Incode Group Workshop #2
Области видимости переменных,
замыкания, каррирование в Javascript

Замыкания (Closure), области видимости переменной (Scope).

- Глобальный объект;
- Замыкания, функции изнутри;
- [[Scope]] для new Function;
- Локальные переменные объекта;
- Модули через замыкания.
- Управление памятью.
- Переменные: let и const.

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Глобальный объект

- **Декларация функции и переменных**
- Порядок инициализации
- Конструкции for, if... не влияют на видимость переменных
- Не важно, где и сколько раз объявлена переменная

В JavaScript все глобальные переменные и функции являются свойствами специального объекта, который называется «глобальный объект» (global object).

Node.JS => global

Browsers => window

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

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

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Глобальный объект

- Декларация функции и переменных
- **Порядок инициализации**
- Конструкции for, if... не влияют на видимость переменных
- Не важно, где и сколько раз объявлена переменная

1) Инциализация
   a) Function Declaration
   b) var => undefined
2) Выполнение

```
// На момент инициализации, до выполнения кода:
// window = { f: function, a: undefined, g: undefined }

var a = 5;
// window = { f: function, a: 5, g: undefined }

function f(arg) { /*...*/ }
// window = { f: function, a: 5, g: undefined } без изменений, f обработана ранее

var g = function(arg) { /*...*/ };
// window = { f: function, a: 5, g: function }
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Глобальный объект

- Декларация функции и переменных
- Порядок инициализации
- **Конструкции for, if... не влияют на видимость переменных**
- Не важно, где и сколько раз объявлена переменная

Фигурные скобки, которые используются в **for**, **while**, **if**, в отличие от объявлений функции, имеют «декоративный» характер.
В JavaScript нет разницы между объявлением вне блока:

```
var i;
{
  i = 5;
}
```

…И внутри него:

```
i = 5;
{
  var i;
}
```

Также нет разницы между объявлением в цикле и вне его:

```
for (var i = 0; i < 5; i++) { }
```

Идентичный по функциональности код:

```
var i;
for (i = 0; i < 5; i++) { }
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Глобальный объект

- Декларация функции и переменных
- Порядок инициализации
- Конструкции for, if... не влияют на видимость переменных
- **Не важно, где и сколько раз объявлена переменная**

Объявлений var может быть сколько угодно:

```
var i = 10;

for (var i = 0; i < 20; i++) {

   ...

}

var i = 5;
```

Все var будут обработаны один раз, на фазе инициализации.

На фазе исполнения объявления **var** будут проигнорированы: они уже были обработаны. Зато будут выполнены присваивания.

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Замыкания,функции изнутри

- **Лексическое окружение**
- Доступ к внешним переменным
- Всегда текущее значение
- Вложенные функции
- Свойства функции

Все переменные внутри функции – это свойства специального внутреннего объекта LexicalEnvironment, который создаётся при её запуске.

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

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

```
function sayHi(name) {
  // LexicalEnvironment = { name: 'Вася',
phrase: undefined }
  var phrase = "Привет, " + name;

  // LexicalEnvironment = { name: 'Вася',
phrase: 'Привет, Вася'}
  alert( phrase );
}

sayHi('Вася');
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Замыкания,функции изнутри

- Лексическое окружение
- **Доступ к внешним переменным**
- Всегда текущее значение
- Вложенные функции
- Свойства функции

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

```
var userName = "Вася";

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

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.

- Каждая функция при создании получает ссылку [[Scope]] на объект с переменными, в контексте которого была создана.
- При запуске функции создаётся новый объект с переменными LexicalEnvironment. Он получает ссылку на внешний объект переменных из [[Scope]].
- При поиске переменных он осуществляется сначала в текущем объекте переменных, а потом – по этой ссылке.

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Замыкания,функции изнутри

- Лексическое окружение
- Доступ к внешним переменным
- **Всегда текущее значение**
- Вложенные функции
- Свойства функции

Значение переменной из внешней области берётся 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 (**)
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Замыкания,функции изнутри

- Лексическое окружение
- Доступ к внешним переменным
- Всегда текущее значение
- **Вложенные функции**
- Свойства 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


getFullName.[[Scope]] = объект
переменных текущего запуска sayHiBye
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Замыкания,функции изнутри

- Лексическое окружение
- Доступ к внешним переменным
- Всегда текущее значение
- Вложенные функции
- **Свойства функции**

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

```
function f() {}

f.test = 5;
alert( f.test );
```

Свойства функции не стоит путать с переменными и параметрами. Они совершенно никак не связаны. Переменные доступны только внутри функции, они создаются в процессе её выполнения. Это – использование функции «как функции».

А свойство у функции – доступно отовсюду и всегда. Это – использование функции «как объекта».

Если хочется привязать значение к функции, то можно им воспользоваться вместо внешних переменных.

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

- 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.
- При создании функция получает системное свойство [[Scope]], которое ссылается на LexicalEnvironment, в котором она была создана.
- При вызове функции, куда бы её ни передали в коде – она будет искать переменные сначала у себя, а затем во внешних LexicalEnvironment с места своего «рождения».

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

[[Scope]] для new Function

- **Функции, создаваемые через new Function, имеют значением [[Scope]] не внешний объект переменных, а window.**
- Следствие – такие функции не могут использовать замыкание. Но это хорошо, так как бережёт от ошибок проектирования, да и при сжатии JavaScript проблем не будет. Если же внешние переменные реально нужны – их можно передать в качестве параметров.

```
var a = 1;

function getFunc() {
  var a = 2;

  var func = new Function('', 'alert(a)');

  return func;
}

getFunc()(); // 1, из window
```

```
var a = 1;

function getFunc() {
  var a = 2;

  var func = function() { alert(a); };

  return func;
}

getFunc()(); // 2, из LexicalEnvironment
функции getFunc
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

[[Scope]] для new Function

- Функции, создаваемые через new Function, имеют значением [[Scope]] не внешний объект переменных, а window.
- **Следствие – такие функции не могут использовать замыкание. Но это хорошо, так как бережёт от ошибок проектирования, да и при сжатии JavaScript проблем не будет. Если же внешние переменные реально нужны – их можно передать в качестве параметров.**

Если внутри функции, создаваемой через new Function, всё же нужно использовать какие-то данные – без проблем, нужно всего лишь предусмотреть соответствующие параметры и передавать их явным образом, например так:

```
var sum = new Function('a, b', '
return a + b; ');

var a = 1, b = 2;

alert( sum(a, b) ); // 3
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Локальные переменные объекта

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

  // возвращаемся к функции
  function counter() {
    return currentCount++;
  }

  // ...и добавляем ей методы!
  counter.set = function(value) {
    currentCount = value;
  };

  counter.reset = function() {
    currentCount = 1;
  };

  return counter;
}

var counter = makeCounter();

alert( counter() ); // 1
alert( counter() ); // 2

counter.set(5);
alert( counter() ); // 5
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Модули через замыкания

- **Зачем нужен модуль?**
- Зачем скобки вокруг функции?
- Экспорт значения
- Экспорт через return

Приём программирования «модуль» имеет громадное количество вариаций.

Его цель – скрыть внутренние детали реализации скрипта. В том числе: временные переменные, константы, вспомогательные мини-функции и т. п.

```
(function() {

  // глобальная переменная нашего
скрипта
  var message = "Привет";

  // функция для вывода этой переменной
  function showMessage() {
    alert( message );
  }

  // выводим сообщение
  showMessage();

})();
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Модули через замыкания

- Зачем нужен модуль?
- **Зачем скобки вокруг функции?**
- Экспорт значения
- Экспорт через return

Общее правило таково:

- Если браузер видит function в основном потоке кода – он считает, что это Function Declaration.
- Если же function идёт в составе более сложного выражения, то он считает, что это Function Expression.
- Для этого и нужны скобки – показать, что у нас Function Expression, который по правилам JavaScript можно вызвать «на месте».

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Модули через замыкания

- Зачем нужен модуль?
- Зачем скобки вокруг функции?
- **Экспорт значения**
- Экспорт через return

Приём «модуль» используется почти во всех современных библиотеках. Ведь что такое библиотека? Это полезные функции, ради которых её подключают, плюс временные переменные и вспомогательные функции, которые библиотека использует внутри себя. Если подключить библиотеку Lodash, то появится специальная переменная lodash (короткое имя _), которую можно использовать как функцию, и кроме того в неё записаны различные полезные свойства,

```
;(function() {

  // lodash – основная функция для библиотеки
  function lodash(value) {
    // ...
  }

  // вспомогательная переменная
  var version = '2.4.1';
  // ... другие вспомогательные переменные и функции

  // код функции size, пока что доступен только внутри
  function size(collection) {
    return Object.keys(collection).length;
  }

  // присвоим в lodash size и другие функции, которые
нужно вынести из модуля
  lodash.size = size
  // lodash.defaults = ...
  // lodash.cloneDeep = ...

  // "экспортировать" lodash наружу из модуля
  window._ = lodash; // в оригинальном коде здесь
сложнее, но смысл тот же

}());
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Модули через замыкания

- Зачем нужен модуль?
- Зачем скобки вокруг функции?
- Экспорт значения
- **Экспорт через return**

Можно оформить модуль и чуть по-другому, например передать значение через return:

```
var lodash = (function() {

  var version;
  function assignDefaults() { ... }

  return {
    defaults: function() {  }
  }

})();
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Управление памятью

Объект переменных внешней функции существует в памяти до тех пор, пока существует хоть одна внутренняя функция, ссылающаяся на него через свойство [[Scope]].

```
function f() {
  var value = 123;

  function g() {}

  return g;
}

var g = f(); // функция g жива
// а значит в памяти остается соответствующий объект переменных f()

g = null; // ..а вот теперь память будет очищена
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Управление памятью

Современные JS-движки делают оптимизации замыканий по памяти. Они анализируют использование переменных и в случае, когда переменная из замыкания абсолютно точно не используется, удаляют её.

Важный побочный эффект в V8 (Chrome, Opera) состоит в том, что удалённая переменная станет недоступна и при отладке!

```
var value = "Сюрприз";

function f() {
  var value = "самое близкое значение";

  function g() {
    debugger; // выполните в консоли alert( value ); Сюрприз!
  }

  return g;
}

var g = f();
g();
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Переменные: let и const

- **let**
  - область видимости переменной let – блок {...}.
  - переменная let видна только после объявления.
  - При использовании в цикле, для каждой итерации создаётся своя переменная.
- const

В ES-2015 предусмотрены новые способы объявления переменных: через let и const вместо var.

Например:

```
let a = 5;
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Переменные: let и const

- let
  - **область видимости переменной let – блок {...}.**
  - переменная let видна только после объявления.
  - При использовании в цикле, для каждой итерации создаётся своя переменная.
- const

```
var apples = 5;

if (true) {
  var apples = 10;

  alert(apples); // 10 (внутри
блока)
}

alert(apples); // 10 (снаружи блока
то же самое)

------------------------------------

let apples = 5; // (*)

if (true) {
  let apples = 10;

  alert(apples); // 10 (внутри
блока)
}

alert(apples); // 5 (снаружи блока
значение не изменилось)
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Переменные: let и const

- let
  - область видимости переменной let – блок {...}.
  - **переменная let видна только после объявления.**
  - При использовании в цикле, для каждой итерации создаётся своя переменная.
- const

```
alert(a); // undefined

var a = 5;


_____________________________________

alert(a); // ошибка, нет такой
переменной

let a = 5;

_____________________________________

let x;
let x; // ошибка: переменная x уже
объявлена
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Переменные: let и const

- let
  - область видимости переменной let – блок {...}.
  - переменная let видна только после объявления.
  - **при использовании в цикле, для каждой итерации создаётся своя переменная.**
- const

```
for(var i=0; i<10; i++) { /* … */ }

alert(i); // 10

-----------------------------------

function makeArmy() {

  let shooters = [];

  for (let i = 0; i < 10; i++) {
    shooters.push(function() {
      alert( i ); // выводит свой
номер
    });
  }

  return shooters;
}

var army = makeArmy();

army[0](); // 0
army[5](); // 5
```

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Переменные: let и const

- let
  - область видимости переменной let – блок {...}.
  - переменная let видна только после объявления.
  - При использовании в цикле, для каждой итерации создаётся своя переменная.
- **const**

Объявление const задаёт константу, то есть переменную, которую нельзя менять:

```
const apple = 5;
apple = 10; // ошибка
```

В остальном объявление const полностью аналогично let.

Заметим, что если в константу присвоен объект, то от изменения защищена сама константа, но не свойства внутри неё:

```
const user = {
  name: "Вася"
};

user.name = "Петя"; // допустимо
user = 5; // нельзя, будет ошибка
```

То же самое верно, если константе присвоен массив или другое объектное значение.

Incode Group Workshop #2
Замыкания (Closure), области видимости переменной (Scope).

Переменные: let и const

Переменные let:

- **Видны только после объявления и только в текущем блоке.**
- **Нельзя переобъявлять (в том же блоке).**
- **При объявлении переменной в цикле for(let …) – она видна только в этом цикле. Причём каждой итерации соответствует своя переменная let.**

Переменная const – это константа, в остальном – как let.

Incode Group Workshop #2
Области видимости переменных,
замыкания, каррирование в Javascript

Каррирование.

- Почему называется каррирование?
- Что это такое?
- Зачем это нужно?

Incode Group Workshop #2
Каррирование.

Каррирование.

- **Почему называется каррирование?**
- Что это такое?
- Зачем это нужно?

Haskell Brooks Curry

Incode Group Workshop #2
Каррирование.

Каррирование.

- Почему называется каррирование?
- **Что это такое?**
- Зачем это нужно?

Преобразование функции нескольких аргументов в фукнцию одного аргумента называется каррированием.

Или же иначе, Карринг (currying) или каррирование – термин функционального программирования, который означает создание новой функции путём фиксирования аргументов существующей.

```
multiply = (n, m) => (n * m)
multiply(3, 4) === 12 // true

curryedMultiply = (n) => ( (m) =>
multiply(n, m) )
triple = curryedMultiply(3)
triple(4) === 12 // true
```

Incode Group Workshop #2
Каррирование.

Каррирование.

- Почему называется каррирование?
- Что это такое?
- **Зачем это нужно?**

Частичное применение:

```
multiply = (n, m) => n * m
multiply(3, 4) === 12 // true

triple = (m) => multiply(3, m)
triple(4) === 12 // true
```

React - Redux

```
export default connect(mapStateToProps)
(TodoApp)
```

Event Handling:

```
const handleChange = (fieldName) => (event) =>
{
  saveField(fieldName, event.target.value)
}
<input type="text"
onChange={handleChange('email')} ... />
```

Html rendering

```
renderHtmlTag = tagName => content => `<$
{tagName}>${content}</${tagName}>`

renderDiv = renderHtmlTag('div')
renderH1 = renderHtmlTag('h1')

console.log(
  renderDiv('this is a really cool div'),
  renderH1('and this is an even cooler h1')
)
```

Incode Group Workshop #2
Каррирование.

Каррирование.

```
handleChangeColor = color => {
    const { colorStylePropName } = this.state
    return () => {
      this.handleCloseColorOptions()
      const { focusedSheet, onCommitSheetChanges } = this.props
      if (!focusedSheet) return

      const hotInstance = TablesContext.getHotInstance()
      const [selection] = hotInstance && hotInstance.getSelected()

      onCommitSheetChanges(focusedSheet, [
        {
          type: CHANGE_TYPES.updateCellStyle,
          selection,
          style: { [colorStylePropName]: color },
        },
      ])
    }
  }
```

Пример кода из реального проекта:

```
<div className="colorOptionsContainer">
              {colorsWithInheritedColor.map((c, index) => {
                const column = c === 'inherit' ? 1 : Math.floor(index / 4) + 1
                const row = c === 'inherit' ? 6 : 5 - (index % 4)
                return (
                  <span
                    className="colorOptionsItem"
                    key={`keyColor${c}`}
                    role="button"
                    style={{
                      backgroundColor: c,
                      gridColumnStart: column,
                      gridColumnEnd: column + 1,
                      gridRowStart: row,
                      gridRowEnd: row + 1,
                      border: (c === 'inherit' && '1px solid black') || 'none',
                    }}
                    tabIndex={0}

onMouseDown={this.handleChangeColor(c)}
                  >
                     
                  </span>
                )
              })}
            </div>
```