Лабораторная работа №4. Продвинутая работа с объектами

Цель работы: освоить основные принципы работы с рекурсивными алгоритмами, методами объектов и дескрипторами свойств объектов.

Краткие теоретические сведения по теме лабораторной работы:

Для простых объектов доступны следующие методы:

Object.keys(obj) – возвращает массив ключей.

Object.values(obj) – возвращает массив значений.

Object.entries(obj) – возвращает массив пар [ключ, значение].

У объектов нет множества методов, которые есть в массивах, например **map**, **filter** и др.

Если нужно их применить, то можно использовать **Object.entries** с последующим вызовом **Object.fromEntries**:

1) Вызов **Object.entries(obj)** возвращает массив пар ключ/значение для **obj**.

2) На нём вызываются методы массивов, например, **map**.

3) Используется **Object.fromEntries(array)** на результате, чтобы преобразовать его обратно в объект.

Помимо значения **value**, свойства объекта имеют три специальных атрибута (так называемые «флаги»).

writable – если **true**, свойство можно изменить, иначе оно только для чтения.

enumerable – если **true**, свойство перечисляется в циклах, в противном случае циклы его игнорируют.

configurable – если **true**, свойство можно удалить, а эти атрибуты можно изменять, иначе этого делать нельзя.

Метод **Object.getOwnPropertyDescriptor** позволяет получить полную информацию о свойстве. Его синтаксис:

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

obj – объект, из которого мы получаем информацию.

propertyName – имя свойства.

Возвращаемое значение – это объект, так называемый «дескриптор свойства»: он содержит значение свойства и все его флаги.

```
let user = {
  name: "John"
};

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/* дескриптор свойства:
{
  "value": "John",
  "writable": true,
  "enumerable": true,
  "configurable": true
}
*/
```

Чтобы изменить флаги, мы можем использовать метод **Object.defineProperty**.

Его синтаксис:

```
Object.defineProperty(obj, propertyName, descriptor)
```

obj, propertyName – объект и его свойство, для которого нужно применить дескриптор.

descriptor – применяемый дескриптор.

Если свойство существует, **defineProperty** обновит его флаги.

В противном случае метод создаёт новое свойство с указанным значением и флагами; если какой-либо флаг не указан явно, ему присваивается значение **false**.

Например, здесь создаётся свойство **name**, все флаги которого имеют значение **false**:

```
let user = {};

Object.defineProperty(user, "name", {
  value: "John"
});

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": "John",
  "writable": false,
  "enumerable": false,
  "configurable": false
}
 */
```

Сделаем свойство user.name доступным только для чтения. Для этого изменим флаг **writable**:

```
let user = {
  name: "John"
};

Object.defineProperty(user, "name", {
  writable: false
});

user.name = "Pete";
// Ошибка: нельзя менять доступное
// только для чтения свойство 'name'
```

Существует метод **Object.defineProperties(obj, descriptors)**, который позволяет определять множество свойств сразу.

Его синтаксис:

```
Object.defineProperties(obj, {
  prop1: descriptor1,
  prop2: descriptor2
  // ...
});
```

Например:

```
Object.defineProperties(user, {
  name: { value: "John", writable: false },
  surname: { value: "Smith", writable: false },
  // ...
});
```

Таким образом, мы можем определить множество свойств одной операцией.

Существуют 2 типа свойств объекта:

1) Свойства-данные (data properties). Все свойства, которые рассматривались нами до текущего момента, были свойствами-данными.

2) Свойства-аксессоры (accessor properties). Это функции, которые используются для присвоения и получения значения, но во внешнем коде они выглядят как обычные свойства объекта.

Свойства-аксессоры представлены методами:

«getter» – для чтения и «setter» – для записи.

При литеральном объявлении объекта они обозначаются как **get** и **set**:

```
let obj = {
  get propName() {
    // геттер, срабатывает при чтении obj.propName
  },

  set propName(value) {
    // сеттер, срабатывает при записи obj.propName = value
  }
};
```

Например, у нас есть объект **user** со свойствами **name** и **surname**. Добавим свойство объекта **fullName**, содержащее полное имя (**"John Smith"**). Разумеется, мы не будем дублировать уже имеющуюся информацию, поэтому реализуем его через аксессор:

```
let user = {
  name: "John",
  surname: "Smith",

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

alert(user.fullName); // John Smith
```

Снаружи свойство-аксессор выглядит как обычное свойство. Мы не вызываем **user.fullName** как функцию, а читаем как обычное свойство: геттер выполняет всю работу за кулисами.

```
let user = {
  get fullName() {
    return `...`;
  }
};

user.fullName = "Тест";
// Ошибка (у свойства есть только геттер)
```

Исправим это, добавив сеттер для **user.fullName**:

```
let user = {
  name: "John",
  surname: "Smith",

  get fullName() {
    return `${this.name} ${this.surname}`;
  },

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  }
};

// set fullName запустится с данным значением
user.fullName = "Alice Cooper";

alert(user.name); // Alice
alert(user.surname); // Cooper
```

Свойства-аксессоры не имеют **value** и **writable**, но взамен предлагают функции **get** и **set**.

Таким образом дескриптор аксессора может иметь:

get – функция без аргументов, которая сработает при чтении свойства,

set – функция, принимающая один аргумент, вызываемая при присвоении свойства,

enumerable – то же самое, что и для свойств-данных,

configurable – то же самое, что и для свойств-данных.

У аксессоров есть интересная область применения – они позволяют в любой момент взять «обычное» свойство и изменить его поведение, поменяв на геттер и сеттер.

Представим, что мы начали реализовывать объект **user**, используя свойства-данные имя **name** и возраст **age**:

```
function User(name, age) {
  this.name = name;
  this.age = age;
}

let john = new User("John", 25);

alert( john.age ); // 25
```

Взамен возраста **age** удобнее дату рождения **birthday**:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;
}

let john = new User("John", new Date(1992, 6, 1));
```

Что делать со старым кодом, который использует свойство **age**?

Неоптимальное решение – найти все такие места и изменить их. Также **age** – хорошее название для свойства в **user**. Давайте его сохраним.

Проблема решается добавлением геттера для **age**:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;

  // возраст рассчитывается из текущей даты и дня рождения
  Object.defineProperty(this, "age", {
    get() {
      let todayYear = new Date().getFullYear();
      return todayYear - this.birthday.getFullYear();
    }
  });
}

let john = new User("John", new Date(1992, 6, 1));

alert( john.birthday ); // доступен как день рождения
alert( john.age );      // ...так и возраст
```

Теперь старый код тоже работает, а также есть отличное дополнительное свойство.

Задачи для самостоятельной работы:

1. Создать пустой объект **obj**. Используя метод **Object.defineProperties**, добавить в этот объект следующие свойства:
   I) **count** – значение счетчика – равно **0**.
   II) **increm** – функция, увеличивающая **count** на **1**. При ее вызове в консоль должно выводиться сообщение о том, что **count** увеличился до **текущее значение count**.
   III) **decrem** – функция, уменьшающая **count** на **1**. При ее вызове в консоль должно выводиться сообщение о том, что **count** уменьшился до **текущее значение count**.

2. Создать функцию-конструктор **ObjConst(first, last)**, которая возвращает объект с 3 свойствами-данными **firstName** (его значение **first**), **lastName** (его значение **last**) и **fullName** (его значение **first last**). Все 3 свойства должны быть доступны только для чтения.

3. Дана функция-конструктор объектов **ObjConstr**:

   ```
   function ObjConstr(first, last) {
   this.firstName = first;
   this.lastName = last;
   }
   ```

   Модифицировать функцию так, чтобы она возвращала объект, в котором помимо свойств **firstName** и **lastName,** было бы свойство **fullName**, значение которого – строка **`${firstName} ${lastName}`**. Свойство **fullName** должно быть как геттером, так и сеттером.

   Пример работы:
   ```
   let joe = new ObjConstr('Joe', 'Smith');
   console.log (joe.firstName) // Joe
   console.log (joe.lastName) // Smith
   console.log (joe.fullName) // Joe Smith

   joe.fullName = 'Bob James';
   console.log (joe.firstName) // Bob
   console.log (joe.lastName) // James
   console.log (joe.fullName) // Bob James
   ```

4. Создать функцию-конструктор **Circle**, принимающую в качестве параметра радиус круга (**R**) и создающую объекты со свойством-данным радиус и свойством-аксессором **area**. Свойство **area** должно иметь геттер (возвращает площадь круга) и сеттер (устанавливает площадь круга, в результате чего должен меняться радиус). При

реализации сеттера учитывать, что площадь должна быть положительным числом.

Срабатывание сеттера должно сопровождаться появлением следующего сообщения в консоли:
'Задано новое значение для площади, равное ***площадь***, новое значение радиуса равно ***радиус*** '

5. Дана строка, в которой содержится потребительская корзина.
basket = '1 × meat, 4 × cheese, 2 × candy, 4 × milk'

Дан объект с ценами продуктов из потребительской корзины в разные годы.

```
let data = {
        2019: { meat: 1.25, cheese: 1, candy: 0.5, milk: 0.4 },
        2020: { meat: 1.25, cheese: 1.25, candy: 0.7, milk: 0.5 },
        2021: { meat: 2.5, cheese: 1, candy: 1, milk: 0.6 }
    }
```

Написать функцию, которая принимает 4 параметра: 1) объект с ценами продуктов из потребительской корзины; 2) строку, которая содержит потребительскую корзину; 3) начальный год; 4) конечный год.
Функция должна вернуть процент инфляции за период от начального года до конечного. Процент инфляции должен быть округлен до 2 знаков после запятой.

Пример работы:
func(data, basket, 2019, 2021) // 38.85

Стоимость потребительской корзины **basket** согласно **data** за **2019** год:
1*1.25 + 4*1 + 2*0.5 + 4*0.4 = **7.85**

Стоимость потребительской корзины **basket** согласно **data** за **2021** год:
1*2.5 + 4*1 + 2*1 + 4*0.6 = **10.9**

10.9 больше **7.85** на 38.85%, если округлить до 2 знаков после запятой