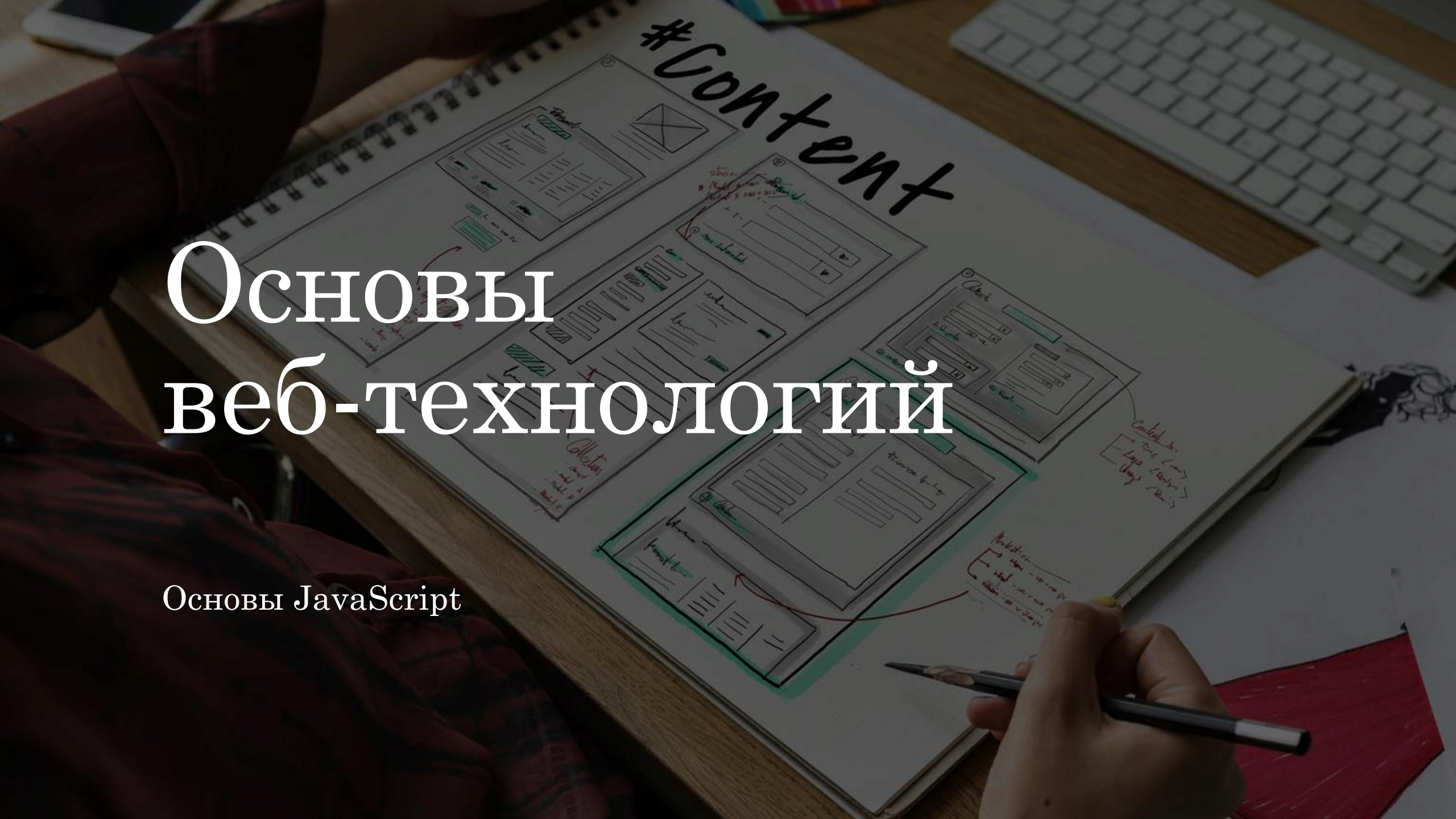

Основы веб-технологий

Основы JavaScript

Задачи на массивы и объекты

- Напишите функцию, которая разворачивает массив в обратном порядке без использования `reverse()`

```
function myReverse(a){
  let a1 = [];
  for(let i = a.length-1; i>=0; i--) a1.push(a[i]);
  return a1;
}

console.log(myReverse([2,4,5,3,1]));
console.log(myReverse([]));
```

Задачи на массивы и объекты

Напишите функцию, которая находит сумму положительных четных элементов массива

```
function mySum1(a) {
  let sum = 0;
  for(let v of a){
    if (v>0 && v%2 == 0){
      sum += v;
    }
  }
  return sum;
}
console.log(mySum1([-2, 2, 4, 5, -8, 7]))
console.log(mySum1([]))

function mySum2(a) {
  return a.filter((v)=>(v%2==0 && v>0)).reduce((sum, v) => sum+v, 0);
}
console.log(mySum2([-2, 2, 4, 5, -8, 7]))
console.log(mySum2([]))

const mySum3 = (a) => {
  return a.filter((v)=>(v%2==0 && v>0)).reduce((sum, v) => sum+v, 0);
}
console.log(mySum3([-2, 2, 4, 5, -8, 7]))
console.log(mySum3([]))
```

Задачи на массивы и объекты

- Напишите функцию, которая сравнивает 2 массива и возвращает `true`, если они одинаковые и `false`, если нет. Считаем, что массивы содержат элементы только примитивных типов.

```
function isEqual1 (a1, a2) {
  if(a1.length!=a2.length) return false;
  for(let i = 0; i<a1.length; i++){
    if (a1[i]!==a2[i]) return false;
  }
  return true;
}
console.log(isEqual1([1,2,3], [1,2,3]));
console.log(isEqual1([1,2,3], [1,2,2]));

const  isEqual2 = (a1, a2) => {
  if(a1.length!=a2.length) return false;
  return !a1.map((item, i) => item === a2[i]).includes(false) ;
}

console.log(isEqual2([1,2,3], [1,2,3]));
console.log(isEqual2([1,2,3], [1,2,2]));
```

Задачи на массивы и объекты

- Напишите функцию, которая сравнивает два объекта на равенство и возвращает `true`, если они одинаковые и `false`, если нет. Считаем, что значения свойств являются примитивами.

```
function isEqual3(o1, o2) {
    const keys1 = Object.keys(o1);
    const keys2 = Object.keys(o2);
    if (keys1.length!=keys2.length) return false;
    for (let key of keys1){
      if (!(key in o2)) return false;
      if (o1[key]!==o2[key]) return false;
    }
    return true;
}

console.log(isEqual3({x: 3, y: '5'}, {x: 3, y: 5, z: 0}))
```

Дата и время

В JavaScript создание даты осуществляется с помощью объекта Date. Объект Date представляет собой точку на оси времени и предназначен для хранения даты и времени с точностью до миллисекунды.

Создание текущей даты и времени

```
let nowDate = new Date();
console.log(nowDate); // Sat Dec 02 2023 17:40:24 GMT+0300 (Москва, стандартное время)
console.log(nowDate.toLocaleDateString()); // '02.12.2023'
console.log(nowDate.toLocaleTimeString()); // '17:40:24'
console.log(nowDate.toLocaleTimeString().slice(0,-3)); // '17:40’
console.log(nowDate.toLocaleString()); // '02.12.2023, 17:40:24’
```

Создание даты путём указания объекту Date количества миллисекунд, прошедших с 1 января 1970 00:00:00 UTC.

```
console.log(new Date(31536000000)); // Fri Jan 01 1971 03:00:00 GMT+0300 (Москва, стандартное время)
```

Создание даты посредством указания её объекту Date в виде строки.

```
let date1 = new Date('2023-12-02'); //YYYY-MM-DD
let date2 = new Date('2023-12-02T17:45'); //YYYY-MM-DDThh:mm:ss.sss
let date3 = new Date('2023-12-02T17:45+03:00'); //(формат YYYY-MM-DDThh:mm:ss.sss±hh:mm)
```

Создание даты посредством указания следующих параметров через запятую: год (4 цифры), месяц (отсчёт ведётся с 0), день (1..31), часы (0..23), минуты (0..59), секунды (0..59), миллисекунды (0..999). Причём обязательными из них являются только первые два параметра.

```
let  date4 = new Date(2023, 11, 6); // Wed Dec 06 2023 00:00:00 GMT+0300 (Москва, стандартное время)
let  date5 = new Date(2023,3,1,18,17); //Wed Dec 06 2023 18:17:00 GMT+0300 (Москва, стандартное время)
```

Дата и время

В JavaScript для получения отдельных компонентов даты и времени предназначены следующие методы:

- getFullYear() – возвращает год, состоящий из 4 чисел;
- getMonth() – возвращает месяц в формате числа от 0 до 11 (0 – январь, 1 – февраль, 2 – март, ..., 11 – декабрь);
- getDate() – возвращает число месяца от 1 до 31;
- getHours() – возвращает количество часов от 0 до 23;
- getMinutes() – возвращает количество минут от 0 до 59;
- getSeconds() – возвращает количество секунд от 0 до 59;
- getMilliseconds() – возвращает количество миллисекунд от 0 до 999.
- getDay() – возвращает номер дня недели. Нумерация начинается с воскресенья, с нуля.

```
let   now = new Date(),   hour = now.getHours(),  message = '';
// определим фразу приветствия в зависимости от местного времени пользователя
if (hour <= 6) {
  message = 'Доброе время суток';
} else if (hour <= 12) {
  message = 'Доброе утро';
} else if (hour <= 18) {
  message = 'Добрый день';
} else {
  message = 'Добрый вечер';
}
alert(message);
```

Задачи на работы с датами

1. Напишите функцию, которая находит число дней до следующего нового года.
2. Напишите функцию, которая находит, является ли переданная дата средой.

Задачи на работы с датами

```
//Напишите функцию, которая находит число дней до следующего нового года.

const daysToNewYear = () =>{
  let now = new Date();
  let newYear = new Date(now.getFullYear()+1, 0, 1);
  return Math.floor((newYear-now)/1000/60/60/24);
}

console.log(daysToNewYear());


//Напишите функцию, которая находит, является ли переданная дата средой.

const isWednesday = (date) => {
  return date.getDay()==3;
}

console.log(isWednesday(new Date(2023, 5, 7)));
```

JSON

JSON (JavaScript Object Notation) – это текстовый формат представления данных в нотации объекта JavaScript. Предназначен JSON, также как и некоторые другие форматы такие как XML и YAML, для обмена данными.
JSON по сравнению с другими форматами обладает достаточно весомым преимуществом. Он в отличие от них является более компактным, а это очень важно при обмене данными в сети Интернет. Кроме этого, JSON более прост в использовании, его намного проще читать и писать.
В JSON, данные организованы также как в объекте JavaScript. Но JSON – это не объект, а строка. При этом не любой объект JavaScript может быть переведён один к одному в JSON. Например, если у объекта есть методы, то они при преобразовании в строку JSON будут проигнорированы и не включены в неё.
Структура строки JSON практически ничем не отличается от записи JavaScript объекта.
Она состоит из набора пар ключ-значения. В этой паре ключ отделяется от значения с помощью знака двоеточия (:), а одна пара от другой - с помощью запятой (,). При этом ключ в JSON, в отличие от объекта обязательно должен быть заключен в двойные кавычки.

```
// строка JSON
const jsonPerson = '{"name":"Иван","age":25}';

// объект JavaScript
const person = {
  name: "Иван",
  age: 25
};
```

JSON

Работа с JSON в JavaScript обычно осуществляется в двух направлениях:

- перевод строки JSON в объект (данный процесс ещё называют парсингом);
- конвертирование объекта в строку JSON (другими словами действие обратное парсингу).

В JavaScript для парсинга строки в JSON рекомендуется использовать метод JSON.parse

```
const personFromJSON = JSON.parse(jsonPerson);
console.log(personFromJSON)
```

Перевод объекта JavaScript в строку JSON осуществляется с помощью метода JSON.stringify. Данный метод осуществляет действие обратное методу JSON.parse.

```
const personString = JSON.stringify(personFromJSON);
console.log(personString)
```

Работа с данными JSON после парсинга осуществляется как с объектом JavaScript.

Обработка ошибок, "try..catch"

Синтаксическая конструкция try..catchпозволяет «ловить» ошибки и вместо «падения» скрипта делать что-то более осмысленное.

```
try {
  // код...
} catch (err) {
  // обработка ошибки
}
```

Как работает:

1. Сначала выполняется код внутри блока `try {...}`.
2. Если в нём нет ошибок, то блок `catch(err)` игнорируется: выполнение доходит до конца try и потом далее, полностью пропуская `catch`.
3. Если же в нём возникает ошибка, то выполнение `try` прерывается, и поток управления переходит в начало `catch(err)`. Переменная `err` (можно использовать любое имя) содержит объект ошибки с подробной информацией о произошедшем.

```
try {
  let sum = 10n +10;
} catch (err) {
  console.log("Ошибка при суммировании: " + err.name + ": " + err.message);
}
console.log("Продолжаем выполнение программы");
```

Генерация собственных ошибок

Оператор `throw` генерирует ошибку. Технически в качестве объекта ошибки можно передать что угодно. Это может быть даже примитив, число или строка, но всё же лучше, чтобы это был объект, желательно со свойствами `name` и `message` (для совместимости со встроенными ошибками).

В JavaScript есть множество встроенных конструкторов для стандартных ошибок: `Error`, `SyntaxError`, `ReferenceError`, `TypeError` и другие. Можно использовать и их для создания объектов ошибки.

```
if (!user.name) {
    throw new SyntaxError("Данные неполны: нет имени");
}
```

Классы

Начиная с ECMAScript 2015 (ES6) в языке JavaScript появился синтаксис классов. В этом синтаксисе вы можете создавать классы, используя ключевое слово class.
А как же дела обстояли до этого стандарта? Как создавались классы? Их просто не было, так как JavaScript не является классическим объектно-ориентированным языком. Создание объектов осуществлялось с помощью конструкторов или, другими словами, посредством обычных функций, которые вызывались с использованием ключевого слова new.
Под классом в JavaScript мы будем понимать шаблон, на основании которого создаются объекты. Чтобы не путаться, классом будем считать как шаблон, созданный с помощью конструктора, так и с помощью ключевого слова class.

```
function Rect(width, height) {
    this.height = height;
    this.width = width;
    this.calcArea = function(){
        return this.width*this.height;
    }
}
```

```
class Rect {
    constructor(width, height) {
        this.height = height;
        this.width = width;
        this.calcArea = function () {
            return this.width * this.height;
        };
    }
}
```

Что же с наследованием? В JavaScript имеется только прототипное наследование и работает оно через объекты. С приходом классов в этом плане ничего не изменилось.
Так что же такое class? class – это специальная функция, предназначенная для создания объектов и организации существующего в языке прототипного наследования. То есть class – это просто более новый способ написания шаблона с большим количеством крутых возможностей, который затем можно использовать для создания нужного количества однотипных объектов.

Класс: базовый синтаксис

```
class MyClass {
  prop = value; // свойство
  constructor(...) { // конструктор
    // ...
  }
  method(...) {} // метод
  get something(...) {} // геттер
  set something(...) {} // сеттер
  [Symbol.iterator]() {} // метод с вычисляемым именем (здесь - символом)
  // ...
}
```

Наследование

Наследование классов – это способ расширения одного класса другим классом. Таким образом, мы можем добавить новый функционал к уже существующему.

```
class Point2D {
    constructor(x, y) {
      this.x = x;
      this.y = y;
    }
    toString() {
      return `(${this.x},${this.y})`;
    }
}
class Point3D extends Point2D {
    constructor(x, y, z) {
      super(x, y);
      this.z = z;
    }
    toString() {
      return `(${this.x},${this.y},${this.z})`;
    }
}

let p2 = new Point2D(4, 6);
let p3 = new Point3D(4, 6, 8);
console.log(p2.toString())
console.log(p3.toString())
```

Приватные и защищённые методы и свойства

Один из важнейших принципов объектно-ориентированного программирования – разделение внутреннего и внешнего интерфейсов. Защищённые поля не реализованы в JavaScript на уровне языка, но на практике они очень удобны, поэтому их эмулируют.

Защищённые свойства обычно начинаются с префикса _, а для доступа к элементу используются сеттеры и геттеры.

```
class CoffeeMachine {
  _waterAmount = 0;
  set waterAmount(value) {
    if (value < 0) throw new Error("Отрицательное количество воды");
    this._waterAmount = value;
  }
  get waterAmount() {
    return this._waterAmount;
  }
  constructor(power) {
    this._power = power;
  }
}
// создаём новую кофеварку
let coffeeMachine = new CoffeeMachine(100);
// устанавливаем количество воды
coffeeMachine.waterAmount = -10; // Error: Отрицательное количество воды
```