Урок 8

Анонимные функции. Замыкания.

Знакомство с функциональными выражениями. Принципы работы функций. Замыкания.

Введение

В течение курса Вы могли заметить, что несмотря на то, что мы работали как на уровне функционального программирования, так и с ООП. Так или иначе, каждый из подходов использовал фундаментальную сущность function. И на этом занятии мы ещё ближе познакомимся со строением, поведением и применениями этой сущности.

Функции call(), apply() и bind() в JavaScript

В языке JavaScript функции, по сути, являются объектами. Как и положено объектам, они имеют свои собственные методы. Среди них есть очень полезные - apply(), call() и bind(). Методы call() и apply() почти идентичны и применяются для явной установки значения this. Метод apply() также может использоваться для изменения количества параметров функции, а метод bind() используется как для установки значения this, так и для преобразования функции со многими аргументами в набор функций с одним аргументом (т.н. каррирование).

Метод bind() позволяет нам явно задать объект, ссылка на который будет значением this при вызове функции. Функция не всегда вызывается в контексте того объекта, в котором она описана (например, при эффекте всплытия). Таким образом, значение this в функции может меняться, что может привести к неожиданным последствиям.

```
var users = {
  data: [
     {name: 'John Smith'},
     {name: 'Ellen Simons'}
  ],

   showFirst: function (event) {
      console.log(this.data[0].name);
   }
}

$('button').click(users.showFirst); // this.data is undefined
```

В данном примере при нажатии на кнопку вызовется метод showFirst. Он принимает на вход некий event. Однако при событии нажатия на кнопку в роли this будет выступать сама кнопка. Давайте решим эту проблему при помощи bind()

```
$("button").click(users.showFirst.bind(users));
```

Обратите внимание на то, что bind не будет работать в браузерах ниже IE9.

В JavaScript можно передавать функции как параметры, возвращать их в качестве результата, присваивать переменным. На этом основан такой приём как заимствование методов. Т.е. если у объекта нет метода, то его можно позаимствовать у другого объекта. Например:

```
var cars = {
   data:[
      { name: 'Mitzubisi Lancer' },
      { name: 'Chevrolet Impala' }
   ]
}

cars.showFirst = users.showFirst.bind(cars);
cars.showFirst();
```

У объекта cars изначально нет метода showFirst. Однако мы можем создать его при помощи bind, не описывая заново, но избегая проблем с изменением значения this.

При каррировании bind применяется следующим образом:

```
// Определим функцию от трех переменных
function greet(gender, age, name) {
    // if a male, use Mr., else use Ms.
    var salutation = gender === "male" ? "Mr. " : "Ms. ";
    if (age > 25) {
        return "Hello, " + salutation + name + ".";
    }
    else {
        return "Hey, " + name + ".";
    }
}
// С помощью bind() мы можем получать функции от меньшего числа переменных
var greetAnAdultMale = greet.bind(null, "male", 45);
greetAnAdultMale("John Hartlove");  // "Hello, Mr. John Hartlove."
var greetAYoungster = greet.bind(null, "", 16);
greetAYoungster("Alex");                // "Hey, Alex."
greetAYoungster("Emma Waterloo"); // "Hey, Emma Waterloo."
```

Методы apply() и call() также являются важными методами объекта Function. Как мы уже узнали выше, они позволяют явно установить значение this для функции.

```
var avgScore = "global avgScore";
function avg(arrayOfScores) {
    var sumOfScores = arrayOfScores.reduce(function(prev, cur, index, array) {
        return prev + cur;
    });
    this.avgScore = sumOfScores / arrayOfScores.length;
}
var gameController = {
    scores  :[20, 34, 55, 46, 77],
    avgScore:null
}
avg(gameController.scores);
console.log(window.avgScore);           // 46.4
console.log(gameController.avgScore); // null
avgScore = "global avgScore";
avg.call(gameController, gameController.scores);
console.log(window.avgScore);           //global avgScore
console.log(gameController.avgScore); // 46.4
```

В данном примере первый параметр метода call() используется в качестве значения this, а остальные — передаются функции как параметры. Т.е. мы говорим «Вызови (call) функцию avg с переменной gameController как this, передав в качестве параметра gameController.scores».
Метод apply() похож на метод call() - первый параметр также используется в качестве значения this, а остальные передаются в виде массива.

```
var clientData = {
    id: 094545,
    fullName: "Not Set",
    setUserName: function(firstName, lastName)  {
        this.fullName = firstName + " " + lastName;
    }
}
function getUserInput(firstName, lastName, callback, callbackObj) {
    callback.apply(callbackObj, [firstName, lastName]);
}
```

Поведение function

Ключевое слово function может применяться двумя способами:

```
function myFunction() {
   for (var i = 0; i < num; i++) {
      console.log("Ping");
   }
}
```

```
var mf = function() {
   for (var i = 0; i < num; i++) {
      console.log("Pong");
   }
};
mf();
```

В первом случае мы объявляем функцию. Объявление при этом генерирует функцию с указанным нами именем, которое впоследствии может применяться для ссылок и вызова функции. Во втором случае мы задаём функцию внутри команды (как будто присваиваем переменную). Такой подход называется функциональным выражением. И если в первом случае функция имеет строго заданное имя, то во втором – нет. Но при этом результат работы функции будет сразу же присвоен переменной mf. Этим результатом будет ссылка на нашу функцию. И вызываем мы её через запись mf().

Конечно, и объявления функций, и функциональные выражения достаточно похожи. Но зачем тогда в JS реализовано два подхода работы с функциями? На самом деле между ними есть кардинальные различия. Для того, чтобы понять их, нужно рассмотреть действия браузера при разборе и выполнении кода.
В процессе чтения кода браузер в первую очередь производит поиск **объявлений** функций. В момент обнаружения происходит создание функции и присвоение ссылки на функцию некой переменной, у которой имя будет совпадать с именем функции. После того, как все объявления будут найдены и обработаны, работа с кодом переключается на выполнение последовательных инструкций от начала кода. Т.е. браузер уже не читает код, а выполняет его. И именно в этом кроется главное отличие объявления функций от функциональных выражений.
Функциональные выражения попадают в работу только после начала работы с переменными. Таким образом, функция создается при выполнении кода. Кроме того, отличием является именование функций. При использовании функционального выражения имя функции, как правило, не указывается, а функция может присваиваться переменной или использоваться другим способом.

Функция – это тоже значение

В течение курса мы расценивали функции в виде сущностей, которые можно вызвать. Но необходимо знать, что функции при этом также можно рассматривать и как значения. Как мы уже выяснили, значение такого рода - это ссылка на функцию. На самом деле, неважно, как определена функция – в

результате все равно получается ссылка на эту функцию.

Таким образом, совершенно естественным представляется присваивание функций переменным.

```
var func1 = function() {
  for (var i = 0; i < num; i++) {
    console.log("Ping");
  }
}
var func2 = func1;
func2();
```

```
function myFunction(){
  for (var i = 0; i < num; i++) {
    console.log("Pong");
  }
}
var myNewFunction = myFunction;
myNewFunction();
```

Как видите, ссылка на функцию всегда остается ссылкой вне зависимости от того, каким образом она была создана – через объявление функции или через функциональное выражение.

В информатике есть понятие **первоклассного значения.** Это значение, с котором можно выполнять следующие операции:

- Присвоение значения переменной или свойству;
- Передача значения функции;
- Возврат значения из функции.

С этими значениями могут выполняться те же операции, что и с любыми другими значениями, включая возможность присваивания переменной, передачи в аргументе и возвращения из функции. И в отличие от многих классических языков программирования, в JavaScript функция является этим самым первоклассным значением. Т.е., помимо уже известного нам присваивания функции переменной, Вы можете:

- передать функцию в функцию, чтобы вызвать её внутри;
- вернуть из функции функцию.

Именно по этому принципу работают callback-функции. Мы говорим верхнеуровневой функции, что нужно выполнить в определённой ситуации, причём она сама не знает, что же будет происходить.

Помимо всего прочего, поскольку объявления функций и функциональные выражения обрабатываются в разное время, объявления функций могут определяться совершенно в любой точке кода.

При работе с функциональными выражениями все обстоит по-другому, ведь они, по сути, не определены до момента их обработки. Даже если функциональное выражение присваивается глобальной переменной, у Вас не выйдет применять эту переменную для вызова до того, как функция будет определена.

Разберём пример:

```
declaration();
 function declaration() {
    console.log("Hi, I'm a function declaration!");
 }
  // Вернёт ошибку: Uncaught TypeError: undefined is not a function
 expression();
  var expression = function () {
    console.log("Hi, I'm a function expression!");
 }
```

Как видно из примера выше, если вызов function expression расположен выше, чем сама функция, то происходит ошибка, однако вызов finction declaration происходит без ошибок. Однако начинающие разработчики используют функциональные выражения слишком часто. Их стоит применять лишь тогда, когда выражение функции является более полезным:

- В качестве замыканий (рассмотрим далее);
- Как аргумент для другой функции (callback, например);
- В случае немедленно вызываемой функции (IIFE).

Анонимные функции

Анонимными функциями называются те функции, которые определяются без указания имени. Зная два известных нам подхода задания функции в JS, мы можем сказать, что так или иначе, у функции существует имя. Однако, если мы определяем функцию с использованием функционального выражения, присваивать функции имя совершенно не обязательно.
Преимущество анонимных функций в том, что в определённых ситуациях они делают код более лаконичным, четким, удобочитаемым и эффективным, зачастую упрощая его сопровождение.
Рассмотрим пример перехода к анонимным функциям. В курсе мы уже встречались с функцией, которая вызывается при загрузке страницы.

```
function handler() { alert("Страница загружена"); }
window.onload = handler;
```

Мы точно знаем, что эта функция не будет выполняться повторно в процессе работы пользователя со страницей. Событие загрузки страницы уникально, поэтому функция выполнится единожды. При этом

свойству window.onload мы присваиваем ссылку на функцию, размещённую под именем handler. С понимание того, что функция выполняется только один раз, резервирование памяти под неё (а именно это мы делаем, когда задаём переменную, пусть и неявно) излишне. Всё это приводит к перерасходу ресурсов.
Применение в таком случае анонимной функции упростит наш код. Такая функция будет представлять собой уже знакомое нам функциональное выражение, но без имени, т.е. мы не будем выделять область памяти, в которой будет хранится ссылка на функцию. На практике это выглядит следующим образом:

```
window.onload = function() { alert("Страница загружена"); };
```

Давайте попробуем решить задачку посложнее. Ниже приведён код. Подумайте, какие места в нём можно оптимизировать, заменив объявления функций или функциональные выражения на анонимные функции?

```
window.onload = init;
var cookies = {
  instructions: "Preheat oven to 350...",
  bake: function(time) {
    console.log("Baking the cookies.");
    setTimeout(done, time);
  }
};
function init() {
  var button = document.getElementById("bake");
  button.onclick = handleButton;
}
function handleButton() {
  console.log("Time to bake the cookies.");
  cookies.bake(2500);
}
function done() {
  alert("Cookies are ready, take them out to cool.");
  console.log("Cooling the cookies.");
  var cool = function() {
    alert("Cookies are cool, time to eat!");
  };
  setTimeout(cool, 1000);
}
```

Теперь рассмотрим правильный ответ

```
window.onload = function() {
  var button = document.getElementById("bake");
  button.onclick = function() {
    console.log("Time to bake the cookies.");
    cookies.bake(2500);
  };
};
var cookies = {
  instructions: "Preheat oven to 350...",
  bake: function(time) {
    console.log("Baking the cookies.");
    setTimeout(done, time);
  }
};
function done() {
  alert("Cookies are ready, take them out to cool.");
  console.log("Cooling the cookies.");
  setTimeout(function() {
    alert("Cookies are cool, time to eat!");
  }, 1000);
}
```

Код переработан с тем, чтобы создать два анонимных функциональных выражения — для функции init и для функции handleButton. Теперь функциональное выражение назначается свойству window.onload, другое выражение назначается свойству button.onclick. Также мы заменили функциональное выражение cool, переместив его содержимое в callback-функцию для setTimeout.

Что мешает нам определять функции внутри других функций? Ничего! Это значит, что объявления функций или функциональные выражения мы можем использовать только внутри этих функций. При этом различия между функцией, определяемой на верхнем уровне кода, и функцией, определяемой внутри другой функции, остаются только в области действия имен, также как это происходит с переменными - размещение функции внутри другой функции будет влиять на видимость этой функции внутри кода.

Функции, которые определяются на верхнем уровне кода, имеют глобальную область видимости, тогда как определяемые внутри других функций функции имеют только локальную видимость.

```
var migrating = true;
  var fly = function(num) {
  var sound = "Flying";
  function wingFlapper() {
    console.log(sound);
  }
  for (var i = 0; i < num; i++) {
    wingFlapper();
  }
};
function quack(num) {
  var sound = "Quack";
  var quacker = function() {
    console.log(sound);
  };
  for (var i = 0; i < num; i++) {
    quacker();
  }
}
if (migrating) {
  quack(4);
  fly(4);
}
```

Для функций, которые мы определяем внутри других функций, для обращения к функции действуют те же правила, что и на верхнем уровне. Звучит запутанно, но внутри функции при определении вложенной функции через объявления сама вложенная функция определена только внутри тела внешней функции, но в любой его точке. С другой стороны, если создание вложенной функции идёт через функциональное выражение, то вложенная функция определена только после обработки функционального выражения.
Самый главный вывод, который необходимо запомнить, это то, что функции в JavaScript всегда выполняются в же окружении, где они были определены. Для указания происхождения переменной внутри функции просто проведите её поиск во внешних функциях, следуя от самой глубоко вложенной функции к самой верхнеуровневой.

Замыкания

Говоря о JavaScript как о современном языке, нельзя не сказать о том, что он поддерживает замыкания. Сами по себе замыкания – весьма сложная тема, но на самом деле мы уже использовали их в процессе обучения, сами того не зная.

Замыкание - это функция вместе с сопутствующим окружением. Для того, чтобы понять это определение, нужно понять концепцию «замкнутости» функции.

В телах наших функций часто присутствуют как локальные переменные (включая все параметры функции), так и переменные, определённые вне функции – свободные переменные. Такие свободные переменные не связываются ни с какими значениями. Таким образом, если мы определим значения для всех свободных переменных, определённых для функции, то сама функция станет замкнутой. А если взять функцию вместе с ее окружением, получится замыкание.

Замыкания являются довольно мощным и распространенным инструментом функциональных языков программирования. Давайте научимся применять замыкания на практике. Для примера возьмём классический счётчик.

```
var count = 0;
function counter() {
   count = count + 1;
   return count;
}
```

Вне функции определена свободная переменная. Функция увеличивает её с каждым вызовом. При разработке в команде это может создать проблемы, т.к. при задании одинаковых имен случится конфликт.
Тогда нужно реализовать счётчик с полностью локальной и защищённой переменной. Тогда она точно не будет ни с кем конфликтовать, сохраняя при этом функционал.

Для перехода на замыкания мы воспользуемся следующим алгоритмом:

1. Мы определяем функцию makeCounter, внутри которой создаем функцию counter, возвращая ее вместе с окружением, содержащим свободную переменную count. Т.е., она создаёт замыкание. Функция, возвращенная из makeCounter, сохраняется в переменной doCount.
2. При необходимости вызова счётчика мы вызываем функцию doCount. Это приводит к выполнению тела функции counter.
3. При обращении к переменной count браузер попытается найти её в окружении. После этого новое значение сохраняется в окружении, возвращая при этом его в точку вызова doCount.

Обратите внимание на то, что мы как раз создаём новые функции внутри функций, чего не делали до этого. Надо понимать, что область видимости у них будет соответствующая.

```
function makeCounter() {
var count = 0;
  function counter() {
    count = count + 1;
    return count;
  }
  return counter;
}
var doCount = makeCounter();
console.log(doCount());
console.log(doCount());
console.log(doCount());
```

При этом, возвращение функции из функции - это далеко не единственный подход при создании замыканий. На самом деле подобные конструкции создаются в любом месте, где появляется ссылка на функцию, имеющую свободные переменные. Такая функция выполняется вне контекста, в котором она была создана. Также замыкания могут создаваться при помощи передачи функций при вызове функций в качестве аргументов. В этом случае передаваемая функция будет выполняться в контексте, который отличается от контекста ее создания.

```
function startNewTimer(userMessage, millis) {
  setTimeout(function() {
    alert(userMessage);
  }, millis);
}
startNewTimer("Работа завершена", 1000);
```

В приведенном выше примере функциональное выражение со свободной переменой userMessage, передается в стандартную функцию setTimeout. При этом функциональное выражение обрабатывается для получения ссылки на функцию, которая затем передается setTimeout. При этом то, что сохраняет метод setTimeout, является функцией с окружением, т.е. замыкание. После этого по прошествии 1000 миллисекунд функция вызывается.

Домашнее задание

1. Для практикума из занятия 7 продумать, где можно применить замыкания
2. Не выполняя кода, ответить, что выведет браузер и почему
    a.

```
if (!("a" in window)) {
    var a = 1;
}
alert(a);
```

b.

```
var b = function a(x) {
    x && a(--x);
};
alert(a);
```

c.

```
function a(x) {
    return x * 2;
}
var a;
alert(a);
```

d.

```
function b(x, y, a) {
    arguments[2] = 10;
    alert(a);
}
b(1, 2, 3);
```

e. *

```
function a() {
    alert(this);
}
a.call(null);
```

Дополнительные материалы

1. https://habrahabr.ru/post/38642/ - хабр о замыканиях
2. https://habrahabr.ru/post/199456/ - bind, call и apply в JS

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. «JavaScript. Подробное руководство» - Дэвид Флэнаган
2. «Изучаем программирование на JavaScript» - Эрик Фримен, Элизабет Робсон