GeekBrains

Основы программирования

Функции

[В какой версии ПО был написан код. Если актуально]

Иллюстрации для обложки:

https://www.dropbox.com/sh/lzla84g77kbk851/AAASYvRaXE-4idVbxkXaC4LYa/Illustration?dl=0&subfolder_nav_tracking=1

Иллюстрация должна быть отцентрирована, уменьшать в зависимости от размера заголовка

Акцентный для выделений: #6654D9

На этом уроке

1. Узнаем что такое функции и какие проблемы они решают;
2. Научимся создавать и использовать функции.

Оглавление

Введение

На прошлом уроке вы познакомились с циклами, научились их применять для решения различных задач. Сегодня вы познакомитесь с важным аспектом программирования — функциями. Функции пришли в программирование из математики, однако для понимания того, как устроены функции в программировании, знание математики не обязательно. Функции помогают избегать дублирования

кода, улучшают читаемость, а также позволяют использовать эффективные приёмы в программировании.

Вынос кода вычисления факториала в функцию

Решим следующую задачу: необходимо запросить у пользователя два числа и вывести в консоль сумму факториалов этих чисел.

Поскольку мы уже решали задачу нахождения факториала, можно взять за основу тот код и решить задачу копированием кода:

```
let value1 = prompt('Введите первое число');
let value2 = prompt('Введите второе число');

let result1 = 1;

for (let n = 2; n <= value1; n = n + 1) {
  result1 = result1 * n;
}

let result2 = 1;

for (let n = 2; n <= value2; n = n + 1) {
  result2 = result2 * n;
}

let finalResult = result1 + result2;

console.log('Результат: ' + finalResult);
```

В глаза бросается то, что кода стало в два раза больше, хотя дублирующийся код, по-сути, совершает то же самое действие. Кроме того, что это выглядит некрасиво, дублирование кода влечёт ещё одну неприятность: в крупных системах, дублирующийся код тяжело поддерживать. В нашем случае код вычисления факториала не будет меняться, потому что это устоявшаяся математическая функция, но в больших программных продуктах возникает необходимость изменять исходный код с целью добавления нового функционала. И разработчик, вносящий изменения, может не знать, что у кода, который он изменяет, есть дубликат(ы), что приведёт к появлению ошибок в программе.

К счастью, у этой проблемы есть решение и название этого решения — функции. Вот как они работают:

```
function factorial(value) {
  let n = 2;
  let result = 1;
```

```
  while (n <= value) {
    result = result * n;
    n = n + 1;
  }
  return result;
}

let value1 = prompt('Введите первое число');
let value2 = prompt('Введите второе число');
let result = factorial(value1) + factorial(value2);

console.log('Результат: ' + result);
```

Функции в JavaScript объявляются с помощью ключевого слова **function**, после которого следует название функции. После названия идут круглые скобки, внутри которых перечисляются аргументы функции, их может быть любое количество или их может не быть совсем, например:

```
function myFunc1() {
}

function myFunc2(arg1) {
}

function myFunc3(arg1, arg2, arg3) {
}
```

Аргументы — это то, что функция получает на вход. При вызове функции, мы передаём в неё аргумент(ы), если это необходимо. В нашем случае мы вызвали объявленную нами функцию *factorial* с аргументом *value1* и прибавили к полученному результату результат второго вызова factorial, в который был передан аргумент *value2*:

```
let result = factorial(value1) + factorial(value2);
```

Вызов функции для нас не новая операция — мы их уже вызывали раньше, например, запрос данных от пользователя осуществляется через вызов функции *prompt*:

```
let value = prompt('Введите число');
```

console.log — это тоже функция и мы её вызывали неоднократно:

```
console.log('Hello, world!');
```

Мы даже можем написать свою собственную функцию для вывода текста в консоль:

```
function log(text) {
  console.log(text);
}
log('Hello, world!');
```

Как можно заметить, функция не обязательно должна возвращать результат.

Рекурсивный вызов функции

Один из интересных приёмов, который открывает использование функций — рекурсивный вызов или просто рекурсия. Рекурсия — это вызов функции из неё же самой.

Попробуем реализовать функцию вычисления факториала без использования цикла, но с использованием рекурсии:

```
function factorial(value) {
  if (value <= 1) {
    return 1;
  }
  return value * factorial(value - 1);
}
```

Несмотря на то, что функция вычисления факториала стала короче, понять этот алгоритм новичку в программировании не всегда просто. Разберём работу алгоритма на примере вызова функции с аргументом 3:

```
function factorial(value) {
  if (value <= 1) {
    return 1;
  }
  return value * factorial(value - 1);
}
console.log('Результат: ' + factorial(3)); // 6
```

Изначально функция вызывается с аргументом 3, условие if (value <= 1) не выполняется, поэтому функция возвращает 3 * factorial(3 - 1), то есть 3 умноженное на результат вызова самой себя с аргументом уменьшенным на единицу. И пока функция не получит результат вызова самой себя, она не сможет вернуть результат. В новом вызове функции с аргументом 2 условие if (value <= 1) также не выполняется, поэтому функция возвращает 2 * factorial (2 - 1). В следующем вызове аргумент функции

равен 1, поэтому условие if (value <= 1) соблюдается и функция возвращает значение 1. Если представить, что мы строим цепочку рекурсивных вызовов, то её можно изобразить так:

```
factorial(3) // Этот вызов ждёт результат от следующего вызова
     ⬇
factorial(2) // Этот вызов тоже ждёт результат от следующего вызова
     ⬇
factorial(1) // Этот вызов просто возвращает 1
```

В программировании это называется стеком вызовов. Важно понимать, что ни один из рекурсивных вызовов функции в стеке не сможет завершиться, пока самый последний вызов не вернёт результат. Соответственно, когда стек вызовов построен, каждая функция ожидает результат от вызова самой себя, происходит следующее (заметьте, что результат начинает формироваться с конца стэка):

```
factorial(3) // Возвращает 3 * 2, что равно 6
     ⬆
factorial(2) // Возвращает 2 * 1, что равно 2
     ⬆
factorial(1) // Возвращает 1
```

По аналогии построим стек вызовов для factorial(4):

```
factorial(4) // Этот вызов ждёт результат от следующего вызова
     ⬇
factorial(3) // Этот вызов тоже ждёт результат от следующего вызова
     ⬇
factorial(2) // Этот вызов тоже ждёт результат от следующего вызова
     ⬇
factorial(1) // Этот вызов просто возвращает 1
```

А вот как происходит формирование результата:

```
factorial(4) // Возвращает 4 * 6, что равно 24
     ⬆
factorial(3) // Возвращает 3 * 2, что равно 6
     ⬆
factorial(2) // Возвращает 2 * 1, что равно 2
     ⬆
factorial(1) // Возвращает 1
```

Читаемость кода

Кроме сокращения объёма кода, вынос участков кода в функции улучшает читаемость кода. Давайте вспомним как мы генерировали случайное целое число на прошлом уроке:

```
const randomNumber = Math.round(Math.random() * 10);
```

Вынесем код генерации случайного числа в отдельную функцию:

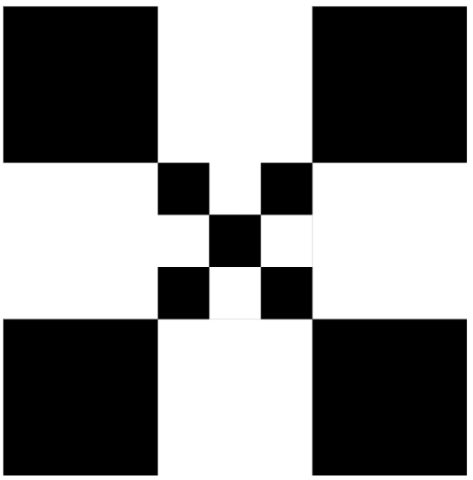
```
function generateRandomNumber(to) {
  return Math.round(Math.random() * to);
}
```

Теперь строка кода, в которой мы задаём переменной случайное значение, выглядит интуитивно понятнее:

```
const randomNumber = generateRandomNumber(10);
```

Функция рисования шахматной доски

Как мы узнали, использование функций позволяют решать более сложные задачи избегая дублирования кода. Для ещё одной демонстрации пользы функций нарисуем такой узор:



Этот узор немного отличается от «шахматной доски», которую мы рисовали на прошлом уроке, отличие состоит в том, что в центральную клетку вписана такая же «шахматная доска», но меньшего размера.

Для начала мы вынесем в функцию код, который написали на прошлом уроке:

```
let isBlack = true;
for (let y = 0; y < 3; y = y + 1) {
  for (let x = 0; x < 3; x = x + 1) {
    let color = 'white';
    if (isBlack) {
      color = 'black';
      isBlack = false;
    } else {
      isBlack = true;
    }
    drawRect(x * 50, y * 50, 50, 50, color);
  }
}
```

Нам необходимо, чтобы функция могла нарисовать шахматную доску указанного нами размера в указанном месте, соответственно она должна принимать следующие аргументы:

1. size — размер доски (сколько клеток в ширину и высоту);
2. left — позиция доски слева от начала холста;
3. top — позиция доски сверху от начала холста;
4. cellWidth — ширина клетки доски.

Теперь, когда мы определились с аргументами функции, мы можем её реализовать:

```
function drawChessboard(size, left, top, cellWidth) {
  let isBlack = true;
  for (let y = 0; y < size; y = y + 1) {
    for (let x = 0; x < size; x = x + 1) {
      let color = 'white';
      if (isBlack) {
        color = 'black';
        isBlack = false;
      } else {
        isBlack = true;
      }
      drawRect(left + x * cellWidth, top + y * cellWidth, cellWidth, cellWidth,
color);
    }
  }
}

drawChessboard(3, 10, 10, 150);
```

Так как количество клеток доски стало зависеть от аргумента *size*, в этих строках мы заменяем число 3 на этот аргумент:

```
for (let y = 0; y < size; y = y + 1) {
```

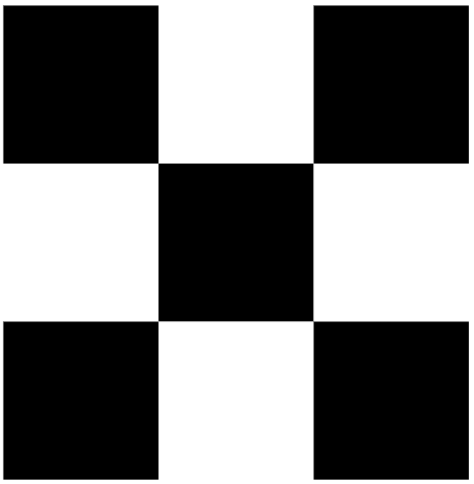
```
    for (let x = 0; x < size; x = x + 1) {
```

Ширина клетки доски теперь тоже регулируется аргументом — cellWidth, поэтому заменяем числа 50 на него:

```
drawRect(left + x * cellWidth, top + y * cellWidth, cellWidth, cellWidth, color);
```

В этой же строке первый аргумент функции drawRect мы увеличиваем на аргумент *left* нашей функции, а второй аргумент на *top*. Это нужно для того, чтобы мы могли нарисовать доску в том месте, в котором захотим. Попробуем как работает наша функция *drawChessboard*:

```
drawChessboard(3, 10, 10, 150);
```



Работает! Осталось в средней клетке нарисовать ещё одну шахматную доску, но меньшего размера. Для этого нужно ещё раз вызвать функции *drawChessboard*:

```
drawChessboard(3, 10, 10, 150);
drawChessboard(3, 160, 160, 50);
```

В качестве второго и третьего аргумента мы передаём 160 потому, что это позиция средней клетки доски, её ещё можно было бы вычислить по формуле *[позиция большой шахматной доски] + [ширина клетки большой шахматной доски]*, то есть *10 + 150*, получается *160*. Последний аргумент равен 50 потому что, чтобы вместить маленькую доску в клетку большой доски, размер клетки маленькой доски должен быть в три раза меньше размера клетки большой доски, чтобы выполнялось следующее равенство: *[размер клетки маленькой доски] + [размер клетки маленькой доски] + [размер клетки маленькой доски] = [размер клетки большой доски]*.

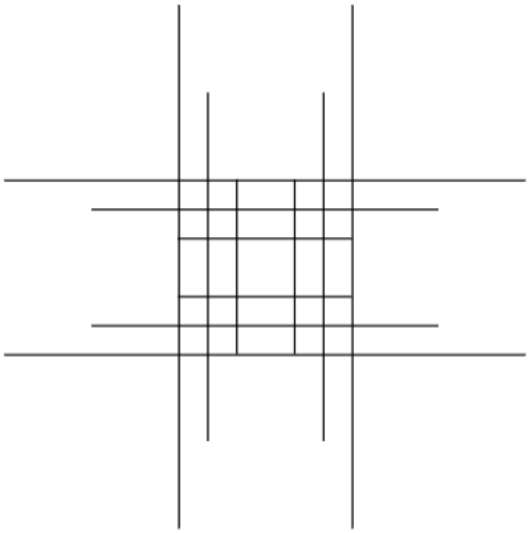
Не забывайте, что функция drawRect не является стандартной для JavaScript, мы добавили её в тренажёр специально для иллюстрации того, как программирование позволяет работать с графикой.

Домашнее задание

Вынесите алгоритм вычисления сложного банковского процента из прошлого домашнего задания в отдельную функцию и усовершенствуйте программу: пусть она выводит сложный процент за количество месяцев, которое указал пользователь, затем за вдвое больший срок, затем за втрое больший срок.

Дополнительно

Вынесите алгоритм рисования сетки для игры в «Крестики-нолики» из прошлого домашнего задания в отдельную функцию и нарисуйте такой узор:



Глоссарий

- Функция — фрагмент программного кода, к которому можно обратиться из другого места программы;
- Рекурсия — вызов функции (процедуры) из неё же самой;
- Стек вызовов — стек, хранящий информацию для возврата управления из подпрограмм (функций) в программу или подпрограмму (при вложенных или рекурсивных вызовах).

Дополнительные материалы

- Цикл for на MDN — https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/for;

- Стрелочные функции — https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Arrow_functions.

Используемые источники

- https://wikipedia.org