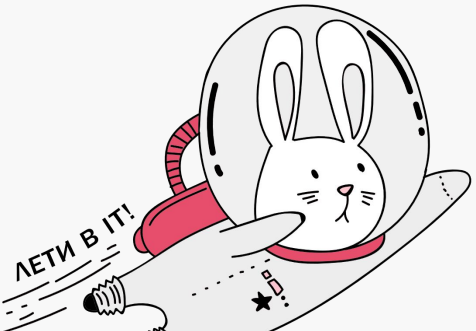
Лексическое окружение
Замыкание
Рекурсия



dev {education}

Глобальный объект

Глобальные означает, что доступные глобально. Глобальными могут быть как переменные так и функции.

Если переменные или функции находятся вне какой либо функции, то они являются глобальными. Такие переменные и функции доступны в любом месте программы.

В JS все глобальные переменные являются свойствами объекта global object.

В браузере этот объект явно доступен под именем window.

Window

Объект window является глобальным объектом и также содержит ряд свойств и методов для работы с окном браузера.

Например:

```
window.open(url, name, params);
```

url - адрес загрузки нового окна.
name - имя окна, свойство target при отправке формы.
params - параметры открытия окна браузера (left/top, width/height).

Также window имеет ряд свойств (location, scrollbars, innerHeight, и другие).

Глобальный объект Node.js

В других окружениях, например NodeJS, глобальный объект может быть недоступен в явном виде, но суть происходящего от этого не меняется.

Присваивая или читая глобальную переменную, мы работаем со свойствами window.

```
var a = 5; // объявление var создает свойство window.a
alert( window.a ); // 5
```

можно присваивать в явном виде

```
window.a = 5;
alert(a); // 5
```

Порядок инициализации

Выполнение скрипта происходит в две фазы:

1. На первой фазе происходит подготовка к запуску.
   Во время инициализации скрипт сканируется на предмет объявления функций, а затем на предмет объявления переменных. Каждое объявление добавляется в window. При этом функции записываются целиком, а переменные равными undefined.

1. На второй фазе - выполнение скрипта.
   Присвоение значений переменных происходит, когда поток доходит до строчки с оператором (=).

Объявление функций

Объявление функций через объект window не рекомендуется, так как может привести к ошибки рекурсии. (IE8)

```
window.recurse = function(times) {     // рекурсия через функцию, явно
  if (times !== 0) recurse(times - 1);     записанную в window
};

recurse(13);
```

Проблема здесь возникает из-за того, что функция напрямую присвоена в window.recurse = ... Проблемы не будет при обычном объявлении функции.

Замыкания

Что это такое – «понимать замыкания», какой смысл обычно вкладывают в эти слова?

«Понимать замыкания» в JavaScript означает понимать следующие вещи:

1. Все переменные и параметры функций являются свойствами объекта переменных LexicalEnvironment.
   Каждый запуск функции создает новый такой объект. На верхнем уровне им является «глобальный объект», в браузере – window.
1. При создании функция получает системное свойство [[Scope]], которое ссылается на LexicalEnvironment, в котором она была создана.
2. При вызове функции, куда бы её ни передали в коде – она будет искать переменные сначала у себя в лексическом окружении, а затем во внешних LexicalEnvironment с места своего «рождения».

Лексическое окружение

Все переменные внутри функции – это свойства специального внутреннего объекта LexicalEnvironment, который создаётся при её запуске. Его называют «лексическое окружение» или просто «объект переменных».

При запуске функция создает объект переменных и записывает туда аргументы, функции и переменные. Процесс инициализации выполняется в том же порядке, что и для глобального объекта, который является частным случаем лексического окружения.

В отличие от window, объект переменных является внутренним и он скрыт от прямого доступа.

Выполнение функции

JS является интерпретируемым языком, поэтому выполнение функции определяется тем, как происходит интерпретация.

До выполнения первой строчки функции, на стадии инициализации, интерпретатор создает объект LexicalEnvironment и заполняет его.

Example

```
sayHi('Вася');

function sayHi(name) {
  // LexicalEnvironment = { name: 'Вася', phrase: undefined }

  var phrase = "Привет, " + name;
  // LexicalEnvironment = { name: 'Вася', phrase: 'Привет, Вася'}

  alert( phrase );
}
```

Очистка лексического окружения

В конце выполнения функции объект переменных обычно очищается из памяти. Однако есть ситуации, при которых объект с переменными сохраняется и после завершения функции.

Доступ к внешним переменным

Из функции мы можем обратиться не только к локальной переменной, но и к внешней.

```
var userName = "Вася";

function sayHi() {
  alert( userName ); // "Вася"
}
```

Доступ к внешним переменным

Интерпретатор, при доступе к переменной, сначала пытается найти переменную в текущем LexicalEnvironment, а затем, если её нет – ищет во внешнем объекте переменных. В данном случае им является window.

Такой порядок поиска возможен благодаря тому, что ссылка на внешний объект переменных хранится в специальном внутреннем свойстве функции, которое называется [[Scope]]. Это свойство закрыто от прямого доступа, но знание о нём очень важно для понимания того, как работает JavaScript.

При создании функция получает скрытое свойство [[Scope]], которое ссылается на лексическое окружение, в котором она была создана.

```
sayHi.[[Scope]] = window;
```

Доступ к внешним переменным

Значение переменной из внешней области берется всегда текущее. Оно может быть уже не то, что было на момент создания функции. Например, в коде ниже функция sayHi берёт phrase из внешней области:

```
var phrase = 'Привет';
function sayHi(name) {
  alert(phrase + ', ' + name);
}

sayHi('Вася');  // Привет, Вася

phrase = 'Пока';
sayHi('Вася'); // Пока, Вася
```

Вложенные функции

Внутри функции можно объявлять не только локальные переменные, но и другие функции. К примеру, вложенная функция может помочь лучше организовать код:

```
function sayHiBye(firstName, lastName) {
  alert( "Привет, " + getFullName() );
  alert( "Пока, " + getFullName() );

  function getFullName() {
     return firstName + " " + lastName;
  }
}
sayHiBye("Вася", "Пупкин"); // Привет, Вася Пупкин ; Пока, Вася Пупкин
```

Вложенные функции

Рассмотрим вариант, при котором внутри одной функции создаётся другая и возвращается в качестве результата.

В разработке UI это совершенно стандартный приём, функция затем может назначаться, как обработчик действий посетителя.

Здесь мы будем создавать функцию-счётчик, которая считает свои вызовы и возвращает их текущее число.

В примере makeCounter создает такую функцию:

Example

```
function makeCounter() {
  var currentCount = 1;

  return function() {
    return currentCount++;
  };
}

var counter = makeCounter(); // каждый вызов увеличивает счётчик и
alert( counter() ); // 1              возвращает результат
alert( counter() ); // 2
alert( counter() ); // 3

var counter2 = makeCounter(); // создать другой счётчик, он будет независим
alert( counter2() ); // 1
```

Свойства функции

Функция в JavaScript является объектом, поэтому можно присваивать свойства прямо к ней, вот так:

```
function f() {}
f.test = 5;
```

Иногда свойства, привязанные к функции, называют «статическими переменными».

В некоторых языках программирования можно объявлять переменную, которая сохраняет значение между вызовами функции. В JavaScript ближайший аналог – такое вот свойство функции.

Замыкание

Замыкание – это функция вместе со всеми внешними переменными, которые ей доступны.

Обычно, говоря «замыкание функции», подразумевают не саму эту функцию, а именно внешние переменные.

Иногда говорят «переменная берётся из замыкания». Это означает – из внешнего объекта переменных.

Рекурсия

В теле функции могут быть вызваны другие функции для выполнения подзадач.

Частный случай - когда функция вызывает сама себя. Это называется рекурсией.

Рекурсия используется для ситуаций, когда выполнение одной сложной задачи можно представить, как некое действие в совокупности с решением той же задачи в более простом варианте.

Example

Рассмотрим возведение в степень

Её можно представить, как совокупность более простого действия и более простой задачи того же типа:

```
function pow(x, n) {
  if (n !== 1) {    // пока n != 1, сводить вычисление pow(x,n) к pow(x,n-1)
    return x * pow(x, n - 1);
  } else {
    return x;
  }
}
```

Глубина рекурсии

Общее количество вложенных вызовов называют глубиной рекурсии. В случае со степенью, всего будет n вызовов.

Максимальная глубина рекурсии в браузерах ограничена, точно можно рассчитывать на 10000 вложенных вызовов, но некоторые интерпретаторы допускают и больше.

Контекст выполнения, Стек

При любом вложенном вызове JavaScript запоминает текущий контекст выполнения в специальной внутренней структуре данных – «стеке контекстов».

Затем интерпретатор приступает к выполнению вложенного вызова. В данном случае вызывается та же pow, однако это абсолютно неважно. Для любых функций процесс одинаков.

Для нового вызова создается свой контекст выполнения, и управление переходит в него, а когда он завершен – старый контекст достается из стека и выполнение внешней функции возобновляется.

Когда вложенных вызовов нет, то функция заканчивает свою работу. Текущий контекст больше не нужен и он удаляется из памяти, из стека восстанавливается предыдущий.