Урок 1

Введение в React JS

Знакомство с ReactJS; сравнение с другими технологиями; современный JS; что нового в ES6 и как это использовать; классы, наследования, модули rest/spread, параметры, промисы, модули.

Что такое ReactJS, в чем его преимущества? Сравнение с другими технологиями

React.js — это фреймворк для создания интерфейсов. Был разработан Facebook как расширение к JS. Отвечает за представление данных, получение и обработку ввода пользователя. Своего рода создаёт всего лишь «вид» вашего приложения.

В React была реализована концепция MVC (Model-View-Controller: модель-вид-контроллер). Суть концепции в чётком разделении разрабатываемых приложений на три части со строго соблюдаемыми правилами и установленными задачами: Model (Обработка данных и логика приложения), View (Представление данных пользователю в любом поддерживаемом формате), Controller (Обработка запросов пользователя и вызов соответствующих ресурсов). Фреймворк работает на втором этапе.

Для справки, википедия нам дает следующее определение: фре́ймворк (иногда фреймво́рк; англицизм, неологизм от framework - каркас,структура) — программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта. Каркасный подход – подходит к построению программ, где любая конфигурация программы строится из двух частей:

1. Постоянная часть — каркас, не меняющийся от конфигурации к конфигурации и несущий в себе гнёзда, в которых размещается вторая,

2. Переменная часть;

3. Сменные модули (или точки расширения).

«Фреймворк» отличается от понятия библиотеки тем, что библиотека может быть использована в программном продукте просто как набор подпрограмм близкой функциональности, не влияя на архитектуру программного продукта и не накладывая на неё никаких ограничений. В то время как «фреймворк» диктует правила построения архитектуры приложения, задавая на начальном этапе разработки поведение по умолчанию — «каркас», который нужно будет расширять и изменять, согласно указанным требованиям. Пример программного фреймворка — C.M.F. (Content Management Framework), а пример библиотеки — модуль электронной почты.

React работает по принципу реактивного программирования, т.е. позволяет создавать динамические страницы с возможностью обновлений состояний, при этом страница легко может вернуться к предыдущему состоянию без обновления (примером может стать приложение типа «телефонный справочник» с возможностью установки фильтра, например, по фамилии).

Можно назвать разные варианты JS-фреймворков, реализующие модельную конструкцию: Angular (фреймворк от Google), Socket (одним из наиболее популярных инструментов для разработки on-line проектов, являясь event-driven фреймворком), Ember (современный JavaScript фреймворк), Meteor (full-stack фреймворк, который поможет на одном JS создать современное, хорошо масштабируемое приложение) и и.д. React не совсем корректно сравнивать с этими программами, т.к. каждая из этих программ охватывают всю концепцию mvc, react работает только на этапе представления. Он делает используемый код js понятным для html. Т.е. результат работы React — это HTML-код. При этом даёт колоссальные возможность для отрисовки состояний приложений или страниц.

Если оценивать react можно выделить следующие плюсы и минусы:

Плюсы:	**Минусы:**
Простой в работе. Отлично подходит для командной разработки, строгое соблюдение UI, и шаблона рабочего процесса. Быстрый (не замедляет работу приложения)	Все-таки его работа заключается просто в переводе js-кода в html, что не позволяет создавать полноценных динамических web-приложений
Вы всегда можете сказать, как ваш компонент будет отрисован, глядя на исходный код (те же возможности есть в Angulsr). Если известно состояние — то известен и результат отрисовки. Не нужно прослеживать ход выполнения программы. Удобно, когда разрабатывается сложное приложение	Не всегда понятная логика работы. Понять, как работают некоторые компоненты, и как они взаимодействуют, бывает непросто.
Возможно рендерить (визуализировать) React на сервере (Angulsr и пр. шаблоны предлагают использовать доп. программы или платные сервисы)	React не поддерживает браузеры от IE8 и младше. Необходимо использовать дополнительные приложения типа webpack, babel.

Современный JS. Что нового в ES6 и как это использовать

Язык Java Script стандартизируется организацией ECMA (European Computer Manufacturers Association), а сам стандарт носит название ECMAScript.

Стандарт определяет следующее:

- Синтаксис языка —ключевые слова, операторы, выражения и прочее;
- Типы — числа, строки, объекты и прочее;
- Правила наследования;
- Стандартные библиотеки, объекты и функций — JSON, Math, методы массивов, методы объектов.

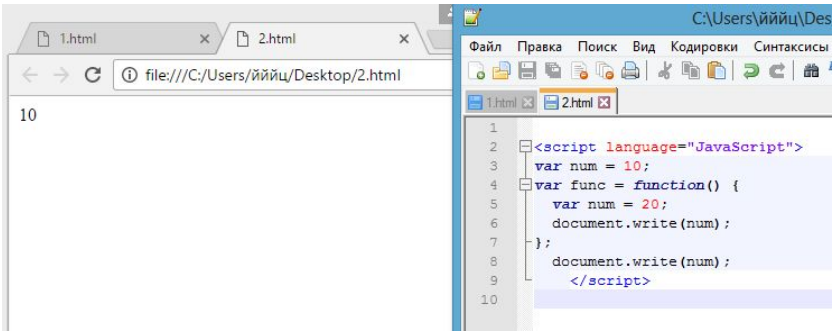
При этом стандарт не определяет всё, что связано с HTML и CSS, DOM.

ECMAScript 6 (ES6) — это новый стандарт JavaScript, принятый в 2015 году. Каждый стандарт должен быть внедрён в браузеры. На данный момент большинство новых возможностей ES6 имплементированы популярными браузерами (Firefox, Chrome и т.п.). Но, тем не менее, не все браузеры могут работать с ES6. Существуют инструменты, называемые трансляторами, конвертирующие код, написанный на ES6, в ES5-совместимый код. Например, ES6Fiddle (http://www.es6fiddle.net/) отличная и простая в использовании площадка, для того, чтобы попробовать ES6 и REPL для Babel(http://babeljs.io/repl/) - способен поддержать огромное количество фреймворков. Дальше разберёем некоторые различия этих двух стандартов.

Let, const, области видимости блока

Оператор let — альтернатива var в ES6. Этот оператор очень чётко даёт возможность разобраться с областью видимости. До введения стандарта ES6 основой всех областей видимости являлись функции. У любой функции существует своя область видимости. Например, что появится на экране при написании следующего кода программы:

```
<script language="JavaScript">
varnum = 10;
        var func = function() {
           varnum = 20;
           document.write(num);
                                };
           document.write(num);
</script>
```

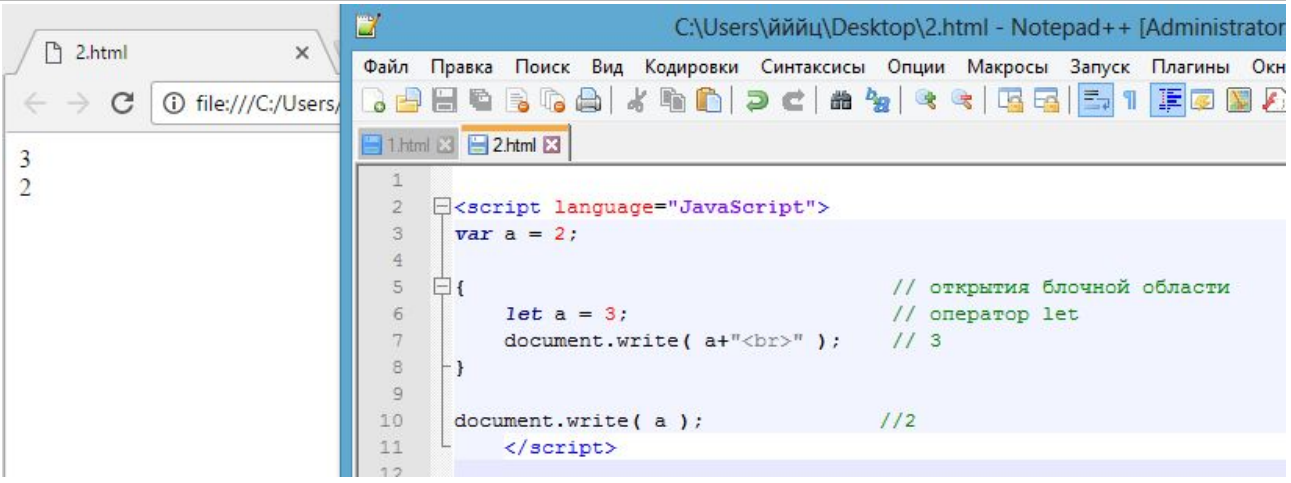


В данном примере наглядно видно, что значит область видимости функции, какие переменные программа «видит» внутри и вне этой области. Объявленные с помощью ключевого слова var переменные внутри функции не влияют на переменные из других областей видимости, в том числе и глобальной. Именно на этом свойстве основана хорошая практика «оборачивания» всего кода в самовызывающуюся анонимную функцию (self-executinganonymousfunction).

С ES6 возможна реализация области видимость в любом виде блока. Это называется блочной областью видимости (blockscoping), в js-коде она обозначается {} без использования функции. Оператор для обозначения переменной: let. При этом Let следует правилам блочной области видимости, а не области видимости, ограниченной функцией, т.е. виден только внутри блока и попытка вывода его из вне вызывает ошибку.

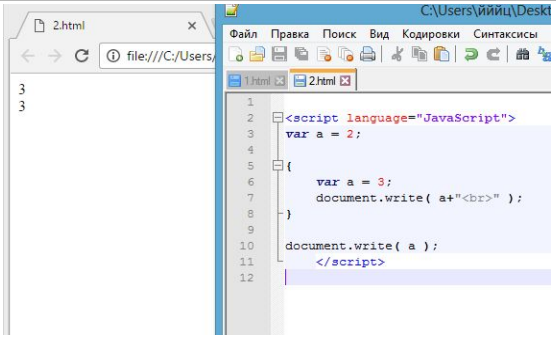
Для наглядности работы оператора let, посмотрим на работу следующего варианта кода:

```
<script language="JavaScript">
var a = 2;
{                                   // открытия блочной области
let a = 3;                          // оператор let
document.write( a+"<br>" );    // 3
}
document.write( a );                //2
</script>
```



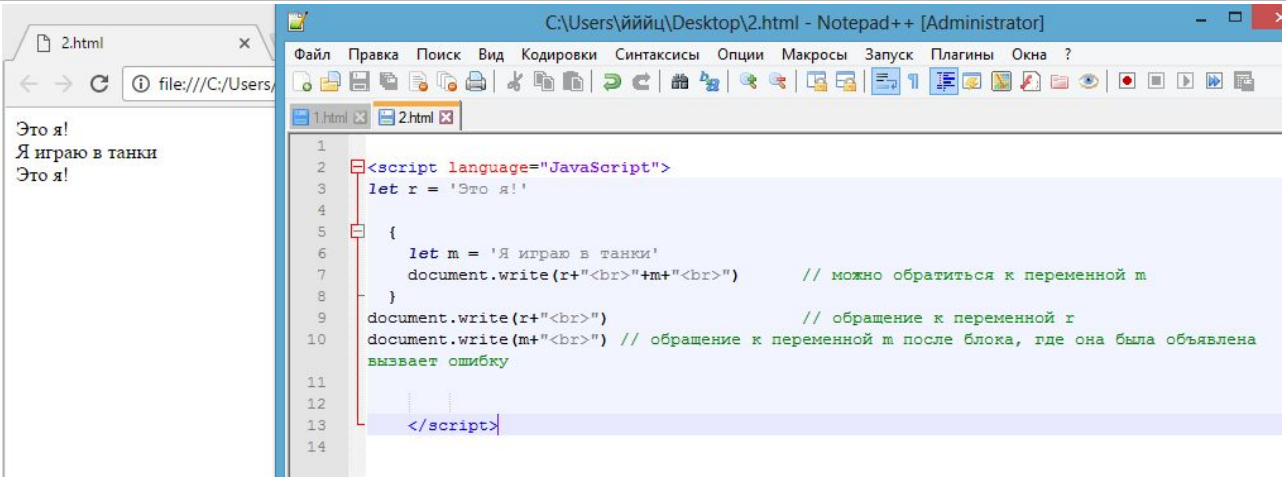
Заменим оператор внутри блока:

```
<script language="JavaScript">
var a = 2;
{
var a = 3;
document.write( a+"<br>" );
}
document.write( a );
</script>
```



С введением ES6 отпадает необходимость введения громоздкого синтаксиса с функциями для обозначения действия локальной или глобальной переменной. Но использование let создаёт определенную опасность в плане «видимости» переменной на протяжении всей программы. Например, у нас есть переменные: letr='Это я!' и let m ='Я играю в танки'. Каждая из них “работает” внутри своего блока.

```
letr='Это я!'
 {
  let m ='Я играю в танки'              // можно обратиться к переменной m и r
 }
// обращение к переменной m после блока, где она была объявлена вызывает ошибку
// возможно обращение только к переменной r
```



Есть ещё одна проблема с оператором let и блочной областью видимости.Оператор var своего рода «прикрепляет» переменную к области видимости и инициализирует её при входе в функцию ещё до её выполнения, вне зависимости от, того где она была объявлена. Данное явление известно как **поднятие.** Оператор let также прикрепляет переменную к области видимости до её выполнения, но не инициализирует её при входе в функцию, что в случае попытки обращения к такой переменной раньше, чем она будет

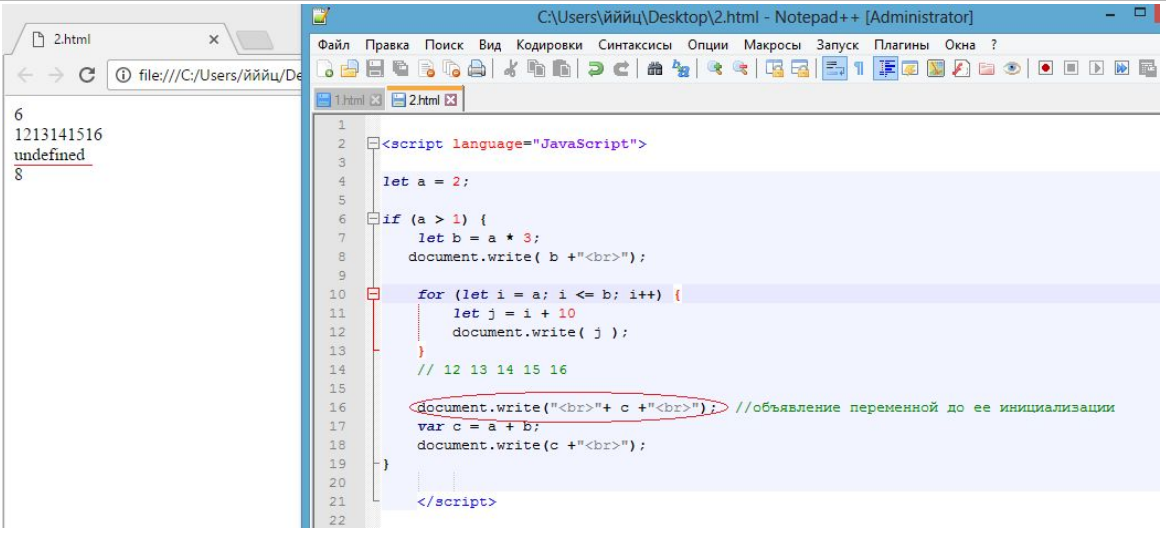
объявлена/инициализирована или вне области видимости, приведёт к ошибке - ошибка TDZ (temporaldeadzone).

Так функции, которые были объявлены в блоке в ES6, будет иметь область видимости (блочную) только внутри этого блока. До ES6, спецификация умалчивала по этому поводу, в отличие от различных конкретных реализаций. Тем не менее, сейчас спецификация соответствует реальности. И функция, объявленная в блочной области видимости, не доступна извне. При этом, в отличие от объявления let, с которым мы не можем работать до того, как такая переменная будет инициализирована, в данном случае ошибки TDZ не возникает.

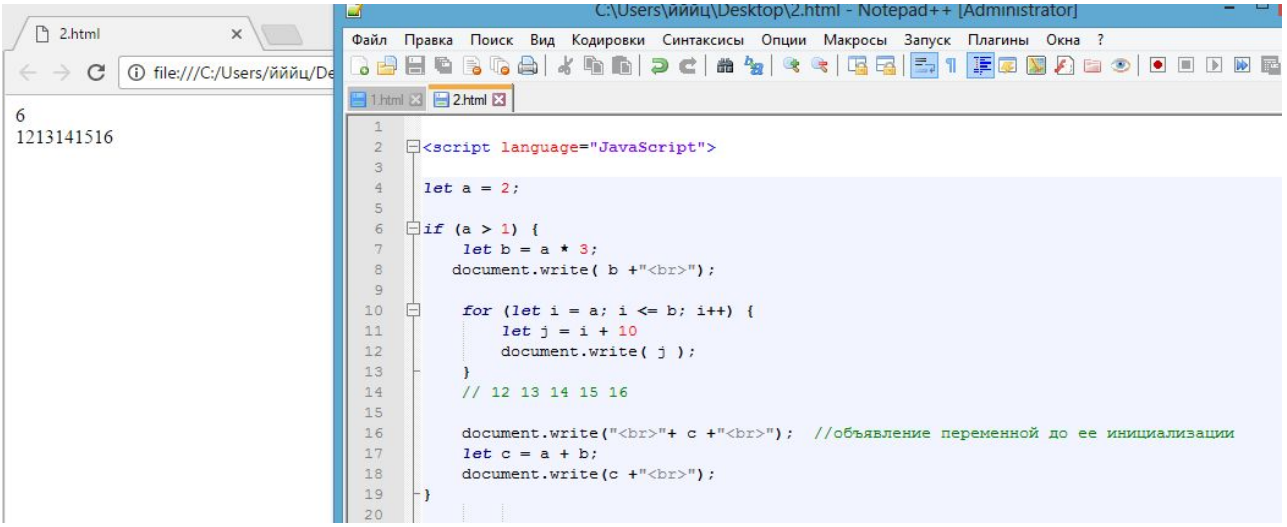
Сравним работу операторов на примере двух кодов:

```
<script language="JavaScript">
let a = 2;
if (a> 1) {
letb = a * 3;
document.write( b +"<br>");                // Область видимости для переменных a и b
          for (let i = a; i<= b; i++) {          // Область видимости i и j
       let j = i + 10
         document.write( j );
                              }                    // решение цикла: 12 13 14 15 16

document.write("<br>"+ c +"<br>");    // обращение к переменной до инициализации
var c = a + b;
document.write(c +"<br>");                // обращение к переменной после инициализации
}
</script>
```

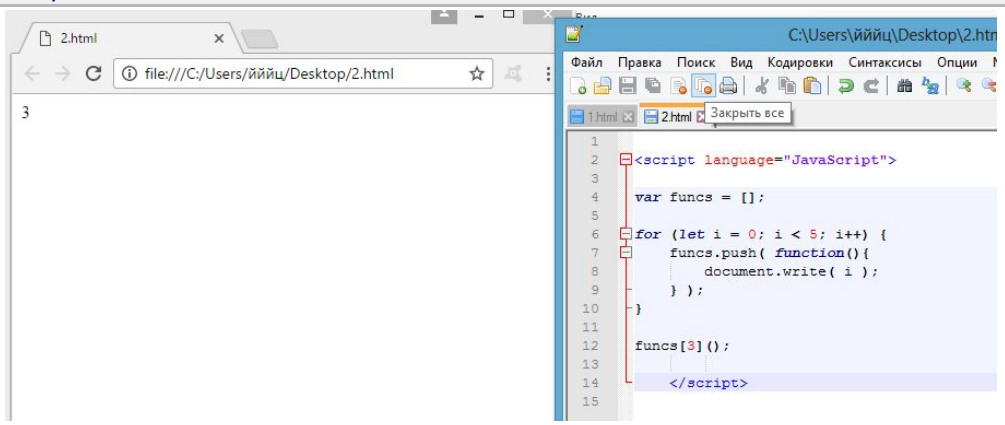


Замена оператора вызывает ошибку, работа программы остановлена (будьте внимательны):

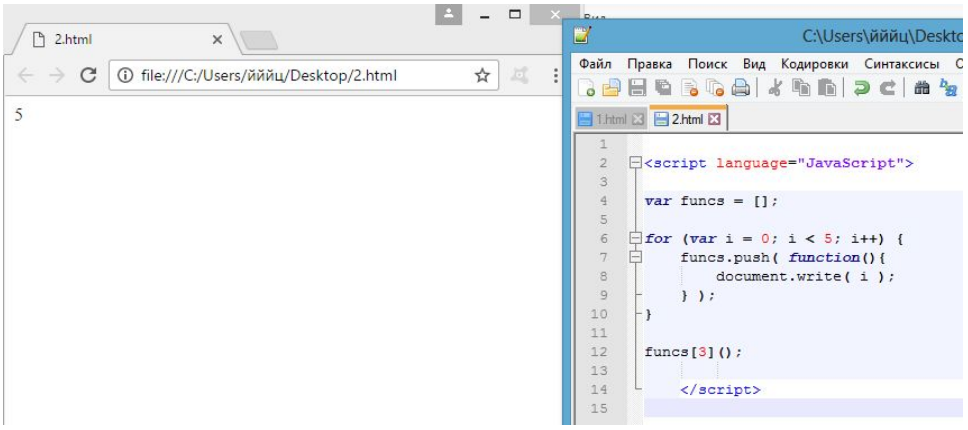


Тем не менее использование let упрощает работу с циклами. Выражение let в шапке цикла for объявляет новую переменную не для всего цикла, а для каждой отдельно взятой итерации. То же самое верно и для замыкания внутри цикла. По факту, мы можем обратиться вне цикла к любому из его итераций. Например:

```
<script language="JavaScript">
var funcs = [];
for (let i = 0; i< 5; i++) {
funcs.push( function(){
document.write( i );
   } );
}
funcs[3]();
</script>
```



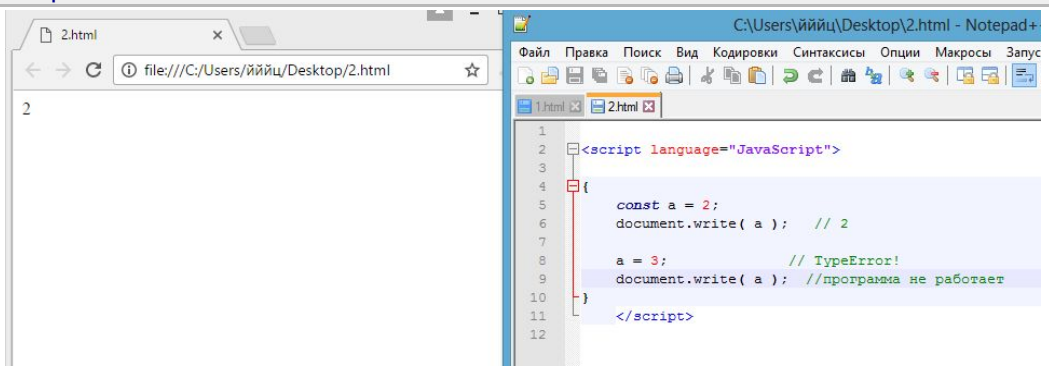
При использовании var:



Существует ещё один оператор, используемый для обозначения области видимости – const. Константа – эта такая переменная, которая после объявления недоступна для редактирования, при этом не имея ограничения на значение переменной. Например:

```
<script language="JavaScript">
{
const a = 2;
document.write( a );              // 2

   a = 3;                         // TypeError!
        document.write( a );      //программа не работает
}
</script>
```



Важно, что оператор константа не получает по умолчанию значения undefined. Для введения данного значения его необходимо задать: const a = undefined.

Деструктуризация, значения по умолчанию при деструктуризации

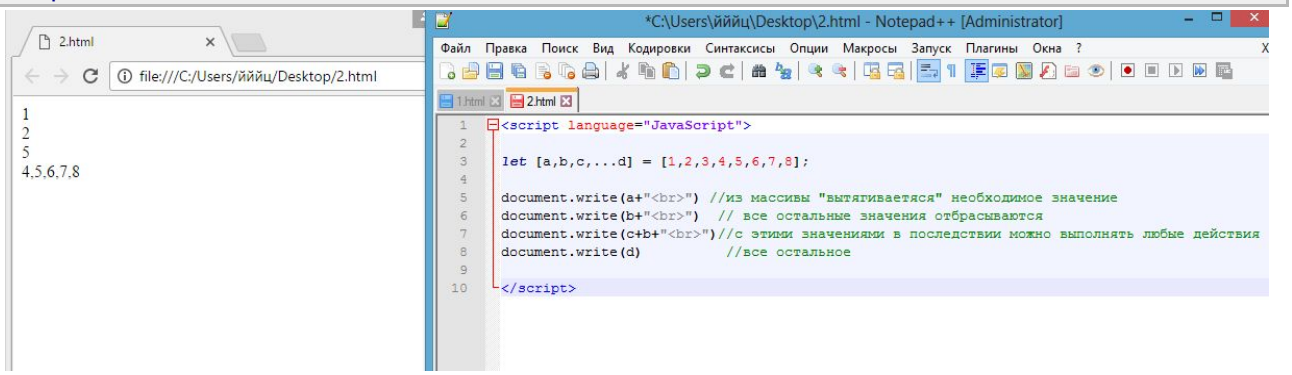
Деструктуризация – это особый синтаксис присваивания, при котором можно присвоить массиву или объект сразу нескольким переменным, разбив его на части.

Синтаксис:

```
let [a,b,c]=[1,2,3]
```

Например, зададим массив со значениями a=1, b=2, c=3. Необходимо вывести значение a, b и найти сумму c+a.

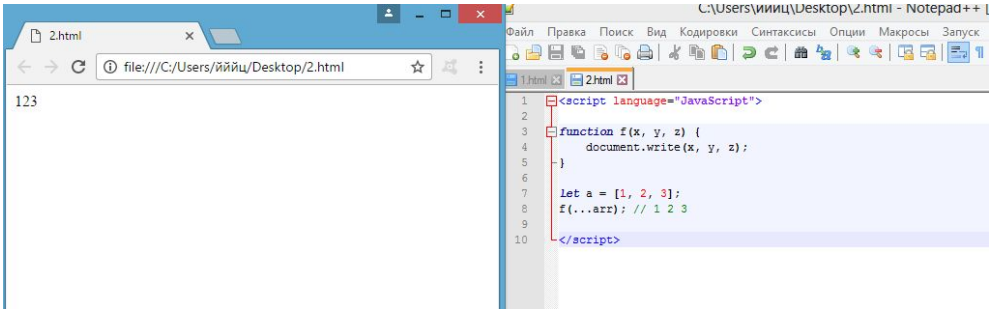
```
<script language="JavaScript">
let [a,b,c,...d] = [1,2,3,4,5,6,7,8];
document.write(a+"<br>")      //из массива "вытягивается" необходимое значение
document.write(b+"<br>")     // все остальные значения отбрасываются
document.write(c+b+"<br>"  //с этими значениями в последствии можно выполнять любые действия
document.write(d)              //все остальное
</script>
```



В данном примере появляется оператор «…» перед переменной, его можно обозначить как «всё остальное». Он должен стоять только последним элементом в списке слева. Само название данного оператора зависит от его расположения

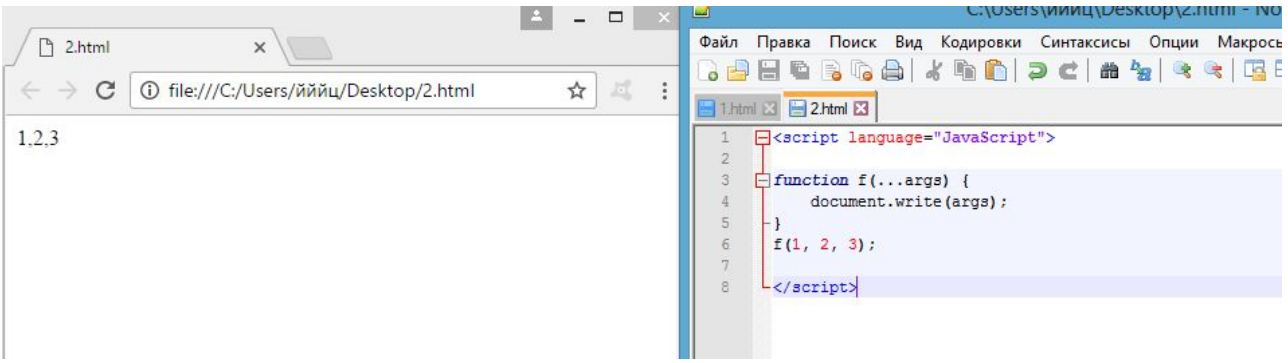
Spread – выполняет разделение объекта на составляющие (присваивает каждой компоненте массива значение оператора a)

```
<script language="JavaScript">
function f(x, y, z) {
    document.write(x, y, z);}
let a = [1, 2, 3];
f(...arr); // 1 2 3
</script>
```



Rest – объединяет объекты массива в один.

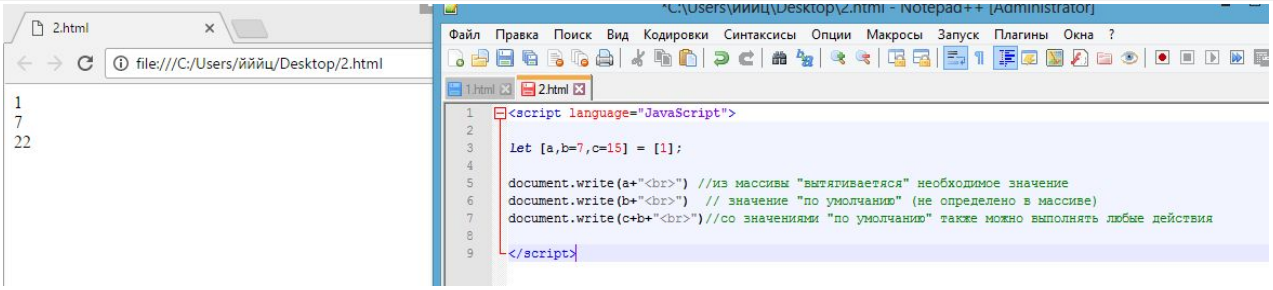
```
<script language="JavaScript">
function f(...args) {
document.write(args);
}
f(1, 2, 3);
</script>
```



Для синтаксиса деструктуризации часто используется значение по умолчанию.

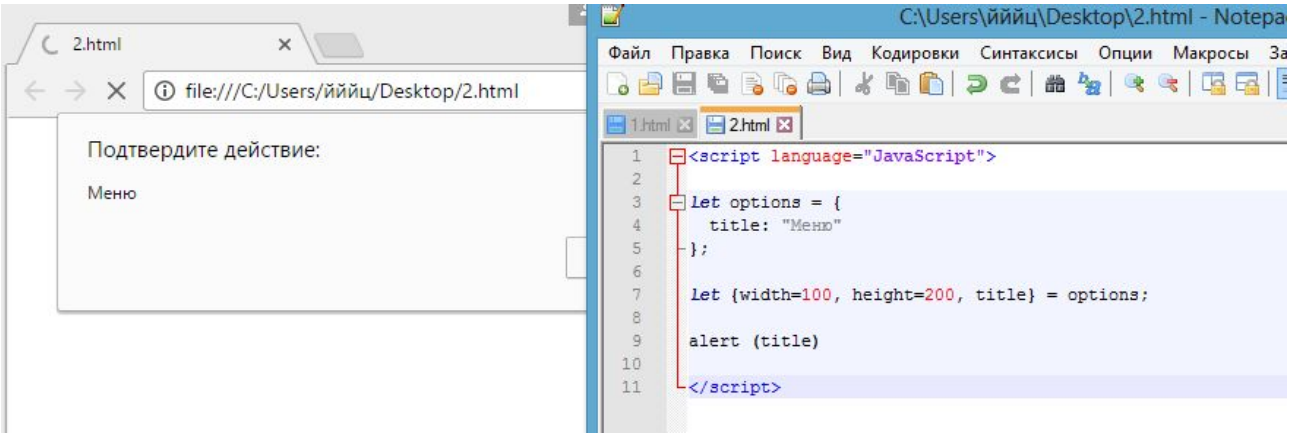
Значение «по умолчанию» вводится в том случае, если значений в массиве меньше, чем переменных. Синтаксис: let [a,b=”f”,c=”d”] = []. Например:

```
<script language="JavaScript">
let [a,b=7,c=15] = [1];
document.write(a+"<br>")     //из массивы "вытягиваетяся" необходимое значение
document.write(b+"<br>")    // значение "по умолчанию" (не определено в массиве)
document.write(c+b+"<br>")//со значениями "по умолчанию" также можно выполнять любые действия
</script>
```



Значение «по умолчанию» можно задать не только для массива, но и для любой используемой функции.
Для объектов используем оператор «:» описывающий свойства. Например возможно описать «меню» размером: width: 100, height: 200.

```
<script language="JavaScript">
let options = {
  title: "Меню"
};
let {width=100, height=200, title} = options;
alert (title)
</script>
```



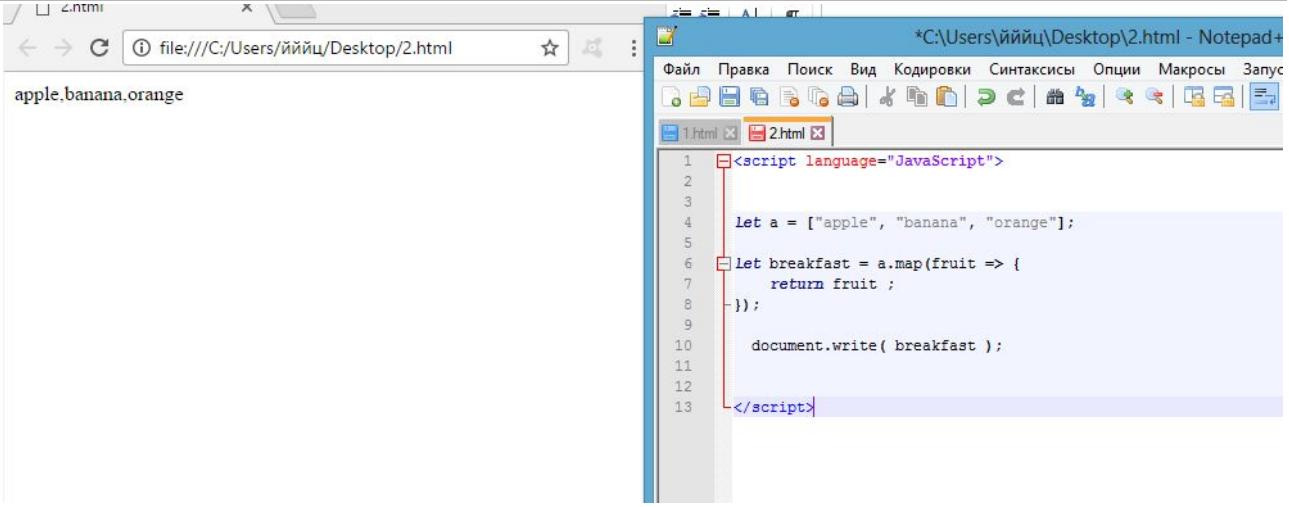
При этом значения параметров по умолчанию в ES6 вычисляются только тогда, когда действительно используются.

Arrow functions, отличие от обычных функций

Стрелочные функции представляют собой сокращённую запись функций в ES6.
Синтаксис:

```
let NameFunction= (Параметр 1, Параметр 2) => a + b;
```

```
<script language="JavaScript">
let a = ['apple', 'banana', 'orange'];
let breakfast = a.map(fruit => {
   return fruit ;
});
document.write( breakfast );
</script>
```



Использование стрелочных функций в определённых случаях делает код более красивым и лаконичным. В примере ниже один и тот же код оформлен в виде стрелочной функции и обычной функции ES5:

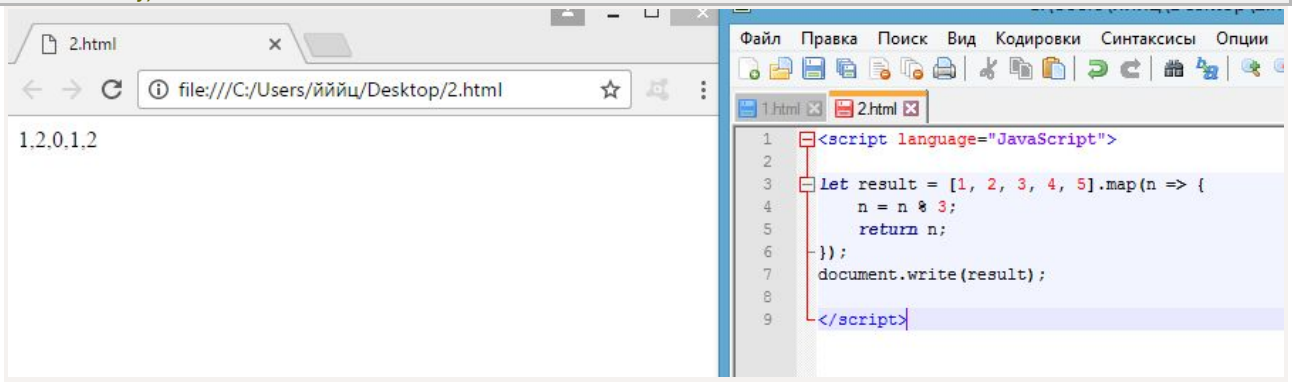
```
let books = [{title: 'X', price: 10}, {title: 'Y', price: 15}];
let titles = books.map( item =>item.title );
// ES5 equivalent:
var titles = books.map(function(item) {
   return item.title;});
```

В синтаксисе стрелочной функции нет ключевого слова function, сначала указывается список параметров, если необходимо, затем «толстая стрелка» и далее тело функции.

Замечание:

- Если параметров нет, указываются пустые круглые скобки: books.map(()=>1);
- Если тело функции содержит больше одного выражения, его необходимо поместить в фигурные скобки.

```
let result =[1,2,3,4,5].map(n =>{
                n = n %3;
    return n;});
```



Стрелочные функции отличаются от обычных не только более коротким синтаксисом. Стрелочная функция наследует значения this и arguments от окружающего контекста. Это означает, что вам не нужно писать уродливое var that = this, контекст выполнения будет правильным

Классы и наследование

ES6 вводит понятие «классов» для javascript. Класс ES6 не представляет собой новую объектно-ориентированную модель наследования. Класс в ES6 представляет собой просто новый синтаксис для работы с прототипами и функциями-конструкторами, которые мы привыкли использовать в ES5.

Для использования классов необходимо:

- Классы создаются в блочном коде ({и });
- Обозначается записью classN, означает, что будет создана функция-конструктор functionN (в ES5);
- Свойство constructor используется для обозначения того, что будет происходить непосредственно в самом конструкторе;
- При перечислении методов не надо использовать запятые (вне класса они запрещены);
- Для вызова функции класса необходимо использовать оператор new.

Пример синтаксиса класса в ES6:

```
class Task {
   constructor() {
document.write("Создан экземпляр task!");
   }
showId() {
document.write(23);
   }

   static loadAll() {
document.write("Загружаем все tasks...");
   }
}
document.write(typeof Task); // function
let task = new Task();           // "Создан экземпляр task!"
task.showId();                   // 23
Task.loadAll();                  // "Загружаем все tasks..."
```

Важно понимать, что мы до сих пор работаем с обычными функциями, т.е. если появляется необходимость проверить тип класса, то обнаруживается «function»

```
document.write(typeof Task);              // function
```

Как и при работе с функциями конструкторами мы можем записать «класс» (на самом деле, только конструктор) в переменную. Иногда подобная запись может быть чрезвычайно полезной, например, когда нужно записать конструктор, как свойство объекта.

```
constPerson=classP{/* делаем свои дела */}
constobj={
Person};
```

Объявления функций поднимаются: объявленные внутри общей области видимости, функции сразу же доступны, независимо от того, где они были объявлены. Это означает, что возможно вызвать функцию, которая будет объявлена позднее.

```
foo(); // работает, так как `foo` _поднялась_
        functionfoo() {}
```

В отличие от функций, определения классов не поднимаются. Таким образом, класс существует только после того, как его определение было достигнуто и выполнено. Попытка создания класса до этого момента приведет к «ReferenceError»:

```
new Foo(); // ReferenceError
classFoo{}
```

Причина этого ограничения в том, что классы могут быть наследниками. Это поддерживается с помощью выражения «extends», значение которого может быть

произвольным. Это выражение должно быть установлено в определённом месте, которое не может быть поднято.

С помощью наследования буквально можно объяснить программе: «У меня есть один конструктор/класс и другой конструктор/класс, который точно такой же, как и первый, кроме вот этого и вот этого». В JavaScript прототипное наследование «динамическое», можно изменять всё налету, классическое же наследование подобным похвастаться не может: всё, что вы объявили в одном классе, останется там навсегда. Грубо говоря, классическое представление наследования предполагает наличие определённой статической схемы, по которой будет строиться каждый объект данного класса. В прототипе наследования мы имеем дело не со схемой, а с живым, постоянно развивающимся организмом, который со временем изменяется и принимает ту форму, которая нужна.

Пример синтаксиса:

```
class Point {
   constructor(x, y) {
this.x = x;
this.y = y;
   }
toString() {
      return '(' + this.x + ', ' + this.y + ')';
   }
}
class ColorPoint extends Point {
   constructor(x, y, color) {
      super(x, y); // (A)
this.color = color;
   }
toString() {
      return super.toString() + ' in ' + this.color; // (B)
}
}
```

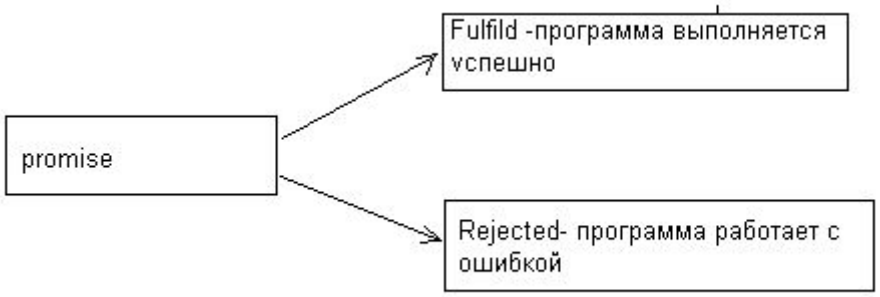
В ES6 ключевое слово extends позволяет классу-потомку наследовать от родительского класса. Важно отметить, что конструктор класса-потомка должен вызывать super().

Есть два способа использовать ключевое слово super:

- Конструктор класса (псевдо-метод «constructor» в теле класса), использует его как вызов функции (_super(•••)_), для того, чтобы вызвать базовый конструктор (строка A).
- Определения методов (в объектах, заданных через литерал, или классах, статических или нет), используют это для вызова свойства (_super.prop_), или вызова метода (_super.method(•••)_), для ссылки на свойства базового класса (строка B).

Промисы

В ES6 появилась встроенная поддержка промисов. Промисы(Promise) - это объекты, которые ждут выполнения асинхронной операции. После выполнения операции промис принимает одно из двух состояний: fulfilled (resolved, успешное выполнение) или rejected (выполнено с ошибкой), позволяя оценить работу кода.



На promise можно навешивать коллбэки двух типов:

- onFulfilled – срабатывают, когда promise в состоянии «выполнен успешно».
- onRejected – срабатывают, когда promise в состоянии «выполнен с ошибкой».

Способ использования:

1. Код, которому надо сделать что-то асинхронно, создает объект promise и возвращает его.
2. Внешний код, получив promise, навешивает на него обработчики.
3. По завершению процесса асинхронный код переводит promise в состояние fulfilled (с результатом) или rejected (с ошибкой). При этом автоматически вызываются соответствующие обработчики во внешнем коде.

Промисы являются альтернативой функциям обратного вызова, которые были стандартом JavaScript до ES6, помогают сделать код чище.

Синтаксис создания Promise:

```
var promise =newPromise(function(resolve, reject){
// Эта функция будет вызвана автоматически
// В ней можно делать любые асинхронные операции,
// А когда они завершатся — нужно вызвать одно из:
// resolve(результат) при успешном выполнении
// reject(ошибка) при ошибке
})
```

Универсальный метод для навешивания обработчиков:

```
promise.then(onFulfilled,onRejected)
```

- onFulfilled – функция, которая будет вызвана с результатом при resolve.
- onRejected – функция, которая будет вызвана с ошибкой при reject.

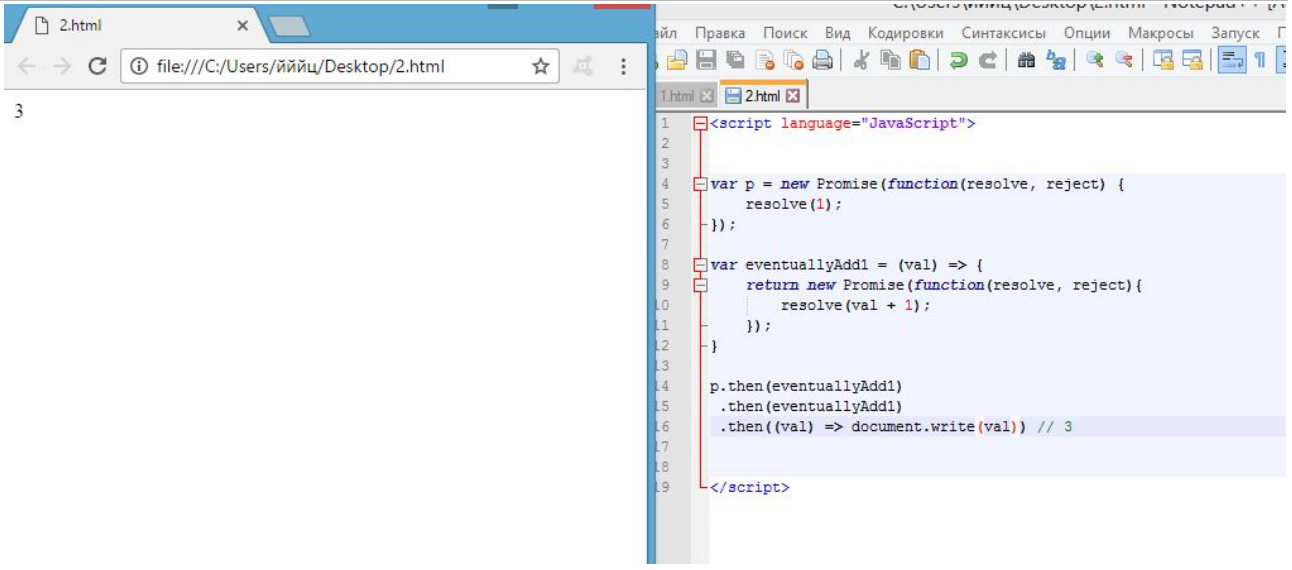
С его помощью можно назначить как оба обработчика сразу, так и только один.

Пример синтаксиса для промисов:

```
var p = newPromise(function(resolve, reject) {
if (/* условие */) {
       resolve(/* значение */);  // fulfilled successfully (успешный результат)
   } else {
reject(/* reason */);               // rejected (ошибка)
   }
});
```

При возвращении промиса успешно обработанное значение промиса пройдёт к следующему коллбэку, для того, чтобы эффективно соединить их вместе.

```
var p = new Promise(function(resolve, reject) {
   resolve(1);
});
var eventuallyAdd1 = (val) => {
   return new Promise(function(resolve, reject){
      resolve(val + 1);
   });
}
p.then(eventuallyAdd1)
 .then(eventuallyAdd1)
 .then((val) =>document.write(val)) // 3
```



Модули

Ни один серьезный JavaScript-проект сегодня не обходится без какой-либо модульной системы — это может быть просто «шаблон модуль» или такие форматы, как AMD или CommonJS. Тем не менее, браузеры до недавнего времени не располагали какой-то модульной системой, т.е. необходимо было настроить сборку или загрузчик модулей AMD или CommonJS с помощью RequireJS, Browserify или Webpack.

Модулем считается файл с кодом. В этом файле ключевым словом export помечаются переменные и функции, которые могут быть использованы снаружи. Другие модули могут подключать их через вызов import. Спецификация ES6 содержит и новый синтаксис, и механизм загрузки модулей:

```
// lib/math.js
export functionsum(x, y){
return x + y;
}
export var pi =3.141593;
// app.js
import{ sum, pi }from"lib/math";
console.log('2π = '+sum(pi, pi));
```

Модуль может содержать несколько выражений export (в примере выше экспортируется функция и переменная). Выражение import в примере имеет синтаксис, схожий с деструктуризацией — в данном случае явно указывается, что именно импортируется из модуля. Чтобы импортировать весь модуль, можно использовать символ *, а ключевое слово as позволяет задать локальное имя модуля:

```
// app.js
import*as math from"lib/math";
console.log('2π = '+math.sum(math.pi,math.pi));
```

Особенностью модульной системы является экспорт по умолчанию. Для этого используется ключевое слово default. Чтобы импортировать значение по умолчанию, достаточно указать локальное имя:

```
// lib/my-fn.js
exportdefaultfunction(){
  console.log('echo echo');
}
// app.js
importdoSomethingfrom'lib/my-fn';
doSomething();
```

Обратите внимание, что выражение import является синхронным, т.е. код модуля не выполнится, пока не загрузятся все зависимости.

Современный стандарт ECMAScript описывает, как импортировать и экспортировать значения из модулей, но он ничего не говорит о том, как эти модули искать, загружать и т.п. Такие механизмы предлагались в процессе создания стандарта, но были убраны по причине недостаточной проработанности. Возможно, они появятся в будущем. Сейчас используются системы сборки, как правило, в сочетании с Babel.JS.

Система сборки обрабатывает скрипты, находит в них import/export и заменяет их на свои внутренние JavaScript-вызовы. При этом, как правило, много файлов-модулей объединяются в один или несколько скриптов, смотря как указано в конфигурации сборки. Модули легче всего использовать с приложением webpack, которое позволит следующее:

- nums.js – модуль, экспортирующий one и two, как описано выше.
- main.js – модуль, который импортирует one, two из nums и выводит их сумму.
- webpack.config.js – конфигурация для системы сборки.
- bundle.js – файл, который создала система сборки из main.js и nums.js.

Практика

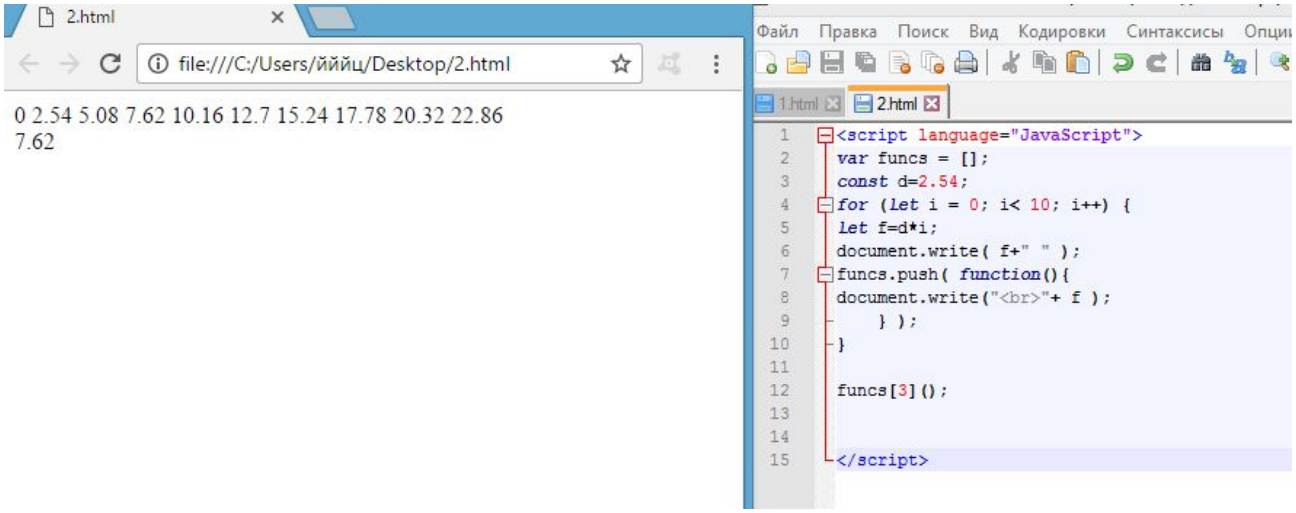
Задача 1.

Написать программу, для демонстрации работы операторов var, let, const и их области видимости. Программа будет переводить дюймы в сантиметры. Константа 1duim=2.54. Вывести значения от 0 до 10.

```
<script language="JavaScript">
var funcs = [];
const d=2.54;
for (let i = 0; i< 10; i++) {
let f=d*i;
document.write( f+" " );
funcs.push( function(){
document.write("<br>"+ f );
    } );
}
funcs[3]();
</script>
```

Вопросы:

1) Обратимся к оператору let вне блока. Что произойдёт?
2) Обратимся к оператору var вне блока. Что произойдёт?
3) Описать область видимости константы.
4) Вызвать значение равное 5 дюймам.



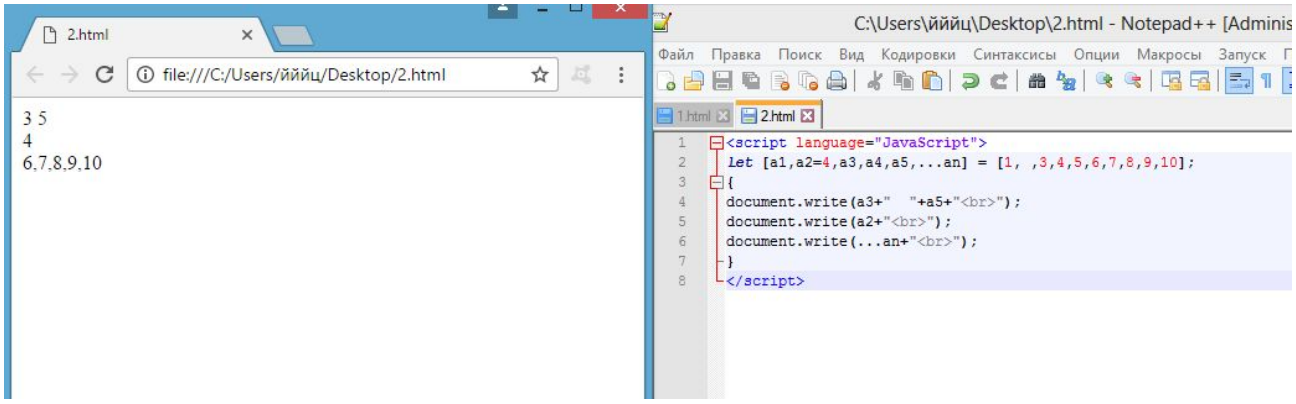
Задача 2.

Дана последовательность чисел An от 0 до 10. Вывести значения.

```
<script language="JavaScript">
let [a1,a2=4,a3,a4,a5,...an] = [1, ,3,4,5,6,7,8,9,10];
{
document.write(a3+"  "+a5+"<br>");
document.write(a2+"<br>");
document.write(...an+"<br>");
}
</script>
```

Вопросы:

1) Вывести значение A3, A5 .
2) Задать значение A2 по умолчанию 4.
3) Вывести все значения начиная с 6.



Задача 3.

Создать стрелочную функцию avg() , которая будет находить среднее значение по всем своим аргументам (аргументы величины числовые).

```
<script>                              // В ES5 эта задача выглядит так
var arr=[1,2,3];
var mean=arr.reduce(function(a, b)
          { return a+b; })/arr.length;
            document.write(mean);  // 2
</script>
```

В ES6 эта задача будет решаться с использованием стрелочной функции

```
<script>            // В ES6 эта задача выглядит так
var arr=[1,2,3];
var mean=arr.reduce((a, b) =>                //function отсутствует
          { return a+b; })/arr.length;

            document.write(mean);    //2
</script>
```

Вопросы:

1) Выполнить то же задание в ES5.

2) Обозначит преимущества стрелочной функции.

Задача 4.

Написать 2 класса с возможностью наследования.

```
<script language="JavaScript">
class Animal {
  constructor(name) {
    this.name = name;
  }
  walk() {
    alert("I walk: " + this.name);
  }
}
class Rabbit extends Animal {
  walk() {
    super.walk();
    alert("...and jump!");
  }
}
new Rabbit("Вася").walk();
// I walk: Вася
// and jump!
</script>
```

Задача 5.

Продемонстрировать работу промисов

```
let promise = new Promise((resolve, reject) => {
                                        // reject вызван раньше, resolve будет проигнорирован
  setTimeout(() => reject(new Error("error")), 1000);
  setTimeout(() => resolve("ignored"), 2000);
});
promise
  .then(
    result => alert("Fulfilled: " + result), // не сработает
    error => alert("Rejected: " + error)  // сработает
  );
```

Задача 6.

Демонстрация работы шаблона модуля.

```
var myObj = (function(){
    // определяем частную (private) переменную
    var myName = "Вася";
    // возвращается объект, содержащий общедоступные члены модуля
    return {
        getName: function(){ return myName; }
    }
}());
myObj.getName();  // Вася
```

Задача 7.

Шаблон модуль через функцию-конструктор.

```
var myObj = (function(){            // определяем частную(private) переменную
   var myName = "Вася";              // конструктор
   var Constr = function(n){
      if(n === undefined)
      {
         this.myName = myName;
      }
      else
      {
         this.myName = n;
      }
   }                                  // возвращается конструктор
   return Constr;
})();
var myObj_ex = new myObj();
console.log(myObj_ex.myName) // Вася
var myObjEx = new myObj("Петя");
console.log(myObjEx.myName)  // Петя
```

Домашнее задание

1. Написать функцию loop, которая будет принимать параметры: times (значение по умолчанию = 0), callback (значение по умолчанию = null) и будет в цикле (times раз), вызывать функцию callback. Если функцию не передана, то цикл не должен отрабатывать ни разу. Покажите применение этой функции
2. Написать функцию calculateArea, которая будет принимать параметры, для вычисления площади (можете выбрать какую то конкретную фигуру, а можете, основываясь на переданных параметрах, выполнять требуемый алгоритм вычисления площади для переданной в параметрах фигуры) и возвращать объект вида: { area, figure, input }, где area - вычисленная площадь, figure - название фигуры, для которой вычислялась площадь, input - входные параметры, по которым было произведено вычисление.
3. Необходимо написать иерархию классов вида:

   Human -> Employee -> Developer

   Human -> Employee -> Manager

   Каждый Менеджер (Manager) должен иметь внутренний массив своих сотрудников (разработчиков), а также методы по удалению/добавлению разработчиков.

   Каждый Разработчик (Developer) должны иметь ссылку на Менеджера и методы для изменения менеджера (имеется ввиду возможность назначить другого менеджера).

   У класса Human должны быть следующие параметры: name (строка), age (число), dateOfBirth (строка или дата)

   У класса Employee должны присутствовать параметры: salary (число), department (строка)

   В классе Human должен присутствовать метод displayInfo, который возвращает строку со всеми параметрами экземпляра Human.

В классе Employee должен быть реализовать такой же метод (displayInfo), который вызывает базовый метод и дополняет его параметрами из экземпляра Employee

Чтобы вызвать метод базового класса, необходимо внутри вызова метода displayInfo класса Employee написать: super.displayInfo(), это вызовет метод disaplyInfo класс Human и вернет строку с параметрами Human'a.

4*. При помощи генератора написать функцию - анкету, которая запрашивает у пользователя на ввод параметры и передаёт их в генератор. В конце, когда генератор завершается, он должен вернуть все введённые входные параметры в виде объекта. Этот объект нужно вывести в консоли.

5*. Написать цикл, который создаёт массив промисов, внутри каждого промиса происходит обращение к ресурсу (https://jsonplaceholder.typicode.com/users/number), где вместо number подставляется число от 1 до 10, в итоге должно получиться 10 промисов. Следует дождаться выполнения загрузки всеми промисами и далее вывести массив загруженных данных

Задачи со * являются заданиями повышенной сложности.

Дополнительные материалы

1. http://www.es6fiddle.net
2. http://babeljs.io/repl/
3. https://babeljs.io/
4. https://facebook.github.io/react/
5. https://habrahabr.ru/post/257005/
6. http://getinstance.info/news/new-features-in-es6/
7. http://getinstance.info/news/new-features-in-es6-2/
8. http://getinstance.info/news/new-features-in-es6-3/
9. https://jsfiddle.net/tw8azarg/
10. http://frontender.info/es6-in-depth-generators/
11. http://frontender.info/es6-in-depth-generators-continued/
12. https://habrahabr.ru/post/210330/
13. https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise
14. https://habrahabr.ru/post/242767/
15. https://jsfiddle.net/wh0vgsnj/
16. https://habrahabr.ru/post/282477/

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. wikipedia.org
2. www.jsraccoon.ru
3. www.habrahabr.ru
4. https://babeljs.io/
5. http://www.es6fiddle.net
6. http://www.es6fiddle.net
7. http://babeljs.io/repl/
8. https://babeljs.io/
9. https://facebook.github.io/react/
10. https://habrahabr.ru/post/257005/

11. http://getinstance.info/news/new-features-in-es6/
12. http://getinstance.info/news/new-features-in-es6-2/
13. http://getinstance.info/news/new-features-in-es6-3/