УДК 004.4: 004.273

Мельник В.И., студентка

РАЗРАБОТКА НОВЫХ ПОДХОДОВ К ВЗАИМОДЕЙСТВИЮ С МОДУЛЯМИ В ES6

Mel'nyk V.I. The development of new approaches to interact with the modules in ES6.

This article solves the problem of interaction with the modules in the new standard ECMAScript 6 language and unusual newfound modular system of language. New JavaScript language standard - the final release of ECMAScript 6 specification (ES6) was recently published. This update is considered to be a phenomenally important as these major changes in language were not around 16 years. The purpose of the introduction of modules in ECMAScript 6 was the construction of the format, which was convenient for CommonJS users, and for users of AMD (Asynchronous Module Definition), - previous module standards applicable in JavaScript. Therefore, they have the same compact syntax as CJS modules. The article also discusses the processes of export, import, alternative ways of loading modules and other functions.

Keywords: module, module system, ECMAScript 6, JavaScript, AMD, CMD.

Мельник В.І. Розробка нових підходів до взаємодії з модулями в ES6.

Виконано аналіз нового і застарілого стандартів ECMAScript на прикладі взаємодії з модульною системою даної мови програмування. ECMAScript служить ядром для такої мови програмування, як JavaScript, і його новий стандарт є суттєвим кроком вперед у розвитку та оптимізації коду безпосередньо JavaScript. У статті розглянуті нові методи експорту, реекспорту та інших операцій з модулями. Модулі в стандарті ECMAScript надають зручні засоби для опису різних функціонально схожих частин коду з метою спрощення читання коду і усунення проблем із його взаємодією. Сучасний стандарт описує, як організувати код в модулі, експортувати і імпортувати значення.

Ключові слова: модуль, модульна система, ECMAScript 6, JavaScript, AMD, CMD.

Мельник В.И. Разработка новых подходов ко взаимодействию с модулями в ES6.

Выполнен анализ нового и устаревшего стандартов ECMAScript на примере взаимодействия с модульной системой данного языка программирования. ECMAScript служит ядром для такого языка программирования, как JavaScript, и его новый стандарт является существенным шагом вперед в развитии и оптимизации кода непосредственно JavaScript. В статье рассмотрены новые методы экспорта, реэкспорта и других операций с модулями. Модули в стандарте ECMAScript предоставляют удобные средства для описания различных функционально схожих частей кода с целью упрощения читаемости кода и устранения проблем со его взаимодействием. Современный стандарт описывает, как организовать код в модули, экспортировать и импортировать значения.

Ключевые слова: модуль, модульная система, ECMAScript 6, JavaScript, AMD, CMD.

Введение

Прежде всего, дадим некоторые определения и расшифровки сокращений, которые не являются общеупотребительными.

ES6 (ECMAScript 6) – язык программирования, используемый в качестве основы для построения других скриптовых языков.

Project Spartan – кодовое название нового браузера компании Microsoft.

CJS – (англ. CommonJS) – стандарт Node.js с нововведениями модульной системы.

AMD – (англ. Asynchronous Module Definition) – подход к разработке на JavaScript, который позволяет создавать модули таким образом, чтобы они и их зависимости могли быть загружены асинхронно.

JavaScript – прототипно-ориентированный сценарный язык программирования.

На очередном собрании Генеральной Ассамблеи ECMA был официально утверждён стандарт ECMAScript 2015, более известный как ECMAScript 6 или "ECMA-262 6th edition". ECMAScript 6 продолжает стандарты, которые определяют базовые функциональные возможности JavaScript, реализованные для всех веб-браузеров.

Прошлый стандарт ECMAScript 5 был принят в 2009 году, а позапрошлый - в 1999 году. Долгое время развитие стандарта было заморожено из-за трудноразрешимых разногласий среди производителей браузеров, некоторые из которых выступали за внесение значительных изменений в JavaScript, а другие настаивали на сохранении полной семантической совместимости.

В статье рассмотрены принципы работы модули в ECMAScript 6, следующей версии JavaScript, затем дается описание инструментов, которые позволяют использовать модули уже сейчас [1-7].

Постановка задачи

Целью создания и внедрения модулей ECMAScript 6 (ES6) было создание формата, удобного как для пользователей CJS, так и для пользователей AMD. В связи с этим они имеют такой же компактный синтаксис, как и модули CJS. С другой стороны, они не такие динамичные (например, вы нельзя условно загрузить модуль с помощью обычного синтаксиса). Кроме того, на этапе компиляции могут возникнуть ошибки, если попытаться импортировать данные, которые не были предварительно экспортированы. При этом можно легко осуществить асинхронную загрузку модулей ES6.

1. Описание модулей ECMAScript

Язык программирования ECMAScript стандартизирован организацией ECMA в спецификации ECMA-262 и является ядром для таких скриптовых языков, как JavaScript (рис. 1).

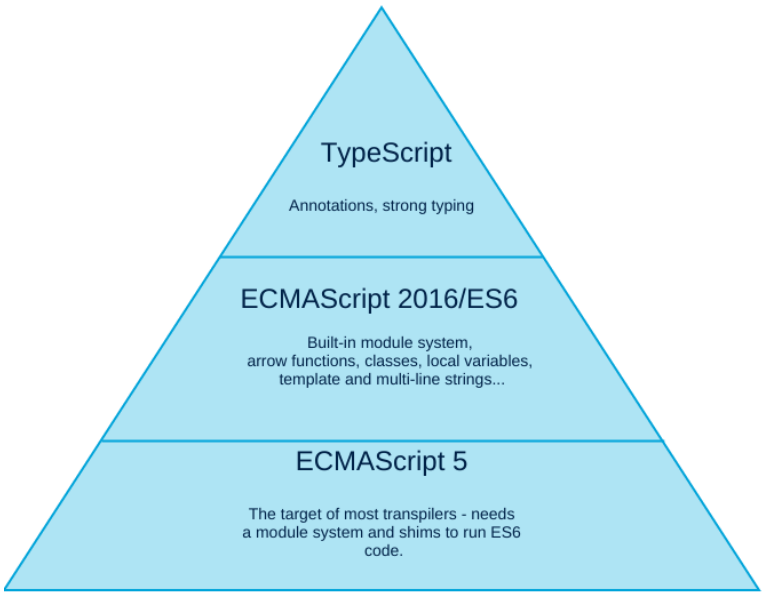

Рис.1. «Пирамида Маслоу» Javascript

На сегодня имеется 70% преимуществ ECMAScript 6 в Project Spartan в соответствии с данной таблицей [1] (рис. 2).

Модуль – определенная инструкция (*statement*), которая вызывается с помощью создания файла или путем его выполнения при помощи интерпретатора ES (при «запуске» файла программистом или в результате импорта другим модулем). В ES6 существует правило соотношения: один файл – один модуль. Каждый модуль имеет отдельную область видимости (*Lexicalenvironment*) – все объявления переменных, функций и классов не доступны за пределами рассматриваемого модуля (файла), если не экспортированы явно. На

верхнем уровне модуля можно использовать операторы import для импорта других модулей и их экспортируемых сущностей, и export - соответственно для экспорта собственных сущностей модуля.

Рис. 2. Особенности ECMAScript 6 в проекте Spartan

Оператор export дает возможность экспортировать модульные сущности так, чтобы они были доступны так же и из других модулей. Каждый модуль имеет неявный объект [[*Exports*]], в котором находятся ссылки на все экспортируемые сущности, ключом к которому, в свою очередь, является идентификатор сущности, имеющий схожесть с *module.exports* из модульной системы NodeJS, но [[*Exports*]] всегда объект, и его нельзя получить напрямую. Единственный способ его изменить – использовать оператор export, который имеет несколько модификаций. Рассмотрим их ниже:

1. Экспорт объявляемой сущности

Представление переменной, функции, класса, с ключевым словом "export" перед ним.

```
export var variable;
export const CONSTANT = 0;
export let scopedVariable = 40;
export function func(){};
export class Fto{};
```

В рассматриваемом случае система экспортов ES6 несколько удобнее, чем в NodeJS, где пришлось бы изначально объявить сущность, а затем добавить её в объект *module.exports*.

```
var variable;
exports.variable = variable;
const CONSTANT = 0;
exports.CONSTANT = CONSTANT;
...
```

Между двумя системами есть и более значимое различие. В NodeJS свойству объекта *exports* присваивается значение выражения. В новом стандарте ES6 оператор export добавляет в [[*Exports*]] ссылку (привязку -*binding*) на объявленную сущность. Это значит, что [[*Exports*]]. <*имя_сущности*> в любом случае будет возвращать действующее значение этой сущности.

```
export var bla = 12; // [[Exports]].bla === 12
bla = 44; // [[Exports]].bla === 44
```

2. Экспорт уже объявленной сущности

Фактически то же самое, только экспортируется сущность, которая была объявлена выше [6]. Следует применить фигурные скобки после ключевого слова export, в которых через запятую нужно указать все сущности, которые надо экспортировать.

```
var btr = 12;
function fto() {}
```

```
export { btr, fto };
```

Посредством ключевого слова "as" мы имеем возможность изменить ключ для [[*Exports*]]) (изменить имя сущности).

```
var bar = 12;
functionfto() {}
export { bar asbla, ftoaschoo };
```

Для данного вида экспорта также верно, что [[*Exports*]] хранит у себя лишь ссылку на сущность, даже в случае «переименования».

```
varbtr = 12;
export { btrasbla }; // [[Exports]].bla === 12
bar = 44; // [[Exports]].bla === 44
```

3. Экспорт по умолчанию

Рассматриваемый вариант использования export различен с двумя предыдущими и отчасти нелогичен. Заключается он в использовании после export ключевого слова default, после которого может идти любая операция из трех: объявление функции, объявление класса или даже выражение [7].

```
exportdefaultfunctionfunc()
{}
export default class Fto{}
export default 44;
```

Каждый вариантов использования добавляет в [[*Exports*]] свойство с ключом «default». Экспортирование по умолчанию *выражения* (третий пример, «export default 44;») – единственный случай с использованием export, когда значением свойства [[*Exports*]] является не ссылка на сущность, а непосредственно само значение выражения. В случае экспорта по умолчанию функции или класса – они будут объявлены в области видимости модуля, а [[*Exports*]].default станет ссылкой на данную сущность.

4. Оператор import

Экспортированное по умолчанию свойство считается «передовым» в этом модуле. Импорт делается с помощью оператора import следующей модификации:

```
import<любое имя>from '<путь к модулю>';
```

В этом вся польза экспорта по умолчанию – при импорте можно назвать его, как угодно.

```
// sub.js
export default class Sub {};
// main.js
Import Base from './sub.js';
```

Импорт обычных экспортируемых свойств выглядит так:

```
// file1.js
export let bla = 30;
// file2.js
import { bla } from './file1.js'; // нужно верно указать имя сущности
// file3.js
import { bla as scopedVariable } from './file1.js'; // но можно переименовать
```

Рассмотрим модуль «file2.js»: оператор import получает объект [[*Exports*]] импортируемого модуля ('file1.js'), находит в нём искомое свойство («bla»), а после создаёт привязку идентификатора "*bla*" к значению [[*Exports*]].bla. Т. е., точно так же, как и [[*Exports*]].bla, *bla* в модуле «file2.js» всегда будет возвращать текущее значение переменной «bla» из модуля «file1.js». Так же, как и *scopedVariable* из модуля «file3.js».

```
// count.js
```

```
export let counter = 0;
export function inc()
{
++counter;
}
// main.js
import { counter, inc } from './count.js';
console.log(counter); // 0
inc();
console.log(counter); // 1
```

Импорт всех экспортируемых свойств

```
import * as sub from'./sub.js';
```

Именно так получаем копию [[*Exports*]] модуля «sub.js».

Включение модуля без импорта

Для того, чтобы файл просто запустился:

```
import'./worker';
```

5. Реэкспорт

Реэкспорт - повторный экспорт модулем свойства, импортируемое им из другого модуля. Осуществляется с помощью оператора export [3];

```
// main.js
export { something } from'./another.js';
```

– *something* после реэкспорта не становится доступной внутри модуля main.js, для этого нужно произвести отдельный импорт;

– система ссылок работает и здесь: модуль, который импортирует из «main.js» *something*, будет получать актуальное значение переменной *something* в «another.js»;

Также имеется возможность реэкспортировать абсолютно все свойства из другого модуля.

```
export * from'./another';
```

Не следует забывать, что в случае объявления в своём модуле exports именем аналогичным реэкспортному, он обязательно сотрёт реэскпортируемое.

```
// sub.js
export const bla = 4, fto = 5;
// another.js
export const bla = 6;
export * from'./sub';
// main.js
import * as another from'./another';
console.log(another); // { bla: 6, fto: 5 }
```

Решается путем переименования конфликтных свойств при реэкспортировании. Синтаксиса для реэкспорта дефолтных свойств у export нет.

2. Сравнительный анализ характеристик модифицированной концепции

Несмотря на то, что JavaScript не поддерживает модули на уровне языка [4], сообществом были созданы впечатляющие решения для их реализации. Два наиболее популярных (но, к сожалению, несовместимых) стандарта [5]:

- CommonJS (CJS): главное воплощение этого стандарта – модульная система Node.js (в Node.js есть несколько особенностей, выходящих за рамки CJS). Характеристики:
  - компактный синтаксис;

- предназначен для синхронной загрузки;
- преимущественно используется на стороне сервера.

- Asynchronous Module Definition (AMD): наиболее популярной реализацией этого стандарта стал RequireJS. Характеристики:
  - синтаксис немного сложнее, что позволяет AMD работать без eval() или этапа компиляции;
  - предназначен для асинхронной загрузки (преимущественно используется на стороне клиента [2]).

Выводы

Чтобы написать серверную часть программы до ES6, нужно пользоваться модульной системой NodeJS, а теперь появилась возможность использования стандартизированной семантики самого языка ECMAScript.

В новом стандарте ECMAScript 6 (2015) для экспорта и импорта модулей введены операторы export и import, а также представлены средства динамической загрузки модулей, пространства имён и изоляция состояния.

Литература

1. Catuhe D. Programming with the Kinect for Windows Software Development Kit [Текст] / D Catuhe – L.: Gardners Books, 2014. – 442 с.

2. Симпсон К. Чего Вы не знали об ECMAScript 6 [Текст] / К. Симпсон – К.: ДМК Пресс, 2015. – 121 с.

3. A TypeScript Angular2 starter project walkthrough [Електронний ресурс] // – Режим доступу: http://chariotsolutions.com/blog/post/angular2-starter-walkthrough-overview/

4. Разработка на ECMAScript6 [Електронний ресурс] // – Режим доступу: https://habrahabr.ru/post/252323/

5. ECMAScript® 2015 Language Specification [Електронний ресурс] // – Режим доступу: http://www.ecma-international.org/ecma-262/6.0/

6. Understanding ECMAScript 6 [Електронний ресурс] // – Режим доступу: https://leanpub.com/understandinges6/

7. Прасти Н. Введение в ECMAScript 6 [Текст] / Н. Прасти – К.: ДМК Пресс, 2016. – 176 с.

Автор статті

Мельник Влада Ігорівна – студентка, Національний авіаційний університет, Київ, Україна. Тел. +38 099 204 40 76. E-mail: vlada.cosmoss@gmail.com

Author of the article

Mel'nyk Vlada Ihorivna – student, National Aviation University, Kyiv, Ukraine. Tel. +38 099 204 40 76. E-mail: vlada.cosmoss@gmail.com

Дата надходження в редакцію: 04.07.2016 р. Рецензент: д.т.н., проф. М.А. Віноградов