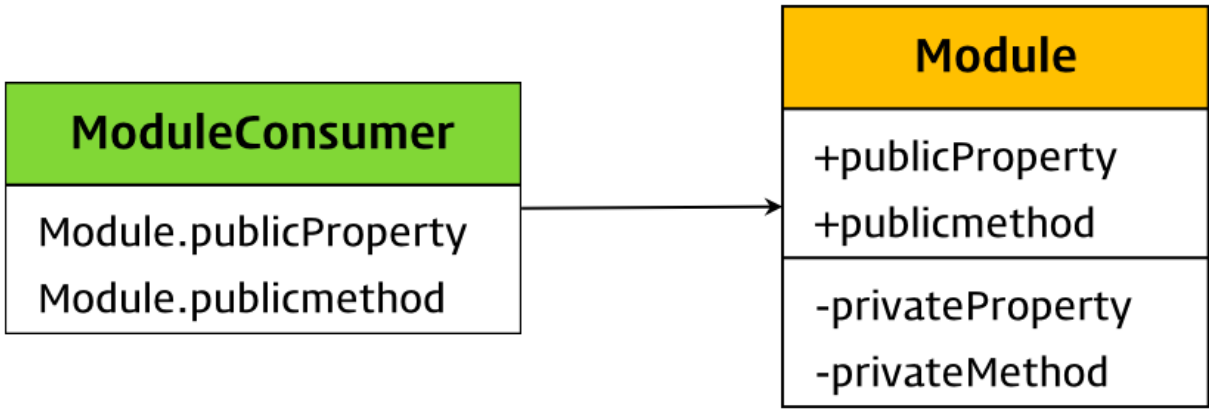
모듈

모듈이란 애플리케이션을 구성하는 개별적 요소로서 재사용 가능한 코드 조각을 말한다. 모듈은 세부 사항을 캡슐화하고 공개가 필요한 API만을 외부에 노출한다.



일반적으로 **모듈은 파일 단위로 분리**되어 있으며 애플리케이션은 필요에 따라 **명시적으로 모듈을 로드하여 재사용**한다. 즉, 모듈은 애플리케이션에 분리되어 개별적으로 존재하다가 애플리케이션의 로드에 의해 비로소 애플리케이션의 일원이 된다. 모듈은 기능별로 분리되어 작성되므로 코드의 단위를 명확히 분리하여 애플리케이션을 구성할 수 있으며 재사용성이 좋아서 개발 효율성과 유지보수성을 높일 수 있다.

자바스크립트는 웹페이지의 보조적인 기능을 수행하기 위해 한정적인 용도로 만들어진 태생적 한계로 다른 언어에 비해 부족한(나쁜) 부분이 있는 것이 사실이다. 그 대표적인 것이 모듈 기능이 없는 것이다.

C 언어는 #include, Java는 import 등 대부분의 프로그래밍 언어는 모듈 기능을 가지고 있다. 하지만 클라이언트 사이드 자바스크립트는 script 태그를 사용하여 외부의 스크립트 파일을 가져올 수는 있지만, 파일마다 독립적인 파일 스코프를 갖지 않고 하나의 전역 객체(Global Object)를 공유한다. 즉, 자바스크립트 파일을 여러 개의 파일로 분리하여 script 태그로 로드하여도 분리된 자바스크립트 파일들이 결국 하나의 자바스크립트 파일 내에 있는 것처럼 하나의 전역 객체를 공유한다. 따라서 분리된 자바스크립트 파일들이 하나의 전역

을 갖게 되어 전역 변수가 중복되는 등의 문제가 발생할 수 있다. 이것으로는 모듈화를 구현할 수 없다.

자바스크립트를 클라이언트 사이드에 국한하지 않고 범용적으로 사용하고자 하는 움직임이 생기면서 모듈 기능은 반드시 해결해야 하는 핵심 과제가 되었다. 이런 상황에서 제안된 것이 CommonJS와 AMD(Asynchronous Module Definition)이다.

결국, 자바스크립트의 모듈화는 크게 CommonJS와 AMD 진영으로 나뉘게 되었고 브라우저에서 모듈을 사용하기 위해서는 CommonJS 또는 AMD를 구현한 모듈 로더 라이브러리를 사용해야 하는 상황이 되었다.

서버 사이드 자바스크립트 런타임 환경인 Node.js는 모듈 시스템의 사실상 표준(de facto standard)인 CommonJS를 채택하였고 독자적인 진화를 거쳐 현재는 CommonJS 사양과 100% 동일하지는 않지만 기본적으로 CommonJS 방식을 따르고 있다. 즉, Node.js에서는 표준은 아니지만 모듈이 지원된다. 따라서 Node.js 환경에서는 모듈 별로 독립적인 스코프, 즉 모듈 스코프를 갖는다.

이러한 상황에서 ES6에서는 클라이언트 사이드 자바스크립트에서도 동작하는 모듈 기능을 추가하였다.

script 태그에 `type="module"` 어트리뷰트를 추가하면 로드된 자바스크립트 파일은 모듈로서 동작한다. ES6 모듈의 파일 확장자는 모듈임을 명확히 하기 위해 mjs를 사용하도록 권장한다.

```
<script type="module" src="lib.mjs"></script>
<script type="module" src="app.mjs"></script>
```

단, 아래와 같은 이유로 아직까지는 브라우저가 지원하는 ES6 모듈 기능보다는 Webpack 등의 모듈 번들러를 사용하는 것이 일반적이다.

- IE를 포함한 구형 브라우저는 ES6 모듈을 지원하지 않는다.
- 브라우저의 ES6 모듈 기능을 사용하더라도 트랜스파일링이나 번들링이 필요하다.
- 아직 지원하지 않는 기능(Bare import 등)이 있다.
- 점차 해결되고는 있지만 아직 몇가지 이슈가 있다

ES6를 사용하여 프로젝트를 진행하려면 ES6로 작성된 코드를 IE를 포함한 모든 브라우저에서 문제 없이 동작시키기 위한 개발환경을 구축하는 것이 필요하다.

모듈 스코프

ES6 모듈 기능을 사용하지 않으면 분리된 자바스크립트 파일에 독자적인 스코프를 갖지 않고 하나의 전역을 공유한다. 아래 예제를 살펴보자.

```
// foo.js
var x = 'foo';

// 변수 x는 전역 변수이다.
console.log(window.x); // foo
```

```
// bar.js
// foo.js에서 선언한 전역 변수 x와 중복된 선언이다.
var x = 'bar';

// 변수 x는 전역 변수이다.
// foo.js에서 선언한 전역 변수 x의 값이 재할당되었다.
console.log(window.x); // bar
```

```
<!DOCTYPE html>
<html>
<body>
  <script src="foo.js"></script>
  <script src="bar.js"></script>
</body>
</html>
```

HTML에서 2개의 자바스크립트 파일을 로드하면 로드된 자바스크립트는 하나의 전역을 공유한다. 위 예제에서 로드된 2개의 자바스크립트 파일은 하나의 전역 객체를 공유하며 하나의 전역 스코프를 갖는다. 따라서 foo.js에서 선언한 변수 x와 bar.js에서 선언한 변수 x는 중복 선언되며 의도치 않게 변수 x의 값이 덮어 써진다.

ES6 모듈은 파일 자체의 스코프를 제공한다. 즉, ES6 모듈은 독자적인 **모듈 스코프**를 갖는다. 따라서, 모듈 내에서 var 키워드로 선언한 변수는 더 이상 전역 변수가 아니며 window 객체의 프로퍼티도 아니다.

```
// foo.mjs
var x = 'foo';

console.log(x); // foo
// 변수 x는 전역 변수가 아니며 window 객체의 프로퍼티도 아니다.
console.log(window.x); // undefined
```

```
// bar.mjs
// 변수 x는 foo.mjs에서 선언한 변수 x와 스코프가 다른 변수이다.
var x = 'bar';

console.log(x); // bar
// 변수 x는 전역 변수가 아니며 window 객체의 프로퍼티도 아니다.
console.log(window.x); // undefined
```

```
<!DOCTYPE html>
<html>
<body>
  <script type="module" src="foo.mjs"></script>
  <script type="module" src="bar.mjs"></script>
</body>
</html>
```

모듈 내에서 선언한 변수는 모듈 외부에서 참조할 수 없다. 스코프가 다르기 때문이다.

```
// foo.js
const x = 'foo';

console.log(x); // foo
```

```
// bar.js
// 다른 모듈에서 선언한 변수는 모듈 외부에서 참조할 수 없다. 스코프가 다르기 때문이다.
console.log(x); // ReferenceError: x is not defined
```

```
<!DOCTYPE html>
<html>
<body>
  <script type="module" src="foo.js"></script>
  <script type="module" src="bar.js"></script>
</body>
</html>
```

export 키워드

모듈은 독자적인 모듈 스코프를 갖기 때문에 모듈 안에 선언한 모든 식별자는 기본적으로 해당 모듈 내부에서만 참조할 수 있다. 만약 모듈 안에 선언한 식별자를 외부에 공개하여 다른 모듈들이 참조할 수 있게 하고 싶다면 export 키워드를 사용한다. 선언된 변수, 함수, 클래스 모두 export할 수 있다.

모듈을 공개하려면 선언문 앞에 export 키워드를 사용한다. 여러 개를 export할 수 있는데 이때 각각의 export는 이름으로 구별할 수 있다.

```
// lib.mjs
// 변수의 공개
export const pi = Math.PI;

// 함수의 공개
export function square(x) {
  return x * x;
}

// 클래스의 공개
export class Person {
  constructor(name) {
```

```
    this.name = name;
  }
}
```

선언문 앞에 매번 export 키워드를 붙이는 것이 싫다면 export 대상을 모아 하나의 객체로 구성하여 한번에 export할 수도 있다.

```
// lib.mjs
const pi = Math.PI;

function square(x) {
  return x * x;
}

class Person {
  constructor(name) {
    this.name = name;
  }
}

// 변수, 함수 클래스를 하나의 객체로 구성하여 공개
export { pi, square, Person };
```

import 키워드

모듈에서 공개(export)한 대상을 로드하려면 import 키워드를 사용한다.

모듈이 export한 식별자로 import하며 ES6 모듈의 파일 확장자를 생략할 수 없다.

```
// app.mjs
// 같은 폴더 내의 lib.mjs 모듈을 로드.
// lib.mjs 모듈이 export한 식별자로 import한다.
// ES6 모듈의 파일 확장자를 생략할 수 없다.
import { pi, square, Person } from './lib.mjs';

console.log(pi);          // 3.141592653589793
```

```
console.log(square(10)); // 100
console.log(new Person('Lee')); // Person { name: 'Lee' }
```

모듈이 export한 식별자를 각각 지정하지 않고 하나의 이름으로 한꺼번에 import할 수도 있다. 이때 import되는 식별자는 as 뒤에 지정한 이름의 객체에 프로퍼티로 할당된다.

```
// app.mjs
import * as lib from './lib.mjs';

console.log(lib.pi);          // 3.141592653589793
console.log(lib.square(10)); // 100
console.log(new lib.Person('Lee')); // Person { name: 'Lee'
}
```

이름을 변경하여 import할 수도 있다.

```
// app.mjs
import { pi as PI, square as sq, Person as P } from './lib.
mjs';

console.log(PI);    // 3.141592653589793
console.log(sq(2)); // 4
console.log(new P('Kim')); // Person { name: 'Kim' }
```

모듈에서 하나만을 export할 때는 default 키워드를 사용할 수 있다.

```
// lib.mjs
export default function (x) {
  return x * x;
}
```

다만, default를 사용하는 경우, var, let, const는 사용할 수 없다.

```
// lib.mjs
export default () => {};
// => OK

export default const foo = () => {};
// => SyntaxError: Unexpected token 'const'
```

default 키워드와 함께 export한 모듈은 {} 없이 임의의 이름으로 import한다.

```
// app.mjs
import square from './lib.mjs';

console.log(square(3)); // 9
```

브라우저가 지원하는 ES6 모듈 기능을 이용하여 import와 export가 동작하는지 확인해보자.

```
// lib.mjs
export default x => x * x;
```

```
// app.mjs
// 브라우저 환경에서는 모듈의 파일 확장자를 생략할 수 없다.
// 모듈의 파일 확장자는 .mjs를 권장한다.
import square from './lib.mjs';

console.log(square(10)); // 100
```

```
<!DOCTYPE html>
<html>
<body>
  <script type="module" src="./lib.js"></script>
  <script type="module" src="./app.js"></script>
```

```
</body>
</html>
```

위 HTML을 실행해보면 콘솔에 100이 출력되는 것을 확인할 수 있다.