Урок 1

Современный JavaScript

Знакомство со стандартом ES2015 и некоторыми его возможностями

Стандарты JavaScript

Объявление переменных

Деструктуризация массивов и объектов

Функции

Шаблонные строки

Практика

Практическое задание

Дополнительные материалы

Используемая литература

Стандарты JavaScript

В 1995 году компания Netscape выпустила JavaScript – язык, который любители могли бы использовать для написания простых скриптов. В 1996 году ассоциация ECMA International начала работу над стандартизацией языка. К тому времени, согласно пресс-релизу самой Netscape, JavaScript уже использовался на 300 тысячах страниц. В июне 1997 года рабочая группа выпустила первую версию стандарта. Использовать имя JavaScript они не смогли из-за проблем с авторскими правами, поэтому стандарт назвали ECMAScript, или сокращённо ES.

Стандарт позволил сообществу вносить предложения по улучшению языка. Сначала спецификации выходили каждый год: ES1 вышла в 1997 году, ES2 – в 1998, ES3 – в 1999. После этого был ряд неудачных попыток выпустить четвёртую версию, но она так и не вышла. ES5, вышедшая в 2009 году, стала последней номерной частью спецификации. Начиная с шестой версии, ECMA решили обновлять стандарт строго раз в год, поэтому в обозначении теперь указывается год.

Спецификация ES2015 принесла в язык много новых функций. К сожалению, не все они поддерживаются браузерами. По состоянию на сентябрь 2018 года ни один браузер не покрывает 100% функций, описанных в стандарте. Однако основные можно применять свободно. Часть из них мы рассмотрим в этом уроке.

Объявление переменных

До ES2015 для объявления переменной использовалась конструкция

```
var a = 1;
```

Область видимости переменной, объявленной через **var**, ограничена функцией. Это значит, что в ситуации

```
function a() {
  var first = 2;
}
function b() {
  var second = 1;
}
```

переменная **first** будет видна только в пределах функции **a()**, а переменная **second** – только в пределах функции **b()**. Невозможно получить доступ к переменной **first** из функции **b()** и наоборот. При этом в примере

```
function a() {
  if (true) {
    var first = 1;
  }
}
```

переменная **first** будет видна в любом месте функции **a()**, потому что условие – не функция.

Переменные, объявленные через **var**, умеют «всплывать», т. е. существуют в пределах всей области видимости. Объявим функцию

```
function a() {
  var first = 1;
}
```

А теперь попробуем вывести в консоль значение переменной **first**. Причём сделаем это до объявления самой переменной.

```
function a() {
  console.log(first);
  var first = 1;
}
```

Казалось бы, мы должны получить ошибку, потому что обратились к переменной, которая ещё не существует. Но на деле в консоль выведется значение **undefined**, как будто переменная есть, но у неё нет значения. По сути, это так и есть. Такая запись равносильна следующей:

```
function a() {
  var first;
  console.log(first);
  first = 1;
}
```

То есть на момент вывода значения переменной **first** в консоль она уже существует, но пока пуста.

Такое поведение чревато проблемами, поэтому потребовался механизм, позволяющий лучше контролировать работу переменных. Так появились **let** и **const**.

Первое отличие переменных, объявленных через **let** и **const**, – блочная область видимости. Теперь переменные видны не в пределах функции, а в пределах блока, ограниченного фигурными скобками.

```
function a() {
  if (v > 10) {
    let k = 0;
  } else {
    let k = 10;
  }
}
```

Здесь **k** в первом и втором блоке – абсолютно разные переменные. Их можно менять в пределах блока, не боясь сломать что-нибудь снаружи.

Второе отличие – переменные, объявленные через **let** и **const**, не «всплывают». Если мы возьмём нашу старую функцию и объявим переменную через **let**…

```
function a() {
```

```
  console.log(first);
  let first = 1;
}
```

…получим ошибку «first is not defined», потому что в момент обращения к переменной **first** её ещё не существует.

Переменные, объявленные через **let** и **const**, отличаются тем, что **const** определяет константу, т. е. неизменяемую переменную. Если мы попробуем записать в неё новое значение, получим ошибку:

```
function a() {
  let first = 1;
  const second = 2;

  first = 5;  // Всё в порядке
  second = 5; // Ошибка «Assignment to constant variable»
}
```

Деструктуризация массивов и объектов

В ES2015 появился способ удобно записывать в разные переменные элементы массива или объекта. Такой синтаксис называется деструктуризацией.

Возьмём массив, состоящий из двух элементов:

```
let array = ['first', 'second'];
```

Теперь запишем в одну переменную первый элемент массива, а в другую – второй. Раньше нам пришлось бы обращаться к элементам массива через индексы. Теперь мы можем использовать деструктуризацию:

```
let array = ['first', 'second'];
let [a, b] = array;
console.log(a); // first
console.log(b); // second
```

Похожий механизм работает и для объектов:

```
let obj = {
  a: 'first',
  b: 'second'
}
let {a, b} = obj;
console.log(a); // first
console.log(b); // second
```

Если попытаться присвоить нашей переменной несуществующее значение (например, переменной, которая создана, но еще не инициализирована), то ошибки не будет. В переменную запишется **undefined.**

```
let array = [];
let object = {};
let [a] = array;
let {b} = object;
console.log(a); // undefined
console.log(b); // undefined
```

При деструктуризации объектов можно изменять имя переменной, в которую записывается значение. Например, есть такой объект:

```
let object = {
  first: 'foo',
  second: 'bar'
}
```

Нужно записать значение поля **first** в переменную **first**, а значение поля **second** – в переменную **s**. Для этого укажем новое имя переменной через двоеточие:

```
let object = {
  first: 'foo',
  second: 'bar'
}
let {first, second: s} = object;
console.log(first); // foo
console.log(s);     // bar
```

Деструктуризации можно комбинировать и вкладывать друг в друга. Такой объект

```
let user = {
  params: {
    firstname: 'John',
    lastname: 'Smith'
  },
  goods: ['Book', 'Phone']
}
```

можно разобрать на переменные одной строкой:

```
let user = {
  params: {
    firstname: 'John',
    lastname: 'Smith'
  },
  goods: ['Book', 'Phone']
}
let {params: {firstname, lastname}, goods: [good1, good2]} = user;
console.log(firstname); // John
console.log(lastname);  // Smith
console.log(good1);     // Book
console.log(good2);     // Phone
```

Функции

В стандарте ES2015 появилась возможность использовать стрелочные функции. До появления стандарта мы объявляли функции так:

```
var f = function(number) {
  return number + 1;
}
```

Теперь мы можем использовать такую запись:

```
let f = (number) => {
  return number + 1;
}
```

Если параметр всего один, как в нашем примере, скобки можно убрать:

```
let f = number => {
  return number + 1;
}
```

Однако если параметров нет совсем, пустые скобки ставить нужно:

```
let f = () => {
  doSomething;
}
```

Иногда функция только возвращает значение:

```
let f = number => {
  return number + 1;
}
```

В таком случае можно сократить запись, убрав фигурные скобки и команду **return**:

```
let f = number => number + 1;
```

Также стандарт ES2015 позволяет устанавливать значения по умолчанию для параметров функции:

```
const f = (param = 5) => {
  console.log(param);
}
f();    // 5
f(10);  // 10
```

У стрелочных функций есть ещё одно важное отличие от обычных: они не создают собственного контекста и наследуют контекст родителя. Создадим объект, одно из полей которого сделаем функцией:

```
const user = {
  name: 'John',
  greet: function() {
    console.log('Hello ' + this.name);
  }
}
```

Если вызвать **user.greet()**, то в консоли появится «Hello John». В функции **greet** ключевое слово **this** указывает на контекст выполнения функции, т. е. на сам объект **user**. Поэтому запись **this.name** позволяет получить значение поля **name** объекта **user**.

Теперь заменим функцию на стрелочную:

```
const user = {
  name: 'John',
  greet: () => {
    console.log('Hello ' + this.name);
  }
}
```

Стрелочная функция не создает контекста, поэтому **this**, находящийся в ней, указывает на контекст родителя и именно в нём будет искать поле **name**. Если это поле не найдётся, значение будет **undefined**. Поэтому стрелочные функции надо использовать осмотрительно.

Шаблонные строки

Раньше, чтобы вставить в строку значение переменной, приходилось складывать несколько значений:

```
var name = 'John';
var greet = 'Hello, ' + name + '. How are you?';
console.log(greet); // Hello, John. How are you?
```

В ES2015 появилась более удобная запись с помощью шаблонных строк:

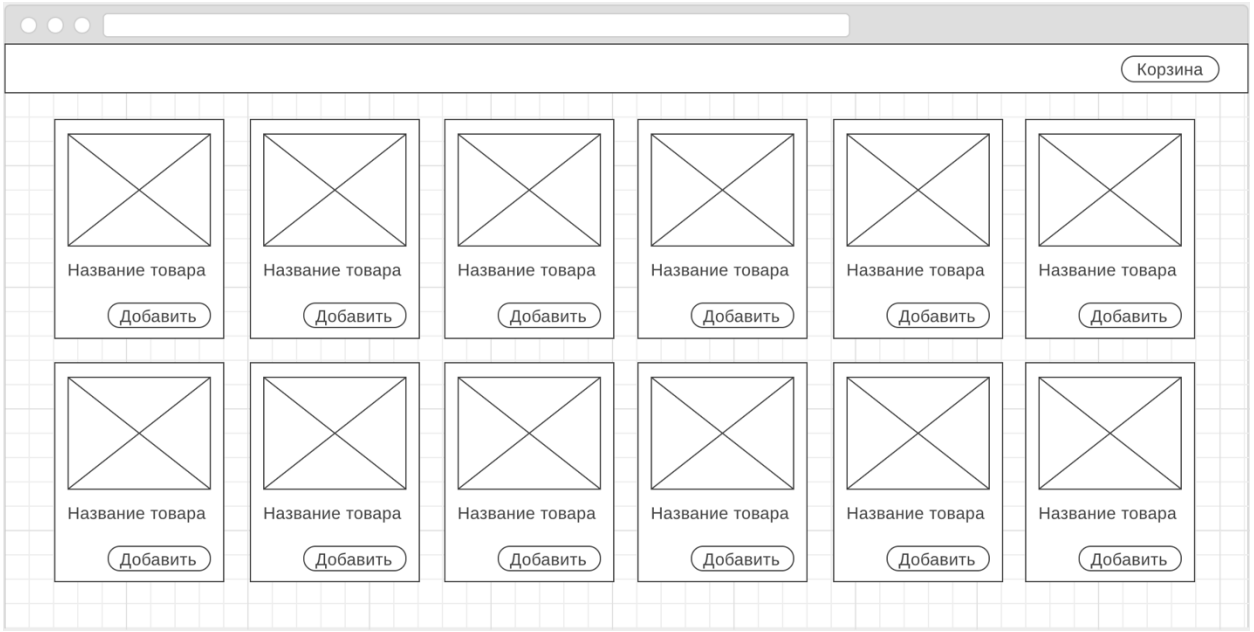
```
let name = 'John';
let greet = `Hello, ${name}. How are you?`;
console.log(greet); // Hello, John. How are you?
```

Внутри фигурных скобок может быть не только имя переменной, но и любые другие операции, возвращающие результат. Это можно использовать, чтобы управлять отображением значения, не меняя самой переменной.

```
let name = 'John';
let greet = `Hello, ${name.toUpperCase()}. How are you?`;
console.log(greet); // Hello, JOHN. How are you?
```

Практика

В течение курса мы создадим клиентскую часть для простого интернет-магазина. В нём будет список товаров и корзина. Товары можно добавить в корзину или удалить из неё.



Создадим файл **index.html** и добавим в него простейшую разметку. Заодно создадим файл **script.js** и подключим его на странице:

```
<!DOCTYPE html>
<html>
<head>
  <title>eShop</title>
  <script src="script.js"></script>
</head>
<body>
  <header>
```

```
    <button class="cart-button" type="button">Корзина</button>
  </header>
  <main>
    <div class="goods-list"></div>
  </main>
</body>
</html>
```

Кнопка **.cart-button** будет отвечать за вызов корзины. В контейнере **.goods-list** будет храниться список товаров. Теперь в **script.js** добавим массив со списком товаров. Для каждого товара определим название и цену:

```
const goods = [
  { title: 'Shirt', price: 150 },
  { title: 'Socks', price: 50 },
  { title: 'Jacket', price: 350 },
  { title: 'Shoes', price: 250 },
];
```

Теперь напишем две функции. Первая будет возвращать разметку для конкретного товара, подставляя его название и цену. Вторая – собирать все товары в один список и записывать его в контейнер **.goods-list**:

```
const goods = [
  { title: 'Shirt', price: 150 },
  { title: 'Socks', price: 50 },
  { title: 'Jacket', price: 350 },
  { title: 'Shoes', price: 250 },
];

const renderGoodsItem = (title, price) => {
  return `<div class="goods-item"><h3>${title}</h3><p>${price}</p></div>`;
};

const renderGoodsList = (list) => {
  let goodsList = list.map(item => renderGoodsItem(item.title, item.price));
  document.querySelector('.goods-list').innerHTML = goodsList;
}

renderGoodsList(goods);
```

Практическое задание

1. Добавьте стили для верхнего меню, товара, списка товаров и кнопки вызова корзины.
2. Добавьте значения по умолчанию для аргументов функции. Как можно упростить или сократить запись функций?
3. * Сейчас после каждого товара на странице выводится запятая. Из-за чего это происходит? Как это исправить?

Дополнительные материалы

1. Современные возможности ES2015.
2. ES6, ES8, ES2017: что такое ECMAScript и чем он отличается от JavaScript.

Используемая литература

1. Современный учебник Javascript.
2. Alberto Montalesi. The Complete Guide to Modern JavaScript.