

ЛАБОРАТОРНАЯ РАБОТА № 1

«Исследование императивного подхода к программированию с использованием JavaScript»

1. Цель работы

Изучение особенностей императивных парадигм разработки ПО, получение практических навыков разработки ПО с использованием JavaScript.

2 Краткие теоретические сведения

2.1 Парадигмы программирования

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Другими словами, парадигмы программирования представляют собой основные концепции и подходы, которые определяют стиль написания программного кода. Парадигмы представляют собой набор принципов, методов и практик, которые определяют, каким образом программисты могут решать задачи с использованием компьютеров. Каждая парадигма программирования предлагает свой уникальный подход к решению проблем и организации кода.

Парадигмы программирования включают в себя различные подходы, такие как императивное программирование, функциональное программирование, логическое программирование и объектно-ориентированное программирование, ... Каждая из этих парадигм предлагает свои уникальные методы для организации кода, управления данными и решения задач. Понимание различий между этими парадигмами помогает программистам выбирать наиболее подходящий подход для конкретной задачи и создавать более эффективный и читаемый код.

Основные группы парадигм делятся по тому, что именно описывает программа:

- **ЧТО** должен вычислить компьютер? («вычисления») – **ДЕКЛАРАТИВНАЯ** парадигма.
- **КАК** он должен это вычислить? («работа») – **ИМПЕРАТИВНАЯ** парадигма.

В данной работе будет подробно рассмотрена императивная парадигма

Императивная парадигма программирования является одной из старейших и наиболее широко используемых парадигм. Основана на идее последовательного выполнения команд, которые изменяют состояние программы. Основные конструкции императивной парадигмы включают в себя присваивание значений переменным, условные операторы (if-else) для принятия решений в зависимости от определенных условий, и циклы (например, for и while), позволяющие многократно выполнять определенные инструкции. Императивная парадигма обычно связана с использованием переменных и изменяемых данных.

Существует несколько ответвлений императивной парадигмы, таких как процедурное программирование и структурное программирование.

Процедурная парадигма программирования возникла как развитие императивной парадигмы. В процедурной парадигме программа разделяется на набор процедур (или функций), каждая из которых выполняет определенную задачу. Это позволяет создавать более модульные и структурированные программы, где каждая процедура отвечает за определенный аспект функционирования программы. Процедурная парадигма также вводит концепцию локальных переменных, что делает код более читаемым и поддерживаемым.

Структурная парадигма – это методология разработки программ, которая подчеркивает **использование структурных блоков**, таких как **последовательности**, **условия** и **циклы**, для построения алгоритмов. Основная идея структурного программирования заключается в том, что **любой алгоритм может быть выражен с помощью трех основных структурных конструкций** (теорема Бёма – Якопини): **последовательности** (последовательное выполнение инструкций), **условия** (выбор инструкций в зависимости от определенных условий) и **циклы** (многократное выполнение инструкций).

Еще одним ответвлением императивной парадигмы является **объектно-ориентированное программирование (ООП)**. ООП расширяет императивную парадигму, добавляя **понятие объектов**, которые могут содержать **данные** (поля) и **методы** (функции), оперирующие этими данными. ООП также включает в себя концепции наследования, полиморфизма и инкапсуляции для управления сложностью программ.

Основные черты императивной парадигмы программирования:

- в исходном коде программы записываются **инструкции (команды)**;
- инструкции должны **выполняться последовательно**;
- при выполнении инструкции **данные, полученные при выполнении предыдущих инструкций, могут читаться из памяти**;
- **данные, полученные при выполнении инструкции, могут записываться в память**. Главное достоинство:

- именно так, и только так, работает вычислительная техника.

Главный недостаток:

- Переменные → присваивание → состояние → побочные эффекты.

Основные исторические моменты императивной парадигмы:

- Машинные коды – первый императивный язык
- Ассемблер

- Алгоритмические языки – Fortran, Algol, и далее

Развитие императивной парадигмы:

- Процедурная парадигма
 - Введено понятие подпрограмм;
 - Является особенностью архитектуры фон Неймана, где код располагается в общей памяти с произвольным доступом.
- Структурная парадигма
 - Отказ от произвольных переходов между подпрограммами;
 - Построение программы из трех базовых управляющих структур:
 - Последовательность
 - Ветвление
 - Цикл

На текущий момент компьютеры собираются на основе архитектуры фон Неймана, в который исполняемый код работает только в императивной парадигме. Какая бы парадигма не использовалась для разработки той или иной системы, при компиляции код переводится в императивный.

2.2 Базовые управляющие структуры в Java Script

2.2.1 Последовательности

В JavaScript последовательное выполнение команд означает, что команды выполняются одна за другой в порядке их следования. Это позволяет программе последовательно выполнять различные действия.

Последовательное выполнение операций:

```
let a = 5;
let b = 3;
let c = a + b;
console.log(c); // Выведет 8
```

В примере переменная `a` инициализируется значением 5, затем переменная `b` инициализируется значением 3, после чего переменной `c` присваивается результат сложения `a` и `b`. Наконец, результат выводится на консоль.

Последовательное выполнение функций:

```
function greet(name) {
    console.log("Привет, " + name + "!");
}

function askQuestion() {
    console.log("Как дела?");
}

greet("Анна"); // Выведет "Привет, Анна!"
askQuestion(); // Выведет "Как дела?«
```

В примере сначала вызывается функция `greet` с аргументом "Анна", затем вызывается функция `askQuestion`. Обе функции выполняются последовательно в порядке их вызова.

2.2.2 Ветвления

Условные операторы в JavaScript используются для выполнения различных действий в зависимости от выполнения определенного условия. Самый распространенный условный оператор в JavaScript – это **if-else** оператор.

```
let x = 10;

if (x > 5) {
  console.log("x больше 5");
} else {
  console.log("x меньше или равен 5");
}
```

В примере, если значение переменной `x` больше 5, будет выполнен блок кода внутри первого блока `{ }`, иначе будет выполнен блок кода внутри второго блока `{ }`.

Также в JavaScript есть условный оператор **else if**, который позволяет проверить несколько условий подряд:

```
let y = 7;

if (y > 10) {
  console.log("y больше 10");
} else if (y > 5) {
  console.log("y больше 5, но меньше или равен 10");
} else {
  console.log("y меньше или равен 5");
}
```

В примере сначала проверяется условие `y > 10`, затем, если оно не выполняется, проверяется условие `y > 5`, иначе выполняется блок кода после `else`.

Существует **тернарный оператор**, который представляет собой сокращенную форму записи условия `if-else`:

```
let z = 12;
let result = (z > 10) ? "z больше 10" : "z меньше или равен 10";
console.log(result);
```

В примере, если *z* больше 10, переменной *result* будет присвоено значение "z больше 10", иначе - "z меньше или равен 10".

Условные операторы в JavaScript позволяют создавать гибкие и мощные программы, которые могут адаптироваться к различным ситуациям.

2.2.3 Циклы

Циклы в JavaScript позволяют многократно выполнять определенный блок кода в зависимости от условий, что делает их очень мощным инструментом для автоматизации повторяющихся задач.

Цикл **for**:

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

Цикл будет выполняться до тех пор, пока переменная *i* меньше 5. На каждой итерации значение *i* увеличивается на 1, и в результате на консоль выводятся числа от 0 до 4.

Цикл **while**:

```
let count = 0;
while (count < 3) {
  console.log("Повторение " + count);
  count++;
}
```

Цикл будет выполняться до тех пор, пока переменная count меньше 3. На каждой итерации значение count увеличивается на 1, и на консоль выводится сообщение "Повторение" с номером текущей итерации.

Цикл **do...while:**

```
let x = 0;
do {
  console.log(x);
  x++;
} while (x < 3);
```

Цикл сначала выполняет блок кода, а затем проверяет условие. В данном случае на консоль выводятся числа от 0 до 2.

2.3 **Массивы в js (упрощенно)**

Массив – это тип данных, в котором хранится **упорядоченный набор однотипных элементов**

В JavaScript массивы **могут содержать любые типы данных**, включая числа, строки, объекты и даже другие массивы. Каждый элемент в массиве имеет свой уникальный индекс, начиная с **0**.

Одномерный массив представляет собой список элементов, организованных одним рядом. Для создания одномерного массива в JavaScript можно использовать следующий синтаксис:

```
let одномерный_массив = [элемент1, элемент2, элемент3, ...];
```

Пример:

```
let числа = [1, 2, 3, 4, 5];
let месяцы = ["Январь", "Февраль", "Март", "Апрель"];
```

Вы можете **получить доступ к элементам в одномерном массиве**, используя их индексы:

```
console.log(числа[0]); // Вывод: 1
console.log(месяцы[2]); // Вывод: "Март"
```

Двумерный массив представляет собой массив массивов, то есть массив, в котором каждый элемент также является массивом. Такая структура данных позволяет моделировать матрицы или таблицы. Для создания двумерного массива в JavaScript можно использовать следующий синтаксис:

```
let    двумерный_массив    =    [[элемент1,    элемент2,    элемент3],
[элемент4, элемент5, элемент6], ...];
```

Пример:

```
let матрица = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
```

Вы можете получить **доступ к элементам в двумерном массиве**, используя двойные индексы:

```
console.log(матрица[0][1]); // Вывод: 2
console.log(матрица[2][0]); // Вывод: 7
```

Для обхода и изменения элементов в массивах можно использовать циклы.

3 Порядок выполнения

3.1 Изучить основные средства языка JavaScript для разработки программ с использованием императивной парадигмы.

3.2 Выполнить две задачи из варианта на языке JavaScript согласно своему варианту.

3.3 Разработать тестовые примеры.

3.4 Выполнить отладку программ.

3.5 Сформулировать выводы, проанализировав созданные программы.

3.6 Оформить отчет по проделанной работе.

4 Варианты заданий

Для каждого варианта требуется создать одномерный и двумерный массивы. Для однокамерного массива нужно добавить минимум 10 элементов, для матрицы 16 элементов.

Вариант 1

Найти сумму всех элементов в массиве.

Удалить все строки матрицы, в которых есть отрицательные элементы.

Вариант 2

Найти наибольший элемент в массиве.

Проверить, содержит ли матрица магический квадрат (сумма элементов всех строк, столбцов и диагоналей одинакова).

Вариант 3

Найти индекс первого вхождения определенного элемента в массив.

Найти среднее арифметическое элементов в каждой строке матрицы.

Вариант 4

Посчитать количество четных элементов в массиве.

Поменять местами две заданные строки матрицы.

Вариант 5

Отсортировать массив по возрастанию.

Найти сумму всех элементов в двумерном массиве.

Вариант 6

Изменить порядок элементов массива на обратный.

Отсортировать строки матрицы по возрастанию суммы их элементов.

Вариант 7

Удалить все дубликаты из массива.

Посчитать сумму элементов каждого столбца и сохранить результаты в одномерный массив.

Вариант 8

Найти среднее арифметическое всех элементов массива.

Посчитать количество строк матрицы, в которых есть хотя бы один отрицательный элемент.

Вариант 9

Проверить, является ли массив палиндромом.

Найти наименьший элемент в двумерном массиве.

Вариант 10

Найти сумму элементов на нечетных позициях массива.

Найти сумму элементов главной диагонали матрицы.