

Глава 5

Контекст исполнения. Объект `this`

5.1. Ключевое слово `this`

5.1.1. Ключевое слово `this` в различных языках программирования

Ключевое слово `this` встречается в различных языках программирования. Например, в языке C# его можно использовать следующим образом. Объявляется класс `User` (Пользователь), который состоит из приватного поля `age` и публичного метода `ShowAge`:

```
class User
{
    private int age = 24;

    public void ShowAge()
    {
        Console.WriteLine(this.age);
    }
}

static void Main()
{
    User mike = new User();

    mike.ShowAge(); // 24
}
```

Метод `ShowAge` выводит на консоль приватное поле `age`. Для этого он

обращается к нему через ключевое слово `this`.

Аналогичный пример можно рассмотреть в языке программирования Java:

```
public class User {
    private int age = 24;

    public void showAge() {
        System.out.println(this.age);
    }
}

public static void main(String []args){
    User mike = new User();

    mike.showAge();
}
```

Также, как и в C#, обращение к приватному полю `age` происходит через ключевое слово `this`.

5.1.2. Свойства `this`

Во всех рассмотренных языках программирования `this` обладает рядом свойств:

- `this` является ключевым словом языка.
- Следовательно, `this` нельзя перезаписать, то есть в него нельзя положить другое значение каким-либо способом.
- `this` всегда ссылается на текущий объект.

5.1.3. Ключевое слово `this` в JavaScript

Аналогичный пример можно написать на JavaScript. Для этого объявим функцию `User`, которая будет возвращать объект. Этот объект состоит из двух полей: поля `age` (аналог свойства) и поля `showAge` (аналог метода).

```
function User () {
    return {
        age: 24,

        showAge: function () {
            console.log(this.age);
        }
    }
}
```

```

    }
  }
}

var mike = new User();

mike.showAge(); // 24

```

В методе `showAge` обращение к свойству `age` происходит также с помощью ключевого слова `this`.

Такое поведение полностью аналогично другим языкам программирования. Однако, в отличие от них, в JavaScript ключевое слово `this` обладает уникальным свойством. А именно `this` в JavaScript можно использовать и за пределом объекта.

Если в консоли браузера вызвать свойство `innerWidth` у ключевого слова `this`, находясь в глобальной области видимости, будет возвращена ширина текущего окна браузера:

```
this.innerWidth; // 1280
```

В интерпретаторе NodeJS, если вызвать свойство `process`, у которого, в свою очередь, вызвать свойство `version`, будет возвращена текущая версия NodeJS:

```
this.process.version; // v7.0.0
```

Таким образом, ключевое слово `this` может быть использовано в глобальной области видимости за пределом какого-либо объекта и это работает корректно.

5.2. Контекст исполнения

5.2.1. Область видимости (повторение)

Ранее была рассмотрена такая тема, как лексическая область видимости. Для того, чтобы понять, как работает ключевое слово `this`, следует углубить эти знания и рассмотреть, что такое контекст исполнения.

Напомним, что любая функция или переменная, которые описаны не в рамках другой функции, попадают в глобальную область видимости:

```

function sum(a, b) {
  return a + b;
}

// Область видимости:
// { sum }
// { a, b }

```

```

}
sum(1, 2);

```



А переменные и функции, которые описаны в рамках другой функции, попадают в область видимости последней функции.

```

function sum(a, b) {
  return a + b;
}

sum(1, 2);

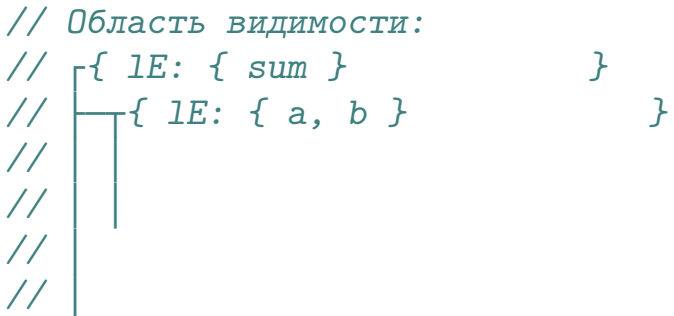
```

// Область видимости:

```

// { lE: { sum } }
// { lE: { a, b } }

```



5.2.2. Контекст исполнения

Контекст исполнения содержит область видимости. Далее область видимости кратко будет обозначаться с помощью аббревиатуры lE (lexical environment). Кроме области видимости, контекст исполнения содержит ключевое слово `this`, которое определяется в момент интерпретации участка кода.

```

function sum(a, b) {
  return a + b;
}

sum(1, 2);

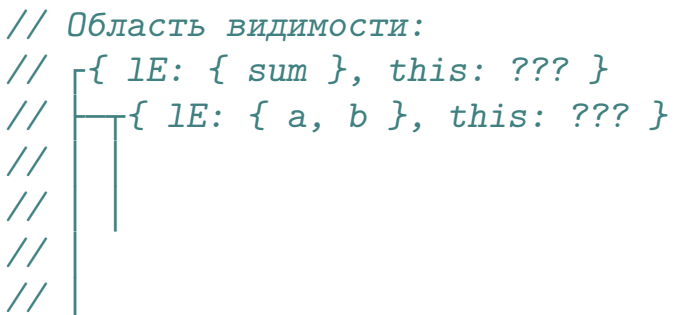
```

// Область видимости:

```

// { lE: { sum }, this: ??? }
// { lE: { a, b }, this: ??? }

```



Значение `this` зависит от следующих факторов:

1. Типа участка кода
2. Как мы попали на этот участок кода
3. Режимы работы интерпретатора

5.2.3. Значение `this` в глобальном участке кода

Как уже было сказано, значение `this` в контексте исполнения зависит от типа участка кода.

Например, если в браузере обратиться к полю `innerWidth` ключевого слова `this`, будет получено некоторое значение, для определенности 1280.

```
this.innerWidth;    // 1280
window.innerWidth;  // 1280
```

Это значение равно ширине окна браузера. Точно также это значение можно получить, если обратиться к полю `innerWidth` глобального объекта `window`.

В интерпретаторе NodeJS обращение к полю `process.version` позволяет получить строку с текущей версией NodeJS:

```
this.process.version;    // "v7.0.0"
global.process.version;  // "v7.0.0"
```

То же самое значение можно получить, если обратиться к полю `process.version` объекта `global`.

Многие, наверняка, использовали `console.log` для вывода на консоль нужные значения.

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

В этом случае на самом деле вызывается свойство `console.log` глобального объекта:

```
global.console.log('Hello!');
this.console.log('Hello!');
```

Тот же самый эффект можно получить, вызвав метод `console.log` у ключевого слова `this`.

Более того, во всех предыдущих примерах `this` равен глобальному объекту:

```
this === global; // true
```

5.2.4. Значение `this` в модуле Node.js

Участок кода может быть не только глобальным, но и быть написанным в рамках модуля NodeJS. Например, пусть дан следующий файл:

```
// year-2016.js

module.exports.days = 366;
```

Это простейший модуль, в рамках которого будут описываться свойства 2016 года. Для того, чтобы свойства стали доступными извне, они должны быть экспортированы: необходимо положить свойства в объект `module.exports`.

Оказывается, что сократить запись можно, если записывать свойства в ключевое слово `this`:

```
this.isLeapYear = true;
```

Это аналог `module.exports` и обе записи работают одинаково.

Для того, чтобы использовать свойство извне модуля, например в файле `index.js`, нужно сперва импортировать модуль:

```
// index.js

var year2016 = require('./year-2016');

year2016.days; // 366;
year2016.isLeapYear; // true;
```

Импортирование модуля производится при помощи вызова функции `require`. На вход функции передается путь до нашего модуля, а в результате будет возвращен как раз тот самый объект, который был экспортирован из модуля.

После этого через обращение к свойствам этого объекта можно получить нужные значения.

5.2.5. Значение `this` при простом вызове функции

Еще одним фактором, который влияет на значение ключевого слова `this`, является то, как именно произошел переход к участку кода.

Например, пусть объявлена функция `getSelf`, которая возвращает текущее значение `this`:

```
function getSelf() {
    return this;
}
```

```
getSelf(); // global
```

Возвращаемое этой функцией значение всегда будет равно значению `this` на том участке кода, на котором она была вызвана. То есть, если функция была вызвана в глобальном контексте, она вернет глобальный объект: `global` (в случае NodeJS) или `window` (в случае работы в браузере).

Если описать эту функцию в контексте NodeJS модуля и вызвать внутри модуля:

```
// year-2016.js

module.exports.days = 366;

function getSelf() {
```

```
        return this;
    }

    getSelf(); // { days: 366 }
```

функция вернет экспортируемое этим модулем значение.

5.2.6. Значение `this` при вызове метода объекта

Еще один способ вызвать функцию — вызвать ее как метод объекта.

```
var block = {
    innerHeight: 300,

    getHeight: function () {
        return this.innerHeight;
    }
}
```

Здесь объект `block` состоит из двух полей: поля `innerHeight` и поля `getHeight`, которое содержит метод, возвращающий значение высоты. Внутри метода значение высоты берется из ключевого слова `this`.

Важно понимать, что значение ключевого слова `this` определяется не в момент определения функции, а в момент ее вызова. Если функция `getHeight` вызывается от объекта `block`, значение `this` будет равно объекту `block`:

```
var block = {
    innerHeight: 300,

    getHeight: function () {
        return this.innerHeight;
    }
}

block.getHeight(); // 300
```

В данном случае будет получено значение 300.

5.2.7. Заимствование метода

Если же сперва функция `getHeight` будет положена в некоторую переменную, то при вызове через эту переменную, значение ключевого слова `this` будет равно глобальному объекту:

```

var block = {
  innerHeight: 300,

  getHeight: function () {
    return this.innerHeight;
  }
}

var getHeight = block.getHeight;
getHeight(); // 1280

```

При работе в браузере значение `this` будет `window` и значение поля `innerHeight` будет взято из объекта `window`. Такое поведение может поначалу сбивать с толку, однако оно получило широкое распространение и называется «заимствование методов».

5.2.8. Заимствование метода. Метод `call`

Чтобы вызвать метод одного объекта в контексте другого объекта, можно использовать метод `call` функции.

Согласно [Function.prototype.call\(\) - JavaScript | MDN](#), метод `call()` вызывает функцию с указанным значением `this` и индивидуально предоставленными аргументами.

Этот метод функции имеет следующую сигнатуру:

```
fun.call(thisArg, arg1, arg2, ...)
```

Первым аргументом передается контекст, а остальные аргументы являются аргументами, с которыми функция будет вызвана.

Например, пусть даны два объекта `mike` (содержит свойство `age` и метод `getAge`) и `anna` (содержит только свойство `age`):

```

var mike = {
  age: 24,

  getAge: function () {
    return this.age;
  }
}

var anna = {
  age: 21
}

```


Значение ключевого слова `this` определяется в момент вызова функции, а не в момент объявления. Поэтому можно воспользоваться методом `getAge` Михаила, чтобы узнать возраст Анны, через `call`:

```
mike.getAge.call(anna); // 21
```

Таким образом, в качестве значения ключевого слова `this` в методе `getAge` будет объект `anna` и, следовательно, будет получено значение свойства `age` объекта `anna`.

На практике заимствование методов используется для того, чтобы превратить массивоподобный объект `arguments` в массив. Для этого заимствуется метод `slice` массива и вызывается в контексте `arguments`:

```
function func() {  
    var args = Array.prototype.slice.call(arguments);  
}
```

Таким образом, в переменной `args` окажется массив.

5.2.9. Заимствование метода. Метод `apply`

Другой метод для работы с контекстом исполнения — это метод `apply`:

Согласно [Function.prototype.apply\(\) - JavaScript | MDN](#), метод `apply()` вызывает функцию с указанным значением `this` и аргументами, представленными в виде массива.

Он ведет себя аналогичным образом, но имеет другую сигнатуру:

```
fun.apply(thisArg, [arg1, arg2]);
```

Метод `apply` всегда принимает два аргумента. Первый аргумент будет использован в качестве ключевого слова `this`. Второй аргумент содержит в себе массив аргументов, с которыми будет вызвана функция `fun`.

Можно продемонстрировать работу метода `apply` на примере функции `min` из библиотеки `Math` (математическая библиотека):

```
Math.min(4, 7, 2, 9); // 2
```

Пусть нужно найти минимум среди элементов некоторого массива. Если передать массив в качестве единственного аргумента, желаемый результат получен не будет:

```
var arr = [4, 7, 2, 9];  
Math.min(arr); // NaN
```

Функция `Math.min`, на самом деле, приводит каждый аргумент к числу, в том числе и переданный в качестве единственного аргумента массив будет приведен к значению `NaN`. Соответственно, результатом выполнения также будет `NaN`.

Чтобы переписать пример правильно, можно воспользоваться методом `apply`:

```
Math.min.apply(Math, arr); // 2
```

Здесь `min` вызвана в контексте `Math` с аргументами из `arr`. Но, поскольку `min` в своей реализации не использует ключевое слово `this`, в качестве `this` можно передать `null`:

```
Math.min.apply(null, arr); // 2
```

5.2.10. Callback

Еще один способ вызова функции — вызов функции как коллбэк:

```
var person = {
  name: 'Sergey',
  items: ['keys', 'phone', 'banana'],

  showItems: function () {
    this.items.map(function (item) {
      return this.name + ' has ' + item;
    });
  }
}
```

В данном случае объект `person` состоит из трех полей: свойств `name` и `items`, а также метода `showItems`.

Как уже много раз упоминалось, значение `this` определяется в момент вызова функции. Пусть метод `showItems` вызывается как метод объекта `person`:

```
person.showItems();
```

В этом случае значение первого `this` будет `person`. В качестве второго ключевого слова `this` будет подставлен объект `global`, поскольку контекст никак явным образом не задан. А значит поле `name` будет браться не у объекта `person`, а у объекта `global`.

Если `name` в `global` не определено, результат будет следующий:

```
'undefined has keys'
'undefined has phone'
'undefined has banana'
```

Такой результат не совсем тот, который требуется.

Существует несколько способов исправить эту проблему. Самый простой из них — сохранить контекст исполнения в некоторую переменную:

```
var person = {                                     // { person }
  name: 'Sergey',                                  //
  items: ['keys', 'phone', 'banana'],              //
                                                    //
  showItems: function () {                         // { _this }
    var _this = this;                              //
                                                    //
    this.items.map(function (item) {               // { item }
      return _this.name+' has '+item;              //
    });                                              //
  }                                                  //
}                                                    //

person.showItems();                                //
```

Callback переписывается таким образом, чтобы обращение происходило не к ключевому слову `this`, а к сохраненному контексту `_this`.

Каждый раз при вызове функции-callback'a переменная `_this` сперва ищется в области видимости callback'a. После того, как найти `_this` там не удалось, она ищется в области видимости родительской функции, то есть в области видимости `showItems`. Значение переменной `_this` как раз такое, какое нужно — объект `person`, а значит `_this.name` вернет правильный результат.

Это пример использования замыкания. В итоге получается желаемый результат:

```
'Sergey has keys'
'Sergey has phone'
'Sergey has banana'
```

Описанная проблема встречается каждый раз, когда нужно обратиться к ключевому слову `this` внутри callback'a, который передан в метод `map`. В JavaScript есть способ сохранить контекст исполнения без создания дополнительных конструкций: контекст исполнения callback'a можно передать в качестве второго аргумента метода `map`.

То есть можно переписать код так:

```
var person = {
  name: 'Sergey',
  items: ['keys', 'phone', 'banana'],
```

```

    showItems: function () {
        this.items.map(function (item) {
            return this.name + ' has ' + item;
        }, this);
    }
}

person.showItems();

```

В этом случае будет получен желаемый результат, поскольку `this` внутри `showItems` и внутри `callback`'а будет равен `person`.

5.2.11. Метод `bind`

Однако не все функции, которые работают с `callback`'ами, принимают в качестве одного из аргументов контекст выполнения `callback`'а. В таком случае можно воспользоваться методом `bind()`.

Метод `bind()` создаёт новую функцию, которая при вызове устанавливает в качестве контекста выполнения `this` предоставленное значение.

<...>

[Function.prototype.bind\(\) - JavaScript | MDN](#)

Метод `bind`, в отличие от методов `call` и `apply`, не вызывает функцию, а возвращает новую. Сигнатура метода `bind` следующая:

```
fun.bind(thisArg, arg1, arg2, ...);
```

Первым аргументом передается контекст исполнения новой функции, а остальные — задают аргументы, с которыми она будет вызвана.

Таким образом, пример можно переписать так:

```

var person = {
    name: 'Sergey',
    items: ['keys', 'phone', 'banana'],

    showItems: function () {
        this.items.map(function (item) {
            return this.name + ' has ' + item;
        }).bind(this));
    }
}

person.showItems();

```

5.3. myBind

Чтобы лучше понять, как работает функция `bind`, рассмотрим ее возможную реализацию:

```
Function.prototype.myBind = function(_this) {  
    var fn = this;  
    var args = [].slice.call(arguments, 1);  
  
    return function () {  
        var curArgs = [].slice.call(arguments);  
  
        return fn.apply(_this, args.concat(curArgs));  
    };  
};
```

Первым аргументом функция `bind` принимает контекст выполнения новой функции и сохраняется в переменной `_this`. В переменной `fn` сохраняется исходная функция. В переменной `args` будет лежать массив аргументов, с которыми вызвали функцию `bind`. Для этого массивоподобный объект `arguments` превращается при помощи заимствования метода `slice` в массив.

В результате возвращается новая функция. Следует обратить внимание, что вызвать новую функцию также можно с аргументами, поэтому исходная функция будет вызвана с объединенным набором аргументов.

5.4. Частичное применение

При помощи метода `bind` можно реализовать так называемое частичное применение.

Например, функция `pow` из библиотеки `Math` возвращает первое переданное как аргумент число, возведенное в степень второго числа.

```
Math.pow(2, 3); // 8  
Math.pow(2, 10); // 1024
```

Если требуется возводить в различную степень только число 2, можно написать функцию `binPow`, которая получается из функции `Math.pow` частичным применением:

```
var binPow = Math.pow.bind(null, 2);
```

Теперь в результате вызова функции `binPow` число 2 будет возводиться в переданную в качестве аргумента степень:

```
binPow(3); // 8  
binPow(10); // 1024
```

5.5. Режим работы интерпретатора

Последний фактор, который влияет на значение ключевого слова `this` — это режим работы интерпретатора. По умолчанию включен режим совместимости, а строгий режим можно включить, указав специальную директиву.

В режиме обратной совместимости, если вызывается функция, внутри которой используется `this`, значение `this` совпадает со значением контекста, внутри которого была вызвана функция.

```
function getSelf(){  
    return this;  
}
```

```
getSelf(); //global
```

В данном примере функция вызывается в глобальном контексте, поэтому внутри этой функции значение `this` равняется `global`.

Для того, чтобы вызвать функцию в строгом режиме, в начале файла или в начале функции нужно добавить директиву `'use strict'`;

```
function getSelf(){  
    'use strict';  
  
    return this;  
}
```

```
getSelf(); //undefined
```

В строгом режиме значение ключевого слова `this` будет `undefined`.

5.6. eval

Функция `eval` используется в JavaScript для того, чтобы интерпретировать код, который написан в виде строки.

```
var temperature = 12;  
  
eval('temperature + 5'); // 17
```

В строке можно использовать один или несколько операторов, перечисленных через точку с запятой, а также обращаться к внешним переменным. В том числе, можно использовать ключевое слово `this`.

```

var person = {
  name: 'Sergey',

  showName: function () {
    return eval('this.name');
  }
}

```

Внутри метода showName происходит обращение к ключевому слову this внутри строки, которая передается в eval. Значение this в данном случае совпадает с контекстом исполнения, в котором был вызван eval.

Это означает, что если showName будет вызван в контексте person, то this будет равен person.

```

person.showName(); // Sergey

```

Если же функцию eval сначала положить в некоторую переменную, например evil, поведение будет совершенно другим: в качестве this всегда будет глобальный объект.

```

var person = {
  name: 'Sergey',

  showName: function () {
    var evil = eval;

    return evil('this.name');
  }
}

person.showName(); // ''

```

В данном случае результат работы showName будет пустая строка.