магазина: роскошный дизайн с золотыми/серебряными акцентами, анимацией бликов на драгоценностях. Для каждого изделия: фото в нескольких ракурсах, проба, вес, каратность камней.

Вариант 10

Создать корзину для магазина творчества и рукоделия: материалы для разных видов творчества (шитье, вязание, рисование). Дизайн — акварельные текстуры, рукописные шрифты, нежные пастельные тона.

Вариант 11

Создать корзину для интернет-аптеки: лекарства, БАДы, медицинские изделия. Чистый, стерильный дизайн с сине-белой палитрой, значками рецептурных препаратов, указанием формы выпуска и дозировки.

Вариант 12

Создать корзину для магазина косметики и парфюмерии: средства по уходу, декоративная косметика, парфюм. Гламурный дизайн с розово-золотой палитрой, глянцевыми поверхностями, фототоваров на моделях.

Вариант 13

Создать корзину для зоомагазина: товары для разных питомцев (кошки,

собаки, грызуны, птицы). Милый дизайн с иконками животных, пастельными тонами, разделением по видам питомцев. Указание возраста и породы животного.

Вариант 14

Создать корзину для мебельного магазина: товары с 3D-превью или фото в интерьере, выбор цвета и материала, габариты. Солидный дизайн с текстурой дерева, схемами сборки, визуализацией в комнате.

Вариант 15

Создать корзину для магазина музыкальных инструментов и оборудования: инструменты, усилители, аксессуары. Креативный дизайн с "музыкальной" тематикой: нотный стан, волны звука, темная тема с неоновыми акцентами.

Лабораторная работа № 20

Практическая работа: разработка модуля «Корзина» (этап 2 – контроллер)

Цель лабораторной работы: реализовать бизнес-логику модуля: добавление/удаление, пересчет итогов, реактивное обновление интерфейса.

Методические указания

Реализация контроллера для модуля корзины — это процесс оживления статического интерфейса, превращение набора HTML-элементов в интерактивную систему, реагирующую на каждое действие пользователя. Контроллер выступает посредником между моделью данных и представлением, обеспечивая их согласованное взаимодействие.

Бизнес-логика добавления и удаления товаров кажется простой только на поверхности. На самом деле это комплексная операция, затрагивающая множество аспектов: проверка наличия товара на складе, обновление счетчика в заголовке корзины, пересчет промежуточных итогов, применение актуальных скидок, обновление рекомендательной системы. Каждое действие должно быть атомарным — либо выполняется полностью, либо не выполняется вообще, что требует тщательной обработки ошибок и отката изменений при неудаче.

Система управления количеством товаров требует особого внимания. Увеличение и уменьшение количества — это не просто изменение числа в интерфейсе, а цепочка связанных операций: проверка максимально доступного количества, обновление стоимости позиции, пересчет общей суммы, возможно — изменение условий доставки или применение дополнительных скидок при достижении определенного порога. Важно предусмотреть все сценарии: что происходит, когда пользователь пытается добавить больше товара, чем есть в наличии; как обрабатывается уменьшение количества до нуля; как ведет себя система при быстром последовательном изменении количества.

Пересчет итогов — математически сложная задача, особенно в современных электронных коммерческих системах. Промежуточная сумма формируется не простым сложением цен, а с учетом множества факторов: скидки на конкретные товары, промокоды на всю корзину, налоги, стоимость доставки, которая может зависеть от суммы заказа. Эти расчеты должны выполняться точно, с правильным

округлением и в корректной последовательности (например, сначала применяются скидки на товары, затем промокод на корзину, затем добавляется доставка, затем начисляются налоги).

Реактивное обновление интерфейса — это искусство баланса между мгновенным откликом и производительностью. При каждом изменении в корзине должны обновляться десятки элементов: стоимость каждой позиции, промежуточные итоги по группам товаров, общая сумма, счетчик товаров, индикаторы скидок, условия доставки. Оптимизация этих обновлений достигается через дебаунсинг (группировку быстрых последовательных изменений), виртуализацию (обновление только видимых элементов при больших списках.

Обработка асинхронных операций становится важной частью контроллера. Проверка актуальности цен, подтверждение наличия товара, расчет стоимости доставки — все это требует обращения к серверу. Контроллер должен управлять этими запросами: показывать состояние загрузки, обрабатывать ошибки сети, кэшировать результаты, обеспечивать работу в оффлайн-режиме с последующей синхронизацией.

Система валидации предотвращает ошибки и улучшает пользовательский опыт. Проверка вводимых данных (например, количества товара), валидация промокодов, контроль целостности корзины при редактировании — все эти проверки выполняются как на клиенте для мгновенной обратной связи, так и на сервере для гарантии безопасности. Сообщения об ошибках должны быть понятными и предлагать конкретные действия для исправления.

Управление состоянием корзины требует реализации сложного состояния с множеством взаимосвязанных полей. Используются паттерны вроде конечного автомата, где корзина может находиться в разных состояниях: пустая, с товарами, в процессе оформления, заблокированная для изменений. Переходы между состояниями четко определены и защищены от неправильных последовательностей действий.

Интеграция с другими системами расширяет функциональность контроллера. Корзина не существует изолированно — она взаимодействует с

системой рекомендаций (предлагает сопутствующие товары), системой лояльности (применяет накопленные бонусы), системой аналитики (отслеживает поведение пользователя). Контроллер координирует эти взаимодействия, обеспечивая целостность данных и последовательность операций.

Тестирование контроллера — многоуровневый процесс. Модульные тесты проверяют отдельные функции (расчет скидок, обновление количества). Интеграционные тесты — взаимодействие между компонентами (добавление товара → обновление итогов → сохранение в localStorage).

Производительность контроллера оптимизируется через различные техники: ленивая загрузка данных, пагинация или бесконечный скролл для больших корзин, оптимизированные алгоритмы пересчета. Особое внимание уделяется мобильным устройствам, где ресурсы процессора и памяти ограничены.

Контроллер должен быть не просто функциональным, но и предсказуемым, отзывчивым, терпимым к ошибкам — настоящим цифровым помощником в процессе покупки.

Задания для выполнения лабораторной работы

Вариант 1

Реализовать логику с учетом особенностей электроники: проверка совместимости товаров (например, наушники к телефону), автодобавление сопутствующих товаров (чехол к телефону), расчет скидки за комплект, обновление итогов с учетом доставки крупногабаритных товаров.

Вариант 2

Реализовать логику с учетом весовых товаров: пересчет цены при изменении веса, проверка минимального веса, округление стоимости, система "3 по цене 2" для акционных товаров, автоматическое добавление рекомендаций (сыр к вину).

Вариант 3

Реализовать логику с учетом размерной сетки: проверка доступности выбранного размера, предложение альтернативных размеров, расчет скидки при покупке комплекта, обновление итогов с учетом промокодов на определенные категории.

Вариант 4

Реализовать логику с учетом специфики книг: проверка электронной/бумажной версии, система рекомендаций "купили вместе с этой книгой", накопление бонусов за покупку книг определенного автора, расчет доставки с учетом веса книг.

Вариант 5

Реализовать логику с учетом сезонности и свежести: проверка доступности цветов на выбранную дату, расчет стоимости доставки "к определенному часу", система сборки букета на заказ, автоматическое добавление ухода за цветами.

Вариант 6

Реализовать логику с учетом программ питания: проверка совместимости добавок, расчет стоимости программы на месяц, система накопления баллов за регулярные покупки, рекомендации по циклу тренировок.

Вариант 7

Реализовать логику с учетом возрастных ограничений: проверка возраста для игрушек (мелкие детали), рекомендации "растем вместе" (одежда на вырост), система подарочной упаковки, расчет скидки на второй товар из той же категории.

Вариант 8

Реализовать логику с учетом строительных норм: расчет необходимого количества материалов по площади, проверка совместимости материалов, система "заказать недостающее", расчет доставки с учетом габаритов и веса.

Вариант 9

Реализовать логику с учетом высокой стоимости: многоуровневая проверка данных, система резервирования товара, расчет страховки доставки, индивидуальные скидки для постоянных клиентов, градация скидки в зависимости от суммы чека.

Вариант 10

Реализовать логику с учетом комплектности: проверка наличия всех материалов для набора, расчет стоимости набора поштучно и целиком, система "докупить недостающее", рекомендации инструментов к материалам.

Вариант 11

Реализовать логику с учетом медицинских ограничений: проверка совместимости лекарств (предупреждение о несочетаемости), ограничение количества рецептурных препаратов, система напоминания о пополнении домашней аптечки.

Вариант 12

Реализовать логику с учетом косметических линий: проверка совместимости средств одной линии, система пробников к основному заказу, персонализированные рекомендации по типу кожи/волос, накопление баллов в программе лояльности.

Вариант 13

Реализовать логику с учетом потребностей животных: проверка корма по возрасту и породе, расчет количества корма на месяц, система автозаказа регулярных товаров, рекомендации аксессуаров к основному товару.

Вариант 14

Реализовать логику с учетом мебельных гарнитуров: проверка совместимости предметов коллекции, расчет стоимости гарнитура целиком, проверка габаритов для доставки, система планирования интерьера.

Вариант 15

Реализовать логику с учетом музыкальных комплектов: проверка совместимости инструментов и оборудования (разъемы, мощность), система сборки домашней студии под бюджет, расчет рассрочки на дорогие инструменты, рекомендации педагогов к инструментам.

Лабораторная работа № 21

Индивидуальный проект: определение темы и требований

Цель лабораторной работы: выбрать предметную область для одностраничного приложения (SPA), сформулировать функциональные требования и описание структуры данных.

Методические указания

Выбор предметной области для одностраничного приложения (SPA) — это стратегическое решение, определяющее всю последующую работу. Этот процесс требует баланса между амбициями и реалистичностью. Хорошая тема должна быть достаточно интересной, чтобы мотивировать разработчика на протяжении всего проекта, и достаточно практичной, чтобы демонстрировать реальные навыки программирования. Тема должна позволять реализовать основные паттерны веб-разработки: работу с данными, состояние приложения, маршрутизацию, обработку событий, взаимодействие с пользователем.

Формулирование функциональных требований — это искусство декомпозиции. Каждое требование должно описывать конкретную возможность, которую получит пользователь. Требования формулируются с позиции пользователя: "пользователь может добавлять задачи в список дел", "пользователь может отмечать задачи как выполненные", "пользователь может фильтровать задачи по статусу". Важно разделить требования на обязательные (без которых приложение не имеет смысла) и опциональные (улучшающие опыт, но не являющиеся критичными). Составление User Stories (пользовательских историй) помогает увидеть приложение глазами конечного пользователя.

Описание структуры данных — это проектирование информационной архитектуры приложения. На этом этапе определяются основные сущности (например, для приложения управления задачами: Task, Project, User, Category) и их взаимосвязи. Для каждой сущности описываются атрибуты: название, тип данных, обязательность, значения по умолчанию, ограничения. Создается схема данных, показывающая, как сущности связаны между собой (один-ко-многим, многие-ко-многим). Эта схема станет основой для моделей данных в JavaScript и структуры JSON-файлов для хранения данных.

Задания для выполнения лабораторной работы

Вариант 1

Планировщик задач с категориями и тегами. Функции: создание задач с заголовком, описанием, сроком выполнения, приоритетом (низкий/средний/высокий), категориями (работа, личное, здоровье) и тегами. Фильтрация задач по приоритету, категории, тегам, статусу выполнения. Поиск по тексту. Статистика: выполненные/невыполненные задачи по категориям. Структура данных: задачи с полями id, title, description, priority, category, tags[], dueDate, completed, createdAt.

Вариант 2

Трекер личных финансов. Функции: добавление доходов/расходов с категориями (еда, транспорт, развлечения), датой, суммой, описанием. Установка месячного бюджета по категориям. Статистика: диаграммы распределения расходов, сравнение с бюджетом. Структура данных: операции с полями id, type (income/expense), amount, category, date, description.

Вариант 3

Кулинарная книга с рецептами. Функции: добавление рецептов с ингредиентами, шагами приготовления, временем, сложностью, тегами (вегетарианское, острое и т.д.). Поиск рецептов по ингредиентам, тегам, времени. Создание списка покупок для выбранных рецептов. Структура данных: рецепты с полями id, title, ingredients[], steps[], time, difficulty, tags[], image.

Вариант 4

Дневник тренировок. Функции: создание тренировок с упражнениями, подходами, весом, повторениями. Отслеживание прогресса по упражнениям. Календарь тренировок. Статистика: прогресс по весам, объем работы. Структура данных: тренировки с полями id, date, exercises[] (name, sets[{weight, reps}]).

Вариант 5

Коллекция просмотренных фильмов. Функции: добавление фильмов с оценкой, жанрами, датой просмотра, рецензией. Фильтрация по жанру, оценке, году. Статистика: распределение по жанрам, средняя оценка. Рекомендации на основе похожих фильмов. Структура данных: фильмы с полями id, title, year, genres[], rating,

review, watchedDate.

Вариант 6

Персональная библиотека прочитанных книг. Функции: добавление книг с автором, жанром, датой прочтения, рецензией, оценкой (1-10), статусом (прочитана/читаю/в планах). Фильтрация по жанру, автору, статусу, оценке. Статистика: количество прочитанных книг по месяцам, распределение по жанрам, средняя оценка. Цитаты из книг с возможностью сохранения. Структура данных: книги с полями id, title, author, genres[], year, pages, status, rating, review, quotes[], readDate, addedDate.

Вариант 7

Планировщик поездок и путешествий. Функции: создание поездок с маршрутами, датами, бюджетом, списком достопримечательностей. Разделение расходов по категориям (транспорт, жилье, питание, развлечения). Интеграция карты для визуализации маршрута. Список вещей для сбора. Погода для дат поездки. Структура данных: поездки с полями id, title, destination, dates {start, end}, budget, categories {transport, accommodation, food, activities}, itinerary[], packingList[], notes.

Вариант 8

Трекер ухода за комнатными растениями. Функции: добавление растений с названием, видом, датой приобретения, фото. Расписание ухода: полив, опрыскивание, удобрение, пересадка. Напоминания о необходимости ухода. Журнал ухода с записями. Диагностика проблем по симптомам. Структура данных: растения с полями id, name, species, location, purchaseDate, careSchedule {watering, misting, fertilizing, repotting}, careLog[], photos[], healthStatus.

Вариант 9

Интерактивный словарь для изучения иностранных языков. Функции: добавление слов с переводом, примером использования, тегами, произношением. Группировка по темам и уровням сложности. Карточки для запоминания с интервальным повторением. Тесты на проверку знаний. Статистика прогресса. Структура данных: слова с полями id, word, translation, example, phonetic, tags[], difficulty, addedDate, reviewDates[], score.

Вариант 10

Каталог настольных игр с подбором. Функции: добавление игр с описанием, количеством игроков, временем игры, сложностью, механиками. Подбор игр по параметрам (количество игроков, доступное время, сложность). Журнал сыгранных партий с результатами. Рейтинг игр. Структура данных: игры с полями id, title, minPlayers, maxPlayers, minTime, maxTime, complexity, mechanics[], description, plays[], rating.

Вариант 11

Трекер настроения и эмоционального состояния. Функции: ежедневные записи настроения по шкале, эмоций, активности, сна, факторов влияния. Анализ корреляций между факторами и настроением. Статистика по периодам. Напоминания о записях. Экспорт данных. Структура данных: записи с полями id, date, moodScore, emotions[], activities[], sleepHours, factors[], notes.

Вариант 12

Каталог виниловых пластинок. Функции: добавление пластинок с информацией об альбоме, исполнителе, годе выпуска, состоянии, цене покупки. Организация по жанрам и исполнителям. Отслеживание прослушивания. Желаемый список (wantlist). Оценка стоимости коллекции. Структура данных: пластинки с полями id, album, artist, year, genre, condition, purchasePrice, purchaseDate, listenCount, lastListened, notes.

Вариант 13

Планировщик регулярной уборки дома. Функции: разделение дома на зоны (комнаты), создание задач уборки для каждой зоны с частотой (ежедневно/еженедельно/ежемесячно). Отметка выполненных задач. Напоминания о необходимости уборки. Ротация сложных задач. Статистика чистоты. Структура данных: зоны с полями id, name, tasks[] {task, frequency, lastCompleted, nextDue}, cleaningHistory[].

Вариант 14

Трекер формирования привычек. Функции: создание привычек с целями, напоминаниями, периодичностью. Отслеживание выполнения по дням. Статистика выполнения (серии, успешность). Мотивационные уведомления. Визуализация прогресса. Структура данных: привычки с полями id, name, description, frequency,

goal, reminders[], streak, completionHistory[].

Вариант 15

Простая CRM для учета клиентов и взаимодействий. Функции: добавление контактов с информацией, история взаимодействий (звонки, встречи, emails), напоминания о следующих контактах. Статусы контактов (новый/в работе/закрыт). Поиск и фильтрация контактов. Экспорт данных. Структура данных: контакты с полями id, name, company, email, phone, status, interactions[], nextContactDate, notes.

Лабораторная работа № 22

Индивидуальный проект: проектирование состояния приложения

Цель лабораторной работы: определить модели данных и переменные состояния, необходимых для реализации логики приложения.

Методические указания

Модели данных в JavaScript — это конструкторы или классы, определяющие структуру объектов. Например, для задачи (Task) модель определяет, что у каждой задачи есть id, title, description, completed, createdAt. Модели обеспечивают целостность данных: при создании объекта проверяются типы, устанавливаются значения по умолчанию. Модели также могут содержать методы валидации, сериализации/десериализации.

Переменные состояния — это данные, которые меняются в процессе работы приложения. Их можно классифицировать:

- **Данные приложения** — основная информация (список задач, пользователи, проекты)
- **Состояние UI** — что сейчас отображается (текущая страница, выбранный элемент, открытые модальные окна)
- **Состояние загрузки** — индикаторы загрузки, флаги выполнения операций
- **Ошибки** — сообщения об ошибках, их тип, статус
- **Фильтры и сортировки** — текущие настройки отображения данных

Проектирование управления состоянием определяет, как состояние изменяется и распространяется по приложению. Даже в небольших SPA важно продумать централизованное хранилище состояния (store), которое содержит все данные приложения и предоставляет методы для их изменения. Это упрощает отслеживание изменений, отладку и синхронизацию разных частей интерфейса. Для управления состоянием можно использовать паттерны Flux или Redux, адаптированные к масштабу проекта.

Задания для выполнения лабораторной работы

Вариант 1

Состояние приложения: текущий список задач (массив объектов), выбранные фильтры (приоритет, категория, теги, статус), строка поиска, статистика задач.

Переменные: tasks, filters, searchQuery, statistics, selectedTaskId для редактирования. Модель Task: id, title, description, priority, category, tags, dueDate, completed, createdAt.

Вариант 2

Состояние: список операций, месячный бюджет по категориям, выбранный период (месяц), статистика за период. Переменные: transactions, budget, selectedMonth, summary. Модель Transaction: id, type, amount, category, date, description.

Вариант 3

Состояние: список рецептов, выбранные рецепты для списка покупок, текущие фильтры поиска. Переменные: recipes, selectedRecipes, shoppingList, filters. Модель Recipe: id, title, ingredients (массив объектов с name, amount, unit), steps, time, difficulty, tags, image.

Вариант 4

Состояние: история тренировок, текущая тренировка, прогресс по упражнениям, календарь занятий. Переменные: workouts, currentWorkout, progress, calendar. Модель Workout: id, date, exercises.

Вариант 5

Состояние: коллекция фильмов, фильтры отображения, статистика, рекомендации. Переменные: movies, filters, stats, recommendations. Модель Movie: id, title, year, genres, rating, review, watchedDate.

Вариант 6

Состояние приложения: коллекция книг, текущие фильтры, статистика чтения, выбранная книга для детального просмотра, режим отображения (сетка/список). Переменные: books, filters, stats, selectedBookId, viewMode. Модель Book: id, title, author, genres, year, pages, status, rating, review, quotes, readDate, addedDate.

Вариант 7

Состояние: список поездок, текущая выбранная поездка, активный маршрут на карте, расчет бюджета. Переменные: trips, currentTripId, mapRoute, budgetSummary. Модель Trip: id, title, destination, dates, budget, categories, itinerary (массив точек маршрута), packingList,

notes.

Вариант 8

Состояние: коллекция растений, текущие задачи по уходу, журнал действий, напоминания. Переменные: plants, careTasks, careLog, reminders. Модель Plant: id, name, species, location, purchaseDate, careSchedule, careLog (массив записей с date, action, notes), photos, healthStatus.

Вариант 9

Состояние: словарь слов, текущий режим (просмотр/обучение/тест), прогресс изучения, настройки тестирования. Переменные: words, studyMode, progress, testSettings. Модель Word: id, word, translation, example, phonetic, tags, difficulty, addedDate, reviewDates, score.

Вариант 10

Состояние: коллекция игр, история игр, текущие параметры подбора, рейтинги. Переменные: games, playHistory, selectionParams, ratings. Модель Game: id, title, minPlayers, maxPlayers, minTime, maxTime, complexity, mechanics, description, plays (массив с date, players, duration, winner), rating.

Вариант 11

Состояние: история записей, текущая запись, статистика настроения, выбранный период анализа. Переменные: entries, currentEntry, moodStats, analysisPeriod. Модель MoodEntry: id, date, moodScore (1-10), emotions, activities, sleepHours, factors, notes.

Вариант 12

Состояние: коллекция пластинок, желаемый список, статистика коллекции, текущие фильтры. Переменные: records, wantlist, collectionStats, filters. Модель Record: id, album, artist, year, genre, condition (1-5), purchasePrice, purchaseDate, listenCount, lastListened, notes.

Вариант 13

Состояние: зоны уборки, текущие задачи, история уборки, настройки напоминаний. Переменные: zones, currentTasks, cleaningHistory, reminderSettings. Модель CleaningZone: id, name, tasks (массив объектов с task, frequency, lastCompleted, nextDue), cleaningHistory.

Вариант 14

Состояние: список привычек, текущие серии, статистика выполнения, календарь трекинга. Переменные: habits, currentStreaks, completionStats, trackingCalendar. Модель Habit: id, name, description, frequency (daily/weekly), goal, reminders, streak, completionHistory (массив дат выполнения).

Вариант 15

Состояние: список контактов, история взаимодействий, напоминания, текущие фильтры. Переменные: contacts, interactions, reminders, filters. Модель Contact: id, name, company, email, phone, status, interactions (массив объектов с type, date, notes), nextContactDate, notes.

Лабораторная работа № 23

Индивидуальный проект: разработка бизнес-логики (Model)

Цель лабораторной работы: создать набор чистых JavaScript-функций, реализующих ключевые операции предметной области.

Методические указания

Чистые функции — основа предсказуемой бизнес-логики. Функция считается чистой, если:

1. Для одних и тех же аргументов всегда возвращает одинаковый результат
2. Не имеет побочных эффектов (не изменяет внешние переменные, не делает запросов к серверу)
3. Не зависит от внешнего состояния (глобальных переменных, времени)

Пример чистой функции для управления задачами:

```
function addTask(tasks, newTask) {
    return [...tasks, { ...newTask, id: Date.now(), createdAt: new Date() }];
}

function toggleTaskCompletion(tasks, taskId) {
    return tasks.map(task =>
        task.id === taskId ? { ...task, completed: !task.completed } : task
    );
}
```

Модульная организация бизнес-логики предполагает разделение кода по функциональным областям. Например:

- **`taskManager.js`** — операции с задачами
- **`projectManager.js`** — управление проектами
- **`filterManager.js`** — фильтрация и сортировка
- **`validation.js`** — валидация данных

Каждый модуль экспортирует только необходимые функции, скрывая детали реализации. Это упрощает тестирование и позволяет заменять реализации модулей без изменения остального кода.

Задания для выполнения лабораторной работы

Вариант 1

Функции: addTask(taskData), updateTask(taskId, updates), deleteTask(taskId), toggleTaskCompletion(taskId), filterTasks(tasks, filters, searchQuery), getStatistics(tasks), sortTasks(tasks, sortBy) (по дате, приоритету). Чистые функции для работы с тегами: добавление/удаление тегов у задач.

Вариант 2

Функции: addTransaction(transaction), calculateSummary(transactions, budget, month), getCategoryStats(transactions, category), checkBudgetExceedance(transactions, budget). Чистые функции для расчета баланса, прогноза на месяц.

Вариант 3

Функции: addRecipe(recipeData), searchRecipes(recipes, filters), generateShoppingList(selectedRecipes), filterByIngredients(recipes, availableIngredients). Чистые функции для объединения ингредиентов в списке покупок.

Вариант 4

Функции: addWorkout(workout), calculateProgress(workouts, exerciseName), getPersonalRecords(workouts), calculateVolume(workout). Чистые функции для анализа прогресса.

Вариант 5

Функции: addMovie(movieData), filterMovies(movies, filters), calculateStatistics(movies), getRecommendations(movies, currentMovie). Чистые функции для поиска похожих фильмов по жанрам.

Вариант 6

Функции: addBook(bookData), updateBookStatus(bookId, newStatus), filterBooks(books, filters), calculateReadingStats(books), getBooksByAuthor(books, author), addQuoteToBook(bookId, quote). Чистые функции для анализа жанровых предпочтений и темпа чтения.

Вариант 7

Функции: createTrip(tripData), calculateBudget(trip), addToItinerary(tripId,

point), generatePackingList(destination, dates, activities), calculateDaysRemaining(trip). Чистые функции для оптимизации маршрута и распределения бюджета.

Вариант 8

Функции: addPlant(plantData), calculateNextCareDate(plant, actionType), logCareAction(plantId, action, notes), checkHealthStatus(plant, symptoms), getPlantsNeedingCare(plants, today). Чистые функции для определения частоты ухода по виду растения.

Вариант 9

Функции: addWord(wordData), getWordsForReview(words, date), calculateProgress(words), generateTest(words, settings), updateWordScore(wordId, correct). Чистые функции для алгоритма интервального повторения (Spaced Repetition System).

Вариант 10

Функции: addGame(gameData), findGames(games, params), logPlay(gameId, playData), calculateGameStats(game), updateRating(gameId, newRating). Чистые функции для подбора игр по критериям.

Вариант 11

Функции: addEntry(entryData), calculateMoodStats(entries, period), findCorrelations(entries), detectPatterns(entries), generateInsights(entries). Чистые функции для статистического анализа корреляций.

Вариант 12

Функции: addRecord(recordData), logListening(recordId), calculateCollectionValue(records), getCollectionStats(records), findSimilarRecords(records, currentRecord). Чистые функции для анализа жанрового состава и стоимости.

Вариант 13

Функции: addZone(zoneData), markTaskCompleted(zoneId, taskIndex), calculateNextDueDate(frequency, lastCompleted), getOverdueTasks(zones), rotateTasks(zones). Чистые функции для расчета графиков уборки.

Вариант 14

Функции: addHabit(habitData), markHabitCompleted(habitId,

date), calculateStreak(completionHistory), getHabitsForToday(habits), calculateSuccessRate(habit). Чистые функции для анализа серий выполнения.

Вариант 15

Функции: addContact(contactData), addInteraction(contactId, interaction), updateContactStatus(contactId, newStatus), getContactsDueForFollowup(contacts), searchContacts(contacts, query). Чистые функции для анализа активности контактов.

Лабораторная работа № 24

Индивидуальный проект: создание пользовательского интерфейса (View)

Цель лабораторной работы: реализовать верстку основных экранов приложения, навигацию между ними без перезагрузки страницы.

Методические указания

Верстка основных экранов — это создание HTML-структуры для каждого значимого состояния приложения. Современный подход предполагает использование компонентов — независимых, переиспользуемых частей интерфейса. Каждый компонент отвечает за свой фрагмент UI и может иметь собственное состояние.

Навигация в SPA реализуется через систему маршрутизации (routing). Основные концепции:

- Маршруты (routes) — сопоставление URL с компонентами
- Параметры маршрута — динамические части URL (например, /tasks/:id)
- История навигации — возможность перехода вперед/назад
- Защита маршрутов — ограничение доступа к определенным страницам

Пример простой маршрутизации:

```
const routes = {
    '/': HomeComponent,
    '/tasks': TasksComponent,
    '/tasks/:id': TaskDetailComponent,
    '/projects': ProjectsComponent
};

function navigate(path) {
    // Разбор URL, поиск компонента, рендеринг
}
```

Компоненты общаются через свойства (props) и события (events). Родительский компонент передает данные дочернему через свойства, дочерний компонент сообщает о своих действиях через события. Это создает однонаправленный поток данных, упрощающий отладку.

Задания для выполнения лабораторной работы

Вариант 1

Экраны: главная (список задач с фильтрами), создание/редактирование задачи, статистика. Навигация без перезагрузки. Компоненты: TaskList, TaskItem, TaskForm, FilterPanel, StatisticsChart. Верстка с разделением на колонки: фильтры слева, список задач справа.

Вариант 2

Экраны: журнал операций, добавление операции, статистика, настройки бюджета. Компоненты: TransactionList, TransactionForm, BudgetChart, CategoryPieChart. Верстка с дашбордом: сводка сверху, список операций ниже.

Вариант 3

Экраны: каталог рецептов, страница рецепта, создание рецепта, список покупок. Компоненты: RecipeCard, RecipeDetail, RecipeForm, ShoppingList. Галерея рецептов с карточками.

Вариант 4

Экраны: календарь тренировок, страница тренировки, статистика прогресса. Компоненты: CalendarView, WorkoutForm, ExerciseProgressChart. Таблица с историей подходов.

Вариант 5

Экраны: коллекция фильмов, добавление фильма, детальная страница, статистика. Компоненты: MovieGrid, MovieCard, MovieForm, StatsDashboard. Постеры фильмов в сетке.

Вариант 6

Экраны: основная библиотека, страница книги, форма добавления книги, статистика. Компоненты: BookShelf (полки по статусу), BookCard, BookDetail, BookForm, ReadingStatsChart. Анимированные переходы между статусами книг.

Вариант 7

Экраны: список поездок, детали поездки, планирование маршрута, упаковка, бюджет. Компоненты: TripCard, TripPlanner, BudgetCalculator, PackingList, RouteMap (использует Leaflet/OpenStreetMap). Интерактивная карта с точками маршрута.

Вариант 8

Экраны: коллекция растений, страница растения, календарь ухода, журнал, диагностика. Компоненты: PlantGrid, PlantCard, CareCalendar, CareLog, HealthDiagnosis. Вид по локациям (комнатам).

Вариант 9

Экраны: словарь, добавление слов, режим обучения, тесты, статистика. Компоненты: DictionaryList, WordCard, StudySession, TestGenerator, ProgressChart. Анимированные карточки для запоминания.

Вариант 10

Экраны: каталог игр, подбор игр, страница игры, журнал партий. Компоненты: GameCollection, GameFinder, GameCard, PlayLogger, StatsDashboard. Интерактивный подбор с ползунками параметров.

Вариант 11

Экраны: календарь настроения, форма записи, статистика, анализ корреляций. Компоненты: MoodCalendar, EntryForm, MoodChart, CorrelationMatrix, InsightsPanel. Визуализация настроения цветами на календаре.

Вариант 12

Экраны: основная коллекция, желаемый список, статистика, добавление пластинки. Компоненты: RecordShelf, RecordCard, Wantlist, CollectionStats, AddRecordForm. Визуализация состояния пластинок.

Вариант 13

Экраны: план дома с зонами, список задач, история уборки, настройки. Компоненты: HousePlan, ZoneCard, TaskList, CleaningCalendar, SettingsPanel. Интерактивная схема дома.

Вариант 14

Экраны: список привычек, календарь выполнения, статистика, создание привычки. Компоненты: HabitsList, HabitCard, StreakCalendar, StatsDashboard, HabitForm. Визуализация серий в календаре.

Вариант 15

Экраны: список контактов, карточка контакта, календарь взаимодействий, добавление контакта. Компоненты: ContactsTable, ContactCard, InteractionTimeline, ReminderList, ContactForm. Таблица с сортировкой и фильтрацией.

Лабораторная работа № 25

Индивидуальный проект: интеграция логики и интерфейса (Controller)

Цель лабораторной работы: написать обработчики событий, который связывают пользовательский ввод с бизнес-логикой и обновляют UI.

Методические указания

Обработчики событий — это функции, которые связывают действия пользователя с бизнес-логикой. Основные типы событий:

- **События мыши** — click, dblclick, mousedown, mouseup
- **События клавиатуры** — keydown, keyup, keypress
- **События форм** — submit, change, input, focus, blur
- **События навигации** — popstate (изменение истории браузера)

Паттерн обработки событий:

```
class TasksController {
    constructor(model, view) {
        this.model = model;
        this.view = view;

        // Привязка обработчиков
        this.view.onAddTask = this.handleAddTask.bind(this);
        this.view.onToggleTask                                    =
this.handleToggleTask.bind(this);
        this.view.onDeleteTask                                    =
this.handleDeleteTask.bind(this);
    }

    handleAddTask(taskData) {
        // Валидация данных
        if (!this.validateTask(taskData)) return;

        // Вызов бизнес-логики
        const updatedTasks = this.model.addTask(taskData);

        // Обновление представления
        this.view.renderTasks(updatedTasks);
```

```
            // Сохранение состояния
            this.saveState(updatedTasks);
        }
    }
```

Обновление UI должно быть эффективным. Вместо полной перерисовки всего интерфейса обновляются только изменившиеся части. Для этого можно использовать:

- **Virtual DOM** — создание виртуального представления DOM и сравнение с предыдущим
- **Реактивные системы** — автоматическое обновление при изменении данных
- **Инкрементальное обновление** — явное указание, какие элементы нужно обновить

Задания для выполнения лабораторной работы

Вариант 1

Обработчики: добавление задачи через форму, изменение статуса по клику, удаление по кнопке, применение фильтров, поиск. Обновление UI при изменении задач: перерисовка списка, обновление статистики, сброс фильтров при очистке.

Вариант 2

Обработчики: добавление операции через форму, удаление операции, изменение бюджета. Реактивное обновление: пересчет статистики при добавлении операции, обновление диаграмм.

Вариант 3

Обработчики: добавление рецепта, выбор рецептов для списка покупок, применение фильтров. Обновление списка покупок при изменении выбора.

Вариант 4

Обработчики: добавление тренировки, ввод подходов, выбор даты в календаре. Обновление графиков прогресса при добавлении тренировки.

Вариант 5

Обработчики: добавление фильма, оценка, применение фильтров. Обновление статистики при добавлении фильма.

Вариант 6

Обработчики: добавление книги через форму, изменение статуса перетаскиванием на виртуальную полку, добавление цитат, фильтрация. Обновление статистики в реальном времени при изменении статуса книги.

Вариант 7

Обработчики: создание поездки, добавление точек маршрута на карте, ввод расходов, отметка собранных вещей. Обновление бюджета при добавлении расходов, пересчет дней до поездки.

Вариант 8

Обработчики: добавление растения, отметка выполненных действий ухода, добавление фото, запись в журнал. Обновление календаря ухода при добавлении растений.

Вариант 9

Обработчики: добавление слов, ответы в тестах, оценка сложности слов, настройка параметров обучения. Обновление прогресса при каждом тестировании.

Вариант 10

Обработчики: добавление игры, подбор по параметрам, запись сыгранной партии, оценка игры. Обновление рекомендаций при изменении параметров подбора.

Вариант 11

Обработчики: создание записи, выбор эмоций и активностей, просмотр статистики за период. Обновление графиков при добавлении новых записей.

Вариант 12

Обработчики: добавление пластинки, отметка прослушивания, перемещение в желаемый список, фильтрация коллекции. Обновление статистики при любых изменениях.

Вариант 13

Обработчики: создание зон, отметка выполненных задач, настройка частоты, просмотр истории. Обновление статуса задач при изменении дат.

Вариант 14

Обработчики: создание привычки, отметка выполнения, просмотр статистики, настройка напоминаний. Обновление серий и статистики при каждой отметке.

Вариант 15

Обработчики: добавление контакта, запись взаимодействия, изменение статуса, настройка напоминаний. Обновление списка напоминаний при добавлении взаимодействий.

Лабораторная работа № 26

Индивидуальный проект: загрузка и инициализация данных

Цель лабораторной работы: организовать работу с данными: загрузка начального состояния из внешнего источника (JSON) или их программная генерация.

Методические указания

Стратегии загрузки данных зависят от источника:

- Локальные JSON-файлы — fetch('./data/tasks.json')
- Моковые данные — предопределенные массивы объектов
- Программная генерация — функции, создающие тестовые данные
- Удаленные API — асинхронные запросы к серверу

Инициализация приложения включает несколько этапов:

1. **Загрузка начальных данных** — получение данных из выбранного источника
2. **Создание начального состояния** — преобразование сырых данных в модели
3. **Восстановление состояния** — проверка localStorage на наличие сохраненных данных
4. **Настройка UI** — применение сохраненных настроек интерфейса
5. **Навигация** — определение начального маршрута

Обработка ошибок загрузки данных критически важна. Приложение должно корректно работать даже если данные не загрузились:

- Показать понятное сообщение об ошибке
- Предложить повторить загрузку
- Использовать данные по умолчанию
- Работать в ограниченном режиме

Задания для выполнения лабораторной работы

Вариант 1

Источник: tasks.json с начальными задачами. Формат: массив объектов с полями модели. Инициализация: загрузка из JSON, парсинг, создание начальных тегов из существующих задач. Генерация демо-данных при отсутствии файла.

Вариант 2

Источник: transactions.json с историей операций. Генерация демо-данных за последние 3 месяца. Инициализация бюджета из budget.json или значений по умолчанию.

Вариант 3

Источник: recipes.json с коллекцией рецептов. Инициализация базовых рецептов. Генерация изображений-заглушек при отсутствии.

Вариант 4

Источник: workouts.json с демо-историей тренировок. Генерация календаря на текущий месяц.

Вариант 5

Источник: movies.json с начальной коллекцией. Загрузка данных о фильмах из открытого API (опционально). Генерация постеров-заглушек.

Вариант 6

Источник: books.json с начальной коллекцией. Интеграция с OpenLibrary API для автоматического заполнения данных по ISBN (опционально). Генерация обложек-заглушек для книг без изображений.

Вариант 7

Источник: trips.json с примером поездок. Интеграция с погодным API для прогноза на даты поездки. Загрузка данных о достопримечательностях из открытых источников.

Вариант 8

Источник: plants.json с базой видов растений и рекомендуемым уходом. my_plants.json с пользовательской коллекцией. Генерация начальных напоминаний на основе вида растений.

Вариант 9

Источник: words.json с начальным словарем по выбранному языку. Интеграция с Dictionary API для автоматического получения переводов и примеров. Генерация тестовых данных для отладки.

Вариант 10

Источник: games.json с базой популярных настольных игр. collection.json с

пользовательской коллекцией. Загрузка данных о механиках игр из открытых источников.

Вариант 11

Источник: mood_entries.json с демо-историей за несколько месяцев. Генерация начальных данных для отображения возможностей анализа. Прелоад списков эмоций и активностей.

Вариант 12

Источник: records.json с начальной коллекцией. Интеграция с Discogs API для автоматического получения данных по штрих-коду (опционально). Генерация обложек-заглушек.

Вариант 13

Источник: house_zones.json с примером планировки. Генерация стандартных задач уборки для разных типов комнат. Начальный график уборки на месяц вперед.

Вариант 14

Источник: habits.json с примером трекера. Генерация демо-данных выполнения за последний месяц. Списки стандартных привычек для быстрого добавления.

Вариант 15

Источник: contacts.json с демо-базой контактов. Генерация примеров взаимодействий для демонстрации. Загрузка стандартных шаблонов взаимодействий**.**

Лабораторная работа № 27

Индивидуальный проект: сохранение состояния и улучшение UX

Цель лабораторной работы: реализовать сохранение прогресса пользователя в локальное хранилище и восстановления состояния приложения.

Методические указания

Сохранение прогресса пользователя в локальное хранилище превращает приложение из эпизодического инструмента в постоянного спутника. Данные, которые стоит сохранять: результаты работы пользователя, настройки интерфейса, история действий, избранные элементы. Сохранение должно происходить автоматически при изменениях, но с учетом производительности — слишком частые записи могут замедлить работу приложения.

Восстановление состояния приложения при повторном посещении создает ощущение непрерывности. При загрузке приложение проверяет наличие сохраненных данных и, если они есть, восстанавливает состояние. Важно обрабатывать несовместимость версий: если структура данных изменилась между версиями приложения, нужна миграция или очистка устаревших данных.

Улучшение пользовательского опыта через локальное хранение проявляется в мелочах: запоминание последнего выбранного элемента, сохранение неотправленной формы, автосохранение текста, восстановление скролл-позиции. Эти детали делают приложение удобным и предсказуемым, уменьшая фрустрацию пользователя при случайном обновлении страницы или закрытии вкладки.

Задания для выполнения лабораторной работы

Вариант 1

Сохранение в localStorage: список задач, последние фильтры. Автосохранение при любых изменениях задач. Восстановление: загрузка из localStorage при старте, объединение с данными из JSON. Сохранение настроек отображения (скрытые колонки).

Вариант 2

Сохранение: все операции, настройки бюджета. Резервное копирование при удалении операций. Восстановление последней удаленной операции. Сохранение

выбранного периода отображения.

Вариант 3

Сохранение: пользовательские рецепты, выбранные для списка покупок. Автосохранение черновика при создании рецепта. Восстановление последнего поиска.

Вариант 4

Сохранение: все тренировки. Автосохранение текущей тренировки. Восстановление незавершенной тренировки.

Вариант 5

Сохранение: пользовательская коллекция, рецензии, оценки. Сохранение последних фильтров. Восстановление черновика рецензии.

Вариант 6

Сохранение: вся библиотека, добавленные цитаты, пользовательские рецензии. Автосохранение при любом изменении. Сохранение текущих фильтров и режима отображения. Восстановление последней просмотренной книги.

Вариант 7

Сохранение: все поездки, текущие планы, списки вещей. Автосохранение черновика новой поездки. Сохранение состояния упаковки. Восстановление последней активной поездки.

Вариант 8

Сохранение: вся коллекция растений, журнал ухода, фото. Автосохранение после каждого действия ухода. Сохранение настроек напоминаний. Восстановление незавершенных действий.

Вариант 9

Сохранение: пользовательский словарь, прогресс изучения, результаты тестов. Автосохранение после каждого теста. Сохранение текущей сессии обучения. Восстановление незавершенного теста.

Вариант 10

Сохранение: вся коллекция игр, история партий, рейтинги. Автосохранение после каждой записанной партии. Сохранение последних параметров подбора. Восстановление незавершенной записи партии.

Вариант 11

Сохранение: все записи настроения. Автосохранение черновика текущей записи. Сохранение настроек напоминаний. Восстановление последней незавершенной записи.

Вариант 12

Сохранение: вся коллекция, история прослушивания, желаемый список. Автосохранение после каждого изменения. Сохранение текущих фильтров просмотра. Восстановление последнего состояния.

Вариант 13

Сохранение: все зоны и задачи, история уборки. Автосохранение после каждой отметки выполнения. Сохранение настроек напоминаний. Восстановление последнего состояния задач.

Вариант 14

Сохранение: все привычки, история выполнения. Автосохранение после каждой отметки. Сохранение настроек уведомлений. Восстановление текущих серий.

Вариант 15

Сохранение: все контакты, история взаимодействий, напоминания. Автосохранение после каждого изменения. Сохранение текущих фильтров и сортировок. Восстановление последнего состояния.

Лабораторная работа № 28

Индивидуальный проект: рефакторинг и отладка

Цель лабораторной работы: структурировать код, улучшить читаемость. Найти и устранить ошибки с использованием инструментов разработчика.

Методические указания

Структурирование кода — это процесс организации существующего кода в более понятную и поддерживаемую форму. Рефакторинг включает выделение повторяющихся фрагментов в функции, группировку связанного кода в модули, переименование переменных для ясности, упрощение сложных выражений. Цель — не изменение функциональности, а улучшение читаемости и уменьшение сложности.

Поиск и устранение ошибок с использованием инструментов разработчика — это систематический процесс, а не случайные попытки. Современные браузерные инструменты предоставляют мощные возможности: отладчик с точками останова, профилировщик производительности, монитор сетевых запросов, инспектор элементов. Важно научиться воспроизводить ошибки, локализовывать их источник, понимать контекст возникновения и проверять исправления.

Улучшение читаемости кода важно не только для текущего разработчика, но и для тех, кто будет работать с кодом в будущем. Читаемый код имеет осмысленные имена переменных и функций, последовательный стиль форматирования, комментарии для сложных алгоритмов, минимальную вложенность конструкций. Хорошо структурированный код служит своей собственной документацией.

Задания для выполнения лабораторной работы

Вариант 1

Рефакторинг: выделение модулей для работы с тегами, фильтрации, статистики. Улучшение имен переменных. Отладка: проверка обработки крайних случаев (пустые теги, просроченные задачи). Инструменты: console.log для отслеживания состояния, debugger для пошагового выполнения.

Вариант 2

Рефакторинг: выделение модулей для расчетов, форматирования денежных сумм. Оптимизация пересчета статистики. Отладка точности расчетов с плавающей точкой.

Вариант 3

Рефакторинг: выделение модулей для работы с ингредиентами, поиска. Нормализация единиц измерения. Отладка работы с массивами ингредиентов.

Вариант 4

Рефакторинг: выделение модулей для расчетов прогресса, работы с датами. Оптимизация пересчета статистики. Отладка обработки единиц измерения.

Вариант 5

Рефакторинг: выделение модулей для рекомендаций, статистики. Оптимизация поиска по большой коллекции. Отладка работы с жанрами.

Вариант 6

Рефакторинг: выделение модулей для работы с жанрами, статистики чтения, форматов дат. Оптимизация поиска по большой библиотеке. Отладка обработки специальных символов в названиях книг.

Вариант 7

Рефакторинг: выделение модулей для работы с датами, геоданными, бюджетными расчетами. Оптимизация работы с картой. Отладка расчетов с различными валютами.

Вариант 8

Рефакторинг: выделение модулей для работы с расписаниями, диагностики, обработки дат. Оптимизация расчета следующих дат ухода. Отладка работы с часовыми поясами.

Вариант 9

Рефакторинг: выделение модулей для алгоритма повторения, генерации тестов, статистики. Оптимизация поиска по словарю. Отладка алгоритма интервального повторения.

Вариант 10

Рефакторинг: выделение модулей для подбора игр, статистики, работы с датами. Оптимизация алгоритма подбора. Отладка расчетов с диапазонами

параметров.

Вариант 11

Рефакторинг: выделение модулей для статистического анализа, работы с датами, визуализации. Оптимизация расчетов корреляций. Отладка обработки эмоциональных меток.

Вариант 12

Рефакторинг: выделение модулей для работы с состоянием пластинок, оценки стоимости, статистики. Оптимизация поиска по коллекции. Отладка обработки музыкальных жанров.

Вариант 13

Рефакторинг: выделение модулей для работы с расписаниями, расчета дат, управления задачами. Оптимизация обновления статусов задач. Отладка работы с повторяющимися задачами.

Вариант 14

Рефакторинг: выделение модулей для работы с сериями, статистики, управления уведомлениями. Оптимизация расчетов серий. Отладка обработки различных периодичностей.

Вариант 15

Рефакторинг: выделение модулей для работы с контактами, взаимодействиями, поиска. Оптимизация работы с большими списками контактов. Отладка обработки дат и напоминаний**.**

Лабораторная работа № 29

Индивидуальный проект: адаптивный дизайн и финальная стилизация

Цель лабораторной работы: обеспечить корректное отображение интерфейса на различных устройствах и разрешениях экрана.

Методические указания

Обеспечение корректного отображения на различных устройствах требует понимания принципов адаптивного веб-дизайна. Приложение должно одинаково хорошо работать на мобильных телефонах, планшетах, ноутбуках и десктопах. Это достигается через гибкие сетки, медиазапросы, responsive-изображения, относительные единицы измерения. Важно тестировать не только разные размеры экранов, но и различные соотношения сторон, ориентации.

Финальная стилизация — это приведение визуального оформления в соответствие с задуманным дизайном. На этом этапе уточняются цвета, шрифты, отступы, анимации, состояния элементов. Стилизация должна быть последовательной: одинаковые элементы выглядят одинаково, схожие действия имеют схожую визуальную обратную связь. Особое внимание уделяется доступности: достаточный цветовой контраст, размер интерактивных элементов, поддержка клавиатурной навигации.

Оптимизация производительности рендеринга включает минимизацию перерисовок и перекомпоновок, ленивую загрузку ресурсов, оптимизацию анимаций. Даже красивое и функциональное приложение будет разочаровывать пользователя, если оно работает медленно. Профилирование помогает найти узкие места и принять решения об оптимизациях.

Задания для выполнения лабораторной работы

Вариант 1

Адаптация: на мобильных — вертикальная компоновка, скрытие второстепенной информации, выдвижное меню фильтров. Стилизация: цветовое кодирование приоритетов (красный/желтый/зеленый), анимации добавления/удаления задач.

Вариант 2

Адаптация: на мобильных — упрощенные диаграммы, компактный вид

операций. Стилизация: цветовое разделение доходов (зеленый) и расходов (красный), визуализация превышения бюджета.

Вариант 3

Адаптация: на мобильных — упрощенное отображение рецептов, вертикальный список шагов. Стилизация: визуальное разделение по сложности, анимации перехода между шагами.

Вариант 4

Адаптация: на мобильных — упрощенный ввод подходов, компактные графики. Стилизация: цветовое кодирование интенсивности тренировок.

Вариант 5

Адаптация: на мобильных — уменьшение колонок в сетке, адаптивные постера. Стилизация: визуализация рейтинга звездами, цветовое кодирование жанров.

Вариант 6

Адаптация: на мобильных — компактные карточки книг, вертикальное расположение полок, упрощенная форма добавления. Стилизация: визуальное различие статусов книг (цветовые маркеры), анимация "перелистывания страниц" при переходе.

Вариант 7

Адаптация: на мобильных — упрощенное отображение карты, вертикальный список точек маршрута, компактные формы ввода. Стилизация: цветовое кодирование категорий расходов, интерактивные элементы на карте.

Вариант 8

Адаптация: на мобильных — упрощенный вид карточек растений, быстрые действия ухода, камера для фото. Стилизация: цветовые индикаторы состояния растений, анимации роста, визуализация графика полива.

Вариант 9

Адаптация: на мобильных — упрощенные карточки для обучения, удобный ввод слов, адаптивные тесты. Стилизация: цветовое кодирование сложности слов, анимации переворота карточек, визуализация прогресса.

Вариант 10

Адаптация: на мобильных — компактные карточки игр, упрощенный подбор, быстрая запись партии. Стилизация: визуальное представление параметров игр, интерактивные фильтры, анимации "броска кубика".

Вариант 11

Адаптация: на мобильных — быстрая запись настроения, компактные графики, упрощенный анализ. Стилизация: цветовая шкала настроения, интерактивные элементы выбора эмоций, плавные переходы в графиках.

Вариант 12

Адаптация: на мобильных — компактное отображение пластинок, упрощенная форма добавления, адаптивные статистические блоки. Стилизация: визуализация состояния пластинок, анимации "вращения пластинки", стилизованные полки.

Вариант 13

Адаптация: на мобильных — упрощенный план дома, быстрые действия по задачам, компактный календарь. Стилизация: цветовые индикаторы срочности задач, анимации "очистки", визуализация прогресса уборки.

Вариант 14

Адаптация: на мобильных — быстрая отметка привычек, компактный календарь, упрощенная статистика. Стилизация: визуализация серий (цепочки), цветовые индикаторы выполнения, анимации отметки.

Вариант 15

Адаптация: на мобильных — упрощенная таблица контактов, быстрые действия, компактные формы. Стилизация: цветовое кодирование статусов, визуализация временных линий, профессиональный дизайн.

Лабораторная работа № 30

Индивидуальный проект: тестирование, презентация и защита

Цель лабораторной работы: продемонстрировать полный функционал приложения

Методические указания

Тестирование полного функционала — это всесторонняя проверка того, что приложение работает так, как задумано. Тестирование включает не только автоматические тесты (модульные, интеграционные), но и ручное тестирование пользовательских сценариев, тестирование на разных устройствах и браузерах, тестирование производительности под нагрузкой. Особое внимание уделяется пограничным случаям и обработке ошибок.

Презентация приложения — это демонстрация его ценности для аудитории. Хорошая презентация рассказывает историю: какую проблему решает приложение, для кого оно предназначено, как им пользоваться. Демонстрация должна быть четкой, сфокусированной на ключевых функциях, с подготовленными сценариями использования. Важно не только показать, что работает, но и объяснить, как это устроено.

Защита проекта включает обоснование принятых архитектурных решений, объяснение выбора технологий, демонстрацию понимания предметной области. Защита — это возможность показать не только результат, но и процесс его достижения, уроки, извлеченные в ходе разработки, понимание того, что можно улучшить в будущем. Успешная защита подтверждает не только технические навыки, но и способность к системному мышлению, проектной работе, рефлексии.

Задания для выполнения лабораторной работы

Вариант 1

Тестирование: проверка всех функций бизнес-логики, обработки крайних случаев. Презентация: демонстрация добавления задачи, фильтрации по тегам, статистики. Защита: объяснение архитектуры, выбор структур данных для тегов.

Вариант 2

Тестирование: проверка расчетов статистики, обработки валюты.

Презентация: демонстрация добавления расхода, анализа статистики, настройки бюджета.

Вариант 3

Тестирование: проверка генерации списка покупок, поиска по ингредиентам. Презентация: демонстрация поиска рецепта, создания списка покупок.

Вариант 4

Тестирование: проверка расчетов прогресса, обработки дат. Презентация: демонстрация добавления тренировки, анализа прогресса.

Вариант 5

Тестирование: проверка рекомендательной системы, фильтрации. Презентация: демонстрация добавления фильма, анализа статистики, получения рекомендаций.

Вариант 6

Тестирование: проверка всех функций работы с книгами, статистики чтения, фильтрации. Презентация: демонстрация добавления книги, изменения статуса, анализа статистики чтения, работы с цитатами.

Вариант 7

Тестирование: проверка бюджетных расчетов, работы с датами, генерации списков. Презентация: демонстрация создания поездки, планирования маршрута, расчета бюджета, подготовки к упаковке.

Вариант 8

Тестирование: проверка расчетов расписания ухода, диагностики проблем, работы с датами. Презентация: демонстрация добавления растения, настройки ухода, использования журнала, диагностики проблем.

Вариант 9

Тестирование: проверка алгоритма интервального повторения, генерации тестов, статистики. Презентация: демонстрация добавления слов, режима обучения, тестирования, анализа прогресса.

Вариант 10

Тестирование: проверка подбора игр по параметрам, расчетов статистики, работы с диапазонами. Презентация: демонстрация добавления игры, подбора по

параметрам, записи партии, анализа статистики.

Вариант 11

Тестирование: проверка статистических расчетов, корреляционного анализа, работы с датами. Презентация: демонстрация записи настроения, анализа статистики, выявления корреляций, получения инсайтов.

Вариант 12

Тестирование: проверка расчетов стоимости коллекции, статистики, работы с состояниями. Презентация: демонстрация добавления пластинки, анализа коллекции, работы с желаемым списком.

Вариант 13

Тестирование: проверка расчетов графиков уборки, работы с частотами, ротации задач. Презентация: демонстрация создания зон, отметки задач, настройки расписания, анализа истории уборки.

Вариант 14

Тестирование: проверка расчетов серий, статистики выполнения, работы с различными частотами. Презентация: демонстрация создания привычки, отслеживания выполнения, анализа статистики, мотивационных элементов.

Вариант 15

Тестирование: проверка работы с контактами, взаимодействиями, поиска, фильтрации. Презентация: демонстрация добавления контакта, записи взаимодействия, работы с напоминаниями, поиска и фильтрации.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1 **Никсон, Р.** Создаем динамические веб-сайты с помощью PHP, MySQL, JavaScript, CSS и HTML5 / Р. Никсон. – 5-е изд. – Санкт-Петербург : Питер, 2019. – 816 с. – (Серия «Бестеллеры O'Reily»).

2 **Роббинс, Дж.** HTML5, CSS3 и JavaScript. Исчерпывающее руководство / Дженнифер Роббинс ; [пер. с англ. М. А. Райтман]. – 4-е изд. – Москва : Эксмо, 2014. – 528 с.

3 Учебники, задачники, справочники по web языкам [Электронный ресурс]. – Форма доступа: http://code.mu/.

4 **Крокфорд, Д.** JavaScript: сильные стороны / Д. Крокфорд. – Санкт-Петербург : Питер, 2012. – 176 с.

5 **Бенедетти, Р.** Изучаем работу с jQuery / Р. Бенедетти, Р. Крэнли. – Санкт-Петербург : Питер, 2012. – 528 с.

6 **Полуэктова, Н. Р.** Разработка веб-приложений: учебное пособие для среднего профессионального образования / Н. Р. Полуэктова. — Москва: Издательство Юрайт, 2022. — 204 с. (образовательная платформа Юрайт https://urait.ru/)

7 **Колисниченко Д.Н.** Разработка веб-приложений. – Спб.: БХВ-Петербург, 2017. – 640 с. [Электронный ресурс]. Форма доступа: https://books.google.ru/books?id=BjExDwAAQBAJ&printsec=frontcover&hl=ru#v=onepage&q &f=false

8 **Ломаш, Д. А.** Интернет-технологии и мультимедиа : учебное пособие / Д. А. Ломаш, О. Г. Ведерникова ; ФГБОУ ВО РГУПС. – Ростов-на-Дону, 2017. – 119 с.

9 **Бибо, Бер.** jQuery. Подробное руководство по продвинутому JavaScript / Бер Бибо, Иегуда Кац ; [пер. с англ.]. – 2-е изд. – Санкт-Петербург : Символ-Плюс, 2011. – 624 с.

10 **Васильев, А. Н.** JavaScript в примерах и задачах / А. Н. Васильев. – Москва : Издательство «Э», 2017. – 720 с. – (Российский компьютерный бестселлер).

Учебное издание

ФРОНТЕНД-РАЗРАБОТКА (КЛИЕНТСКАЯ ЧАСТЬ)

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Ростовский государственный университет путей сообщения»
(ФГБОУ ВО РГУПС)

Адрес университета:
344038, г. Ростов н/Д, пл. Ростовского Стрелкового Полка
Народного Ополчения, д. 2, www.rgups.ru