Функции

Объявление функции (Function Declaration)

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

Как правило, в имени используются опередеенные префиксы, обозначающие действие, после которых следует объект действия. Например, функции, начинающиеся с
"show..." обычно что-то показывают,
"get..." – возвращают значение,
"calc..." – что-то вычисляют,
"create..." – что-то создают,
"check..." – что-то проверяют и возвращают логическоезначение, и т.д.

Запуск функции: showMessage();
alert(showMessage); - не вызывает функцию, если после ее имени нет круглых скобок, **выведет код функции**

Локальные переменные: переменные, объявленные внутри функции, видны только внутри этой функции. Функция обладает полным доступом к **внешним переменным** и может изменять их значение. Внешняя переменная используется, только если внутри функции нет такой локальной. Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю.

Мы можем передать внутрь функции любую информацию, используя **параметры** (также называемые **аргументами функции**). Если параметр не указан, то его значением становится undefined. Если мы хотим задать параметру text значение по умолчанию, мы должны указать его после =:

```
function showMessage(from, text = "текст не добавлен") {
  alert( from + ": " + text );
}
```

Функция может вернуть результат, который будет передан в вызвавший её код. Директива **return** может находиться в любом месте тела функции. Как только выполнение доходит до этого места, функция останавливается, и значение возвращается в вызвавший её код (присваивается переменной result выше). Результат функции **с пустым return или без него** – undefined

Функциональное выражение (Function Expression)

```
let showMessage = function () {
        console.log('Привет!');
};
showMessage();
```

В отличие от Function Declaration Function Expression создаётся, когда выполнение доходит до него, и затем уже может использоваться. В большинстве случаев, когда нам нужно создать функцию, предпочтительно использовать Function Declaration, т.к. функция будет видима до своего объявления в коде.

Важная особенность Function Declaration заключается в их **блочной области видимости**. В строгом режиме, когда Function Declaration находится в блоке {...}, функция доступна везде внутри блока. Но не снаружи него. Если нужно, чтобы функция была видна снаружи, то нужно воспользоваться Function Expression, и присвоить значение переменной, объявленной снаружи блока, что обеспечит нам нужную видимость:

```
let age = prompt("Сколько Вам лет?", 18);
let welcome;
if (age < 18) {
  welcome = function() {
    alert("Привет!");
  };
} else {
  welcome = function() {
    alert("Здравствуйте!");
```

```
  };
}
welcome(); // теперь всё в порядке
```

Стрелочная функция (arrow function)

let имя переменной = (параметр, ...параметр) => выражение

```
H-p: let sum = (a, b) => a + b;
```

Если у нас только один аргумент, то круглые скобки вокруг параметров можно опустить, сделав запись ещё короче:

```
 let double = n => n * 2;
```

Если нет аргументов, указываются пустые круглые скобки:

```
let sayHi = () => alert("Hello!");
```

Многострочная стрелочная функция:

```
let getMessage = (text, name) => {
        let message = text + ', ' + name + '!';
        return message;
};
console.log(getMessage('Привет', 'Вася'));
```

Функция-конструктор

Функции-конструкторы являются обычными функциями. Но есть два соглашения:

- Имя функции-конструктора должно начинаться с большой буквы.
- Функция-конструктор должна вызываться при помощи оператора "new"

Например:

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}
let user = new User("Вася");
alert(user.name); // Вася
alert(user.isAdmin); // false
```

Если в нашем коде большое количество строк, создающих один сложный объект, мы можем обернуть их в функцию-конструктор следующим образом:

```
let user = new function() {
  this.name = "Вася";
  this.isAdmin = false;
  // ...другой код для создания пользователя
  // возможна любая сложная логика и выражения
  // локальные переменные и т. д.
};
```

Такой конструктор не может быть вызван дважды, так как он нигде не сохраняется, просто создаётся и тут же вызывается.

Обычно конструкторы ничего не возвращают явно. Их задача – записать все необходимое в this, который в итоге станет результатом. Но если return всё же есть, то применяется простое правило:

- При вызове return с объектом, будет возвращён объект, а не this.
- При вызове return с примитивным значением, примитивное значение будет отброшено.

Другими словами, return с объектом возвращает объект, в любом другом случае конструктор вернёт this (возвращается объект неявно).

Мы можем не ставить скобки после new, если вызов конструктора идёт без аргументов.

В this мы можем добавлять не только свойства, но и методы:

```
function User(name) {
  this.name = name;
  this.sayHi = function() {
    alert( "Меня зовут: " + this.name );
  };
}
let vasya = new User("Вася");
vasya.sayHi(); // Меня зовут: Вася
```

Глобальный объект

Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения.

В браузере он называется **window**, в Node.js — **global**, в другой среде исполнения может называться иначе.

Недавно **globalThis** был добавлен в язык как стандартизированное имя для глобального объекта, которое должно поддерживаться в любом окружении. В некоторых браузерах, например Edge не на Chromium, globalThis ещё не поддерживается, но легко реализуется с помощью полифила. Ко всем свойствам глобального объекта можно обращаться напрямую:

```
alert("Привет");
// это то же самое, что и
window.alert("Привет");
```

В браузере глобальные функции и переменные, объявленные с помощью var (не let/const!), становятся свойствами глобального объекта.

Объект функции, NFE

В JavaScript функции – это объекты. Объект функции содержит несколько полезных свойств. Например, имя функции нам доступно как свойство «**name**»:

```
function sayHi() {
  alert("Hi");
}
alert(sayHi.name); // sayHi
```

Ещё одно встроенное свойство «**length**» содержит количество параметров функции в её объявлении. Например:

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}
alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2
```

Как мы видим, троеточие, обозначающее «остаточные параметры», здесь как бы «не считается»
Мы также можем добавить свои **собственные свойства**.

Давайте добавим свойство counter для отслеживания общего количества вызовов:

```
function sayHi() {
  alert("Hi");
  // давайте посчитаем, сколько вызовов мы сделали
  sayHi.counter++;
}
sayHi.counter = 0; // начальное значение
sayHi(); // Hi
sayHi(); // Hi
alert( `Вызвана ${sayHi.counter} раза` ); // Вызвана 2 раза
```

Свойство не есть переменная. Переменные – это не свойства функции и наоборот. Это два параллельных мира.

Named Function Expression или NFE – это термин для Function Expression, у которого есть имя. Например,

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
```

```
};
```

Есть две важные особенности имени func, ради которого оно даётся:

- Оно позволяет функции ссылаться на себя же.
- Оно не доступно за пределами функции.

Синтаксис "new Function"

Существует ещё один вариант объявлять функции. Он используется крайне редко, но иногда другого решения не найти.

```
let func = new Function([arg1, arg2, ...argN], functionBody);
```

Функция создаётся с заданными аргументами arg1...argN и телом functionBody

Главное отличие от других способов объявления функции, которые были рассмотрены ранее, заключается в том, что функция создаётся полностью «на лету» из строки, переданной во время выполнения.

```
let str = ... код, полученный с сервера динамически ...
let func = new Function(str);
func();
```

Это используется в очень специфических случаях, например, когда мы получаем код с сервера для динамической компиляции функции из шаблона, в сложных веб-приложениях.

Когда функция создаётся с использованием new Function, в её [[Environment]] записывается ссылка не на внешнее лексическое окружение, в котором она была создана, а на глобальное. Поэтому такая функция имеет доступ только к глобальным переменным.

Замыкание

В JavaScript у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый лексическим окружением **LexicalEnvironment**. Обычно функция запоминает, где родилась, в специальном свойстве **[[Environment]]**. Это ссылка на лексическое окружение (Lexical Environment), в котором она создана

Объект лексического окружения состоит из двух частей:

- **Environment Record** – объект, в котором как свойства хранятся все локальные переменные (а также некоторая другая информация, такая как значение this). "Переменная" – это просто свойство Environment Record. «Получить или изменить переменную», означает, «получить или изменить свойство этого объекта».
- **Ссылка на внешнее лексическое окружение** – то есть то, которое соответствует коду снаружи (снаружи от текущих фигурных скобок). У глобального лексического окружения нет внешнего окружения, так что она указывает на null

В процессе вызова функции у нас есть два лексических окружения: **внутреннее** (для вызываемой функции) и **внешнее** (глобальное). У внутреннего лексического окружения есть ссылка outer на внешнее. Когда код хочет получить доступ к переменной – сначала происходит поиск во внутреннем лексическом окружении, затем во внешнем, затем в следующем и так далее, до глобального.

Если переменная не была найдена, это будет ошибкой в strict mode. Без strict mode, для обратной совместимости, присваивание несуществующей переменной создаёт новую глобальную переменную с таким именем.

Замыкание – это функция, которая запоминает свои внешние переменные и может получить к ним доступ. В JavaScript, все функции изначально являются замыканиями (есть только одно исключение, про которое будет рассказано в Синтаксис "new Function"). То есть, они автоматически запоминают, где были созданы, с помощью скрытого свойства [[Environment]] и все они могут получить доступ к внешним переменным.

Лексическое окружение существует также для любых блоков кода **{...},** в том числе и для циклов.

```
for (let i = 0; i < 10; i++) {
 // У каждой итерации цикла своё собственное лексическое окружение
 // {i: value}
alert(i); // Ошибка, нет такой переменной
```

В прошлом в JavaScript не было лексического окружения на уровне блоков кода.Так что программистам пришлось что-то придумать. И то, что они сделали, называется «immediately-invoked function expressions» (аббревиатура **IIFE**), что означает функцию, запускаемую сразу после объявления.

```
// Пути создания IIFE

(function() {
  alert("Скобки вокруг функции");
})();

(function() {
  alert("Скобки вокруг всего");
}());

!function() {
  alert("Выражение начинается с логического оператора NOT");
}();

+function() {
  alert("Выражение начинается с унарного плюса");
}();
```

Скобки вокруг функции – это трюк, который позволяет показать JavaScript, что функция была создана в контексте другого выражения, и, таким образом, это функциональное выражение: ей не нужно имя и её можно вызвать немедленно.