Частное учреждение образования

«Колледж бизнеса и права»

|  |
| --- |
| УТВЕРЖДАЮ  Заведующий  методическим кабинетом  \_\_\_\_\_\_\_\_\_\_Е.В.Паскал  «\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_ |

|  |  |
| --- | --- |
| Специальность:  2-40 01 01 «Программное обеспечение информационных технологий» | Дисциплина: «Скриптовые языки программирования» |

**Лабораторная работа № 17**

**Инструкционно-технологическая карта**

Тема: Методы обработки массивов

Цель: Научиться обрабатывать массивы с помощью методов обработки массивов.

Время выполнения: 2 часа

# Порядок выполнения работы

1. Изучить теоретические сведения к лабораторной работе.

2. Реализовать алгоритм решения задачи.

3. Отлаженную, работающую программу сдать преподавателю. Работу программы показать с помощью самостоятельно разработанных тестов.

4. Ответить на контрольные вопросы.

# Теоретические сведения

## Добавление/удаление элементов

Методы, которые добавляют и удаляют элементы из начала или конца:

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

Есть и другие.

#### splice

Как удалить элемент из массива?

Так как массивы – это объекты, то можно попробовать delete:

let arr = ["I", "go", "home"];

delete arr[1]; // удалить "go"

alert( arr[1] ); // undefined

// теперь arr = ["I",  , "home"];

alert( arr.length ); // 3

Вроде бы, элемент и был удалён, но при проверке оказывается, что массив всё ещё имеет 3 элемента arr.length == 3.

Это нормально, потому что всё, что делает delete obj.key – это удаляет значение с данным ключом key. Это нормально для объектов, но для массивов мы обычно хотим, чтобы оставшиеся элементы сдвинулись и заняли освободившееся место. Ожидается, что массив станет короче.

Поэтому для этого нужно использовать специальные методы.

Метод **arr.splice(str)** – это универсальный «швейцарский нож» для работы с массивами. Умеет всё: добавлять, удалять и заменять элементы.

Его синтаксис:

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

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

Этот метод проще всего понять, рассмотрев примеры.

Начнём с удаления:

let arr = ["Я", "изучаю", "JavaScript"];

arr.splice(1, 1); // начиная с позиции 1, удалить 1 элемент

alert( arr ); // осталось ["Я", "JavaScript"]

Начиная с позиции **1**, он убрал **1** элемент.

В следующем примере удаляется 3 элемента и заменяются их двумя другими.

let arr = ["Я", "изучаю", "JavaScript", "прямо", "сейчас"];

// удалить 3 первых элемента и заменить их другими

arr.splice(0, 3, "Давай", "танцевать");

alert( arr ) // теперь ["Давай", "танцевать", "прямо", "сейчас"]

Здесь видно, что **splice** возвращает массив из удалённых элементов:

let arr = ["Я", "изучаю", "JavaScript", "прямо", "сейчас"];

// удалить 2 первых элемента

let removed = arr.splice(0, 2);

alert( removed ); // "Я", "изучаю" <-- массив из удалённых элементов

Метод **splice** также может вставлять элементы без удаления, для этого достаточно установить **deleteCount** в **0**:

let arr = ["Я", "изучаю", "JavaScript"];

// с позиции 2

// удалить 0 элементов

// вставить "сложный", "язык"

arr.splice(2, 0, "сложный", "язык");

alert( arr ); // "Я", "изучаю", "сложный", "язык", "JavaScript"

**Отрицательные индексы разрешены**

В этом и в других методах массива допускается использование отрицательного индекса. Он позволяет начать отсчёт элементов с конца, как тут:

let arr = [1, 2, 5];

// начиная с индекса -1 (перед последним элементом)

// удалить 0 элементов,

// затем вставить числа 3 и 4

arr.splice(-1, 0, 3, 4);

alert( arr ); // 1,2,3,4,5

#### slice

Метод **arr.slice** намного проще, чем похожий на него **arr.splice**.

Его синтаксис:

arr.slice([start], [end])

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

Это похоже на строковый метод **str.slice**, но вместо подстрок возвращает подмассивы.

Например:

let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) ); // e,s (копирует с 1 до 3)

alert( arr.slice(-2) ); // s,t (копирует с -2 до конца)

Можно вызвать **slice** и вообще без аргументов: **arr.slice()** создаёт копию массива **arr**. Это часто используют, чтобы создать копию массива для дальнейших преобразований, которые не должны менять исходный массив.

#### Concat

Метод **arr.concat** создаёт новый массив, в который копирует данные из других массивов и дополнительные значения.

Его синтаксис:

arr.concat(arg1, arg2...)

Он принимает любое количество аргументов, которые могут быть как массивами, так и простыми значениями.

В результате получаем новый массив, включающий в себя элементы из **arr**, а также **arg1**, **arg2** и так далее…

Если аргумент **argN** – массив, то все его элементы копируются. Иначе скопируется сам аргумент.

Например:

let arr = [1, 2];

// создать массив из: arr и [3,4]

alert( arr.concat([3, 4]) ); // 1,2,3,4

// создать массив из: arr и [3,4] и [5,6]

alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6

// создать массив из: arr и [3,4], потом добавить значения 5 и 6

alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6

Обычно он просто копирует элементы из массивов. Другие объекты, даже если они выглядят как массивы, добавляются как есть:

let arr = [1, 2];

let arrayLike = {

  0: "что-то",

  length: 1

};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]

Но если объект имеет специальное свойство **Symbol.isConcatSpreadable**, то он обрабатывается **concat** как массив: вместо него добавляются его числовые свойства.

Для корректной обработки в объекте должны быть числовые свойства и **length**:

let arr = [1, 2];

let arrayLike = {

  0: "что-то",

  1: "ещё",

  [Symbol.isConcatSpreadable]: true,

  length: 2

};

alert( arr.concat(arrayLike) ); // 1,2,что-то,ещё

#### Перебор: forEach

Метод **arr.forEach** позволяет запускать функцию для каждого элемента массива.

Его синтаксис:

arr.forEach(function(item, index, array) {

    // ... делать что-то с item

  });

Например, этот код выведет на экран каждый элемент массива:

// Вызов alert для каждого элемента

["Bilbo", "Gandalf", "Nazgul"].forEach(alert);

А этот вдобавок расскажет и о своей позиции в массиве:

["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {

    alert(`${item} имеет позицию ${index} в ${array}`);

  });

Результат функции (если она вообще что-то возвращает) отбрасывается и игнорируется.

## Поиск в массиве

#### indexOf/lastIndexOf и includes

Методы **arr.indexOf**, arr.**lastIndexOf** и **arr.includes** имеют одинаковый синтаксис и делают по сути то же самое, что и их строковые аналоги, но работают с элементами вместо символов:

1. **arr.indexOf(item, from)** ищет **item**, начиная с индекса **from**, и возвращает индекс, на котором был найден искомый элемент, в противном случае **-1**.
2. **arr.lastIndexOf(item, from)** – то же самое, но ищет справа налево.
3. **arr.includes(item, from)** – ищет **item**, начиная с индекса **from**, и возвращает **true**, если поиск успешен.

Например:

let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1

alert( arr.indexOf(false) ); // 2

alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true

Эти методы используют строгое сравнение ===. Таким образом, если ищем **false**, он находит именно **false**, а не ноль.

Если надо проверить наличие элемента, и нет необходимости знать его точный индекс, тогда предпочтительным является **arr.includes**.

Кроме того, очень незначительным отличием **includes** является то, что он правильно обрабатывает **NaN** в отличие от **indexOf/lastIndexOf**:

const arr = [NaN];

alert( arr.indexOf(NaN) ); // -1 (должен быть 0, но === проверка на равенство не работает для NaN)

alert( arr.includes(NaN) );// true (верно)

#### find и findIndex

Есть массив объектов. Как найти объект с определённым условием?

Здесь пригодится метод **arr.find**.

Его синтаксис таков:

let result = arr.find(function(item, index, array) {

    // если true - возвращается текущий элемент и перебор прерывается

    // если все итерации оказались ложными, возвращается undefined

  });

Функция вызывается по очереди для каждого элемента массива:

1. item – очередной элемент.
2. index – его индекс.
3. array – сам массив.

Если функция возвращает **true**, поиск прерывается и возвращается **item**. Если ничего не найдено, возвращается **undefined**.

Например, есть массив пользователей, каждый из которых имеет поля **id** и **name**. Нахождение того, кто с **id == 1**:

let users = [

    {id: 1, name: "Вася"},

    {id: 2, name: "Петя"},

    {id: 3, name: "Маша"}

  ];

  let user = users.find(item => item.id == 1);

  alert(user.name); // Вася

В реальной жизни массивы объектов – обычное дело, поэтому метод **find** крайне полезен.

В данном примере передается **find** функцию **item** **=> item.id == 1**, с одним аргументом. Это типично, дополнительные аргументы этой функции используются редко.

Метод **arr.findIndex** – по сути, то же самое, но возвращает индекс, на котором был найден элемент, а не сам элемент, и -1, если ничего не найдено.

#### filter

Метод **find** ищет один (первый попавшийся) элемент, на котором функция-колбэк вернёт **true**.

На тот случай, если найденных элементов может быть много, предусмотрен метод **arr.filter(fn)**.

Синтаксис этого метода схож с **find**, но **filter** возвращает массив из всех подходящих элементов:

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

  });

Например:

let users = [

    {id: 1, name: "Вася"},

    {id: 2, name: "Петя"},

    {id: 3, name: "Маша"}

  ];

  // возвращает массив, состоящий из двух первых пользователей

  let someUsers = users.filter(item => item.id < 3);

  alert(someUsers.length); // 2

## Преобразование массива

#### map

Метод **arr.map** является одним из наиболее полезных и часто используемых.

Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции.

Синтаксис:

let result = arr.map(function(item, index, array) {

    // возвращается новое значение вместо элемента

  });

Например, здесь преобразуем каждый элемент в его длину:

let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);

alert(lengths); // 5,7,6

#### sort(fn)

Вызов **arr.sort()** сортирует массив на месте, меняя в нём порядок элементов.

Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам **arr**.

Например:

let arr = [ 1, 2, 15 ];

// метод сортирует содержимое arr

arr.sort();

alert( arr );  // 1, 15, 2

Порядок стал 1, 15, 2. Это неправильно! Но почему?

По умолчанию элементы сортируются как строки.

Буквально, элементы преобразуются в строки при срав-нении. Для строк применяется лексикографический порядок, и действительно выходит, что "2" > "15".

Чтобы использовать наш собственный порядок сортировки, нам нужно предоставить функцию в качестве аргумента **arr.sort().**

Функция должна для пары значений возвращать:

function compare(a, b) {

    if (a > b) return 1; // если первое значение больше второго

    if (a == b) return 0; // если равны

    if (a < b) return -1; // если первое значение меньше второго

  }

Например, для сортировки чисел:

function compareNumeric(a, b) {

    if (a > b) return 1;

    if (a == b) return 0;

    if (a < b) return -1;

}

let arr = [1, 2, 15];

arr.sort(compareNumeric);

alert(arr);  // 1, 2, 15

Теперь всё работает как надо.

Упомянутый ранее массив **arr** может быть массивом чего угодно, верно? Он может содержать числа, строки, объекты или что-то ещё. Есть набор каких-то элементов. Чтобы отсортировать его, нам нужна функция, определяющая порядок, которая знает, как сравнивать его элементы. По умолчанию элементы сортируются как строки.

Метод **arr.sort(fn)** реализует общий алгоритм сортировки. Не нужно заботиться о том, как он работает внутри (в большинстве случаев это оптимизированная быстрая сортировка). Она проходится по массиву, сравнивает его элементы с помощью предоставленной функции и переупорядочивает их. Всё, что остаётся, это предоставить **fn**, которая делает это сравнение.

Кстати, если когда-нибудь надо будет узнать, какие элементы сравниваются – ничто не мешает вывести их на экран:

[1, -2, 15, 2, 0, 8].sort(function (a, b) {

    alert(a + " <> " + b);

});

В процессе работы алгоритм может сравнивать элемент с другими по нескольку раз, но он старается сделать как можно меньше сравнений.

**Функция сравнения может вернуть любое число**

На самом деле от функции сравнения требуется любое положительное число, чтобы сказать «больше», и отрицательное число, чтобы сказать «меньше».

Это позволяет писать более короткие функции:

let arr = [ 1, 2, 15 ];

arr.sort(function(a, b) { return a - b; });

alert(arr);  // 1, 2, 15

#### Reverse

Метод **arr.reverse** меняет порядок элементов в **arr** на обратный.

Например:

let arr = [1, 2, 3, 4, 5];

arr.reverse();

alert( arr ); // 5,4,3,2,1

Он также возвращает массив **arr** с изменённым порядком элементов.

#### split и join

Ситуация из реальной жизни. Приложение для обмена сообщениями, и посетитель вводит имена тех, кому его отправить, через запятую: Вася, Петя, Маша. Но гораздо удобнее работать с массивом имён, чем с одной строкой. Как его получить?

Метод **str.split(delim)** именно это и делает. Он разбивает строку на массив по заданному разделителю **delim**.

В примере ниже таким разделителем является строка из запятой и пробела.

let names = 'Вася, Петя, Маша';

let arr = names.split(', ');

for (let name of arr) {

  alert( `Сообщение получат: ${name}.` ); // Сообщение получат: Вася (и другие имена)

}

У метода **split** есть необязательный второй числовой аргумент – ограничение на количество элементов в массиве. Если их больше, чем указано, то остаток массива будет отброшен. На практике это редко используется:

let arr = 'Вася, Петя, Маша, Саша'.split(', ', 2);

alert(arr); // Вася, Петя

Вызов split(s) с пустым аргументом s разбил бы строку на массив букв:

let str = "тест";

alert( str.split('') ); // т,е,с,т

Вызов **arr.join(glue)** делает в точности противоположное **split**. Он создаёт строку из элементов **arr**, вставляя glue между ними.

Например:

let arr = ['Вася', 'Петя', 'Маша'];

let str = arr.join(';'); // объединить массив в строку через ;

alert( str ); // Вася;Петя;Маша

#### reduce/reduceRight

Если нужно перебрать массив – можно использовать **forEach**, for или for..of.

Если нужно перебрать массив и вернуть данные для каждого элемента – используется map.

Методы **arr.reduce** и **arr.reduceRight** похожи на методы выше, но они немного сложнее. Они используются для вычисления какого-нибудь единого значения на основе всего массива.

Синтаксис:

let value = arr.reduce(function (previousValue, item, index, array) {

    // ...

}, [initial]);

Функция применяется по очереди ко всем элементам массива и «переносит» свой результат на следующий вызов.

Аргументы:

1. previousValue – результат предыдущего вызова этой функции, равен initial при первом вызове (если передан initial),
2. item – очередной элемент массива,
3. index – его индекс,
4. array – сам массив.

При вызове функции результат её вызова на предыдущем элементе массива передаётся как первый аргумент.

Звучит сложновато, но всё становится проще, если думать о первом аргументе как «аккумулирующем» результат предыдущих вызовов функции. По окончании он становится результатом reduce.

Этот метод проще всего понять на примере.

Тут мы получим сумму всех элементов массива всего одной строкой:

let arr = [1, 2, 3, 4, 5];

let result = arr.reduce((sum, current) => sum + current, 0);

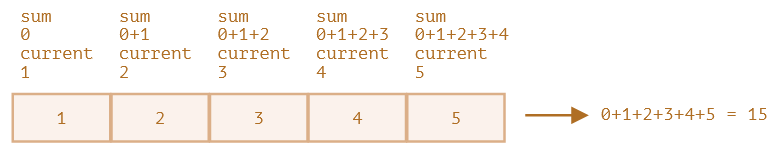
alert(result); // 15

Здесь используется наиболее распространённый вариант reduce, который использует только 2 аргумента.

Как он работает.

1. При первом запуске **sum** равен **initial** (последний аргумент **reduce**), то есть **0**, а **current** – первый элемент массива, равный **1**. Таким образом, результат функции равен **1**.
2. При втором запуске **sum = 1**, и к нему добавляется второй элемент массива (2).
3. При третьем запуске **sum = 3**, к которому добавляется следующий элемент, и так далее…

Поток вычислений получается такой:



#### Array.isArray

Массивы не образуют отдельный тип языка. Они основаны на объектах.

Поэтому **typeof** не может отличить простой объект от массива:

alert(typeof {}); // object

alert(typeof []); // тоже object

Но массивы используются настолько часто, что для этого придумали специальный метод: **Array.isArray(value).** Он возвращает **true**, если **value** массив, и **false**, если нет.

alert(Array.isArray({})); // false

alert(Array.isArray([])); // true

#### Большинство методов поддерживают «thisArg»

Почти все методы массива, которые вызывают функции – такие как **find**, **filter**, **map**, за исключением метода **sort**, принимают необязательный параметр **thisArg**.

Этот параметр не объяснялся выше, так как очень редко используется, но для наиболее полного понимания темы мы обязаны его рассмотреть.

Вот полный синтаксис этих методов:

arr.find(func, thisArg);

arr.filter(func, thisArg);

arr.map(func, thisArg);

// ...

// thisArg - это необязательный последний аргумент

Значение параметра **thisArg** становится **this** для **func**.

Например, вот тут мы используем метод объекта **army** как фильтр, и **thisArg** передаёт ему контекст:

let army = {

    minAge: 18,

    maxAge: 27,

    canJoin(user) {

        return user.age >= this.minAge && user.age < this.maxAge;

    }

};

let users = [

    { age: 16 },

    { age: 20 },

    { age: 23 },

    { age: 30 }

];

// найти пользователей, для которых army.canJoin возвращает true

let soldiers = users.filter(army.canJoin, army);

alert(soldiers.length); // 2

alert(soldiers[0].age); // 20

alert(soldiers[1].age); // 23

Если в примере выше использовать просто **users.filter(army.canJoin)**, то вызов **army.canJoin** был бы в режиме отдельной функции, с **this=undefined**. Это тут же привело бы к ошибке.

Вызов **users.filter(army.canJoin, army)** можно заменить на **users.filter(user => army.canJoin(user))**, который делает то же самое. Последняя запись используется даже чаще, так как функция-стрелка более наглядна.

## Итого

Методы массива:

1. Для добавления/удаления элементов:
   1. push (...items) – добавляет элементы в конец,
   2. pop() – извлекает элемент с конца,
   3. shift() – извлекает элемент с начала,
   4. unshift(...items) – добавляет элементы в начало.
   5. splice(pos, deleteCount, ...items) – начиная с индекса pos, удаляет deleteCount элементов и вставляет items.
   6. slice(start, end) – создаёт новый массив, копируя в него элементы с позиции start до end (не включая end).
   7. concat(...items) – возвращает новый массив: копирует все члены текущего массива и добавляет к нему items. Если какой-то из items является массивом, тогда берутся его элементы.
2. Для поиска среди элементов:
   1. indexOf/lastIndexOf(item, pos) – ищет item, начиная с позиции pos, и возвращает его индекс или -1, если ничего не найдено.
   2. includes(value) – возвращает true, если в массиве имеется элемент value, в противном случае false.
   3. find/filter(func) – фильтрует элементы через функцию и отдаёт первое/все значения, при прохождении которых через функцию возвращается true.
   4. findIndex похож на find, но возвращает индекс вместо значения.
3. Для перебора элементов:
   1. forEach(func) – вызывает func для каждого элемента. Ничего не возвращает.
4. Для преобразования массива:
   1. map(func) – создаёт новый массив из результатов вызова func для каждого элемента.
   2. sort(func) – сортирует массив «на месте», а потом возвращает его.
   3. reverse() – «на месте» меняет порядок следования элементов на противоположный и возвращает изменённый массив.
   4. split/join – преобразует строку в массив и обратно.
   5. reduce(func, initial) – вычисляет одно значение на основе всего массива, вызывая func для каждого элемента и передавая промежуточный результат между вызовами.
5. Дополнительно:
   1. Array.isArray(arr) проверяет, является ли arr массивом.

Методы sort, reverse и splice изменяют исходный массив.

Изученных методов достаточно в 99% случаев, но существуют и другие.

* arr.some(fn)/arr.every(fn) проверяет массив.
* Функция fn вызывается для каждого элемента массива аналогично map. Если какие-либо/все результаты вызовов являются true, то метод возвращает true, иначе false.
* arr.fill(value, start, end) – заполняет массив повторяющимися value, начиная с индекса start до end.
* arr.copyWithin(target, start, end) – копирует свои элементы, начиная со start и заканчивая end, в собственную позицию target (перезаписывает существующие).

Полный список есть в справочнике MDN.

# Контрольные вопросы

1. Какие функции используются для добавления и удаления элементов в массив?
2. Какие функции используются для поиска информации в массиве?
3. Какие функции используются для преобразования информации в массиве?

# Задания

1. Переведите текст вида border-left-width в borderLeftWidth

Напишите функцию camelize(str), которая преобразует строки вида «my-short-string» в «myShortString».

То есть дефисы удаляются, а все слова после них получают заглавную букву.

Примеры:

camelize("background-color") == 'backgroundColor';

camelize("list-style-image") == 'listStyleImage';

camelize("-webkit-transition") == 'WebkitTransition';

1. Фильтрация по диапазону

Напишите функцию filterRange(arr, a, b), которая принимает массив arr, ищет в нём элементы между a и b и отдаёт массив этих элементов.

Функция должна возвращать новый массив и не изменять исходный.

Например:

let arr = [5, 3, 8, 1];

let filtered = filterRange(arr, 1, 4);

alert( filtered ); // 3,1 (совпадающие значения)

alert( arr ); // 5,3,8,1 (без изменений)

1. Фильтрация по диапазону "на месте"

Напишите функцию filterRangeInPlace(arr, a, b), которая принимает массив arr и удаляет из него все значения кроме тех, которые находятся между a и b. То есть, проверка имеет вид a ≤ arr[i] ≤ b.

Функция должна изменять принимаемый массив и ничего не возвращать.

Например:

let arr = [5, 3, 8, 1];

filterRangeInPlace(arr, 1, 4); // удалены числа вне диапазона 1..4

alert( arr ); // [3, 1]

1. Скопировать и отсортировать массив

У нас есть массив строк arr. Нужно получить отсортированную копию, но оставить arr неизменённым.

Создайте функцию copySorted(arr), которая будет возвращать такую копию.

let arr = ["HTML", "JavaScript", "CSS"];

let sorted = copySorted(arr);

alert( sorted ); // CSS, HTML, JavaScript

alert( arr ); // HTML, JavaScript, CSS (без изменений)

1. Трансформировать в массив имён

У вас есть массив объектов user, и в каждом из них есть user.name. Напишите код, который преобразует их в массив имён.

Например:

let vasya = { name: "Вася", age: 25 };

let petya = { name: "Петя", age: 30 };

let masha = { name: "Маша", age: 28 };

let users = [vasya, petya, masha];

let names = /\* ... ваш код \*/

alert(names); // Вася, Петя, Маша

# Литература

**Диков, А.В.** Клиентские технологии веб программирования: JavaScript и DOM: учебное пособие / А.В. Диков. – СПб: Лань, 2020 – 124 с.

**Читанамбри, Кирупа.** Изучаем React / Кирупа Читанамбри – 2-е изд. – М.: Эксмо, 2019. 368 с.

**Хавербеке, Марейн.** Выразительный JavaScript. Современное веб-программирование. / Марейн Хавербеке – 3-е изд. – СПб.: Питер, 2019. – 480 с.:

Преподаватель Рогалевич А.В.

Рассмотрено на заседании цикловой

комиссии программного обеспечения

информационных технологий №10

Протокол № \_\_\_ от \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Председатель ЦК\_\_\_\_\_\_\_\_ В.Ю.Михалевич