Функции

Функции — это механизм для многократного использования частей кода. Они позволяют запускать один и тот же код из разных мест программы без необходимости его копировать. Кроме того, если вы «спрячете» сложные фрагменты кода внутри функций, вам будет легче сосредоточиться на проектировании программы — так вы будете налаживать взаимодействие между функциями, а не барахтаться в мелких деталях, из которых состоит код этих фрагментов. Организация кода в виде небольших, легко контролируемых частей позволяет видеть общую картину и думать о строении программы на более высоком уровне.

Функции очень удобны, когда нужно многократно выполнять в программе некие расчеты или другие действия. Мы уже пользовались готовыми функциями, такими как Math.random, Math.floor, alert, prompt и confirm. И сейчас мы научимся создавать свои функции.

Создаем простую функцию

Давайте создадим функцию, которая печатает фразу «Привет, мир!». Введите в консоли браузера следующий код. Чтобы перейти к новой строке без выполнения уже введенных команд, используйте SHIFT-ENTER.

```
var ourFirstFunction = function()
{
  console.log("Привет, мир!");
};
```

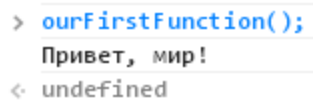
Этот код создает новую функцию, сохраняя ее в переменной ourFirstFunction.

Вызов функции

Чтобы запустить код функции (то есть ее тело), нужно эту функцию вызвать. Для этого укажите ее имя, а следом — открывающую и закрывающую скобки, вот так:

```
ourFirstFunction();
Привет, мир!
```

Однако, вызвав эту функцию из браузера, можно заметить в консоли еще одну строчку — с маленькой, указывающей влево стрелкой. Это значение, которое возвращает функция.



Возвращаемое значение — это значение, которое функция выдает наружу, чтобы потом его можно было использовать где угодно в программе. В данном случае это undefined, поскольку мы не указывали возвращаемое значение в теле функции, мы лишь дали команду вывести текст в консоль. Функция всегда будет возвращать undefined, если в теле функции нет указания вернуть что-нибудь другое. (Но скоро мы выясним, как это можно сделать.)

Передача аргументов в функцию

Наша функция ourFirstFunction выводит одну и ту же строку при каждом вызове, однако хотелось бы, чтобы поведением функции можно было управлять. Чтобы функция могла изменять поведение в зависимости от значений, нам понадобятся аргументы. Список аргументов указывается в скобках после имени функции — как при ее создании, так и при вызове.

Функция sayHelloTo использует аргумент (name), чтобы поздороваться с человеком, имя которого передано в аргументе:

```
var sayHelloTo = function(name)
{
  console.log("Привет, " + name + "!");
};
```

Здесь мы создали функцию и сохранили ее в переменной sayHelloTo. При вызове функция печатает строку "Привет, " + name + "!", заменяя name на значение, переданное в качестве аргумента.

Вызывая функцию, которая принимает аргумент, введите значение, которое вы хотите использовать в качестве этого аргумента в скобках после имени функции. Например, чтобы поздороваться с Анечкой, можно ввести:

```
sayHelloTo("Анечка");
```

Привет, Анечка!

А Саида можно поприветствовать так:

```
sayHelloTo("Саид");
```

Привет, Саид!

Каждый раз при вызове функции переданный нами аргумент name подставляется в строку, которую печатает функция. Поэтому, когда мы передаем значение "Анечка", в консоли появляется «Привет, Анечка!», а когда пишем "Саид", мы видим в консоли «Привет, Саид!».

Печатаем котиков!

Кроме того, переданный в функцию аргумент может указывать, сколько раз требуется что-то сделать. Например, функция drawCats выводит в консоль смайлы — кошачьи мордочки (вот такие: =^.^=). Задавая аргумент howManyTimes, мы сообщаем ей, сколько таких смайлов нужно напечатать:

```
var drawCats = function (howManyTimes)
{
    for(var i = 0; i < howManyTimes; i++)
    {
       console.log(i + " =^.^=");
    }
};
```

Тело функции представляет собой цикл for, который повторяется столько раз, сколько указано в аргументе howManyTimes (поскольку переменная i сначала равна 0, а затем возрастает до значения howManyTimes - 1). На каждом повторе цикла функция выводит в консоль строку i + " =^.^=".

Вот что мы увидим, вызвав эту функцию со значением 5 в качестве аргумента howManyTimes:

```
drawCats(5);
0 =^.^=
1 =^.^=
2 =^.^=
3 =^.^=
4 =^.^=
```

Попробуйте задать howManyTimes значение 100, чтобы напечатать 100 кошачьих мордочек!

Передача в функцию нескольких аргументов

В функцию можно передать больше одного значения, задав несколько аргументов. Для этого перечислите аргументы в скобках после имени функции, разделив их запятыми.

Фунция printMultipleTimes похожа на drawCats, однако она принимает еще один аргумент с именем whatToDraw.

```
var printMultipleTimes = function (howManyTimes, whatToDraw)
{
  for(var i = 0; i < howManyTimes; i++)
  {
    console.log(i + " " + whatToDraw);
  }
};
```

Функция printMultipleTimes печатает строку, переданную в аргументе whatToDraw столько раз, сколько указано в аргументе howManyTimes. Второй аргумент сообщает функции, что печатать, а первый — сколько раз это нужно печатать.

Вызывая функцию с несколькими аргументами, перечислите нужные вам значения через запятую в скобках после имени функции. Например, чтобы напечатать кошачьи мордочки с помощью функции printMultipleTimes, вызывайте ее так:

```
printMultipleTimes(5, "=^.^=");
0 =^.^=
1 =^.^=
2 =^.^=
```

```
3 =^.^=
4 =^.^=
```

А чтобы четыре раза напечатать смайлик «^_^», вызывайте printMultipleTimes так:

```
printMultipleTimes(4, "^_^");
0 ^_^
1 ^_^
2 ^_^
3 ^_^
```

Здесь при вызове printMultipleTimes мы указали значение 4 для аргумента howManyTimes и строку «^_^» для аргумента whatToDraw. В результате цикл выполнил четыре повтора (переменная i менялась от 0 до 3), каждый раз печатая i + " " + "^_^".

Чтобы дважды напечатать «(>_<)», введите:

```
printMultipleTimes(2, "(>_<)");
0 (>_<)
1 (>_<)
```

На этот раз мы передали число 2 для аргумента howManyTimes и строку «(>_<)» для whatToDraw.

Возврат значения из функции

До сих пор все наши функции выводили текст в консоль с помощью console.log. Это простой и удобный способ отображения данных, однако мы не сможем потом взять это значение из консоли и использовать его в коде. Вот если бы наша функция выдавала значение так, чтобы его потом можно было использовать в других частях программы...

Как я уже говорил, функции могут возвращать значение. Вызвав функцию, которая возвращает значение, мы можем затем использовать это значение в своей программе (сохранив его в переменной, передав в другую функцию или объединив с другими данными). Например, следующий код прибавляет 5 к значению, которое возвращает вызов Math .floor (1.2345):

```
5 + Math.floor(1.2345);
6
```

Math.floor — функция, которая берет переданное ей число, округляет его вниз до ближайшего целого значения и возвращает результат. Глядя на вызов функции Math.floor(1.2345), представьте, что вместо него в коде стоит значение, возвращаемое этой функцией, — в данном случае это число 1.

Теперь давайте создадим функцию, которая возвращает значение. Вот функция double, которая принимает аргумент number и возвращает произведение number * 2. Иными словами, значение, которое возвращает эта функция, вдвое больше переданного ей аргумента.

```
var double = function(number)
{
  return number * 2;
};
```

Чтобы вернуть из функции значение, используйте оператор return, после которого укажите само это значение. Мы воспользовались return, вернув из функции double число number * 2.

Теперь можно вызывать нашу функцию double и удваивать числа:

```
double(11);
22
```

Возвращаемое значение (22) показано здесь второй строкой. Хоть функции и способны принимать несколько аргументов, вернуть они могут лишь одно значение. А если вы не укажете в теле функции, что именно надо возвращать, она вернет undefined.

Вызов функции в качестве значения

Когда функция вызывается из кода программы, значение, возвращаемое этой функцией, подставляется туда, где происходит вызов. Давайте воспользуемся функцией double, чтобы удвоить пару чисел и затем сложить результаты:

```
double(5) + double(6);
22
```

Здесь мы дважды вызвали функцию double и сложили значения, которые вернули эти два вызова. То же самое было бы, если бы вместо вызова double (5) стояло число 10, а вместо double (6) —число 12.

Также вызов функции можно указать в качестве аргумента другой функции, при вызове которой в аргумент попадет значение, возвращенное первой функцией. В следующем примере мы вызываем функцию double, передавая ей в качестве аргумента вызов double с аргументом 3. Вызов double (3) даст 6, так что double (double (3)) упрощается до double (6), что, в свою очередь, упрощается до 12.

```
double(double(3));
12
```

Упрощаем код с помощью функций

Ранее мы использовали методы Math.random и Math.floor, чтобы выбирать случайные слова из массивов и генерировать дразнилки. Давайте мы перепишем генератор дразнилок, упростив его с помощью функций.

Функция для выбора случайного слова

Вот как мы выбирали случайное слово из массива:

```
randomWords[Math.floor(Math.random() * randomWords.length)];
```

Если поместить этот код в функцию, можно многократно вызывать его для получения случайного слова из массива — вместо того чтобы вводить тот же код снова и снова. Например, давайте определим такую функцию pickRandomWord:

```
var pickRandomWord = function(words)
{
  return words[Math.floor(Math.random() * words.length)];
};
```

Все, что мы сделали, — поместили прежний код в функцию. Теперь можно создать массив randomWords.

```
var randomWords = ["Планета", "Червяк", "Цветок", "Компьютер"];
```

С помощью функции pickRandomWord мы можем получить случайное слово из этого массива, вот так:

```
pickRandomWord(randomWords);
"Цветок"
pickRandomWord(randomWords);
"Планета"
```

При этом нашу функцию можно использовать с любым массивом. Например, получить случайное имя из массива имен:

```
pickRandomWord(["Зифа", "Осип", "Гульнара", "Эдуард", "Ралина"]);
"Эдуард"
pickRandomWord(["Зифа", "Осип", "Гульнара", "Эдуард", "Ралина"]);
"Ралина"
```

Генератор случайных дразнилок

Теперь давайте перепишем генератор дразнилок, используя нашу функцию для выбора случайных слов. Для начала вспомним, как выглядел его код:

```
var randomBodyParts = ["глаз", "нос", "череп", "гузно", "нога", "чёлка"];
var randomAdjectives = ["вонючая", "поганая", "смердящая",
                        "унылая", "дурацкая", "паскудная"];
var randomWords = ["коровья лепёшка", "глиста", "рыба", "свинья", "метла", "жвачка"];
// Выбор случайной части тела из массива randomBodyParts:
var randomBodyPart = randomBodyParts[Math.floor(Math.random() * randomBodyParts.length)];
// Выбор случайного прилагательного из массива randomAdjectives:
var randomAdjective = randomAdjectives[Math.floor(Math.random() * randomAdjectives.length)];
// Выбор случайного слова из массива randomWords:
var randomWord = randomWords[Math.floor(Math.random() * randomWords.length)];
// Соединяем случайные строки в предложение:
```

```
var randomInsult = "У тебя " + randomBodyPart + " словно " +
                   randomAdjective + " " + randomWord + "!!!";
```

Обратите внимание — конструкция words [Math.floor(Math.random!) * length)] повторяется здесь несколько раз. Воспользовавшись функцией pickRandomWord, можно переписать программу таким образом:

```
var randomBodyParts = ["глаз", "нос", "череп", "гузно", "нога", "чёлка"];
var randomAdjectives = ["вонючая", "поганая", "смердящая",
"унылая", "дурацкая", "паскудная"];
var randomWords = ["коровья лепёшка", "глиста", "рыба", "свинья", "метла", "жвачка"];
var randomString = "У тебя " + pickRandomWord(randomBodyParts) + " словно " +
pickRandomWord(randomAdjectives) + " " + pickRandomWord(randomWords) + "!!!";
randomString;
"У тебя гузно словно унылая коровья лепёшка!!!"
```

Этот код отличается от прежнего двумя моментами. Во-первых, мы использовали функцию pickRandomWord для выбора случайного слова из массива вместо того, чтобы каждый раз писать words[Math.floor(Math.random() * length)]. А во-вторых, вместо того чтобы сохранять каждое случайное слово в переменной перед тем, как добавлять его к итоговой строке, мы сразу объединяем возвращаемые из функции значения, формируя таким образом строку. Вызов функции можно рассматривать как значение, которое эта функция возвращает, поэтому все, что мы тут делаем, — это объединяем строки. Как видите, новую версию программы гораздо легче читать. Да и писать ее тоже было легче, поскольку часть повторяющегося кода мы вынесли в функцию.

Делаем генератор дразнилок функцией

Можно еще усовершенствовать наш генератор случайных дразнилок, сделав его функцией, которая возвращает дразнилки:

```
var generateRandomInsult = function ()
{
  var randomBodyParts = ["глаз", "нос", "череп", "гузно", "нога", "чёлка"];
  var randomAdjectives = ["вонючая", "поганая", "смердящая", "унылая", "дурацкая",
"паскудная"];
  var randomWords = ["коровья лепёшка", "глиста", "рыба", "свинья", "метла", "жвачка"];
  var randomString = "У тебя " + pickRandomWord(randomBodyParts) + " словно " +
  pickRandomWord(randomAdjectives) + " " + pickRandomWord( randomWords) + "!!!";
  return randomString;
};
generateRandomInsult();
"У тебя нога словно дурацкая жвачка!!!"
generateRandomInsult();
"У тебя чёлка словно вонючая коровья лепёшка!!!"
generateRandomInsult();
"У тебя нос словно дурацкая свинья!!!"
```

Наша новая функция generateRandomInsult представляет собой все тот же код, помещенный в тело функции без аргументов. Мы добавили лишь одну строку, где мы возвращаем сгенерированную строку randomString. Трижды вызвав функцию generateRandomInsult, мы каждый раз получали новую дразнилку.

Теперь весь код находится в функции, и это означает, что для генерации дразнилки мы можем просто вызывать эту функцию, а не копировать в консоль один и тот же код каждый раз, когда понадобится кого-нибудь подразнить.

Ранний выход из функции по return

Как только JavaScript, выполняя код функции, встречает оператор return, он завершает функцию, даже если после return еще остался какой-нибудь код.

Оператор return часто используют, чтобы выйти из функции в самом начале, если какие-нибудь из переданных аргументов имеют некорректные значения — то есть если с такими

аргументами функция не сможет правильно работать. Например, следующая функция возвращает строку с информацией о пятой букве вашего имени. Если в имени, переданном в аргументе name, меньше пяти букв, будет выполнен return, чтобы сразу же выйти из функции. При этом оператор return в конце функции (тот, что возвращает сообщение о пятой букве) так и не будет выполнен.

```
var fifthLetter = function(name)
{
  if(name.length < 5)
    return;
  return "Пятая буква вашего имени: " + name[4] + ".";
};
```

В первой строке мы проверяем длину переданного имени — уж не короче ли оно пяти символов? Если это так, мы выполняем return, чтобы незамедлительно выйти из функции.

Давайте попробуем эту функцию в деле.

```
fifthLetter("Роман");
"Пятая буква вашего имени: н."
```

В имени «Роман» пять букв, так что функция fifthLetter благополучно завершается, вернув пятую букву имени, то есть «н». Попробуем вызвать ее еще раз с именем покороче:

```
fifthLetter("Яна");
undefined
```

Когда мы вызвали fifthLetter для имени «Яна», функция распознала, что имя недостаточно длинное, и сразу завершилась, выполнив оператор return во второй строке. Поскольку никакого значения после этого return не указано, функция вернула undefined.

Многократное использование return вместо конструкции if... else

Можно многократно использовать return внутри разных конструкций if, чтобы возвращать из функции разные значения в зависимости от входных данных. Предположим, вы пишете игру, в которой игроки награждаются медалями согласно набранным очкам. Счету меньше трех очков соответствует бронзовая медаль, счету от трех до шести — серебряная, а счету от семи и выше — золотая.

```
var medalForScore = function(score)
{
  if(score < 3)
    return "Бронзовая";
  if(score < 7)
    return "Серебряная";
  return "Золотая";
};
```

В первой строке мы возвращаем значение "Бронзовая" и выходим из функции, если счет меньше трех очков. Если мы достигли третьей строки, значит, счет как минимум равен трем очкам, поскольку, будь он меньше трех, мы бы уже вышли из функции (выполнив return в первом операторе if). И наконец, если мы достигли пятой строки, значит, на счету как минимум семь очков, проверять больше нечего и можно спокойно вернуть значение "Золотая".

Хотя мы проверяем здесь несколько условий, необходимости использовать цепочку конструкций if... else нет. Мы используем if... else, когда хотим убедиться, что будет выбран лишь один из вариантов. Однако если в каждом варианте выполняется return, это также гарантирует однозначный выбор (поскольку выйти из функции можно лишь один раз).

Сокращенная запись при создании функций

Есть длинный и короткий способы записи функций. Пока мы использовали длинную запись, поскольку она наглядно демонстрирует, что функция хранится в переменной. Тем не менее вам стоит знать и о короткой записи, поскольку ее используют многие JavaScript-разработчики. Возможно, и вы сами, достаточно поработав с функциями, предпочтете короткую запись.

Вот пример длинной записи:

```
var double = function(number)
{
```

```
  return number * 2;
};
```

Короткая запись той же функции выглядит более привычно для нас как знатоков C/C++:

```
function double(number)
{
  return number * 2;
}
```

Как видите, при длинной записи мы явно создаем переменную и сохраняем в ней функцию, так что имя double записывается прежде ключевого слова function. Напротив, при короткой записи сначала идет ключевое слово function, а затем название функции. В этом случае JavaScript создает переменную double неявным образом.

На техническом сленге длинная запись называется функциональным выражением, а короткая — объявлением функции.

DOM и jQuery

До сих пор мы использовали JavaScript для относительно простых задач, вроде вывода текста в консоли браузера или отображения диалогов alert и prompt. Однако помимо этого JavaScript позволяет взаимодействовать с HTML-элементами на веб-страницах, меняя их поведение и внешний вид. Сейчас мы поговорим о двух технологиях, которые помогут вам писать гораздо более мощный JavaScript-код: это DOM и jQuery.

DOM (Document Object Model — объектная модель документа) — это средство, позволяющее JavaScript-коду взаимодействовать с содержимым веб-страниц. Браузеры используют DOM для структурирования страниц и их элементов (параграфов, заголовков и т. д.), а JavaScript может разными способами манипулировать элементами DOM. Например, скоро вы узнаете, как при помощи JavaScript-программы менять заголовки HTML-документов, подставляя туда значение, полученное из диалога prompt.

Также мы познакомимся с удобным инструментом под названием jQuery, который кардинально упрощает работу с DOM. jQuery содержит набор функций, которые позволяют найти нужные вам элементы и произвести с ними определенные действия.

Мы узнаем, как с помощью DOM и jQuery изменять существующие элементы DOM, а также создавать новые, полностью контролируя содержимое веб-страниц из JavaScript-кода. Также мы выясним, как использовать jQuery для анимации элементов DOM — например, для плавного появления и исчезновения изображений и текста.

Поиск элементов DOM

Когда вы открываете HTML-документ, браузер преобразовывает его элементы в древовидную структуру — дерево DOM. Мы уже обсуждали его ранее, когда говорили об иерархии HTML. Браузер дает JavaScript-программистам возможность доступа к этой древовидной структуре при помощи специальных методов DOM.

Идентификация элементов по id

HTML-элементу можно присвоить уникальное имя, или *идентификатор*, с помощью атрибута id. Например, у элемента h1 задан атрибут id:

<h1 id="main-heading">Привет, мир</h1l>

Задав атрибуту id значение (в данном случае "main-heading"), мы получаем возможность впоследствии найти этот конкретный заголовок по его id и что-нибудь с ним сделать, не затрагивая остальные элементы, даже если это другие заголовки уровня h1.

Поиск элемента с помощью getElementByld

Обозначив элемент уникальным id (каждый id в документе должен иметь собственное, отличное от других значение), мы можем воспользоваться DOM-методом document.getElementByld, чтобы найти элемент "main-heading":

var headingElement = document .getElementByld ("main-heading");

Вызовом document.getElementByld("main-heading") мы даем браузеру команду отыскать элемент, id которого равен "main-heading".

Этот вызов вернет DOM-объект с соответствующим id, и мы сохраним этот объект в переменной headingElement.

Когда элемент найден, им можно управлять при помощи JavaScript-кода. Например, через свойство innerHTML мы можем узнать, что за текст находится внутри элемента, или заменить этот текст:

headingElement.innerHTML;

Эта команда возвращает содержимое headingElement — элемента, который мы нашли с помощью getElementByld. В данном случае содержимое — это текст "Привет, мир!", находящийся между тегов <h1>.

Меняем текст заголовка через DOM

Вот пример того, как менять текст заголовка с помощью DOM. Давайте создадим новый HTML-документ dom.html со следующим кодом:

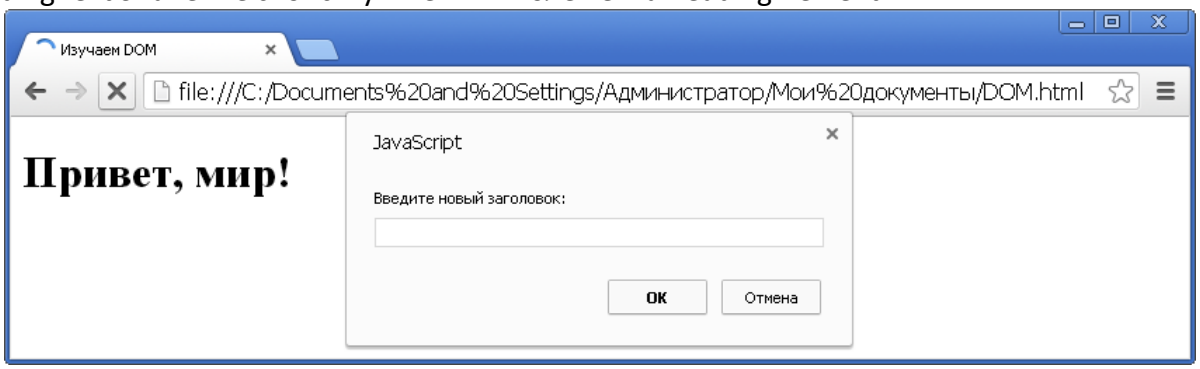
<!DOCTYPE html>

<html>

<head>

```
<meta charset="UTF-8">
<title>Изучаем DOM</title>
</head>
<body>
<h1 id="main-heading">Привет, мир!</h1>
<script>
  var headingElement = document.getElementById("main-heading");
  console.log(headingElement.innerHTML);
  var newHeadingText = prompt("Введите новый заголовок:");
  headingElement.innerHTML = newHeadingText;
</script>
</body>
</html>
```

С помощью document.getElementById мы нашли элемент h1 (id которого равен «main-heading») и сохранили его в переменной headingElement. Далее мы вывели в консоль строку, возвращенную вызовом headingElement.innerHTML — то есть "Привет, мир!". В третьей строке открыли диалог prompt, чтобы получить от пользователя новый заголовок, и сохранили введенный пользователем текст в переменной newHeadingText. И наконец, в строке О присвоили сохраненное в newHeadingText значение свойству innerHTML элемента headingElement.



Введите в этом диалоге свою строку и нажмите «ОК». Заголовок должен тотчас поменяться.

Обращаясь к свойству innerHTML, можно поменять содержимое любого элемента DOM через JavaScript-код.

Работа с деревом DOM через jQuery

Встроенные в браузер методы DOM всем хороши, но пользоваться ими бывает нелегко, поэтому многие программисты применяют специальную библиотеку под названием jQuery.

jQuery — это набор инструментов (в основном функций), которые сильно упрощают работу с DOM-элементами. Подключив эту библиотеку к нашей страничке, мы сможем вызывать ее функции и методы в дополнение к функциям и методам, встроенным в JavaScript и в браузер.

Подключаем jQuery к HTML-странице

Прежде чем воспользоваться библиотекой jQuery, нужно, чтобы браузер ее загрузил, для чего достаточно одной строки HTML-кода:

```
<script src="jquery-3.1.1.js"></script>
```

Обратите внимание, что у тега <script> нет содержимого, зато есть атрибут src. Этот атрибут позволяет загрузить на страницу JavaScript-файл, указав его URL (то есть веб-адрес). В данном случае jquery-3.1.1.js— это URL конкретной версии jQuery.

Меняем текст заголовка с помощью jQuery

В разделе «Меняем текст заголовка через DOM» ранее мы выяснили, как поменять заголовок, вызывая встроенные методы DOM. В этом же разделе мы доработаем код страницы, чтобы менять заголовок через jQuery. Откройте файл dom.html и внесите в него следующие правки:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Изучаем DOM</title>
```

```
</head>
<body>
<h1 id="main-heading">Привет, мир!</h1>
<script src="jquery-3.1.1.js"></script>
<script>
  var newHeadingText = prompt("Введите новый заголовок:");
  $("#main-heading").text(newHeadingText);
</script>
</body>
</html>
```

Мы добавили новый тег <script>, подгружающий jQuery. Когда библиотека загружена, можно использовать jQuery-функцию $ для поиска HTML-элементов.

Функция $ принимает один аргумент, который называется *строка селектора*. Эта строка указывает, какой элемент или элементы нужно найти в дереве DOM. В нашем случае это "#main-heading". Символ # означает id, то есть селектор "#main-heading" ссылается на элемент, id которого равен main-heading.

Функция $ возвращает объект jQuery, соответствующий найденным элементам. Например, вызов $("#main-heading") вернет jQuery-объект для элемента h1 (id которого равен "main-heading").

Теперь, когда у нас есть объект jQuery для элемента h1, мы можем изменить его текст, вызвав метод jQuery-объекта text с новым заголовком в качестве аргумента. Заголовок поменяется на введенную пользователем строку, которая хранится в переменной newHeadingText. Как и прежде, при открытии нашей страницы должен появиться диалог с запросом нового текста для элемента h1.

Создание новых элементов через jQuery

Помимо изменения существующих элементов, с помощью jQuery можно создавать новые элементы и добавлять их в дерево DOM. Для этого мы будем использовать метод jQuery-объекта append, передавая ему нужный HTML-код. Append преобразует HTML в DOM-элемент (соответствующий заданным в коде тегам) и добавит его к содержимому элемента, для которого он был вызван.

Например, чтобы поместить в конец страницы новый элемент p, добавим в наш JavaScript такой код:

```
$("body").append("<p>Это новый параграф</p>");
```

Первая часть этой команды вызывает функцию $ со строкой селектора "body", чтобы найти тело (содержимое) нашего HTML-документа. Поиск не обязательно должен происходить по id — код $("body") ищет элемент body, и точно так же мы можем вызвать $("p") для поиска всех элементов p.

Далее мы вызываем для найденного объекта метод append — переданная ему строка преобразуется в DOM-элемент, а затем добавляется внутрь элемента body, сразу перед закрывающим тегом.

Также append можно использовать в цикле for для добавления нескольких элементов:

```
for(var i = 0; i < 3; i++)
{
  var hobby = prompt("Назови одно из своих хобби!");
$("body").append("<p>" + hobby + "</p>");
}
```

Этот цикл повторяется трижды. При каждом повторении создается диалог prompt, запрашивающий у пользователя его хобби, после чего строка с хобби помещается между тегов <p> и передается методу append, который добавляет ее в конец элемента body.

Анимация элементов средствами jQuery

На многих сайтах при показе и скрытии частей страницы используется анимация. Например, добавляя на страницу новый параграф с текстом, вы можете сделать так, чтобы он проявлялся постепенно, а не весь целиком сразу.

С помощью jQuery анимировать элементы совсем не сложно. К примеру, чтобы элемент медленно исчезал, мы можем воспользоваться методом fadeOut. Замените содержимое второго элемента script в dom.html на такой код:

```
$("h1").fadeOut(3000);
```

Чтобы найти все элементы h1, мы использовали функцию $. Поскольку в dom.html элемент h1 всего один (это заголовок с текстом «Привет, мир!»), именно его мы и получим в виде jQuery-объекта. Вызывая для этого объекта метод fadeout(3000), мы запускаем затухание заголовка до полного его исчезновения в течение трех секунд. (Аргумент fadeOut передается в миллисекундах, то есть тысячных долях секунды, поэтому значение 3000 даст анимацию в три секунды длиной.)

Когда вы перегрузите страницу с этим кодом, вы увидите, как элемент h1 постепенно исчезает.

Цепной вызов и анимация на jQuery

Если вызвать метод jQuery-объекта, этот метод, как правило, вернет первоначальный объект — тот, для которого он и был вызван. Например, $("h1") возвращает jQuery-объект со всеми элементами h1, а $("h1").fadeOut(3000) возвращает все тот же jQuery-объект с элементами h1. Тогда изменить текст заголовка и включить его затухание можно так:

```
$("h1").text("Этот текст скоро исчезнет").fadeOut(3000);
```

Подобный вызов нескольких методов подряд называют *цепным вызовом*.

Можно запустить несколько анимаций одного и того же элемента. Например, использовать цепной вызов методов fadeOut и fadeIn, чтобы элемент исчез и тут же снова проявился:

```
$("h1").fadeOut(3000).fadeIn(2000);
```

Анимация fadeIn заставляет невидимый элемент проявиться. jQuery понимает, что, когда вы делаете цепной вызов двух анимаций, вы, скорее всего, хотите, чтобы они сработали по очереди, одна после другой. В результате элемент h1 будет затухать в течение трех секунд, а затем в течение двух секунд проявляться.

В jQuery есть еще два метода для анимации, похожие на fadeOut и fadeIn, — это slideUp и slideDown. При вызове slideUp элементы исчезают, уплывая вверх, а при вызове slideDown появляются, опускаясь сверху. Замените второй элемент script в dom.html на следующий код и перегрузите страницу:

```
$("h1").slideUp(1000).slideDown(1000);
```

Здесь мы нашли элемент h1, скрыли его с эффектом уплывания вверх в течение одной секунды, а затем показали снова, опустив сверху вниз за одну секунду.