**Лабораторная работа №12**

**Тема работы:** Разработка программ с применением объектов в функциях

**Цель работы:** закрепить навыки по работе с объектами.

**Теоретические сведения**

Объектно-ориентированное программирование на сегодняшний день является одной из господствующих парадигм в разработке приложений, и в JavaScript мы также можем использовать все преимущества ООП. В то же время применительно к JavaScript объектно-ориентированное программирование имеет некоторые особенности.

**Объекты**

В прошлых темах мы работали с примитивными данными - числами, строками, но данные не всегда представляют примитивные типы. Например, если в нашей программе нам надо описать сущность человека, у которого есть имя, возраст, пол и так далее, то естественно мы не сможем представить сущность человека в виде числа или строки. Нам потребуется несколько строк или чисел, чтобы должным образом описать человека. В этом плане человек будет выступать как сложная комплексная структура, у которого будут отдельные свойства - возраст, рост, имя, фамилия и т.д.

Для работы с подобными структурами в JavaScript используются **объекты**. Каждый объект может хранить свойства, которые описывают его **состояние**, и методы, которые описывают его **поведение**

**Создание нового объекта**

Есть несколько способов создания нового объекта.

Первый способ заключается в использовании конструктора Object:

|  |  |
| --- | --- |
| 1 | var user = new Object(); |

В данном случае объект называется user. Он определяется также, как и любая обычная переменная с помощью ключевого слова var.

Выражение new Object() представляет вызов конструктора - функции, создающей новый объект. Для вызова конструктора применяется оператор **new**. Вызов конструктора фактически напоминает вызов обычной функции.

Второй способ создания объекта представляет использование фигурных скобок:

|  |  |
| --- | --- |
| 1 | var user = {}; |

На сегодняшний день более распространенным является второй способ.

**Свойства объекта**

После создания объекта мы можем определить в нем свойства. Чтобы определить свойство, надо после названия объекта через точку указать имя свойства и присвоить ему значение:

|  |  |
| --- | --- |
| 1  2  3 | var user = {};  user.name = "Tom";  user.age = 26; |

В данном случае объявляются два свойства name и age, которым присваиваются соответствующие значения. После этого мы можем использовать эти свойства, например, вывести их значения в консоли:

|  |  |
| --- | --- |
| 1  2 | console.log(user.name);  console.log(user.age); |

Также можно определить свойства при определении объекта:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var user = {    name: "Tom",  age: 26  }; |

В этом случае для присвоения значения свойству используется символ двоеточия, а после определения свойства ставится запятая (а не точка с запятой).

Кроме того, доступен сокращенный способ определения свойств:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var name = "Tom";  var age = 34;  var user = {name, age};  console.log(user.name); // Tom  console.log(user.age); // 34 |

В данном случае названия переменных также являются и названиями свойств объекта. И таким образом можно создавать более сложные конструкции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var name = "Tom";  var age = 34;  var user = {name, age};    var teacher = {user, course: "JavaScript"};  console.log(teacher.user); // {name: "Tom", age: 34}  console.log(teacher.course); // JavaScript |

**Методы объекта**

Методы объекта определяют его поведение или действия, которые он производит. Методы представляют собой функции. Например, определим метод, который бы выводил имя и возраст человека:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var user = {};  user.name = "Tom";  user.age = 26;  user.display = function(){    console.log(user.name);  console.log(user.age);  };    // вызов метода  user.display(); |

Как и в случае с функциями методы сначала определяются, а потом уже вызываются.

Также методы могут определяться непосредственно при определении объекта:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var user = {    name: "Tom",  age: 26,  display: function(){    console.log(this.name);  console.log(this.age);  }  }; |

Как и в случае со свойствами, методу присваивается ссылка на функцию с помощью знака двоеточия.

Чтобы обратиться к свойствам или методам объекта внутри этого объекта, используется ключевое слово **this**. Оно означает ссылку на текущий объект.

Также можно использовать сокращенный способ определения методов, когда двоеточие и слово function опускаются:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var user = {    name: "Tom",  age: 26,  display(){    console.log(this.name, this.age);  },  move(place){  console.log(this.name, "goes to", place);  }  };  user.display(); // Tom 26  user.move("the shop"); // Tom goes to the shop |

**Синтаксис массивов**

Существует также альтернативный способ определения свойств и методов с помощью синтаксиса массивов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var user = {};  user["name"] = "Tom";  user["age"] = 26;  user["display"] = function(){    console.log(user.name);  console.log(user.age);  };    // вызов метода  user["display"](); |

Название каждого свойства или метода заключается в кавычки и в квадратные скобки, затем им также присваивается значение. Например, user["age"] = 26.

При обращении к этим свойствам и методам можно использовать либо нотацию точки (user.name), либо обращаться так: user["name"]

**Строки в качестве свойств и методов**

Также следует отметить, что названия свойств и методов объекта всегда представляют строки. То есть мы могли предыдущее определение объекта переписать так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var user = {  "name": "Tom",  "age": 26,  "display": function(){    console.log(user.name);  console.log(user.age);  }  };  // вызов метода  user.display(); |

С одной стороны, разницы никакой нет между двумя определениями. С другой стороны, бывают случаи, где заключение названия в строку могут помочь. Например, если название свойства состоит из двух слов, разделенных пробелом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var user = {  name: "Tom",  age: 26,  "full name": "Tom Johns",  "display info": function(){    console.log(user.name);  console.log(user.age);  }  };  console.log(user["full name"]);  user["display info"](); |

Только в этом случае для обращении к подобным свойствам и методам мы должны использовать синтаксис массивов.

**Удаление свойств**

Выше мы посмотрели, как можно динамически добавлять новые свойства к объекту. Однако также мы можем удалять свойства и методы с помощью оператора **delete**. И как и в случае с добавлением мы можем удалять свойства двумя способами. Певый способ - использование нотации точки:

|  |  |
| --- | --- |
| 1 | delete объект.свойство |

Либо использовать синтаксис массивов:

|  |  |
| --- | --- |
| 1 | delete объект["свойство"] |

Например, удалим свойство:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var user = {};  user.name = "Tom";  user.age = 26;  user.display = function(){    console.log(user.name);  console.log(user.age);  };    console.log(user.name); // Tom  delete user.name; // удаляем свойство  // альтернативный вариант  // delete user["name"];  console.log(user.name); // undefined |

После удаления свойство будет не определено, поэтому при попытке обращения к нему, программа вернет значение undefined.

**Вложенные объекты и массивы в объектах**

Одни объекты могут содержать в качестве свойств другие объекты. Например, есть объект страны, у которой можно выделить ряд свойств. Одно из этих свойств может представлять столицу. Но у столицы мы также можем выделить свои свойства, например, название, численность населения, год основания:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var country = {    name: "Германия",  language: "немецкий",  capital:{    name: "Берлин",  population: 3375000,  year: 1237  }  };  console.log("Столица: " + country.capital.name); // Берлин  console.log("Население: " + country["capital"]["population"]); // 3375000  console.log("Год основания: " + country.capital["year"]); // 1237 |

Для доступа к свойствам таких вложенных объектов мы можем использовать стандартную нотацию точки:

|  |  |
| --- | --- |
| 1 | country.capital.name |

Либо обращаться к ним как к элементам массивов:

|  |  |
| --- | --- |
| 1 | country["capital"]["population"] |

Также допустим смешанный вид обращения:

|  |  |
| --- | --- |
| 1 | country.capital["year"] |

В качестве свойств также могут использоваться массивы, в том числе массивы других объектов:

|  |  |
| --- | --- |
| 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 | var country = {    name: "Швейцария",  languages: ["немецкий", "французский", "итальянский"],  capital:{    name: "Берн",  population: 126598  },  cities: [  { name: "Цюрих", population: 378884},  { name: "Женева", population: 188634},  { name: "Базель", population: 164937}  ]  };    // вывод всех элементов из country.languages  document.write("<h3>Официальные языки Швейцарии</h3>");  for(var i=0; i < country.languages.length; i++)  document.write(country.languages[i] + "<br/>");    // вывод всех элементов из country.cities  document.write("<h3>Города Швейцарии</h3>");  for(var i=0; i < country.cities.length; i++)  document.write(country.cities[i].name + "<br/>"); |

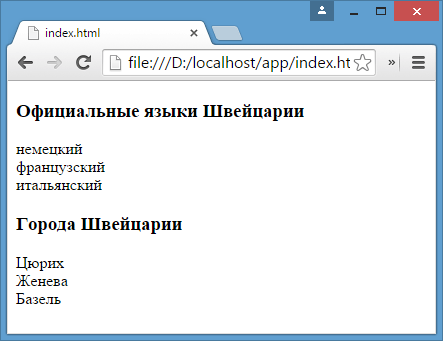
В объекте country имеется свойство languages, содержащее массив строк, а также свойство cities, хранящее массив однотипных объектов.

С этими массивами мы можем работать также, как и с любыми другими, например, перебрать с помощью цикла for.

При переборе массива объектов каждый текущий элемент будет представлять отдельный объект, поэтому мы можем обратиться к его свойствам и методам:

|  |  |
| --- | --- |
| 1 | country.cities[i].name |

В итоге браузер выведет содержимое этих массивов:



**Объекты в функциях**

Функции могут возвращать значения. Но эти значения не обязательно должны представлять примитивные данные - числа, строки, но также могут быть сложными объектами.

Например, вынесем создание объекта user в отдельную функцию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | function createUser(pName, pAge) {  return {  name: pName,  age: pAge,  displayInfo: function() {  document.write("Имя: " + this.name + " возраст: " + this.age + "<br/>");  }  };  };  var tom = createUser("Tom", 26);  tom.displayInfo();  var alice = createUser("Alice", 24);  alice.displayInfo(); |

Здесь функция createUser() получает значения pName и pAge и по ним создает новый объект, который является возвращаемым результатом.

Преимуществом вынесения создания объекта в функцию является то, что далее мы можем создать несколько однотипных объектов с разными значениями.

Кроме того объект может передаваться в качестве параметра в функцию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | function createUser(pName, pAge) {  return {  name: pName,  age: pAge,  displayInfo: function() {  document.write("Имя: " + this.name + " возраст: " + this.age + "<br/>");  },  driveCar: function(car){  document.write(this.name + " ведет машину " + car.name + "<br/>");  }  };  };    function createCar(mName, mYear){  return{  name: mName,  year: mYear  };  };  var tom = createUser("Том", 26);  tom.displayInfo();  var bently = createCar("Бентли", 2004);  tom.driveCar(bently); |

Здесь используются две функции для создания пользователей и объекта машины. Метод driveCar() объекта user в качестве параметра принимает объект car.

В итоге браузер нам выведет:

Имя: Том возраст: 26

Том ведет машину Бентли

**Конструкторы объектов**

Кроме создания новых объектов JavaScript предоставляет нам возможность создавать новые типы объектов с помощью **конструкторов**. Так, одним из способов создания объекта является применение конструктора типа Object:

|  |  |
| --- | --- |
| 1 | var tom = new Object(); |

После создания переменной tom она будет вести себя как объект типа Object.

Конструктор позволяет определить новый тип объекта. Тип представляет собой абстрактное описание или шаблон объекта. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке - наличие двух рук, двух ног, головы, пищеварительной, нервной системы и т.д. Есть некоторый шаблон - этот шаблон можно назвать типом. Реально же существующий человек является объектом этого типа.

Определение типа может состоять из функции конструктора, методов и свойств.

Для начала определим конструктор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function User(pName, pAge) {  this.name = pName;  this.age = pAge;  this.displayInfo = function(){  document.write("Имя: " + this.name + "; возраст: " + this.age + "<br/>");  };  } |

Конструктор - это обычная функция за тем исключением, что в ней мы можем установить свойства и методы. Для установки свойств и методов используется ключевое слово **this**:

|  |  |
| --- | --- |
| 1 | this.name = pName; |

В данном случае устанавливаются два свойства name и age и один метод displayInfo.

Как правило, названия конструкторы в отличие от названий обычных функций начинаются с большой буквы.

После этого в программе мы можем определить объект типа User и использовать его свойства и методы:

|  |  |
| --- | --- |
| 1  2  3 | var tom = new User("Том", 26);  console.log(tom.name); // Том  tom.displayInfo(); |

Чтобы вызвать конструктор, то есть создать объект типа User, надо использовать ключевое слово **new**.

Подобным образом мы можем определить и другие типы и использовать их вместе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | // конструктор типа Car  function Car(mName, mYear){  this.name = mName;  this.year = mYear;  this.getCarInfo = function(){  document.write("Модель: " + this.name + " Год выпуска: " + this.year + "<br/>");  };  };  // конструктор типа User  function User(pName, pAge) {  this.name = pName;  this.age = pAge;  this.driveCar = function(car){  document.write(this.name + " ведет машину " + car.name + "<br/>");  };  this.displayInfo = function(){  document.write("Имя: " + this.name + "; возраст: " + this.age + "<br/>");  };  };    var tom = new User("Том", 26);  tom.displayInfo();  var bently = new Car("Бентли", 2004);  tom.driveCar(bently); |

**Оператор instanceof**

Оператор **instanceof** позволяет проверить, с помощью какого конструктора создан объект. Если объект создан с помощью определенного конструктора, то оператор возвращает true:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var tom = new User("Том", 26);    var isUser = tom instanceof User;  var isCar = tom instanceof Car;  console.log(isUser); // true  console.log(isCar); // false |

**Расширение объектов. Prototype**

Кроме непосредственного определения свойств и методов в конструкторе мы также можем использовать свойство **prototype**. Каждая функция имеет свойство **prototype**, представляющее прототип функции. То есть свойство User.prototype представляет прототип объектов User. И любые свойства и методы, которые будут определены в User.prototype, будут общими для всех объектов User.

Например, после определения объекта User нам потребовалось добавить к нему метод и свойство:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | function User(pName, pAge) {  this.name = pName;  this.age = pAge;  this.displayInfo = function(){  document.write("Имя: " + this.name + "; возраст: " + this.age + "<br/>");  };  };    User.prototype.hello = function(){  document.write(this.name + " говорит: 'Привет!'<br/>");  };  User.prototype.maxAge = 110;    var tom = new User("Том", 26);  tom.hello();  var john = new User("Джон", 28);  john.hello();  console.log(tom.maxAge); // 110  console.log(john.maxAge); // 110 |

Здесь добавлены метод hello и свойство maxAge, и каждый объект User сможет их использовать. Но важно заметить, что значение свойства maxAge будет одно и то же для всех объектов, это разделяемое статическое свойство. В отличие, скажем, от свойства this.name, которое хранит значение для определенного объекта.

В то же время мы можем определить в объекте свойство, которое будет назваться также, как и свойство прототипа. В этом случае собственное свойство объекта будет иметь приоритет перед свойством прототипа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | User.prototype.maxAge = 110;  var tom = new User("Том", 26);  var john = new User("Джон", 28);  tom.maxAge = 99;  console.log(tom.maxAge); // 99  console.log(john.maxAge); // 110 |

И при обращении к свойству maxAge javascript сначала ищет это свойство среди свойств объекта, и если оно не было найдено, тогда обращается к свойствам прототипа. То же самое касается и методов.

**Инкапсуляция**

Инкапсуляция является одним из ключевых понятий объектно-ориентированного программирования и представляет сокрытие состояния объекта от прямого доступа извне. По умолчанию все свойства объектов являются публичными, общедоступными, и мы к ним можем обратиться из любого места программы.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | function User(pName, pAge) {  this.name = pName;  this.age = pAge;  this.displayInfo = function(){  document.write("Имя: " + this.name + "; возраст: " + this.age);  };  };  var tom = new User("Том", 26);  tom.name=34;  console.log(tom.name); |

Но мы можем их скрыть от доступа извне, сделав свойства локальными переменными:

|  |  |
| --- | --- |
| 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 | function User (name) {  this.name = name;  var \_age = 1;  this.displayInfo = function(){  console.log("Имя: " + this.name + "; возраст: " + \_age);  };  this.getAge = function() {  return \_age;  }  this.setAge = function(age) {  if(typeof age === "number" && age >0 && age<110){  \_age = age;  } else {  console.log("Недопустимое значение");  }  }  }    var tom = new User("Том");  console.log(tom.\_age); // undefined - \_age - локальная переменная  console.log(tom.getAge()); // 1  tom.setAge(32);  console.log(tom.getAge()); // 32  tom.setAge("54"); // Недопустимое значение  tom.setAge(123); // Недопустимое значение |

В конструкторе User объявляется локальная переменная \_age вместо свойства age. Как правило, названия локальных переменных в конструкторах начинаются со знака подчеркивания.

Для того, чтобы работать с возрастом пользователя извне, определяются два метода. Метод getAge() предназначен для получения значения переменной \_age. Этот метод еще называется геттер (getter). Второй метод - setAge, который еще называется сеттер (setter), предназначен для установки значения переменной \_age.

Плюсом такого подхода является то, что мы имеем больший контроль над доступом к значению \_age. Например, мы можем проверить какие-то сопутствующие условия, как в данном случае проверяются тип значение (он должен представлять число), само значение (возраст не может быть меньше 0).

Также, для выполнения роботы вам необходимо будет ознакомиться с документом «Регулярные выражения"

**Порядок выполнения работы**

**Практическая часть**

**Простые задачи**

1. Реализуйте класс **Worker** (Работник), который будет иметь следующие свойства: **name** (имя), **surname** (фамилия), **rate** (ставка за день работы), **days** (количество отработанных дней). Также класс должен иметь метод **getSalary()**, который будет выводить зарплату работника. Зарплата - это произведение (умножение) ставки **rate** на количество отработанных дней **days**.

Вот так должен работать наш класс:

var worker **=** new Worker('Иван', 'Иванов', 10, 31);

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

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

console.log(worker.rate); //выведет 10

console.log(worker.days); //выведет 31

console.log(worker.getSalary()); //выведет 310 - то есть 10\*31

С помощью нашего класса создайте двух рабочих и найдите сумму их зарплат.

**Геттеры и сеттеры**

1. Модифицируйте класс **Worker** из предыдущей задачи следующим образом: сделайте все его свойства приватными, а для их чтения сделайте методы-геттеры. Наш класс теперь будет работать так:

var worker **=** new Worker('Иван', 'Иванов', 10, 31);

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

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

console.log(worker.getRate()); //выведет 10

console.log(worker.getDays()); //выведет 31

console.log(worker.getSalary()); //выведет 310 - то есть 10\*31

1. Модифицируйте класс **Worker** из предыдущей задачи следующим образом: для свойства **rate** и для свойства **days** сделайте еще и методы-сеттеры. Наш класс теперь будет работать так:

var worker **=** new Worker('Иван', 'Иванов', 10, 31);

console.log(worker.getRate()); //выведет 10

console.log(worker.getDays()); //выведет 31

console.log(worker.getSalary()); //выведет 310 - то есть 10\*31

//Теперь используем сеттер:

worker.setRate(20); //увеличим ставку

worker.setDays(10); //уменьшим дни

console.log(worker.getSalary()); //выведет 200 - то есть 20\*10

**Практика**

1. Реализуйте класс **MyString**, который будет иметь следующие методы: метод **reverse()**, который параметром принимает строку, а возвращает ее в перевернутом виде, метод **ucFirst()**, который параметром принимает строку, а возвращает эту же строку, сделав ее первую букву заглавной и метод **ucWords**, который принимает строку и делает заглавной первую букву каждого слова этой строки.

Наш класс должен работать так:

var str **=** new MyString();

console.log(str.reverse('abcde')); //выведет 'edcba'

console.log(str.ucFirst('abcde')); //выведет 'Abcde'

console.log(str.ucWords('abcde abcde abcde')); //выведет 'Abcde Abcde Abcde'

1. Реализуйте класс **Validator**, который будет проверять строки. К примеру, у него будет метод **isEmail** параметром принимает строку и проверяет, является ли она корректным емейлом или нет. Если является - возвращает true, если не является - то false. Кроме того, класс будет иметь следующие методы: метод **isDomain** для проверки домена, метод **isDate** для проверки даты и метод **isPhone** для проверки телефона:

var validator **=** new Validator();

console.log(validator.isEmail('phphtml@mail.ru'));

console.log(validator.isDomain('phphtml.net'));

console.log(validator.isDate('12.05.2020'));

console.log(validator.isPhone('+375 (29) 817-68-92')); //тут можете формат своей страны