**Лабораторная работа №19**

**Тема работы:** Разработка программ с применением указателя this в объектах и функциях

**Цель работы:** закрепить навыки по использованию указателя контекста this.

**Теоретические сведения**

**Режим use strict**

В JavaScript существует специальный режим, по которому код будет выполнятся по современному стандарту ES5, а не по более старому (это по умолчанию).

Этот режим включается директивой **"use strict";** или **'use strict';** и ставится в начале скрипта:

"use strict";

тут код

Строгий режим можно активировать только для кода функции:

function func() {

"use strict";

};

**Исходный код функции и результат**

Пусть у нас есть какая-то функция. Напоминаю вам, что есть вызвать эту функцию с круглыми скобками - вы увидите **результат** работы этой функции, а если без круглых скобок - ее **исходный код**:

function func() {

return '!';

});

alert(func()); //результат

alert(func); //исходный код функции

**Функция как переменная**

В JavaScript функции - это такие же переменные. К примеру, если у нас есть какая-то функция - мы можем записать ее в другую переменную - и эта функция станет доступна с другим именем:

function func() {

alert('!');

});

var test **=** func;

test(); //выведет '!'

func(); //выведет '!'

Можно также создать безымянную функцию и записать ее в какую-то переменную:

var func **=** function() {

alert('!');

});

func();

Так как функция - это переменная, то невозможно существование переменной и функции с одинаковым именем. В следующем примере функция **func** будет затерта и вместо нее станет строка **'hello'**:

function func() {

return '!';

});

var func **=** 'hello';

func(); //получим ошибку

**Функциональные выражения и объявление функций**

Функцию можно объявить двумя способами: первый способ вы изучали в предыдущих уроках, а второй способ - это сделать безымянную функцию и записать ее в какую-либо переменную:

//Первый способ:

function func() {

alert('!');

});

//Второй способ:

var func **=** function() {

alert('!');

});

Первый способ называется **Function Declaration** (*объявление функции*), а второй - **Function Expression** (*функциональное выражение*):

//Function Declaration:

function func() {

alert('!');

});

//Function Expression:

var func **=** function() {

alert('!');

});

По сути это одно и то же, но есть существенная разница: функции, объявленные как Function Declaration, создаются **до выполнения кода**. Поэтому их можно вызвать до объявления, например:

func(); //выведет '!'

function func() {

alert('!');

});

А функциональные выражения создаются в момент выполнения кода и недоступны выше. Поэтому такой код выдаст ошибку:

func(); //ошибка, такой функции еще нет!

var func **=** function () {

alert('!');

});

**Функциональные выражения**

Функциональные выражения достаточно удобная штука - можно, например, записать в переменную одну из функций, в зависимости от условия:

"use strict";

var bool **=** true;

if (bool) {

var func **=** function () {

alert('!');

}

} else {

var func **=** function () {

alert('!!');

}

}

func();

**Особенности Function Declaration при use strict**

Function Declaration при **use strict** видны только внутри блока, в котором объявлены. В данном случае функция **func** объявлена внутри ифа, но не будет видна снаружи:

"use strict";

var bool **=** true;

if (bool) {

function func() {

alert('!');

}

}

func(); //выдаст ошибку - тут функция не видна!

Без use strict все будет нормально.

**Передача функции по ссылке**

Функции в JavaScript - это объекты. А объекты в JavaScript копируются *по ссылке* - это значит, что если у нас есть две переменные с одной и той же функцией - в них не лежит копия этой функции, а обе эти переменные ссылаются на одну и ту же функцию:

function func() {

alert('!');

});

var test **=** func; //И test и func указывают на одну и ту же функцию

**Анонимные функции**

Функции, не имеющие имени, называются *анонимными*. Такие функции бывают полезны в тех случаях, когда имя функции нам без надобности и нет смысла тратить время на придумывание этого имени.

Пример: пусть у нас есть переменная **elem**, в которой лежит ссылка на какой-то элемент. Привяжем в качестве события **onclick** анонимную функцию:

elem.onclick **=** function() {

alert('!');

});

Тоже самое при **addEventListener**:

elem.addEventListener('click', function() {

alert('!');

});

Впрочем, если имя функции вам нужно, например, для того, чтобы открепить ее через **removeEventListener** - можно дать и имя:

elem.addEventListener('click', function show() {

this.removeEventListener('click', show);

});

Это имя будет доступно только внутри самой функции - и нигде извне.

**Функция как параметр другой функции**

Пусть у нас даны 2 функции:

var get1 **=** function() {

return 1;

}

var get2 **=** function() {

return 2;

}

Сделаем третью функцию **go**, которая будет ожидать, что первым и вторым параметром ей передаются функции, которые возвращают какие-то числа.

Наша функция **go** запишет первую функцию в переменную **func1**, а вторую - в **func2**, затем просуммирует числа, возвращаемые этими функциями и выведет их на экран:

function go(func1, func2) {

alert(func1() **+** func2());

}

go(get1, get2); //выведет 3

Получается, что в функцию можно передавать параметрами другие функции, записывать их в переменные и использовать внутри.

Рассмотрим еще пример: сейчас в функцию и **go** будем передавать параметром разные функции - и будем видеть разный результат:

function show1() {

alert('!');

});

function show2() {

alert('!!');

});

function go(func) {

func();

});

go(show1); //выведет '!'

go(show2); //выведет '!!'

А можем вообще использовать анонимную функцию, создаваемую в момент передачи параметра:

function go(func) {

func();

});

go(function() {

alert('!');

});

**Функция в функции**

Одну функцию можно объявить внутри другой. В этом случае внутренняя функция **не будет доступна извне**:

function func() {

function test() {

alert('!');

};

});

test(); //выдаст ошибку

Зачем это нужно: внутренняя функция может быть использована как вспомогательная и это удобно, что она не доступна извне - мы не захламляем глобальную область видимости лишними функциями.

**Функция, возвращающая функцию**

Функция может возвращать другую функцию, например так:

function func() {

return function() {

return '!';

};

};

В этом случае, если мы посмотрим результат работы внешней функции - мы увидим исходный код внутренней функции:

function func() {

return function() {

return '!';

};

}

alert( func() ); //увидим код внутренней функции

Чтобы увидеть результат работы внутренней функции - нужно вызвать внешнюю функции с двумя круглыми скобками:

function func() {

return function() {

return '!';

};

}

alert( func()() ); //увидим '!'

Могут быть и такие вызовы функций: **func()()()** и **func()()()()** - и так далее до бесконечности. Для этого нужно, чтобы внутренняя функция тоже возвращала функцию, та - еще одну и так далее.

**Область видимости переменных**

Давайте вспомним про область видимости переменных. Напомню, что внешняя переменная видна внутри функции:

var num **=** 5;

function test() {

//Внешняя переменная num видна внутри функции:

alert(num);

};

test(); //выведет 5

То же самое будет, если у нас функция содержит внутри другую функцию - переменные внешней функции видны во внутренней:

function func() {

var num **=** 5;

function test() {

//Внешняя переменная num видна внутри функции:

alert(num);

};

test();

};

func(); //выведет 5

**Замыкания**

Пусть у нас есть переменная **num**, определенная снаружи функции:

var num **=** 0;

function func() {

num**++**;

return num;

}

Если вызвать нашу функцию - то она сначала увеличит переменную **num** на единицу, а затем вернет новое значение **num**:

var num **=** 0;

function func() {

num**++**;

return num;

}

alert(func()); //выведет 1

Если вызвать нашу функцию несколько раз - каждый раз она будет выводить на единицу больше, так как каждый вызов **func** приводит к увеличению внешней переменной **num**:

var num **=** 0;

function func() {

num**++**;

return num;

}

alert(func()); //выведет 1

alert(func()); //выведет 2

alert(func()); //выведет 3

alert(func()); //выведет 4

alert(func()); //выведет 5

В нашем примере есть недостаток - переменная **num** видна во всем скрипте. Любая другая функция может случайно затереть **num** - и вся наша конструкция перестанет работать правильно. Давайте реализуем более продвинутый счетчик, избавленный от этого недостатка.

Обернем всю нашу конструкцию в функцию (назовем ее **createCounter**), а функцию **func**, которая у нас была ранее, сделаем анонимной и сделаем так, чтобы новая функция createCounter возвращала эту анонимную функцию:

function createCounter() {

var num **=** 0;

return function() {

num**++**;

return num;

};

}

var counter **=** createCounter();

Рассмотрим подробнее, что тут происходит: переменная num является локальной внутри функции createCounter, но при этом она доступна в анонимной функции (это мы видели в предыдущих примерах). В строчке **var counter = createCounter()** анонимная функция запишется в переменную **counter**. Получится, что у нас далее есть функция **counter**, внутри которой доступна переменная num из **createCounter**.

Давайте убедимся в этом:

function createCounter() {

var num **=** 0; return function() {

num**++**;

return num;

};

}

var counter **=** createCounter();

alert(counter()); //выведет 1

Давайте теперь запустим функцию **counter** несколько раз - мы увидим, что каждый раз она будет выводить на единицу больше:

function createCounter() {

var num **=** 0;

return function() {

num**++**;

return num;

};

};

var counter **=** createCounter();

alert(counter()); //выведет 1

alert(counter()); //выведет 2

alert(counter()); //выведет 3

Еще раз: преимуществом такого подхода является то, что переменная **num** не видна снаружи **createCounter** и ее никто не сможет случайно затереть. Снаружи она не видна, но доступ к ней есть - через функцию **counter**, но только через нее.

Такая штука называется *замыкание*. **Замыкание** - это функция со всеми доступными внешними переменными (типа num в нашем случае). Эти переменные называются *лексическим окружением* функции.

Давайте вернемся назад к счетчику - осталось самое интересное: если **createCounter** вызвать несколько раз, записав результат в разные переменные - каждая из них станет **независимым** счетчиком.

В следующем примере у нас есть 2 функции: **counter1** и **counter2** - и они работают совершенно не мешая друг другу (получается, что у каждой из них своя переменная **num**):

function createCounter() {

var num **=** 0;

return function() {

num**++**;

return num;

};

};

var counter1 **=** createCounter();

alert(counter1()); //выведет 1

alert(counter1()); //выведет 2

alert(counter1()); //выведет 3

var counter2 **=** createCounter();

alert(counter2()); //выведет 1

alert(counter2()); //выведет 2

alert(counter2()); //выведет 3

**Вызов функции на месте**

При программировании иногда возникает задача сделать анонимную функцию и сразу ее вызвать в момент создания.

Зачем это нужно: те переменные и вспомогательные функции, которые мы объявим внутри нашей анонимной функции останутся локальными и не будут засорять своими именами глобальную область видимости.

Для вызова функции на месте хотелось бы сделать что-то в таком роде:

function() {

alert('!');

}();

Однако, это не будет работать, так как на месте разрешен вызов только функциональных выражений. Поэтому необходимо сделать из нашей функции выражение, например, поставив перед ней плюс, такое уже будет работать:

**+**function() {

alert('!');

}();

Так тоже будет работать:

**!**function() {

alert('!');

}();

Но более принято брать такие функции в круглые скобки, вот так:

(function() {

alert('!');

}());

Давайте посмотрим, что же нам дает вызов функции на месте: сделаем внутри такой функции переменную **message**. Наша переменная будет доступна только внутри функции, и не будет доступна снаружи:

(function() {

var message **=** '!';

alert(message);

}());

alert(message); //здесь message недоступна

Иногда во избежания ошибок в начале ставится точка с запятой (ошибки могут возникнуть при сжатии файла минимизатором):

;(function() {

alert('!');

}());

**Создание модулей**

https://learn.javascript.ru/closures-module

файл module.js:

var message **=** 'сообщение модуля';

function showMessage() {

alert(message);

}

Наш код

<script>

var message **=** 'наше сообщение';

</script>

<script src="module.js"></script>

<script>

alert(message); //выведет 'сообщение модуля', а не 'наше сообщение'

</script>

Допилим код модуля

(function() {

var message **=** 'сообщение модуля';

function showMessage() {

alert(message);

}

}());

вот теперь проблем не будет

экспорт функций наружу

(function() {

var message **=** 'сообщение модуля';

function showMessage() {

alert(message);

}

window.showMessage **=** showMessage;}());

экспорт как в библиотеке

(function() {

var message **=** 'сообщение модуля';

var $ **=** { showMessage: function (message) {

alert(message);

}

}

window.$ **=** showMessage;}());

**Замыкания и вызов на месте**

Замыкания и вызов функции на месте можно комбинировать. В следующем примере внешняя анонимная функция выполнится на месте и вернет внутреннюю анонимную функцию - она запишется в переменную **func**. А переменная **counter** попадет в замыкание:

var func **=** (function() {

var counter **=** 0;

//Эта функция запишется в func:

return function() {

counter**++**;

return counter;

}

})();

**Применение замыканий**

Пусть у нас дано несколько кнопочек. Давайте сделаем так, чтобы каждая кнопочка при нажатии увеличивала свое значение на единицу. Реализуем задачу на замыкании:

<button>0</button><button>0</button><button>0</button>

var buttons **=** document.getElementsByTagName('button');

for (var i **=** 0; i **<** buttons.length; i**++**) {

buttons[i].onclick **=** (function() {

var counter **=** 0;

//Эта функция привяжется к onclick:

return function() {

counter**++**; //берется из замыкания - для каждой кнопки своя переменная

this.innerHTML **=** counter;

}

})();

}

**Порядок выполнения работы**

### На работу с функциями

1. Если переменная **a** больше нуля - то в ggg запишем функцию, которая выводит один !, иначе запишем функцию, которая выводит два !.
2. Функция ggg принимает 2 параметра: число и анонимную функцию, которая возводит число в квадрат. Возведите число в 4-тую степень с помощью ggg.
3. Функция ggg принимает 2 параметра: анонимную функцию, которая возвращает 3 и анонимную функцию, которая возвращает 4. Верните результатом функции ggg сумму 3 и 4.
4. Дана функция ggg. Она требует первым параметром число, вторым функцию, которая возводит в квадрат, а третьим параметром функцию, которая возводит в куб. Эти функции есть как Function Declaration - kvadrat, kub. Пусть функция ggg вернет сумму квадрата и куба числа.
5. Сделайте функцию **each**, которая первым параметром принимает массив, а вторым - функцию, которая применится к каждому элементу массива. Функция each должна вернуть измененный массив.
6. Сделайте функцию **each**, которая первым параметром принимает массив, а вторым - массив функций, которые по очереди применятся к каждому элементу массива: к первому элементу массива - первая функция, ко второму - вторая и так далее пока функции в массиве не закончатся, после этого возьмется первая функция, вторая и так далее по кругу.

### На замыкания

1. Сделайте функцию, которая считает и выводит количество своих вызовов.

func(); //выведет 1

func(); //выведет 2

func(); //выведет 3

func(); //выведет 4

1. Даны кнопки. Привяжите к каждой кнопке событие по клику, которое будет считать количество нажатий по кнопке и выводить его в текст кнопки. Количество нажатий для каждой кнопки должно хранится в замыкании.
2. Дан массив цветов. Даны абзацы. По первому нажатию на абзац он должен покраситься в первый цвет из массива, по второму нажатию - во второй и так далее. Все абзацы работают независимо.
3. Даны кнопки. Каждая кнопка по нажатию на нее выводить следующее число Фибоначчи. Кнопки работают независимо. Решить через замыкания.
4. Даны инпуты. Сделайте так, чтобы каждый инпут хранил историю своих изменений. Каждый инпут свою. Изменением считается событие onchange. История должна хранится в замыкании. Над каждым инпутом должны быть стрелочки назад и вперед, с помощью которых можно передвигаться по истории.
5. Сделайте функцию, каждый вызов который будет генерировать случайные числа от 1 до 100, но так, чтобы они не повторялись, пока не будут перебраны все числа из этого промежутка. Решите задачу через замыкания - в замыкании должен хранится массив чисел, которые уже были сгенерированы функцией.