Web-дизайн

Лабораторная работа №8

Элементы объектно-ориентированного программирования

Объекты

Создание нового объекта

Есть несколько способов создания нового объекта.

Первый способ заключается в использовании конструктора **Object**:

```
var user = new Object();
```

Второй способ создания объекта представляет использование **фигурных скобок**:

```
var user = {};
```

Свойства объекта

По факту объект является так называемым **ассоциативным массивом**. Свойства определяются парой *ключ: значение.*

Можно определить свойства при определении объекта:

```
var user = {
    name: "Tom",
    age: 26
};
```

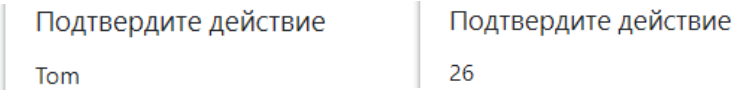
Также свойства можно определить уже после создания объекта:

```
var user = {};
user.name = "Tom"; // синтаксис объектов
user['age'] = 26;  // синтаксис массивов
```

Методы объекта

Например, определим метод, который бы выводил имя и возраст человека:

```
var user = {};
user.name = "Tom";
user.age = 26;
user.display = function(){
    alert(user.name);
    alert(user.age);
};
// вызов метода
user.display();
```



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

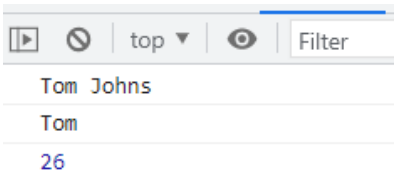
```
display: function(){
      alert(this.name);
      alert(this.age);
}
```

Строки в качестве свойств и методов

Названия свойств и методов объекта всегда представляют **строки**. Но в некоторых ситуациях имена свойств можно определить только *строкой в кавычках*.

В этом случае для обращении к свойствам и методам используется синтаксис массивов:

```
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"]); // пробел в имени свойства
user["display info"]();
```



Удаление свойств

Можно удалять свойства и методы с помощью оператора **delete**.

Первый способ – использование нотации точки:

```
delete объект.свойство
```

Второй способ – использовать синтаксис массивов:

```
delete объект["свойство"]
```

Проверка наличия и перебор методов и свойств

При динамическом определении в объекте новых свойств и методов перед их использованием бывает важно проверить, а есть ли уже такие методы и свойства. Для этого может использоваться оператор **in**.

Оператор *in* имеет следующий синтаксис:

```
"свойство|метод" in объект
```

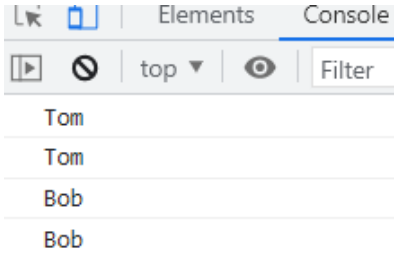
В кавычках идет название свойства или метода, а после *in* – название объекта. Если свойство или метод с подобным именем имеется, то оператор возвращает *true*. Иначе – *false*.

```
var user = {};
user.name = "Tom";
user.age = 26;
user.display = function(){
    alert(user.name);
    alert(user.age);
};
var hasNameProp = "name" in user; // проверка name
alert(hasNameProp); // true - свойство name есть в user
```

Копирование объектов

В отличие от примитивных типов данные объектов копируются **по ссылке**. Рассмотрим следующий пример:

```
const tom = { name: "Tom"};
const bob = tom;
// проверяем свойство name у обоих констант
console.log(tom.name);  // Tom
console.log(bob.name);  // Tom
// меняем свойство name у константы bob
bob.name = "Bob";
// повторно проверяем свойство name у обоих констант
console.log(tom.name);  // Bob
console.log(bob.name);  // Bob
```



В данном случае константа *bob* получает ссылку константы *tom*, поэтому после этого присвоения обе константы, по сути, указывают на один и тот же объект в памяти.

Если же мы хотим **скопировать** свойства объекта, то в этом случае мы можем воспользоваться встроенным методом **Object.assign()**.

Метод **Object.assign()** принимает два параметра:

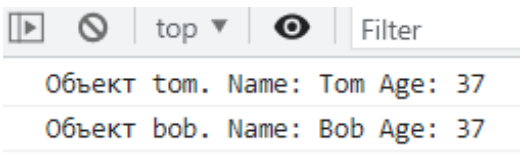
```
Object.assign(target, ...sources)
```

Параметр *target* представляет объект, в который надо скопировать свойства.

Параметр *...sources* набор объектов, из которых надо скопировать свойства (можно скопировать свойства сразу из нескольких объектов).

Возвращает метод объект *target*, в который скопированы свойства из объектов *sources*.

```
const tom = { name: "Tom", age: 37};
const bob = Object.assign({}, tom); // в пустой объект копируем tom
bob.name = "Bob";
console.log(`Объект tom. Name: ${tom.name} Age: ${tom.age}`); // Tom
console.log(`Объект bob. Name: ${bob.name} Age: ${bob.age}`); // Bob
```



В данном случае вызов *Object.assign({}, tom)* означает, что мы копируем данные из объекта *tom* в пустой объект. Результатом этого копирования сохраняется в *bob*.

Паттерн Модуль

Паттерн Модуль (module pattern) базируется на замыканиях и состоит из двух компонентов: *внешняя функция*, которая определяет *лексическое окружение*, и возвращаемый набор *внутренних функций*, которые имеют доступ к этому окружению.

Также здесь могут пригодится *самовызывающиеся функции* (IIFE).

```
let value = (function(){
    let obj = { data: "hello" }; // данные
    return {
        show: function(){
            alert( obj.data );
        }
    }
})();
value.show();  // hello
```

Здесь определена переменная *value*, которая представляет результат функции IIFE. Внутри функции определен объект *obj* с некоторыми данными.

Сама функция IIFE возвращает объект, который определяет функцию *show*. Возвращаемый объект определяет общедоступный интерфейс, через который можно обращаться к данным, определенным внутри модуля.

Конструкторы объектов

Обычный синтаксис объектов «{...}» позволяет создать только один экземпляр. Но зачастую нам нужно создать множество однотипных объектов.

Это можно сделать при помощи функции, выполняющую роль **конструктора**, (функция-конструктор) и оператора «*new*».

```
//функция-конструктор
function User(name) {
  this.name = name;
  this.isAdmin = false;
}
let user = new User("Вася"); // создается объект user
alert(user.name); // Вася
alert(user.isAdmin); // false
```

Таким образом, вызывая данную функцию, можно создавать новые объекты с одинаковым набором полей, заданным в функции.

Оператор instanceof

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

```
var tom = new User("Том");
var isUser = tom instanceof User; // true
var isCar = tom instanceof Car; // false
```

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.

```
function User (name) {
    this.name = name;
    var _age = 1;     // сокрытое свойство
    this.displayInfo = function(){
        console.log(this.name + " " + _age);
    };
    this.getAge = function() {
        return _age;
    }
    this.setAge = function(age) {
        if(typeof age === "number" && age >0 && age<100){
            _age = age;
        } else {
            console.log("Недопустимое значение");
        }
    }
}
var tom = new User("Том");
alert(tom.getAge()); // 1
tom.setAge(32);  // _age становится 32
alert(tom.getAge()); // 32
tom.setAge(123); // Недопустимое значение, _age не изменяется
```

В конструкторе *User* объявляется локальная переменная *_age* вместо свойства *age*.

Для того, чтобы работать с *_age* пользователя извне, определяются два метода.

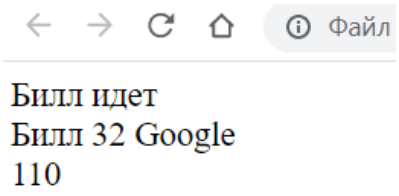
- Метод *getAge*() предназначен для получения значения переменной *_age* – **геттер**.
- Метод *setAge*() предназначен для установки значения переменной *_age* – **сеттер**.

Наследование

Рассмотрим **Наследование** на примере:

```
// 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(this.name + " " + 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(this.name+ " "+ this.age+ " "+ this.company+ "<br>");
    };
}
Employee.prototype = Object.create(User.prototype);//копируем прототип
var bill = new Employee("Билл", 32, "Google");
bill.go();
bill.displayInfo();
document.write(bill.maxage);
```



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

```
User.call(this, name, age);
```

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

```
Employee.prototype = Object.create(User.prototype);
```

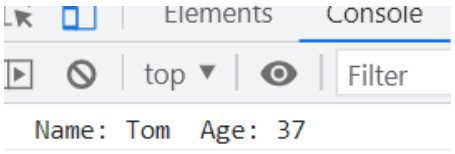
Классы

Еще один способ описывать и создавать объекты – это **класс**. **Класс** представляет описание объекта, его состояния и поведения, а объект является конкретным воплощением или экземпляром класса.

```
class Название [extends Родитель]  {
  constructor
  методы
}
```

Для создания объекта с помощью конструктора сначала ставится ключевое слово **new**. Затем собственно идет вызов конструктора – по сути, вызов функции по имени класса. По умолчанию классы имеют один конструктор без параметров.

```
class Person{
    name;
    age;
    constructor(personName, personAge){
        this.name = personName;
        this.age = personAge;
    }
    print(){
        console.log(`Name: ${this.name}  Age: ${this.age}`);
    }
}
const tom = new Person("Tom", 37);
tom.print();
```



Конструктор определяется с помощью метода с именем **constructor**.

Приватные поля и методы

Чтобы сделать свойства и методы класса **приватными**, их название должно начинаться с символа решетки #:

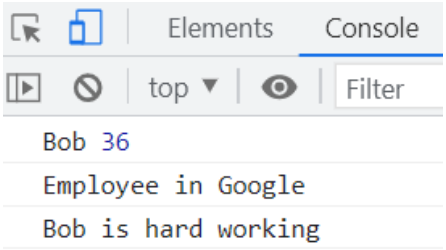
```
class Person{
    #name;
    #age;
    constructor(name, age){
        this.#name = name;
        this.#age = age;
    }
    print(){
        console.log(`Name: ${this.#name}  Age: ${this.#age}`);
    }
}
const tom = new Person("Tom", 37);
// tom.#name = "Sam";   // Ошибка - нельзя обратиться к приватному полю
```

Наследование в классах

Одни классы могут наследоваться от других. Для наследования одного класса от другого в определении класса применяется оператор **extends**, после которого идет название базового класса. Например:

```
class Person{
    name;
    age;
    constructor(name, age){
        this.name = name;
        this.age = age;
    }
    display(){
        console.log(this.name, this.age);
    }
}
class Employee extends Person{ // наследуется от Person
    company;
    constructor(name, age, company){
        super(name, age); // запускает конструктор класса-родителя
        this.company = company;
    }
    display(){
        super.display();// запускает display из класса-родителя
        console.log("Employee in", this.company);
    }
```

```
    work(){
        console.log(this.name, "is hard working");
    }
}
let bob = new Employee("Bob", 36, "Google");
bob.display();
bob.work();
```



Класс *Employee* наследуется от класса *Person*.

Ключевое слово **super** позволяет обращаться к базовому классу.

Знакомство с JS - https://webref.ru/dev/learn-javascript

Самоучитель JS - https://learn.javascript.ru/

Самоучитель JS - https://metanit.com/web/javascript/

Справочник JS - https://javascript.ru/manual

Справочник JS - https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference

Общие задания

Выполните следующие задания с использованием JS:

1) Создайте двухмерный **ассоциативный массив** (объект). У него должны быть два ключа – '*ru*' и '*en*'. Пусть *первый* ключ содержит элемент, являющийся массивом названий дней недели на русском языке, а *второй* – на английском.
   На странице должно появляется окно с вопросом «*Отобразить список дней недели на английском?*» (*confirm*). Если пользователь подтвердит, то отобразить на странице список из элементов массива под ключом '*en*', иначе – под '*ru*'.
2) Создайте конструктор для формирования объектов данных человека с полями «*ФИО*», «*возраст*» и «*рост*» и методом отображения данных в строке. Этот метод должен формировать *html*-код, где все данные представлены в таблице. На странице считайте данные нескольких объектов (*prompt*, также можно воспользоваться функцией *split* для разделения строки). Для создания объектов используйте конструктор. Отобразите данные на странице, воспользовавшись соответствующем методом каждого объекта. Заранее пропишите стиль таблицы.

3) Создайте базовый класс *Person* с полями «*ФИО*» и «*возраст*», а также двумя методами.

   Первый метод *getFieldStr* – отображение данных пользователя в виде *html*-кода, где каждое поле в отдельном абзаце <*p*> последовательно друг за другом. Возвращает полученную строку.

   Второй метод *getDataStr* – вызывает *getFieldStr* и полученную от него строку размещает в блоке <*div*>. Возвращает строку с соответствующем *html*-кодом.

   Создайте дочерний класс *Employee*, наследник *Person*, добавляющий поле «должность» и переопределяющий метод отображения данных пользователя *getDataStr*. Этот метод должен использовать метод базового класса *getFieldStr*, но добавлять к результату абзац <*p*> с полем «должность» и также размещать результат в блоке <*div*>. Возвращает строку с соответствующем *html*-кодом.

   Любым способом сформируйте массив из объектов классов *Person* и *Employee*, заполненных данными. С помощью цикла и вызова функции *getDataStr* у объектов отобразите полученные объекты на странице.

Индивидуальное задание

Дополните сайт, разработанный по выбранной теме по индивидуальному заданию прошлой лабораторной работы.

Измените способ добавления гиперссылки, которая открывает случайную страницу сайта. Для этого создайте массив *объектов* гиперссылок. Каждый объект должен представлять собой ссылку на страницу, текст ссылки, подсказку и CSS-класс стилевого оформления (сделайте несколько CSS-классов, свой для каждой ссылки с разным оформлением).

Также напишите функцию, которая на основе объекта из данного массива формирует строку с html-кодом (тег <*a*> со всеми необходимыми атрибутами и содержимым). Теперь при загрузке страницы должен выбираться случайный объект из массива, на его основе функция должна собирать html-код, который и размещается на странице.

На одной из страниц сайта должна быть **таблица с данными**. Выберите данные в соответствии с темой вашего сайта. Разработайте класс, представляющий данные этой таблицы. Для создания объектов класса используйте конструктор. Представьте объекты в виде массива. Напишите соответствующие методы и функции для отображения данных массива в таблице. То есть таблица с данными должна генерироваться на основе массива.