Разработка web-приложений на языке JavaScript

Изучение функций

Цели изучения темы:

- ознакомление с понятием функции;
- ознакомление с порядком объявления и вызова функции;
- ознакомление с параметрами и аргументами функции;
- ознакомление с понятием замыкания;
- ознакомление с понятием лексическое окружение;
- ознакомление с понятием контекст выполнения.

Задачи темы:

- изучение понятия функции;
- изучение ключевых методов работы с функциями;
- изучение понятия возвращаемые значения функций;
- изучение понятия замыкания;
- изучение ключевых методов работы с замыканиями;
- изучение примеров работы замыканий.

В результате изучения данной темы Вы будете
Знать:

- парадигмы, архитектурные черты, семантику и синтаксис языка программирования JavaScript;
- назначение и свойства основных структур данных и конструкций языка JavaScript;
- модули и пакеты для решения различных прикладных и научных задач.

Уметь:

- разрабатывать математические методы и алгоритмы проектирования, отладки, проверки работоспособности, создании (модификации) и сопровождении web-приложений на языке JavaScript.

Владеть:

- навыками разработки web-приложений на языке JavaScript;
- навыками модификации и отладки web-приложений на языке JavaScript.

Учебные вопросы темы:

1. Функции
2. Замыкания

Вопрос 1. Функции

Функция – это некоторый фрагмент кода, который можно описать один раз, а затем вызвать на выполнение в разных частях программы какое угодно число раз.

Это классическое назначение функции. В этом сценарии код, который повторяется несколько раз на странице, нужно вынести в функцию, а затем использовать ее в тех местах программы, в которых необходимо его выполнить.

Еще один вариант традиционного применения функций – это когда они используются в качестве основных строительных блоков приложения. В этом случае поставленная задача разбивается на подзадачи, а затем каждая из них решается посредством создания отдельных функций. После этого обычно переходят к разработке центрального кода, используя в нем все ранее разработанные функции.

В результате такая программа становится более структурированной. В нее более просто вносить различные изменения и добавлять новые возможности.

В JavaScript создать функцию можно различными способами:

- Function Declaration
- Function Expression
- Arrow Function

Способ создания функции с помощью Function Declaration. ***Function Declaration*** – это «классический» вариант объявления функции.

Объявление и вызов функций

Операции с функцией в JavaScript можно разделить на 2 шага:

- объявление (создание) функции;
- вызов (выполнение) этой функции.

Объявление функции

Написание функции посредством Function Declaration начинается с написания ключевого слова ***function***. После этого указывается имя функции, круглые скобки в которых при необходимости перечисляются через запятую параметры и код функции, заключенный в фигурные скобки.

```
function имя (параметры) {
  // код функции
}
```

При составлении имени функции необходимо руководствоваться теме же правилами, что и при создании имени переменной. Т.е. оно может содержать буквы, цифры (0-9), знаки «$» и «_». В качестве букв рекомендуется использовать только буквы английского алфавита (a-z, A-Z). Имя функции, также, как и имя переменной не может начинаться с цифры.

Параметров у функции может быть сколько угодно много или не быть вообще. Круглые скобки в любом случае указываются. Если параметров несколько, то их между собой необходимо разделить посредством запятой. Они позволяют более удобно (по имени) получить переданные аргументы функции при ее вызове.

Код функции – это набор инструкций, заключенный в фигурные скобки, которые необходимо выполнить при ее вызове. Код функции называют еще ее телом.

Вызов функции

Объявленная функция сама по себе не выполняется. Для того чтобы функцию запустить, ее необходимо вызвать. Вызов функции осуществляется посредством указания ее имени и двух круглых скобок. Внутри скобок при необходимости ей можно передать аргументы (дополнительные данные) отделяя их друг от друга с помощью запятой.

```
// выполнение функции, приведённой в предыдущем примере (без передачи ей аргументов)
someName();
```

Параметры и аргументы функции

Параметры функции – это локальные переменные функции, которые определяются на этапе объявления функции в круглых скобках. Обратиться к параметрам можно только внутри функции, снаружи они не доступны.

Аргументы функции – это значения, переданные в функцию при ее вызове.

Например,

```
// объявление функции sayWelcome (userFirstName и userLastName - это параметры)
// userFirstName – будет принимать значения 1 аргумента, а userLastName соответственно значение 2 аргумента
function sayWelcome (userFirstName, userLastName) {
  console.log('Добро пожаловать, ' + userLastName + ' ' + userFirstName);
}

// вызов функции sayWelcome ('Иван' и 'Иванов' - это аргументы)
sayWelcome('Иван', 'Иванов');

// ещё один вызов функции sayWelcome ('Петр', 'Петров' - это аргументы)
sayWelcome('Петр', 'Петров');
```

Таким образом, ***параметры*** – это один из способов в JavaScript, с помощью которого можно получить аргументы функции.

В JavaScript при вызове функции количество аргументов не обязательно должно совпадать с количеством параметров. Если функции не был передан аргумент, который мы хотим получить с помощью параметра, то в этом случае он будет иметь значение undefined.

Специальный объект внутри функции arguments. Получить переданные функции аргументы в JavaScript можно не только с помощью параметров. Это еще можно выполнить через специальный массивоподобный объект arguments. Он позволяет получить все аргументы, переданные в функцию, а также их количество с помощью свойства length.

```
function f() {
  // проверим является ли arguments обычным массивом
  console.log(Array.isArray(arguments));
}
```

```
f(); // false
```

Доступ к аргументам через arguments выполняется точно также как к элементам обычного массива, т.е. по их порядковым номерам. Таким образом, argument[0] – позволяет получить первый аргумент, arguments[1] – второй аргумент и т.д.

```
// объявление функции sum
function sum() {
  const num1 = arguments[0] // получаем значение 1 аргумента
  const num2 = arguments[1] // получаем значение 2 аргумента
  console.log(num1 + num2);
}

sum(7, 4); // 11
```

Получение аргументов через arguments в основном используется, когда заранее неизвестно точное их количество.

Передачи одной функции в другую. Колбэки

Передаче одной функции в другую в основном используется для создания колбэков. ***Колбэк (callback)*** – это функция, которая должна быть выполнена после завершения работы другой функции. Ее еще называют функцией обратного вызова. Колбэки в основном применяются в асинхронном коде.

```
// функция, которая будет выводить в консоль то, что ей передали в виде аргумента

function outputResult(result) {
  console.log(result);
}

// функция, которая принимает на вход 2 аргумента и колбэк
function sum(num1, num2, callback) {
  // вычислим сумму 2 значений и сохраним его в result
  const result = num1 + num2;

  // вызовем колбэк
  callback(result);
}

sum (5, 11, outputResult);
```

В JavaScript это возможно, т.к. функция в этом языке является значением. Ее можно передавать в другие функции в виде аргументов. По сути, функция в JavaScript – это определенный тип объектов, который можно вызывать.

Узнать является ли некоторый идентификатор функцией можно с помощью typeof:

```
function myfunc() {};

console.log(typeof myfunc); // function
```

Например, проверим является ли колбэк функцией перед тем как его вызвать:

```
function sum(num1, num2, callback) {
  const result = num1 + num2;

  if (typeof callback === 'function') {
    callback(result);
  }
}
```

Возвращаемые значения

Оператор return предназначен для возвращения значения выражения в качестве результата выполнения функции. Значение выражения должно быть указано после ключевого слова return.

```
function f() {
  return выражение;
}
```

Если оно не указано, то вместо этого значения будет возвращено undefined.

Без использования return:

```
function sum(a, b) {
  const result = a + b;
}

// вызовем функцию и сохраним ее результат в константу result
const result = sum(4, 3);
// выведем результат функции sum(4, 3) в консоль
console.log(result); // undefined
```

С использованием return:

```
function sum(a, b) {
  // вернем в качестве результата сумму a + b
  return a + b;
}

// вызовем функцию и сохраним ее результат в константу result
const result = sum(4, 3);
// выведем результат функции sum(4, 3) в консоль
console.log(result); // 7
```

Инструкции, расположенные после return никогда не выполняются:

```
function sum(a, b) {
  // вернем в качестве результата сумму a + b
  return a + b;
  // код, расположенный после return никогда не выполняется
  console.log('Это сообщение не будет выведено в консоль');
}

sum(4, 90);
```

Функция, которая возвращает функцию

В качестве результата функции мы можем также возвращать функцию.

Например:

```
function outer(a) {
  return function(b) {
    return a * b;
  }
}

// в one будет находиться функция, которую возвращает outer(3)
const one = outer(3);
// в two будет находиться функция, которую возвращает outer(4)
const two = outer(4);

// выведем в консоль результат вызова функции one(5)
console.log(one(5)); // 15
// выведем в консоль результат вызова функции two(5)
console.log(two(5)); // 20
```

Вызовы функции outer(3) и outer(4) возвращают одну и туже функцию, но первая запомнила, что a = 3, а вторая - что a = 4. Это происходит из-за того, что функции в JavaScript «запоминают» окружение, в котором они были созданы. Этот прием довольно часто применяется на практике. Так как с помощью него мы можем, например, на основе одной функции создать другие, которые нужны.

Рекурсия

Функцию можно также вызвать внутри самой себя. Это действие в программировании называется ***рекурсией***.

Кроме этого необходимо предусмотреть условия для выхода из рекурсии. Если это не сделать функция будет вызывать сама себя до тех пор, пока не будет брошена ошибка, связанная с переполнением стека.

Например, использование рекурсии для вычисления факториала числа:

```
function fact(n) {
  // условие выхода из рекурсии
  if (n === 1) {
    return 1;
  }

  // возвращаем вызов функции fact(n - 1) умноженное на n
  return fact(n - 1) * n;
}
console.log(fact(5)); // 120
```

Вопрос 2. Замыкания

Замыкание – это функция, у которой есть доступ к области видимости, сформированной внешней по отношению к ней функции даже после того, как эта внешняя функция завершила работу. Это значит, что в замыкании могут храниться переменные, объявленные во внешней функции и переданные ей аргументы.

Понятие ***«лексическое окружение»*** или «статическое окружение» в JavaScript относится к возможности доступа к переменным, функциям и объектам на основе их физического расположения в исходном коде. Рассмотрим пример:

```
let a = 'global';
  function outer() {
    let b = 'outer';
    function inner() {
      let c = 'inner'
      console.log(c);   // 'inner'
      console.log(b);   // 'outer'
      console.log(a);   // 'global'
    }
    console.log(a);     // 'global'
    console.log(b);     // 'outer'
    inner();
  }
outer();
console.log(a);         // 'global'
```

Здесь у функции inner() есть доступ к переменным, объявленным в ее собственной области видимости, в области видимости функции outer() и в глобальной области видимости. Функция outer() имеет доступ к переменным, объявленным в ее собственной области видимости и в глобальной области видимости.

Цепочка областей видимости вышеприведенного кода будет выглядеть так:

```
Global {
  outer {
    inner
  }
}
```

Функция inner() окружена лексическим окружением функции outer(), которая, в свою очередь, окружена глобальной областью видимости. Именно поэтому функция inner() может получить доступ к переменным, объявленным в функции outer() и в глобальной области видимости.

Для того чтобы понять замыкания, нам нужно разобраться с двумя важнейшими концепциями JavaScript. Это – контекст выполнения (Execution Context) и лексическое окружение (Lexical Environment).

Контекст выполнения

Контекст выполнения – это абстрактное окружение, в котором вычисляется и выполняется JavaScript-код. Когда выполняется глобальный код, это происходит внутри глобального контекста выполнения. Код функции выполняется внутри контекста выполнения функции.

В некий момент времени может выполняться код лишь в одном контексте выполнения (JavaScript – однопоточный язык программирования). Управление этими процессами ведется с использованием так называемого стека вызовов (Call Stack).

Стек вызовов – это структура данных, устроенная по принципу LIFO (Last In, First Out – последним вошел, первым вышел). Новые элементы можно помещать только в верхнюю часть стека, и только из нее же элементы можно изымать.

Текущий контекст выполнения всегда будет в верхней части стека, и когда текущая функция завершает работу, ее контекст выполнения извлекается из стека и управление передается контексту выполнения, который был расположен ниже контекста этой функции в стеке вызовов.

Стек вызовов рассмотрен на рисунке 1.

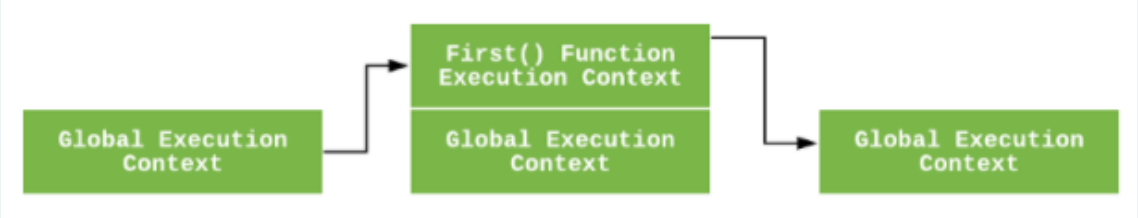

Рис. 1. Стек вызовов

Когда завершается выполнение функции first(), ее контекст выполнения извлекается из стека вызовов и управление передается контексту выполнения, находящемуся ниже его, то есть – глобальному контексту. После этого будет выполнен оставшийся в глобальной области видимости код.

Лексическое окружение

Каждый раз, когда JS-движок создает контекст выполнения для выполнения функции или глобального кода, он создает и новое лексическое окружение для хранения переменных, объявляемых в этой функции в процессе ее выполнения.

Лексическое окружение – это структура данных, которая хранит сведения о соответствии идентификаторов и переменных. Здесь «идентификатор» – это имя переменной или функции, а «переменная» – это ссылка на объект (сюда входят и функции) или значение примитивного типа.

Лексическое окружение содержит два компонента:

- *Запись окружения (environment record)* – место, где хранятся объявления переменных и функций.
- *Ссылка на внешнее окружение (reference to the outer environment)* – ссылка, позволяющая обращаться к внешнему (родительскому) лексическому окружению. Это – самый важный компонент, с которым нужно разобраться для того, чтобы понять замыкания.

Концептуально лексическое окружение выглядит так:

```
lexicalEnvironment = {
  environmentRecord: {
    <identifier> : <value>,
    <identifier> : <value>
  }
  outer: < Reference to the parent lexical environment>
}
```

Посмотрим на следующий фрагмент кода:

```
let a = 'Hello World!';
function first() {
  let b = 25;
  console.log('Inside first function');
}
```

```
first();
console.log('Inside global execution context');
```

Когда JS-движок создает глобальный контекст выполнения для выполнения глобального кода, он создает и новое лексическое окружение для хранения переменных и функций, объявленных в глобальной области видимости. В результате лексическое окружение глобальной области видимости будет выглядеть так:

```
globalLexicalEnvironment = {
  environmentRecord: {
     a     : 'Hello World!',
     first : < reference to function object >
  }
  outer: null
}
```

Обратите внимание на то, что ссылка на внешнее лексическое окружение (outer) установлена в значение null, так как у глобальной области видимости нет внешнего лексического окружения.

Когда движок создает контекст выполнения для функции first(), он создает и лексическое окружение для хранения переменных, объявленных в этой функции в ходе ее выполнения. В результате лексическое окружение функции будет выглядеть так:

```
functionLexicalEnvironment = {
  environmentRecord: {
     b    : 25,
  }
  outer: <globalLexicalEnvironment>
}
```

Ссылка на внешнее лексическое окружение функции установлена в значение <globalLexicalEnvironment>, так как в исходном коде код функции находится в глобальной области видимости.

Обратите внимание на то, что, когда функция завершит работу, ее контекст выполнения извлекается из стека вызовов, но ее лексическое окружение может быть удалено из памяти, а может и остаться там. Это зависит от того, существуют ли в других лексических окружениях ссылки на данное лексическое окружение в виде ссылок на внешнее лексическое окружение.

Вопросы для самопроверки:

1. Что такое функции?
2. Какие вы знаете методы работы с функциями?
3. Что входит в понятие возвращаемые значения функций?
4. Что входит в понятие замыкания?
5. Перечислите ключевые методы работы с замыканиями.
6. Приведите примеры работы замыканий.