Урок 8

Анонимные функции. Замыкания

Знакомство с функциональными выражениями. Принципы работы функций. Замыкания.

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

В течение курса мы работали как на уровне функционального программирования, так и с ООП. Каждый из подходов использовал фундаментальную сущность **function**. Познакомимся ближе с ее строением, поведением и применениями.

В JavaScript функции, по сути, являются объектами. Как и положено объектам, они имеют 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 можно применять для изменения количества параметров функции, а **bind()** — для каррирования, то есть преобразования функции со многими аргументами в набор функций с одним аргументом.

Метод **bind()** позволяет 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. Так что значение **this** в функции может меняться, и последствия могут быть неожиданными.

```
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 **event**. Однако при событии нажатия на кнопку в роли **this** будет выступать сама кнопка. Решим эту проблему при помощи **bind()**:

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

Обратите внимание, 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 прием заимствования методов. Если у объекта нет метода, его можно позаимствовать у другого:

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

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

У объекта **cars** изначально нет метода **showFirst**. Но можно 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()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()** тоже важны для объекта **Function**. 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. Здесь мы диктуем: «Вызови (**call**) функцию **avg** с переменной **gameController** как **this**, передав в качестве параметра **gameController.scores**».

Метод **apply()** похож на **call()**: первый параметр тоже 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 объявляем функцию. Объявление при этом генерирует функцию с указанным нами именем, которое впоследствии может применяться для ссылок и вызова функции. Во втором случае задаем 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 — ссылка на нашу функцию — будет сразу же присвоена переменной **mf**. Вызываем ее через запись **mf()**.

Объявления функций и функциональные выражения похожи, но между ними есть кардинальные различия. Чтобы понять их, нужно рассмотреть действия браузера при разборе и выполнении кода.

В процессе чтения кода браузер в первую очередь производит поиск **объявлений** функций. Когда обнаруживает, создается функция и переменной, у которой имя будет совпадать с именем функции, присваивается ссылка на функцию. Когда все объявления будут найдены и обработаны, работа с кодом переключится на выполнение последовательных инструкций от начала кода. Браузер уже не читает код, а выполняет его. Это главное отличие объявления функций от функциональных выражений.

Функциональные выражения попадают в работу только после начала работы с переменными. Функция создается при выполнении кода. Отличие есть и в именовании функций. При использовании функционального выражения имя функции, как правило, не указывается, а функция может присваиваться переменной или использоваться другим способом.

Функция — это тоже значение

На протяжении курса мы расценивали функции как сущности, которые можно вызвать. Но необходимо знать, что функции можно рассматривать и как значения — ссылки на функции. Неважно, как определена функция — в результате все равно получается ссылка на нее.

Поэтому присваивание функций переменным вполне естественно.

```
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();
```

Ссылка на функцию всегда остается ссылкой — вне зависимости от того, каким образом она была создана: через объявление функции или 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, включая присваивание переменной, передачу в аргументе и возвращение из функции. И в отличие от многих классических языков программирования, в JavaScript функция является первоклассным значением. Помимо присваивания функции переменной, можно:

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

Именно по этому принципу работают callback-функции. Мы говорим о верхнеуровневой функции, которую нужно выполнить в определенной ситуации (причем она сама не знает, что же будет происходить).

Поскольку объявления функций и функциональные выражения обрабатываются в разное время, первые могут определяться в любой точке кода.

При работе с функциональными выражениями все обстоит иначе, ведь они не определены до момента обработки. Даже если функциональное выражение присваивается глобальной переменной, не получится применить ее для вызова до того, как функция будет определена.

Разберем пример:

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

Если вызов **function expression** расположен выше, чем сама функция, то происходит ошибка, но вызов **function declaration** происходит без нее. Однако начинающие разработчики используют функциональные выражения слишком часто. Их стоит применять, только когда выражение функции является более полезным:

- в качестве замыканий;
- как аргумент для другой функции (**callback**, например);
- в случае немедленно вызываемой функции (**IIFE**).

Анонимные функции

Анонимными называются функции, которые определяются без указания имени. Зная два подхода задать функцию в JS, мы можем сказать, что так или иначе у функции есть имя. Но если определять функцию с использованием функционального выражения, присваивать ей имя не обязательно.

Преимущество анонимных функций в том, что в определенных 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** присваиваем ссылку на функцию, размещенную под именем **handler**. С пониманием того, что функция выполняется только один раз, резервирование памяти под нее излишне (а именно это мы делаем, когда задаем переменную, пусть и неявно). Все это приводит к перерасходу ресурсов.

В таком случае применение анонимной функции упростит код. Она будет представлять собой функциональное выражение, но без имени. То есть мы не будем выделять область памяти, в которой будет храниться ссылка на функцию. На практике это выглядит так:

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

Решить задачу посложнее. Рассмотрим код и подумаем, какие места в нем 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 проанализируем 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, чтобы создать два анонимных функциональных выражения — для функций **init** и

handleButton. Теперь одно функциональное выражение назначается свойству **window.onload**, другое — **button.onclick**. Также мы заменили функциональное выражение **cool**, переместив его содержимое в callback-функцию для **setTimeout**.

Ничего не мешает определять одни функции внутри других. Это значит, что объявления функций или функциональные выражения можно использовать только внутри этих функций. При этом различия между функцией, определяемой на верхнем уровне кода, и той, что определяется внутри другой функции, остаются только в области действия имен. Это происходит так же, как с переменными — размещение одной функции внутри другой будет влиять на ее видимость внутри кода.

Функции, которые определяются на верхнем уровне кода, имеют глобальную область видимости, а определяемые внутри других функций — только локальную.

```
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 обращения к функции, определенной внутри другой, действуют те же правила, что и на верхнем уровне. Внутри функции — при определении вложенной функции через объявления — сама вложенная функция определена не только внутри тела внешней, но в любой его точке. С другой стороны, если создание вложенной функции идет через функциональное выражение, то она будет определена только после его обработки.

Вывод: функции в JavaScript всегда выполняются в том же окружении, где были определены. Чтобы указать происхождение переменной внутри функции, просто проведите ее поиск во внешних функциях, следуя от самой глубоко вложенной к верхнеуровневой.

Замыкания

JavaScript — современный язык и поддерживает замыкания. Это сложная тема, но на самом деле мы уже использовали их в этом курсе.

Замыкание — это функция с сопутствующим окружением. Чтобы понять это определение, нужно

вникнуть в концепцию «замкнутости» функции.

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

Замыкания — мощный и распространенный инструмент функциональных языков программирования. Научимся применять замыкания на практике, и для примера возьмем классический счетчик:

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

Вне функции определена свободная переменная. Функция увеличивает ее с каждым вызовом. При разработке в команде это может создать проблемы, так как при задании одинаковых имен случится конфликт.

Тогда нужно реализовать счетчик с полностью локальной и защищенной переменной. Она точно не будет ни с кем конфликтовать, сохраняя при этом функционал.

Для перехода на замыкания воспользуемся алгоритмом:

1. Определяем функцию **makeCounter**, внутри которой создаем функцию **counter**. Возвращаем ее вместе с окружением, содержащим свободную переменную **count**. Она создает замыкание. Функция, возвращенная из **makeCounter**, сохраняется в переменной **doCount**.

2. Если нужно вызвать счетчик, вызываем функцию **doCount**. Это приводит к выполнению тела функции **counter**.

3. При обращении к переменной **count** браузер попытается найти ее в окружении. После этого новое значение сохраняется в окружении, возвращая при этом его в точку вызова **doCount**.

Обратите внимание: мы создаем 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());
```

Возвращение функции из функции — не единственный подход при создании замыканий. Подобные конструкции создаются в любом месте, где появляется ссылка на функцию, имеющую свободные переменные. Такая функция выполняется вне контекста, в котором была создана. Также замыкания

можно создавать, передавая одни функции при вызове других 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 этом примере функциональное выражение со свободной переменной **userMessage** передается в стандартную функцию **setTimeout**. При этом функциональное выражение обрабатывается для получения ссылки на функцию, которая затем передается **setTimeout**. То, что сохраняет метод **setTimeout**, является функцией с окружением, то есть замыканием. После этого по прошествии 1000 миллисекунд функция вызывается.

Практическое задание

1. Продумать, где можно применить замыкания для практикума из седьмого урока.
2. Не выполняя код, 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. Замыкания в JavaScript.
2. Bind, Call и Apply в JavaScript.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

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