Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Филиал
«Минский радиотехнический колледж»

Учебная дисциплина «Программные средства создания Internet-приложений»

Инструкция
по выполнению лабораторной работы
«Использование функций при процедурном подходе в программировании на языке JavaScript. Управление видимостью переменных при помощи замыканий»

Минск
2020

Лабораторная работа № 19

Тема работы: Использование функций при процедурном подходе в программировании на языке JavaScript. Управление видимостью переменных при помощи замыканий

1. Цель работы

Формирование умений объявления и вызова пользовательских функции в JavaScript.

2. Задание

Выполнить задания в соответствии с порядком выполнения лабораторной работы.

3. Оснащение работы

ПК, редактор исходного кода, браузер.

4. Основные теоретические сведения

Функция в JavaScript – специальный тип объектов, позволяющий формализовать средствами языка определённую логику поведения и обработки данных.

В Javascript функции являются полноценными объектами встроенного класса Function. Поэтому их можно присваивать переменным, передавать и, конечно, у них есть свойства.

Существуют следующие способы объявления функции: Function Declaration, Function Expression и Named Function Expression, Arrow Function Expression.

1. Function Declaration (FD, декларативный стиль, *function statement*) – это "классическое" объявление функции. В JavaScript функции объявляются с помощью литерала функции. Синтаксис объявления FD:

```
function имяФункции(пер1, пер2){
   Код функции
}

//Вызов функции
имяФункции(пер1,пер2);
```

```
            function sayHi() {
              alert("Hello");
            }
```

2. Function Expression (FE, функциональное выражение) – это объявление функции, которое является частью какого-либо выражения (например, присваивания). Такие функции, как правило, **анонимны**: Синтаксис объявления FE:

```
//Объявление функции
let имяфункции=function(пер1, пер2){Код функции}

//Вызов функции
имяфункции(пер1,пер2);

            let sayHi = function () {
              alert("Hello");
            }
```

3. Named Function Expression (сокращённо NFE) – это объявление функции, которое является частью какого-либо выражения (например, присваивания). Синтаксис объявления NFE:

```
            let sayHi = function foo() {
```

```
  alert("Hello");
};
sayHi();           // Hello
foo();             // ReferenceError: foo is not defined
```

Объявления FE и NFE обрабатываются интерпретатором точно так же, как и объявление FD: интерпретатор создаёт функцию и сохраняет ссылку на неё в переменной sayHi.

3. Arrow Function Expression – более простой и краткий синтаксис для создания функций, Он называется «функции-стрелки» или «стрелочные функции» (arrow functions).

Это – анонимные функции с особым синтаксисом, которые принимают фиксированное число аргументов и работают в контексте включающей их области видимости, то есть – в контексте функции или другого кода, в котором они объявлены.

Базовая структура стрелочной функции выглядит так:

```
(argument1, argument2, ... argumentN) => {  // тело функции }
```

Выражение стрелочных функций не позволяют задавать имя, поэтому стрелочные функции **анонимны**, если их ни к чему не присвоить.

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
// эквивалентно: (param1, param2, …, paramN) => { return expression; }

// Круглые скобки не обязательны для единственного параметра:
(singleParam) => { statements }
singleParam => { statements }
                      let double = n => n * 2;
                      const getFirst = array => array[0];

// Функция без параметров нуждается в круглых скобках:
() => { statements }
() => expression
// Эквивалентно: () => { return expression; }
                      let sayHi = () => alert("Hello!");

// Когда возвращаете литеральное выражение объекта, необходимо заключить тело в скобки
params => ({foo: bar})

Пример:
let sum = (a, b) => a + b;

/* Более короткая форма для:

let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

Существует ещё один способ создания функции, который используется очень редко. Он позволяет создавать функцию полностью «на лету» из строки, вот так:

```
let sum = new Function('a,b', ' return a+b; ');

let result = sum(1, 2);
alert( result ); // 3
```

В JavaScript есть три области видимости: глобальная, область видимости функции и блочная.

Переменная, которая была объявлена с помощью ключевого слова var или let за пределами функции, называется **глобальной переменной** (global variable). Глобальные

переменные уничтожаются, когда закрывается страница. Такая переменная имеет глобальную область видимости, это означает, что она доступна в любом месте исходного кода:

```
let str = "Hello!";   // Глобальная переменная

function foo() {
  alert(str);
}

foo();                // Доступна внутри функции
alert(str);           // Доступна вне функции
```

Переменная, объявленная внутри блока с помощью ключевого слова **let**, называется **переменной блочного уровня** (block level variable) или кратко блочной переменной. Такая переменная находится в блочной области видимости, это означает, что она доступна только внутри блока:

```
{
  let a = 5;
}

alert(a);   // Ошибка. Переменная a не видна за пределами блока

пример с var
for (var i = 0; i < 5; i++) {
  console.log('Что-то было сделано ' + i + ' раз'); // 0 1 2 3 4
}

console.log('Переменная i до сих пор доступна и равна ' + i); // 5

пример с let
for (let i = 0; i < 5; i++) {
  console.log(i); // 0 1 2 3 4
}

console.log(i); // ReferenceError: i is not defined
```

“Временно мёртвая зона”: при запуске кода из блока происходит резервирование имён всех переменных, объявленных в любом месте блока, но, чтобы переменные были доступны, необходимо объявлять все используемые переменные в самом начале блока.

```
let b = 20;
if (true) {
  console.log(b);
  let b = 10; // ReferenceError: b is not defined
}
```

Переменная, объявленная с помощью ключевого слова **var** или **let** внутри функции, называется **локальной переменной** (local variable). Такая переменная находится в области видимости функции, это означает, что она доступна в любом месте внутри тела функции, в которой была объявлена. Локальная переменная создаётся каждый раз при вызове функции и уничтожается при выходе из неё (при завершении работы функции):

```
function foo() {
  let str = "Hello!";   // Локальная переменная
  alert(str);
}

foo();                  // Доступна внутри функции
alert(str);             // Ошибка. Не доступна вне функции
```

Локальная переменная имеет преимущество перед глобальной переменной с тем же именем, это означает, что внутри функции будет использоваться локальная переменная, а не глобальная.

Желательно сводить использование глобальных переменных к минимуму. В современном коде обычно мало или совсем нет глобальных переменных. Хотя они иногда полезны для хранения важнейших «общепроектовых» данных.

```
(function() {
  'use strict';
  // Переменные a и b находятся в области видимости
  // самовызывающейся анонимной функции и не доступны
  // на более высоких уровнях
  let a = 10;
  let b = 20;
  // Для вывода переменной в глобальную область видимости
  // используется подобная конструкция
  window.b = b;
})();

console.log(a); // undefined
console.log(b); // 20
```

При вызове функции, ей могут передаваться значения, которыми будут инициализированы параметры. Значения, которые передаются при вызове функции, называются аргументами. Аргументы, указываются через запятую:

```
function foo(a, b) {
  let sum = a + b;
  alert(sum);
}

foo(5, 7);      // 12
foo(5);         // NaN
foo(5, 7, 9);   // 12
```

Когда при вызове функции ей передаётся список аргументов, эти аргументы присваиваются параметрам функции в том порядке, в каком они указаны: первый аргумент присваивается первому параметру, второй аргумент – второму параметру и т. д.

Если число аргументов отличается от числа параметров, то никакой ошибки не происходит. В JavaScript подобная ситуация разрешается следующим образом:

- если при вызове функции ей передаётся больше аргументов, чем задано параметров, "лишние" аргументы просто не присваиваются параметрам;
- если функция имеет больше параметров, чем передано ей аргументов, то параметрам без соответствующих аргументов присваивается значение undefined.

Параметры по умолчанию позволяют задавать формальным параметрам функции значения по умолчанию в случае, если функция вызвана без аргументов, или если параметру явным образом передано значение undefined.

```
function multiply(a, b = 1) {
  return a*b;
}

multiply(5, 2); // 10
multiply(5); // 5
multiply(5, undefined); // 5
```

С помощью инструкции **return** функция может возвратить некоторое значение (результат работы функции) программе, которая её вызвала. Возвращаемое значение передаётся в точку вызова функции.

Инструкция return имеет следующий синтаксис:

```
return выражение;
```

В программу возвращается не само выражение, а результат его вычисления.

Для дальнейшего использования возвращаемого значения, результат выполнения функции можно присвоить, к примеру, переменной:

```
function calc(a) {
  return a * a;
}

let x = calc(5);
alert(x);   // 25
```

Инструкция return может быть расположена в любом месте функции. Как только будет достигнута инструкция return, функция возвращает значение и немедленно завершает своё выполнение. Код, расположенный после инструкции return, будет проигнорирован:

```
function foo() {
  return 1;
  alert('Не выполнится');
}

let x = foo();
alert(x);   // 1
```

Внутри функции можно использовать несколько инструкций return:

```
function check(a, b) {
  if(a > b) return a;
  else return b;
}

alert(check(3, 5));   // 5
```

Возможно использовать **return** и без значения. Это приведёт к немедленному выходу из функции.

```
function showMovie(age) {

  if ( !checkAge(age) ) {
    return;
  }

  alert( "Вам показывается кино" ); // (*)
  // ...
}
```

Если инструкция **return** не указана или не указано возвращаемое значение, то функция вернёт значение **undefined**:

```
function bar() {}
function foo() { return; }

alert(bar());   // undefined. Инструкция return не указана
alert(foo());   // undefined. Возвращаемое значение не указано
```

Функция обратного вызова – это функция, которая передаётся в качестве аргумента другой функции для последующего её вызова.

Т.е. коллбэк — это функция, которая должна быть выполнена после того, как другая функция завершила выполнение (отсюда и название: callback – функция обратного вызова).

```
function foo(callback) { return callback(); }
```

```
foo (function() { alert("Hello!"); });
```

Функции обратного вызова часто используются для продолжения выполнения кода после завершения асинхронной операции – они называются асинхронными обратными вызовами.

Коллбэки позволяют быть уверенными в том, что определенный код не начнет исполнение до того момента, пока другой код не завершит исполнение.

5. Порядок выполнения работы

1. Реализуйте функцию, которая запрашивает два значения и возвращает в качестве результата сумму введенных чисел. В случае, если какое-либо из чисел не введено, функция должна вместо отсутствующего операнда подставить ноль.

2. Реализуйте функцию, которая объявляет и инициализирует (значением, введенным пользователем) локальную переменную, которая хранит число от 1 до 7, и выводит эту переменную в глобальную область видимости. Затем реализуйте вторую функцию, которая, в зависимости от значения глобальной переменной, объявленной в первой функции, выводит название дня недели.

3. Создайте пустой массив. С помощью метода prompt реализуйте добавление элементов в массив. Реализуйте функцию, которая находит среднее значение элементов массива, после чего происходит вызов другой функции, которая, проверяет: имеется в массиве элемент, соответствующий найденному среднему значению.

4. Реализуйте стрелочную функцию, которая принимает в качестве аргумента строку, и возвращает в качестве результата исходную строку в обратном порядке.

5. Реализуйте функцию, которая принимает три аргумента: ФИО пользователя, возраст пользователя, и функцию, которая выводит приветственное сообщение с указанием введенного имени в зависимости от возраста до 18 лет – «Привет», от 18 до 30 – «Как поживаете?», от 30 до 40 – «Добрый день», и больше 40 – «Как себя чувствуете?».

6.* Напишите функцию, которая принимает один строковый аргумент и возвращает другую функцию. Возвращенная функция должна вернуть результат в виде строки следующего вида: повторить строку из возвращающей функции переданное в качестве аргумента количество раз (на реализацию замыкания).

6. Форма отчета о работе

Лабораторная работа № ___

Номер учебной группы _____

Фамилия, инициалы учащегося _____

Дата выполнения работы _____

Тема работы: _____

Цель работы: _____

Оснащение работы: _____

Результат выполнения работы: _____

_____

_____

7. Контрольные вопросы и задания

1. Перечислите способы объявления функции в JavaScript. В чем их отличие?
2. Что представляет собой область видимости переменных?

3. Как можно создать глобальную переменную внутри функции?

4. Для чего в функциях используется инструкция return?

5. Что представляет собой замыкание?

8. Рекомендуемая литература

1. **JAVASCRIPT.RU** [Электронный ресурс] / Современный учебник JavaScript – 2007—2020 Илья Кантор. – Режим доступа: https://learn.javascript.ru. – Дата доступа: 04.03.2020.

2. **Никсон, Р.** Создаем динамические веб-сайты с помощью PHP, MySQL, JavaScript, CSS и HTML5 / Р. Никсон. 4-е изд. – СПб.: Питер, 2018.

3. **Симпсон, К.** ES6 и не только / К. Симпсон. – СПб.: Питер, 2017.

4. **Хавербеке, М.** Выразительный JavaScript. Современное веб-программирование / М. Хавербеке – СПб.: Питер, 2019.