# Front-end

JavaScript

# JS. Import, Export

- Import
- Export

# JS. Import, Export

Каждая программа со временем становится большой. Чем больше в проекте кода, тем сложнее в нём ориентироваться, писать и поддерживать.

**Модули** помогают организовать код так, чтобы с проектом было проще и удобнее работать, позволяют переиспользовать код и структурировать программу.

**Модуль** — обособленная часть системы. Как правило, один модуль — это один файл. Модули можно воспринимать, как части конструктора, из которых собирается программа.

# JS. Import, Export

В модуле по умолчанию всё скрыто. Наружу предоставляется только то, что разработчики намеренно хотят предоставить — **экспорты**. Эту функциональность можно импортировать в других модулях и использовать.

Таким образом, **модуль** — это чёрный ящик с одним входом (импорт) и одним выходом (экспорт), через которые этот модуль «общается» с другими.

Модули можно сравнить с функциями: они решают те же проблемы, но в более крупном масштабе. Функции помогают разбивать большую задачу на мелкие, а модули — разбивать большой проект на мелкие части. Функции помогают структурировать выстраивать архитектуру решения конкретной задачи, а модули — проекта целиком.

# JS. Import, Export

Идея и польза модулей в первую очередь в том, что они помогают справляться со сложностью программы. Если мы работаем с проектом на 20 тысяч строк, будет совершенно неудобно хранить весь код в одном файле

# JS. Import, Export

Изначально модулей в JavaScript не существовало. Считалось, что скрипты, подключаемые к страницам очень простые, и модульная система не нужна.

Время шло, приложения на JS становились всё сложнее, и необходимость модулей становилась очевидной. Тогда появились первые попытки «принести» модули в JS.

Сейчас модули уже появились и поддерживаются, но их «становление» проходило медленно. Существовало несколько версий модульных систем: AMD, CommonJS, UMD и **ES-модули**.

Современной системой считаются **ES-модули**. Другие модульные системы считаются устаревшими.

# JS. Import, Export

## ECMAScript или ES-модули

**ES-модули** — модульная система на уровне языка, которая появилась в спецификации ES2015.

В ES-модулях для экспорта используется ключевое слово **export**, а для импорта — **import**.

# JS. Import, Export

```
// module1.js


// При добавлении ключевого слова export
// выражение становится экспортированным.
export function sum(a, b) {
  return a + b
}


// Экспортировать можно не только функции,
// но и константы:
export const SOME_SETTINGS_FLAG = false
export const user = {}
export const books = ['Война и мир', 'Мастер и Маргарита']
```

# JS. Import, Export

```
// module2.js

// Таким образом мы можем получить
// доступ к этой функциональности
// в другом модуле через импорт:
import { sum } from './module1.js'

// Точно так же, мы можем импортировать и константы:
import { user, books } from './module1.js'
```

# JS. Import, Export

Обратите внимание, что перечисляя названия при импорте через запятую, можно в одном импорте получить доступ сразу к нескольким переменным или функциям.

Если вдруг мы хотим изменить имя той функции или переменной, которую импортируем, мы можем использовать ключевое слово **as**:

```
import { user as admin } from './module1.js'
```

# JS. Import, Export

**Это также работает и со множественным импортом:**

```
import { books as library, SOME_SETTINGS_FLAG as turnedOn } from './module1.js'
```

Экспортировать функциональность можно также и уже после того, как она определена:

```
const user = {}

export { user }
```

**Это иногда бывает полезно, если мы хотим описать все экспорты в конце файла.**

**Кроме того, это же помогает изменять названия при экспорте.**

Например, если

```
const user = {}

export { user as admin }
```

# JS. Import, Export

При использовании ключевого слова **export** рядом с функцией или переменной, мы экспортируем конкретную функцию или переменную.

У таких переменных и функций **обязательно должно быть имя**, потому что именно **по этому имени мы будем получать доступ** к ним из других модулей. Поэтому такие экспорты называются **именованными** — у экспортируемой функциональности есть имя и по этому же имени мы будем импортировать её в других модулях.

# JS. Import, Export

## Экспорты по умолчанию

Существуют также **экспорты по умолчанию**. Когда мы из модуля экспортируем какую-то функциональность по умолчанию, мы можем опустить имя, но обязаны использовать ключевое слово **default** после **export**

# JS. Import, Export

// sum.js

Функция может не иметь имени, потому что используется экспорт по умолчанию.

```
export default function (a, b) {
  return a + b
}
```

// other-module.js

При импорте такой функциональности в другом модуле нам уже не требуется использовать {}.

```
import sum from './sum.js'
```

Более того, мы сразу можем использовать другое имя при импорте:

```
import superCoolSummator from './sum.js'
```

# JS. Import, Export

Сейчас сообщество считает экспорты по умолчанию менее удачной практикой. В первую очередь потому, что с именованными экспортами проще работать: их проще переименовывать, с ними проще работать автоматизированными средствами рефакторинга.

Мы рекомендуем использовать именованные экспорты вместо экспортов по умолчанию там, где это возможно.

# JS. Import, Export

## Особенности в браузере

В браузере модули работают через подключение скриптов с атрибутом type='module':

```
<body>
  <script src="module1.js" type="module"></script>
</body>
```

# JS. Import, Export

## Особенности в браузере

Должен быть прописан путь до файла.

То есть:

```
import user from 'user' // Неправильно...

// Должен быть либо абсолютный путь:
import user from 'https://some-site.com/js/user.js'

// ...Либо относительный:
import user from './user.js'
```

# JS. Import, Export

В браузере модули сами по себе используются пока редко. Сейчас чаще используются инструменты сборки типа Gulp, Webpack, Parcel, Rollup и другие.

Код, использующий импорты и экспорты, или использующий скрипты с type="module", «прогоняется» через этот инструмент, соединяется в бандлы, минифицируется и уже в таком виде отправляется в продакшен.

В итоге мы получаем скрипты, готовые для использования в большем количестве браузеров, при этом польза модулей в организации кодовой базы и структуризации проекта сохраняется.