Front-end

JavaScript

JS. Closures

- Замыкания
- Сборщик мусора
- Чистая функция

JS. Closures

В JavaScript функции могут находиться внутри других функций. Когда одна функция находится внутри другой, то **внутренняя** функция имеет доступ к переменным **внешней** функции. Другими словами, внутренняя функция, при вызове как бы «запоминает» место в котором она родилась (имеет ссылку на внешнее окружение).

Замыкание - это такой механизм в JavaScript, который даёт нам доступ к переменным внешней функции из внутренней.

JS. Closures

```
function sayHello() {
  const message = 'Привет, ';
  return function(name) {
    return message + name + '!';
  }
}

const result = sayHello(); // => ƒ (name) { return message + name + '!'; }
console.log(result('Вася')); // => "Привет, Вася!"
```

JS. Closures

Лексическое окружение - это скрытый объект, который связан с функцией и создаётся при её запуске. В нём находятся все локальные переменные этой функции, ссылка на внешнее лексическое окружение, а также некоторая другая информация.

JS. Closures

Поиск переменной всегда начинается с текущего лексического окружения. Т.е., если переменная будет сразу найдена в текущем лексическом окружении, то её дальнейший поиск прекратится и возвратится значение, которая эта переменная имеет здесь. Если искомая переменная в текущем окружении не будет найдена, то произойдёт переход к следующему окружению (ссылка на которое имеется в текущем). Если она не будет найдена в этом, то опять произойдёт переход к следующему окружению, и т.д. Если при поиске переменной, она будет найдена, то её дальнейший поиск прекратится и возвратится значение, которая она имеет здесь.

JS. Closures

Лексические окружения создаются и изменяются в процессе выполнения кода.

```
function sayHello() {
  return function(name) {
    return message + name + '!';
  }
}

const result = sayHello(); // ƒ (name) { return message + name + '!'; }
let message = 'Привет, ';
console.log(result('Вася')); // => "Привет, Вася!"
message = 'Здравствуйте, ';
console.log(result('Вася')); // => "Здравствуйте, Вася!"
```

JS. Closures

Когда мы первый раз вызываем функцию result('Вася'), в глобальном лексическом окружении переменная message имеет значение 'Привет, '. В результате мы получим строку "Привет, Вася!". При втором вызове переменная message имеет уже значение 'Здравствуйте, '. В результате мы уже получим строку "Здравствуйте, Вася!«

Для чего нужны замыкания? Замыкания, например, могут использоваться для «запоминания» параметров, защиты данных (инкапсуляции), привязывания функции к определённому контексту и др

JS. Closures

Примеры:

```
function getCounter() {
  let counter = 0;
  return function() {
    return counter++;
  }
}
let count = getCounter();
console.log(count());  // => 0
console.log(count());  // => 1
console.log(count());  // => 2
```

JS. Closures

Примеры:

```
function person() {
  let name = 'Peter';
    return function displayName() {
    console.log(name);
  };
}
let peter = person();
peter(); // => выведет 'Peter'
```

JS. Closures

Сборщик мусора

В JavaScript лексическое окружение обычно удаляется после того, как функция выполнилась. Это происходит только тогда, когда у нас нет ссылок на это окружение.

```
function sayHello(name) {
  return 'Привет, ' + name + '!';
}

console.log(sayHello('Вася')); // => "Привет, Вася!"
```

Но в вышеприведённых примерах со вложенными функциями, у нас лексическое окружение внешней функции оставалась доступным после её выполнения. Т.к. на на неё оставалась ссылка у вложенной функции. А пока есть доступ к лексическому окружению, автоматический сборщик мусора не может его удалить, и оно остаётся держаться в памяти.

JS. Closures

Сборщик мусора

Управление памятью в JavaScript выполняется автоматически и незаметно. Мы создаём примитивы, объекты, функции… Всё это занимает память.

Основной концепцией управления памятью в JavaScript является принцип достижимости.

Если упростить, то «достижимые» значения – это те, которые доступны или используются. Они гарантированно находятся в памяти.

Существует базовое множество достижимых значений, которые не могут быть удалены.

JS. Closures

Сборщик мусора

Например:

Выполняемая в данный момент функция, её локальные переменные и параметры.

Другие функции в текущей цепочке вложенных вызовов, их локальные переменные и параметры.

Глобальные переменные.

(некоторые другие внутренние значения)

Эти значения мы будем называть корнями.

Любое другое значение считается достижимым, если оно доступно из корня по ссылке или по цепочке ссылок.

Например, если в глобальной переменной есть объект, и он имеет свойство, в котором хранится ссылка на другой объект, то этот объект считается достижимым. И те, на которые он ссылается, тоже достижимы. В движке JavaScript есть фоновый процесс, который называется сборщиком мусора. Он отслеживает все объекты и удаляет те, которые стали недоступными.

JS. Closures

Сборщик мусора

Сборка мусора выполняется автоматически. Мы не можем ускорить или предотвратить её.

Объекты сохраняются в памяти, пока они достижимы.

JS. Pure function

Чистая функция

Чистая функция — это функция, которая не вызывает побочных эффектов (side effects), то есть никак не влияет на состояние внешнего мира.

Чистую функцию можно сравнить с понятием функции из математики: это нечто, что преобразует входные данные по заданным правилам.

Чистые функции всегда при вводе одинаковых аргументов выдают одинаковый результат. По этому свойству легко отличить чистую функцию от нечистой.

Например, pureFn() при вводе 10 и 20 всегда будет возвращать 15, значит она чистая:

```
function pureFn(a, b) {
  return ((a + b) * a) / b
}
```

JS. Pure function

Чистая функция

A impureFn() нечистая — она будет возвращать разные значения, потому что использует случайное число:

```
function impureFn(a, b) {

  return ((a + b) * a) / Math.random()

}
```

Функция производит побочный эффект, потому что обращается к глобальному объекту Math. Любое взаимодействие с чем-либо «снаружи» функции считается побочным эффектом, даже получение значений.

Дело в том, что мы не знаем, как именно устроены методы random() в объектах снаружи. Они могут не только возвращать результат, но и менять состояние окружающего мира, например, меняя какую-то переменную.

JS. Pure function

Чистая функция

«Нечистые» функции бывают разных форм и размеров. Вот некоторые примеры:

- функции, вывод которых зависит от внешнего / глобального состояния;
- функции, которые возвращают разные исходные данные при одинаковых входных;
- функции, которые изменяют состояние приложения;
- функции, которые изменяют «внешний мир».