**Лабораторная работа №13**

**Тема работы:** Разработка программ с применением вложенных объектов и массивов объектов

**Цель работы:** закрепить навыки по работе со вложенными массивами и объектами.

**Теоретические сведения**

В JavaScript функция тоже является объектом - объектом **Function** и тоже имеет прототип, свойства, методы. Все функции, которые используются в программе, являются объектами Function и имеют все его свойства и методы.

Например, мы можем создать функцию с помощью конструктора Function:

|  |  |
| --- | --- |
| 1  2 | var square = new Function('n', 'return n \* n;');  console.log(square(5)); |

В конструктор Function может передаваться ряд параметров. Последний параметр представляет собой само тело функции в виде строки. Фактически строка содержит код javascript. Предыдущие аргументы содержат названия параметров. В данном случае определяется функция возведения числа в квадрат, которая имеет один параметр n.

Среди свойств объекта Function можно выделить следующие:

* **arguments**: массив аргументов, передаваемых в функцию
* **length**: определяет количество аргументов, которые ожидает функция
* **caller**: определяет функцию, вызвавшую текущую выполняющуюся функцию
* **name**: имя функции
* **prototype**: прототип функции

С помощью прототипа мы можем определить дополнительные свойства:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function display(){    console.log("привет мир");  }  Function.prototype.program ="Hello";    console.log(display.program); // Hello |

Среди методов надо отметить методы **call()** и **apply()**.

Метод **call()** вызывает функцию с указанным значением this и аргументами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function add(x, y){    return x + y;  }  var result = add.call(this, 3, 8);    console.log(result); // 11 |

this указывает на объект, для которого вызывается функция - в данном случае это глобальный объект window. После this передаются значения для параметров.

При передаче объекта через первый параметр, мы можем ссылаться на него через ключевое слово this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | function User (name, age) {  this.name = name;  this.age = age;  }  var tom = new User("Том", 26);  function display(){  console.log("Ваше имя: " + this.name);  }  display.call(tom); // Ваше имя: Том |

В данном случае передается только одно значение, поскольку функция display не принимает параметров. То есть функция будет вызываться для объекта tom.

Если нам не важен объект, для которого вызывается функция, то можно передать значение null:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function add(x, y){    return x + y;  }  var result = add.call(null, 3, 8);    console.log(result); // 11 |

На метод call() похож метод **apply()**, который также вызывает функцию и в качестве первого параметра также получает объект, для которого функция вызывается. Только теперь в качестве второго параметра передается массив аргументов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function add(x, y){    return x + y;  }  var result = add.apply(null, [3, 8]);    console.log(result); // 11 |

JavaScript поддерживает наследование, то позволяет нам при создании новых типов объектов при необходимости унаследовать их функционал от уже существующих. Например, у нас может быть объект User, представляющий отдельного пользователя. И также может быть объект Employee, который представляет работника. Но работник также может являться пользователем и поэтому должен иметь все его свойства и методы. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | // конструктор пользователя  function User (name, age) {  this.name = name;  this.age = age;  this.go = function(){document.write(this.name + " идет <br/>");}  this.displayInfo = function(){  document.write("Имя: " + this.name + "; возраст: " + this.age + "<br/>");  };  }  User.prototype.maxage = 110;    // конструктор работника  function Employee(name, age, comp){  User.call(this, name, age);  this.company = comp;  this.displayInfo = function(){  document.write("Имя: " + this.name + "; возраст: " + this.age + "; компания: " + this.company + "<br/>");  };  }  Employee.prototype = Object.create(User.prototype);    var tom = new User("Том", 26);  var bill = new Employee("Билл", 32, "Google");  tom.go();  bill.go();  tom.displayInfo();  bill.displayInfo();  console.log(bill.maxage); |

Здесь в начале определяет конструктор User и к его прототипу добавляется свойство maxage. Затем определяется тип Employee.

В конструкторе Employee происходит обращение к конструктору User с помощью вызова:

|  |  |
| --- | --- |
| 1 | User.call(this, name, age); |

Передача первого параметра позволяет вызвать функцию конструктора User для объекта, создаваемого конструктором Employee. Благодаря этому все свойства и методы, определенные в конструкторе User, также переходят на объект Employee.

Кроме того, необходимо унаследовать также и прототип User. Для этого служит вызов:

|  |  |
| --- | --- |
| 1 | Employee.prototype = Object.create(User.prototype); |

Метод Object.create() позволяет создать объект прототипа User, который затем присваивается прототипу Employee. При этом при необходимости в прототипе Employee мы также можем определить дополнительные свойства и методы.

При наследовании мы можем переопределять наследуемый функционал. Например, Employee переопределяет метод displayInfo(), унаследованный от User, чтобы включить в вывод этого метода новое свойство company.

В итоге браузер предоставит следующий вывод:

Поведение ключевого слова **this** зависит от контекста, в котором оно используется, и от того, в каком режиме оно используется - строгом или нестрогом.

**Глобальный контекст**

В глобальном контексте this ссылается на глобальный объект. В данном случае поведение не зависит от режима (строгий или нестрогий):

|  |  |
| --- | --- |
| 1  2  3  4 | this.alert("global alert");  this.console.log("global console");    var currentDocument = this.document; |

Но в данном случае использвание this избыточно.

**Контекст функции**

В пределах функции this ссылается на внешний контекст. Для функций, определенных в глобальном контексте, - это объект Window, который представляет окно браузера. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | function foo(){  var bar = "bar2";  console.log(this.bar);  }    var bar = "bar1";    foo(); // bar1 |

Если бы мы не использовали бы this, то обращение шло бы к локальной переменной, определенной внутри функции.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | function foo(){  var bar = "bar2";  console.log(bar);  }    var bar = "bar1";    foo(); // bar2 |

Но если бы мы использовали строгий режим (strict mode), то this в этом случае имело бы значение undefined:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | "use strict";  var obj = {  function foo(){  var bar = "bar2";  console.log(this.bar);  }  }    var bar = "bar1";    foo(); // ошибка - this - undefined |

**Контекст объекта**

В контексте объекта, в том числе в его методах, ключевое слово this ссылается на этот же объект:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var o = {  bar: "bar3",  foo: function(){  console.log(this.bar);  }  }  var bar = "bar1";  o.foo(); // bar3 |

**Примеры**

Рассмотрим более сложный пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | function foo(){  var bar = "bar2";  console.log(this.bar);  }    var o3 = {bar:"bar3", foo: foo};  var o4 = {bar:"bar4", foo: foo};    var bar = "bar1";    foo(); // bar1  o3.foo(); // bar3  o4.foo(); // bar4 |

Здесь определена глобальная переменная bar. И также в функции foo определена локальная переменная bar. Значение какой переменной будет выводиться в функции foo? Функция foo выводит значение глобальной переменной, так как данный скрипт запускается в нестрогом режиме, а значит ключеое слово this в функции foo ссылается на внешний контекст.

Иначе дело обстоит с объектами. Они определяют свой собственный контекст, в котором существует свое свойство bar. И при вызове метода foo внешним контекстом по отношению к функции будет контекст объектов o3 и o4.

Подобное поведение может привести к некоторому непонимаю в отдельных случаях. Так, рассмотрим другую ситуацию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var o1 = {  bar: "bar1",  foo: function(){  console.log(this.bar);  }  }  var o2 = {bar: "bar2", foo: o1.foo};    var bar = "bar3";  var foo = o1.foo;    o1.foo(); // bar1  o2.foo(); // bar2  foo(); // bar3 |

Несмотря на то, что объект o2 использует метод foo из объекта o1, тем не менее функция o1.foo также будет искать значение для this.bar во внешнем котексте, то есть в контексте объекта o2. А в объекте o2 это значение равно bar: "bar2".

То же самое с глобальной переменной foo, которая ссылается на ту же функцию, что и метод o1.foo. В этом случае также будет происходить поиск значения для this.bar во внешним контексте, то есть в глобальном контексте, где определена переменная var bar = "bar3".

Однако если мы вызываем функцию из другой функции, вызываемая функция также будет использовать внешний контекст:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var bar = "bar2";    function daz(){  var bar = "bar5";  function maz(){    console.log(this.bar);  }  maz();  }  daz(); // bar2 |

Здесь функция daz в качестве this.bar использует значение переменной bar из внешнего контекста, то есть значение глобальной переменной bar. Функция maz также в качестве this.bar использует значение переменной bar из внешнего контекста, а это значение this.bar из внешней функции daz, которое в свою очередь представляет значение глобальной переменной bar. Поэтому в итоге консоль выведет "bar2", а не "bar5".

**Явная привязка**

С помощью методов call() и apply() можно задать явную привязку функции к определенному контексту:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | function foo(){  console.log(this.bar);  }    var o3 = {bar: "bar3"}  var bar = "bar1";  foo(); // bar1  foo.apply(o3); // bar3  // или  // foo.call(o3); |

Во втором случае функция foo привязывается к объекту o3, который и определяет ее контекст. Поэтому во втором случае консоль выведет "bar3".

**Метод bind**

Метод f.bind(o) позволяет создать новую функцию с тем же телом и областью видимости, что и функция f, но с привязкой к объекту o:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | function foo(){  console.log(this.bar);  }    var o3 = {bar: "bar3"}  var bar = "bar1";  foo(); // bar1  var func = foo.bind(o3);  func(); // bar3 |

**this и стрелочные функции**

При работе с несколькими контекстами мы вынуждены учитывать, в каком контексте определяется переменная. Например, возьмем следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var school ={  title: "Oxford",  courses: ["JavaScript", "TypeScript", "Java", "Go"],  printCourses: function(){  this.courses.forEach(function(course){  console.log(this.title, course);  })  }  }  school.printCourses(); |

Функция printCourses проходит по всем курсам из массива и при их выводе предваряет их значением свойства title. Однако на консоли при запуске программы мы увидим следующее:

undefined "JavaScript"

undefined "TypeScript"

undefined "Java"

undefined "Go"

Мы видим, что значение this.title не определено, так как this как контекст объекта замещается глобальным контекстом. В этом случае нам надо передать подобное значение this.title или весь контекст объекта.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var school ={  title: "Oxford",  courses: ["JavaScript", "TypeScript", "Java", "Go"],  printCourses: function(){  var that = this;  this.courses.forEach(function(course){  console.log(that.title, course);  })  }  }  school.printCourses(); |

Стрелочные функции также позволяют решить данную проблему:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var school ={  title: "Oxford",  courses: ["JavaScript", "TypeScript", "Java", "Go"],  printCourses: function(){  this.courses.forEach((course)=>console.log(this.title, course))  }  }  school.printCourses(); |

Контекстом для стрелочной функции в данном случае будет выступать контекст объекта school. Соответственно, нам недо определять дополнительные переменые для передачи данных в функцию.

**Порядок выполнения работы**

**Практическая часть**

Реализуйте класс **Student** (Студент), который будет наследовать от класса **User**. Этот класс должен иметь следующие свойства: **name** (имя, наследуется от User), **surname** (фамилия, наследуется от User), **year** (год поступления в вуз). Класс должен иметь метод **getFullName()** (наследуется от User), с помощью которого можно вывести одновременно имя и фамилию студента. Также класс должен иметь метод **getCourse()**, который будет выводить текущий курс студента (от 1 до 5). Курс вычисляется так: нужно от текущего года отнять год поступления в вуз. Текущий год получите самостоятельно.

Вот так должен работать наш класс:

var student **=** new Student('Иван', 'Иванов', 2017);

console.log(worker.name); //выведет 'Иван'

console.log(worker.surname); //выведет 'Иванов'

console.log(worker.getFullName()); //выведет 'Иван Иванов'

console.log(worker.year); //выведет 2017console.log(worker.getCourse()); //выведет 3 - третий курс, так как текущий год 2020

Вот так должен выглядеть класс **User**, от которого наследуется наш **Student**:

class User {

constructor(name, surname) {

this.name **=** name;

this.surname **=** surname;

}

getFullName() {

return this.name **+** ' ' **+** this.surname;

}

}