

# Объекты и классы в JavaScript

---

**JS**  
**COURSE**  
**ORT DNIPRO**

---

**ORT****DNIPRO**.ORG/**JS**

# 1. Объекты

# Объекты в JavaScript

```
2
3   let person = {
4       name: "Jhon",
5       lastName: "Smith",
6       sayHello: function(){
7           return `Hello my name is ${this.name} ${this.lastName}`;
8       }
9   }
10
11 console.log( person.sayHello() );
```

**Объект** в JavaScript представляет собой ассоциативный массив содержащий данные (свойства) и функции (методы) которые эти данные обрабатывают. **Объект** в JavaScript один из шести базовых типов данных.

Подробнее: <https://learn.javascript.ru/object>

# Ключевое слово **this**

```
2
3   let person = {
4       name: "Jhon",
5       lastName: "Smith",
6       sayHello: function(){
7           return `Hello my name is ${this.name} ${this.lastName}`;
8       }
9   }
10
11 console.log( person.sayHello() );
```

Ключевое слово **this** – ссылка на сам объект. Другими словами **this** указывает на тот ассоциативный массив (объект) которому принадлежит функция, в которой **this** используется встречается. **this** используется только в функциях объекта. **Важно: у arrow-функций нет своего this.**

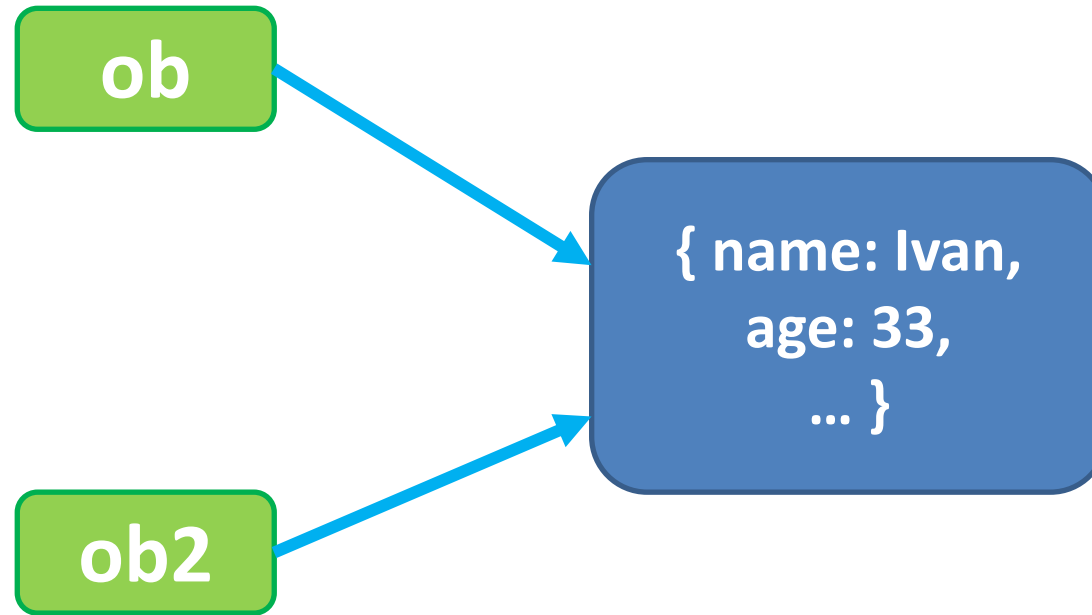
Подробнее: <https://learn.javascript.ru/object-methods>

# Объекты в JavaScript

```
2
3   let person = {
4       name: "Jhon",
5       lastName: "Smith"
6   }
7
8   person = null;
9
10  console.log(person, typeof person);
11
```

**null** – заглушка на случай “когда объекта нет”.

# Объекты в JavaScript



**object** - ссылочная структура данных, т.е сам объект находится где-то в памяти, а в переменной находится только ссылка на него, поэтому когда мы копируем такую переменную в другую, то копируются только ссылки, а сам объект остаётся одним и тем же.

# this привязывается в динамике

```
2
3  let func = function(){
4      return `Hello my name is ${this.name} ${this.lastName}`;
5  }
6
7  let person_1 = {
8      name: "Jhon",
9      lastName: "Smith",
10     sayHello: func
11 }
12
13 let person_2 = {
14     name: "Alice",
15     lastName: "Gates",
16     sayHello: func
17 }
18
19 console.log( person_1.sayHello() );
20 console.log( person_2.sayHello() );
21
```

**this** привязывается к объекту в момент вызова метода, поэтому одна и та же функция может входить в состав двух и большего количества объектов.

Подробнее: <https://learn.javascript.ru/object-methods>

# Конструктор – Когда нужно много однотипных объектов

```
2
3   let func = function(){
4       |   return `Hello my name is ${this.name} ${this.lastName}`;
5   }
6
7   function Person(name, lastName){
8       |   this.name      = name;
9       |   this.lastName  = lastName;
10      |   this.sayHello  = func;
11  }
12
13  let person_1 = new Person('Jhon', 'Smith');
14  let person_2 = new Person('Alice', 'Gates');
15  let person_3 = new Person('Bill', 'Roberts');
16
17  console.log(person_1.sayHello());
18  console.log(person_2.sayHello());
19  console.log(person_2.sayHello());
20
```

**Функция-конструктор** - позволяет создавать много однотипных объектов. Функция конструктор всегда должна использоваться с оператором **new**, иначе у неё не будет доступа к **this** нового созданного объекта.

Использовать оператор **return** не нужно. Конструктор может (и как правило должен) иметь параметры.

Подробнее: <https://learn.javascript.ru/constructor-new>



## 2. Прототипы

# Прототипы

У объекта может быть объект-предок, в **JavaScript** его называют **прототипом**. Если требуемое свойство (или метод) не найден в объекте, то оно ищется у **прототипа**.

**Прототип** это объект который «дополняет» своими свойствами и методами другой (дочерний) объект. Установить кто у объекта будет **прототипом** можно при помощи свойства **\_\_proto\_\_**.

Благодаря **прототипам** в **JavaScript** можно организовать объекты в «**цепочки**» так, чтобы свойство, не найденное в одном объекте, автоматически искалось бы в другом (родительском).

Подробнее: <https://learn.javascript.ru/prototypes>

# Прототипы

```
2
3   let func = function(){
4       return `Hello my name is ${this.name} ${this.lastName}`;
5   }
6
7   let family = {
8       lastName: "Smith",
9       sayHello: func
10  }
11
12  function Person(name){
13      this.name      = name;
14      this.__proto__ = family;
15  }
16
17  let person_1 = new Person('Jhon');
18  let person_2 = new Person('Alice');
19  let person_3 = new Person('Bill');
20
21  console.log(person_1.sayHello());
22  console.log(person_2.sayHello());
23  console.log(person_2.sayHello());
24
```

Свойство или метод не найденные в объекте – будут взяты из **прототипа** (или *прототипа* *прототипа*, если в цепочке прототипов искомое свойство или метод есть).

# 3. Методы

`.toString()` / `.valueOf()` / `[Symbol.toPrimitive]()`

## Методы `.toString()` / `.valueOf()` / `[Symbol.toPrimitive]()` у объектов

```
2
3   let auto_1 = {
4       title: "Ford Focus",
5       id: "AE5589BH"
6   }
7
8   let auto_2 = {
9       title: "Honda Accord",
10      id: "CH5633TB",
11      toString: function(){
12          return `${this.title} (${this.id})`;
13      }
14   }
15
16   alert(auto_1);
17   alert(auto_2);
18
```

localhost:5000 says

[object Object]

OK

localhost:5000 says

Honda Accord (CH5633TB)

OK

Метод `.toString()`, если он определен у объекта – позволяет браузеру корректно преобразовать объект **к строке**. Также есть метод `.valueOf()` для преобразования **к числу**. В **ES2015** появился универсальный метод: `object[Symbol.toPrimitive](typeName)` позволяющий реализовать преобразование к нужному типу (но опять же: числу или строке).

Подробнее: <https://learn.javascript.ru/object-toprimitive>

## 4. Обект Date

# Дата/Время в JavaScript

```
2
3   let currentDate = new Date();
4   console.log(currentDate);
5   console.log(currentDate.toISOString());
6
7   let dateA = new Date(2019, 10, 18, 17, 23, 56);
8
9   console.log(dateA, +dateA);
10
```

В JavaScript есть (относительно) удобная возможность работы с датой и временем – объект **Date**. Дату можно преобразовать к **UTC**-виду или **timestamp**'у, и получить отдельные её компоненты (*год, месяц, ... минуты, секунды*).

Подробнее: <https://learn.javascript.ru/datetime>

# Дата/Время в JavaScript

```
2
3   let newYear2020 = new Date(2020, 0, 1, 0,0,0);
4   let now          = new Date();
5
6   let diff = newYear2020 - now;
7
8   diff = Math.floor(diff / (1000 * 60 * 60 * 24));
9
10  console.log(`New Year 2020 after ${diff} days`);
11
```

Две даты можно вычитать одну из другой, в результате мы можем получить разницу в миллисекундах между этими датами. Это возможно за счёт преобразования даты к числу (**Timestamp'y**) которое показывает кол-во миллисекунд прошедшее от начала Unix-эпохи.

Подробнее: <https://learn.javascript.ru/datetime>



# Дата/Время в JavaScript

Важные моменты при работе с **датой/временем**:

- 1) Не забывать про разницу между местным и UTC-временем;
- 2) Не забывать про смещение ( метод: **.getTimezoneOffset()**);
- 3) Помнить о возможности преобразования даты времени в **timestamp** и обратно;
- 4) Помнить о возможности выполнять **вычитание** дат (и тем самым находить продолжительность какого-либо процесса);
- 5) JavaScript даёт определённые возможности по форматирование вывода даты/времени, при помощи методов **.toLocaleString()**, **.toLocaleDateString()**, **.toLocaleTimeString()**. Но эти возможности крайне ограничены.

Подробнее: <https://habr.com/ru/company/mailru/blog/438286/>

# 5. Классы

# Классы в ECMAScript 2015-2019

**Классы** пришли в JavaScript из других (типизированных) языков программирования. В которых классы применяли для описание структуры объектов которые на основе класса создаются. **Класс** выступают своего рода «чертежом» по которому будут создаваться объекты.

Подробнее: <https://learn.javascript.ru/class>

```
class Parcel{
  #code;
  #width;
  #length;
  #height;

  constructor(code, w, l, h){
    this.#code = code;
    this.#width = w;
    this.#length = l;
    this.#height = h;
  }

  getVolume(){
    return this.#width * this.#length * this.#height;
  }

  getReport(){
    return `Parcel ${this.code}: ${this.getVolume()}`;
  }
}

let box = new Parcel(100, 20, 45);

console.log(box.getReport());
```

# Классы в ECMAScript 2015-2019

**Классы** в JavaScript'е являются лишь надстройкой («маскировкой», «синтаксическим сахаром») над **прототипной** моделью построения объектов. И не являются её заменой.

Подробнее: <https://learn.javascript.ru/class>

# Классы в ECMAScript 2015-2019

По сути описывая **класс** мы создаём функцию **конструктор** в которой идёт перечисление свойств и методом будущего объекта. А далее эта функция вызывается через оператор **new**.

В **ES2019** была добавлена возможность создавать **приватные** (закрытые) свойства и методы. К этим методам есть возможность обратиться только из методов объекта. Из вне они недоступны.

Их легко отличить по символу **#** в начале имени.

Подробнее: <https://learn.javascript.ru/private-protected-properties-methods>

# Наследование

Класс может расширять функционал (наследовать) другого (родительского) класса, а по сути добавлять в него дополнительные методы и свойства. Для указания этого применяется ключевое слово **extends**. В конструкторе дочернего класса (и в его методах) можно обращаться к конструктору (и методам) класса родителя, для этого применяется ключевое слово **super**.

Подробнее:

<https://learn.javascript.ru/extend-natives>

```
class Person{
  #name;
  #lastName;

  constructor(name, lastName){
    this.#name = name;
    this.#lastName = lastName;
  }
  getInfo(){
    return `${this.#name} ${this.#lastName}`;
  }
}

class Driver extends Person{
  #driverLicense;

  constructor(name, lastName, driverLicense){
    super(name, lastName);
    this.#driverLicense = driverLicense;
  }

  getInfo(){
    return super.getInfo() + `, Driver License: ${this.#driverLicense}`;
  }
}

let driver = new Driver('Jhon', 'Smith', 'AA324356');

console.log(driver.getInfo());
```

# Статические свойства и методы

```
2
3   class Demo{
4       static #counter = 0;
5
6       static getNext(){
7           return ++this.#counter;
8       }
9   }
10
11   console.log( Demo.getNext() );
12   console.log( Demo.getNext() );
13   console.log( Demo.getNext() );
14
```

Статические свойства и методы помечаются ключевым словом **static** к ним можно обращаться без создания экземпляра класса (объекта). Статические свойства и методы хороши для данных которые являются общими для всех объектов класса.

Подробнее: <https://learn.javascript.ru/static-properties-methods>

# 6. *AJAX*



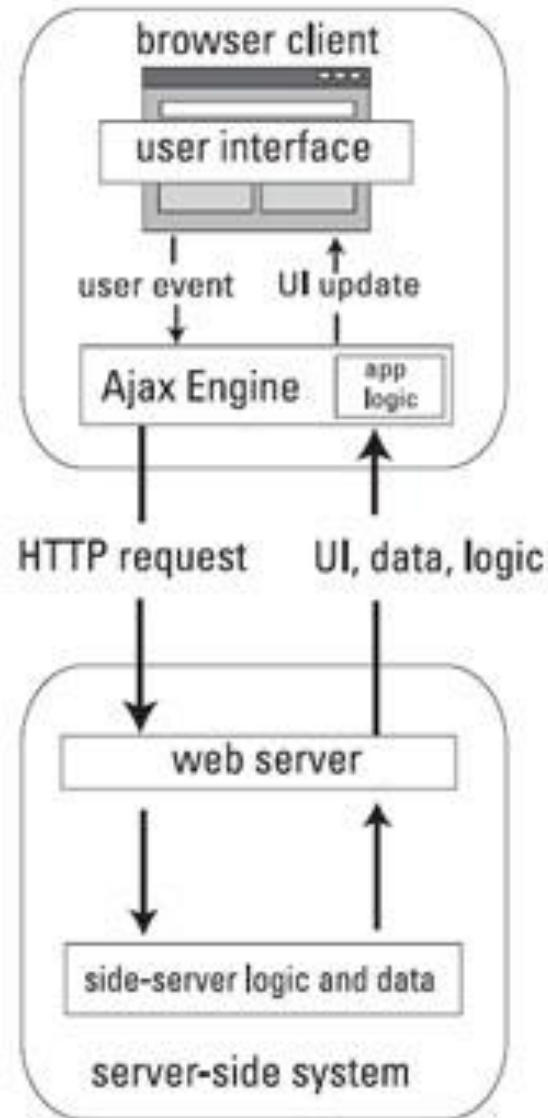
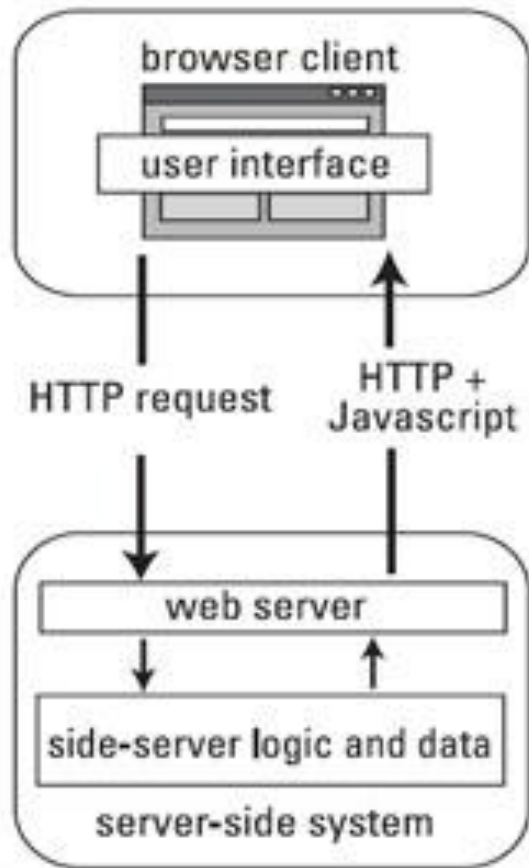
# Asynchronous JavaScript And XML



За изменение страницы в браузере пользователя отвечает JavaScript, но до этого момента JavaScript изменял страницу только на основе данных полученные еще при загрузке страницы в браузер и/или в зависимости от действий пользователя. Получить какие-то новые (дополнительные) данные JavaScript не мог.

С появлением в браузерах специального объекта **XMLHttpRequest** у **JavaScript** появилась возможность делать HTTP-запросы к сайтам, и изменять страницу уже на основе данных которых не было при загрузке странице. Т.е. дозагружать HTML и/или другие данные и вставлять их на страницу.

# Asynchronous JavaScript And XML



Идея заложенная в **AJAX** - не перезагружая страницу полностью, запросить у сервера данные и вставить их в дерево документа.

# 7. Обект

**XMLHttpRequest**

# Объект XMLHttpRequest

Объект **XMLHttpRequest** позволяет использовать функциональность HTTP-клиента, а по простому – делать HTTP-запросы когда страница уже в браузере.

*Несмотря на наличие **XML** в названии объекта, с его помощью можно передавать и другие форматы данных.*

Подробнее: <http://xmlhttprequest.ru>

# Объект XMLHttpRequest

```
2
3   let url = 'https://api.privatbank.ua/p24api/pubinfo?exchange&json&coursid=11';
4
5   let xhr = new XMLHttpRequest();
6
7   xhr.open("GET", url);
8
9   xhr.onload = function(){
10       let data = JSON.parse(this.response);
11       console.log(data);
12   }
13
14   xhr.send();
15
```

Объект **XMLHttpRequest** позволяет использовать функциональность HTTP-клиента, а по простому – делать HTTP-запросы когда страница уже в браузере.

Подробнее: <http://xmlhttprequest.ru>

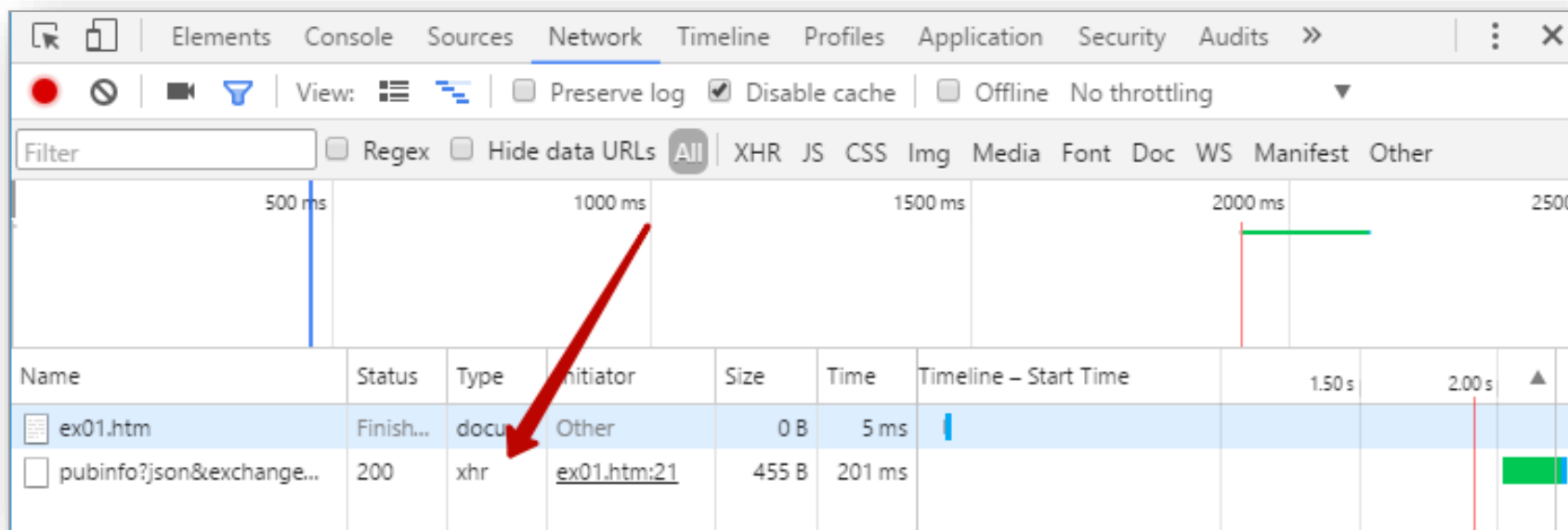
# Объект XMLHttpRequest

**XMLHttpRequest** – поддерживает событийную модель, и в зависимости от развития ситуации генерирует те или иные события.

**Синхронный запрос** – при котором браузер ждёт ответа, скрипт при этом «замирает» до прихода ответа.  
**Асинхронный** – скрипт продолжает выполняться, при поступлении ответа будет вызвана функция зарегистрированная как обработчик события **onload**.

Подробнее: <http://xmlhttprequest.ru>

# Объект XMLHttpRequest



В консоли разработчика хорошо заметны запросы которые делались через **XMLHttpRequest** – по характерной метке **type** равной **xhr**.

## 8. Глобальный объект `globalThis` (`window`)



# Глобальный объект **window**

Браузер добавляет в JavaScript всего один объект – **window**. Но этот объект содержит все необходимые инструменты для манипуляции HTML-документом.

Подробнее: <https://learn.javascript.ru/global-object>

# Глобальный объект **window**

Объект **window** можно использовать неявно, т.е. опускать его имя при написании кода.

## Свойства и методы window

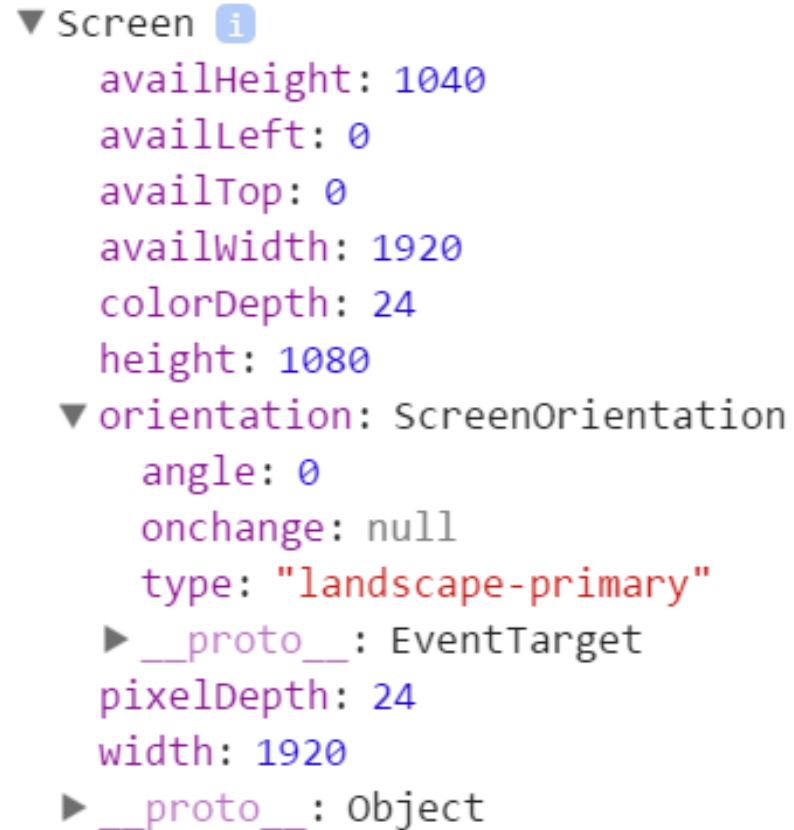
```
.setInterval() ;  
.setTimeout() ;  
.alert() ;  
.prompt() ;  
.confirm() ;  
...
```

# Глобальный объект **window**

```
▼ Location ⓘ  
  ▶ ancestorOrigins: DOMStringList {length: 0}  
  ▶ origin: "http://localhost:5000"  
  ▶ protocol: "http:"  
  ▶ host: "localhost:5000"  
  ▶ hostname: "localhost"  
  ▶ port: "5000"  
  ▶ pathname: "/"  
  ▶ search: ""  
  ▶ hash: ""  
  ▶ href: "http://localhost:5000/"  
  ▶ assign: f assign()  
  ▶ reload: f reload()  
  ▶ toString: f toString()  
  ▶ replace: f replace()  
  ▶ valueOf: f valueOf()  
  ▶ Symbol(Symbol.toPrimitive): undefined  
  ▶ __proto__: Location
```

**window.location** –  
свойство определяющее  
какую страницу содержит  
окно браузера.

# Глобальный объект **window**



```
▼ Screen ⓘ  
  availHeight: 1040  
  availLeft: 0  
  availTop: 0  
  availWidth: 1920  
  colorDepth: 24  
  height: 1080  
  ▼ orientation: ScreenOrientation  
    angle: 0  
    onchange: null  
    type: "landscape-primary"  
    ► __proto__: EventTarget  
  pixelDepth: 24  
  width: 1920  
  ► __proto__: Object
```

**window.screen** – информация об экране, размерах, ориентации и т.д.

# Глобальный объект **window**

```
▼ Navigator ⓘ
  appCodeName: "Mozilla"
  appName: "Netscape"
  appVersion: "5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/49.0.2623.87 Sa
  cookieEnabled: true
  doNotTrack: null
  ▶ geolocation: Geolocation
  hardwareConcurrency: 4
  language: "ru"
  ▶ languages: Array[4]
  maxTouchPoints: 0
  ▶ mediaDevices: MediaDevices
  ▶ mimeTypeArray: MimeTypeArray
  onLine: true
  ▶ permissions: Permissions
  platform: "Win32"
  ▶ plugins: PluginArray
  ▶ presentation: Presentation
  product: "Gecko"
  productSub: "20030107"
  ▶ serviceWorker: ServiceWorkerContainer
  userAgent: "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/49.0.262
  vendor: "Google Inc."
  vendorSub: ""
  ▶ webkitPersistentStorage: DeprecatedStorageQuota
  ▶ webkitTemporaryStorage: DeprecatedStorageQuota
  ▶ __proto__: Object
```

**window.navigator** – информация о браузере.

# **window.document** (корень DOM-дерева) хранилище HTML-документа

Подробнее: <https://learn.javascript.ru/dom-nodes>

**Но это уже другая история....**

## **9. Обработка ошибок (исключительных ситуаций)**

# Обработка ошибок (исключений)

```
2
3   let f = function(a, b){
4       |   return a + b;
5   }
6
7   f = 42;
8
9   try{
10      |   let result = f(2, 3);
11  }catch(e){
12      |   console.log("This code if we have error", e);
13  }finally{
14      |   console.log("This code work always");
15  }
16
```

Если в блоке **try** произойдёт ошибка, выполнение блока прекратится и перейдёт к блоку **catch**, в котором могут быть выполнены какие-либо действия направленные на нивелирования влияния ошибки на работу скрипта. Если в блоке **try** ошибка не произошла, то блок **catch** не выполняется. Независимо от того произошла ошибка или нет, после **try-catch** скрипт пойдёт выполняться дальше, как ни в чём не бывало. Блок **finally** выполняется в любом случае. В этом блоке обычно размещается код который должен при любом варианте развития событий завершить те или иные действий (например убрал иконку-лоадер с экрана независимо от того успешна ли была загрузка).

Подробнее: <https://learn.javascript.ru/try-catch>



## Генерация ошибки | оператор **throw**

```
2
3     try{
4
5         throw new Error("Info about error!");
6
7     }catch(e){
8         console.log("This code if we have error", e);
9     }finally{
10        console.log("This code work always");
11    }
12
```

При помощи оператора **throw** мы можем «выбросить» свою ошибку, для этого оператору достаточно передать любое значение, но хорошей практикой является использование для этих целей объекта **Error** или производного от него.

Подробнее: <https://developer.mozilla.org/uk/docs/Web/JavaScript/Reference/Statements/throw>

# 10. Принципы модульного тестирования (Unit Testing)

# Unit testing – модульное тестирование

```
2
3  function calc_sum(a, b){
4      let result = a + b;
5      return result;
6  }
7
8  (function(){
9      let control = calc_sum(2, 3);
10
11     if(control === 5){
12         console.log("calc_sum() - OK");
13     }else{
14         console.log("calc_sum() - FAIL");
15     }
16 })();
17
```

Идея **модульного тестирования (Unit testing)** в том, чтобы писать код который будет проверять работу основного кода. Функция, как пример модуля, может быть протестирована другой, написанной нами функцией. Основная польза модульного тестирования в том, что при изменении кода функции мы может оперативно определить не поломался ли её функционал.

# Unit testing – модульное тестирование

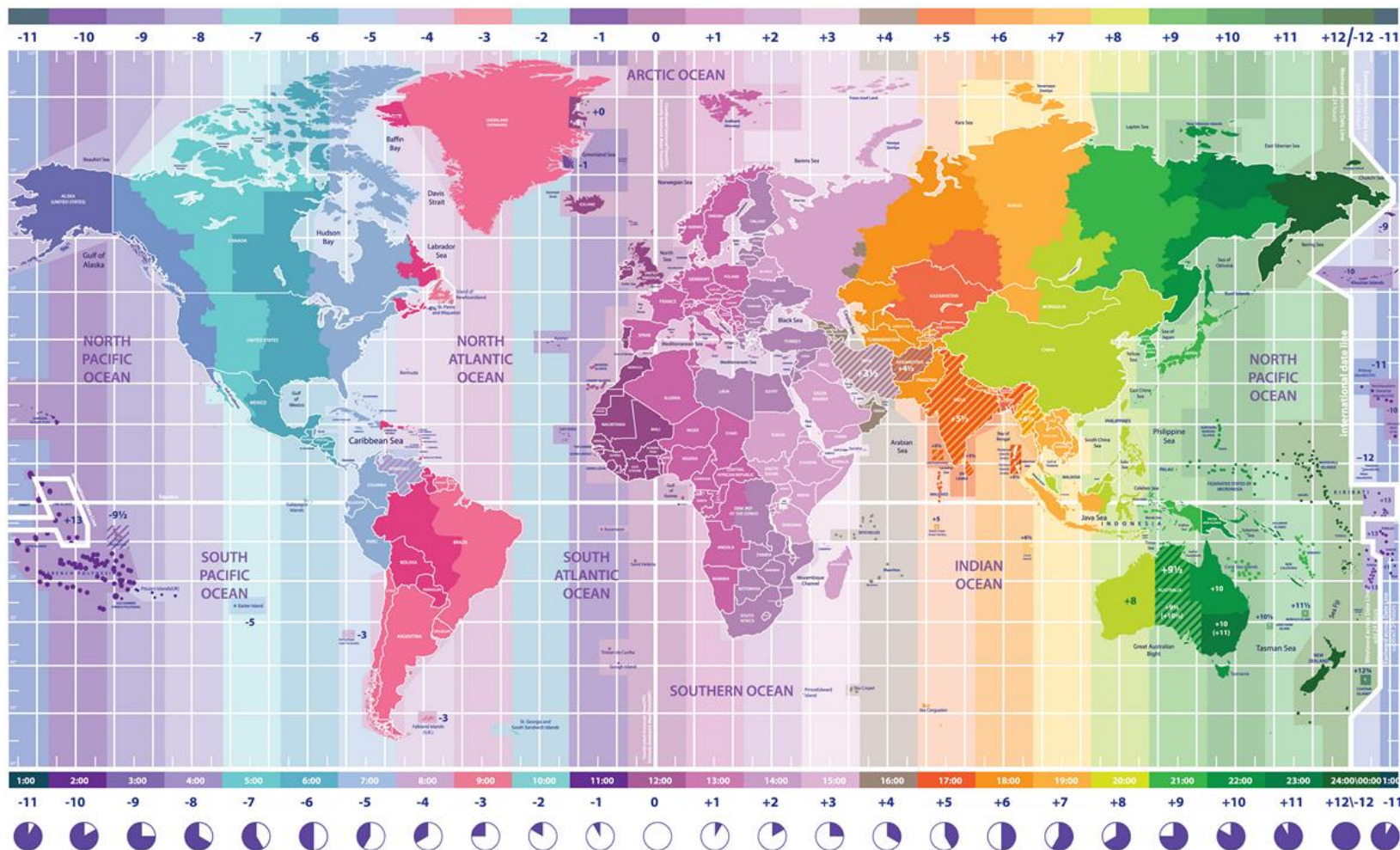
```
2
3     function calc_sum(a, b){
4         let result = a + b+1;
5         return result;
6     }
7
8     (function(){
9         let control = calc_sum(2, 3);
10
11         console.assert(control === 5, "TEST: calc_sum(2,3)");
12
13     })();
14
```

Метод **console.assert()** – удобный способ добавить вывод информации об ошибках в консоль разработчика.

Подробнее: <https://developer.mozilla.org/ru/docs/Web/API/Console/assert>

**Будет полезным**

# Работа с часовыми поясами в JavaScript



<https://habr.com/ru/company/mailru/blog/438286/>

## GET/SET методы у объекта

В составе объектов в JavaScript могут использоваться т.н. **геттеры** и **сеттеры** (**get** и **set** методы) – узнайте о них по подробнее.

**К следующему занятию будет  
полезно почитать о...**



# К следующему занятию...

1. Объект **Promise**;
2. Оператор **async**;
3. Оператор **await**;
4. Установите (если еще не установили) **Node.JS**

**Домашнее задание**  
**/сделать**

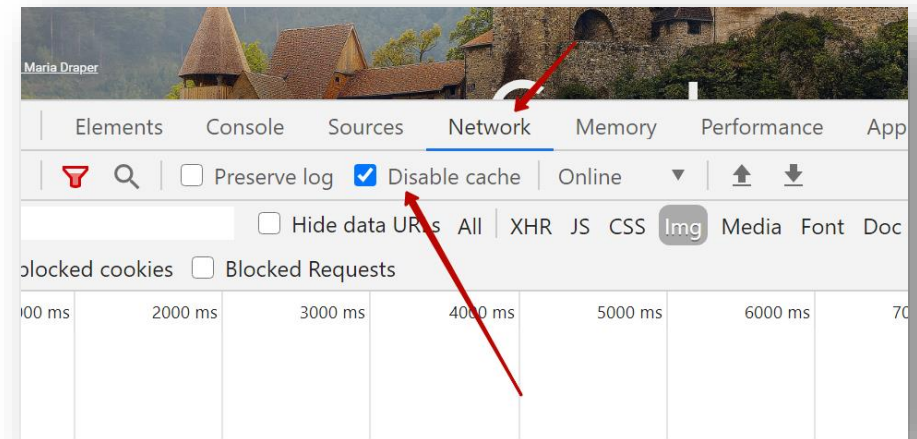
# Домашнее задание #Е.1

Воспользуйтесь API Национального Банка Украины и выведите в консоль последовательный список **курсов доллара** за период с **1 по 28 февраля 2021 г.** по дням.

*По такой структуре:*

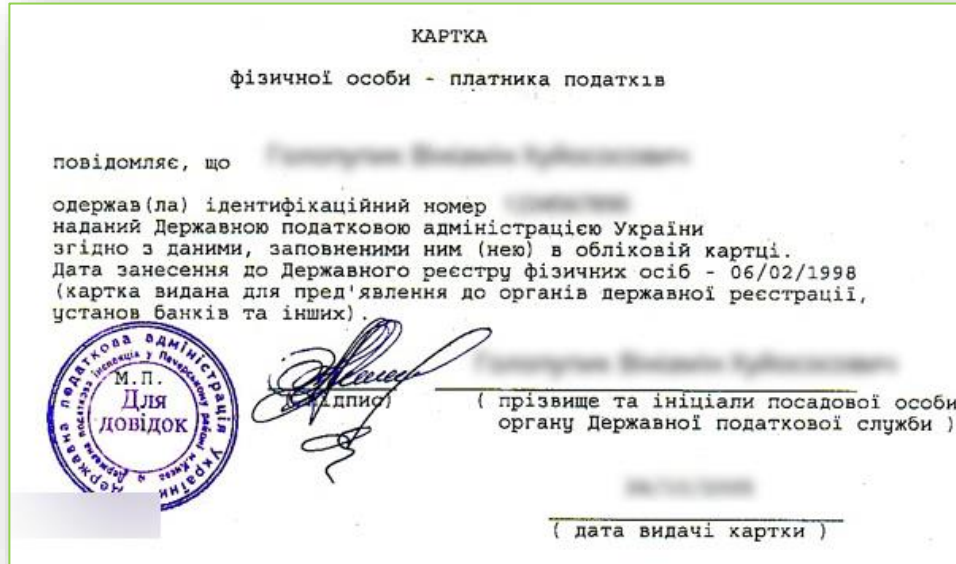
01.02.2021 – 24.56 грн.  
02.02.2021 – 24.86 грн.  
03.02.2021 – 25.01 грн.  
...  
27.02.2021 – 24.21 грн.  
28.02.2021 – 24.98 грн.

<https://bank.gov.ua/ua/open-data/api-dev>



*Не забудьте отключить кеширование, чтобы в процессе разработки браузер не путал вас данными из кеша.*

# Домашнее задание #Е.2 | «Проверка ИНН» v.2



Для проверки:

**3463463460** – пол женский, д.р. 28.10.1994;

**2063463479** – пол мужской, д.р. 29.06.1956.

Пользователь вводит ИНН (физ. лица Украины), Необходимо определить: **нет ли ошибки в коде**, узнать **дату рождения**, определить **пол** и сколько **полных лет** человеку.

Скрипт должен содержать **функцию**, которая принимает **ИНН** в виде строки (строка может содержать проблемы, необходимо отчистить её). По результатам работы функция должна возвращать объект следующей структуры (поля **sex**, **dateOfBirth** и **fullYears** для некорректного номера не создаются):

```
{  
  
  code: "1234567890",  
  isCorrect: true, //or false  
  sex: "female", //or "male"  
  dateOfBirth: "1988-12-23",  
  fullYears: 29  
  
}
```