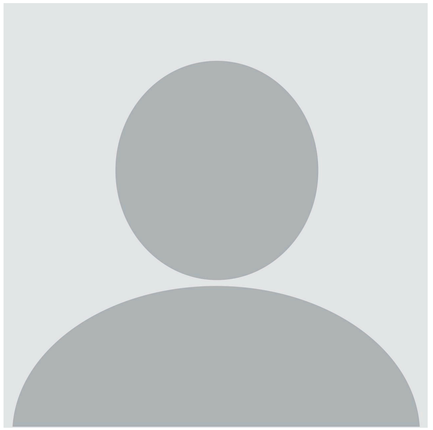
“코드를 흔들어 가볍게”

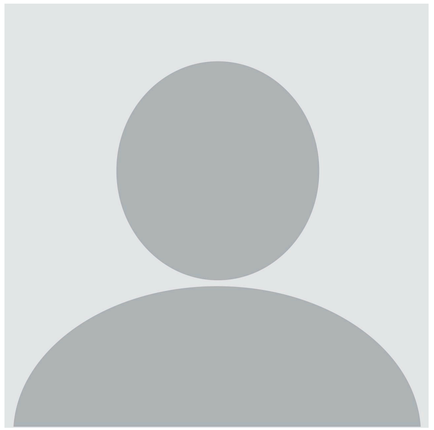
Tree Shaking

조원희(팀장) | 김윤미 | 김효민 | 신경남



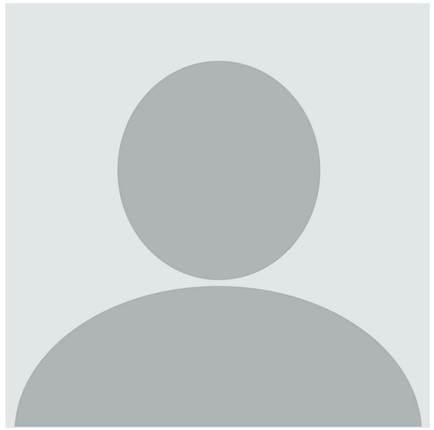
조원희

발표



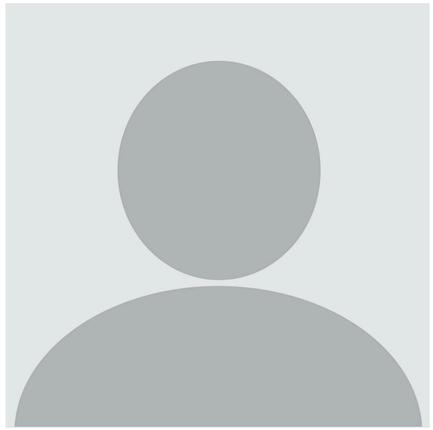
김윤미

발표



김효민

PPT



신경남

PPT

목차

1. 번들링이란?

2. 번들링 최적화 기법 - Tree Shaking

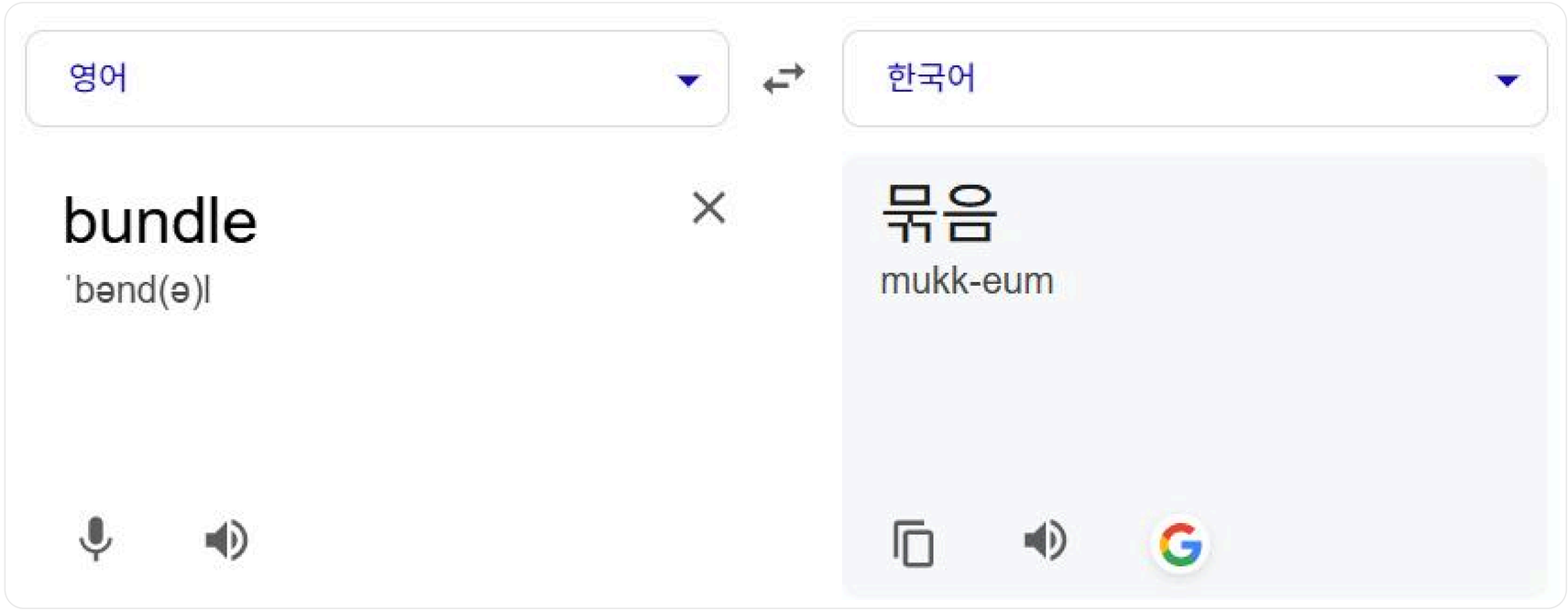
3. Tree Shaking의 원리 & webpack에서의 고려사항

4. Tree Shaking in RollupJS

5. 정리

1. 번들링이란?

번들러란 무엇일까?



번들러 없이 개발한다면?

계산기

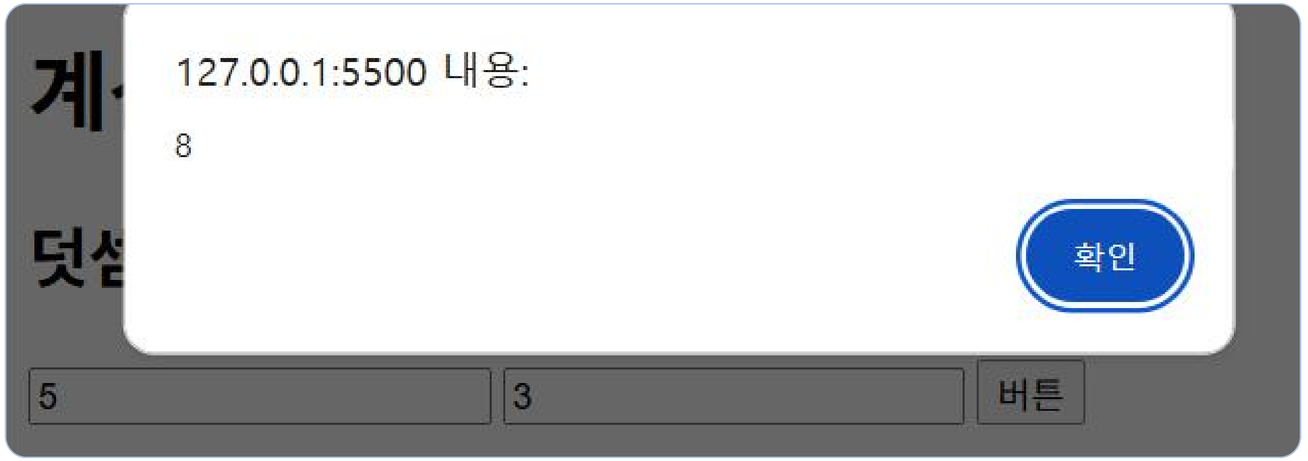
덧셈

덧셈 기능 추가

버튼

뺄셈

버튼



Q. 뺄셈 기능 추가 한다면 ?

번들러 없이 개발한다면?

뺄셈 기능 추가

❌ 전역 변수 문제

```
const num1 = Number(fristInput.value ?? 0);
const num2 = Number(secondInput.value ?? 0);
```

```
Uncaught SyntaxError: Identifier 'firstInput' has already been declared (at minus.js:1:1)
> |
```

💡 즉시 실행 함수 스코프 만들기

❌ 네트워크 요청 문제

Name	Status	Type	Initiator	Size	Time	Waterfall
add.js	200	script	index.html:23	0.8 kB	9 ms	
minus.js	200	script	index.html:24	0.9 kB	9 ms	

💡 한 파일로 합치기

번들러 없이 개발한다면?

만약 기능이 많아진다면?

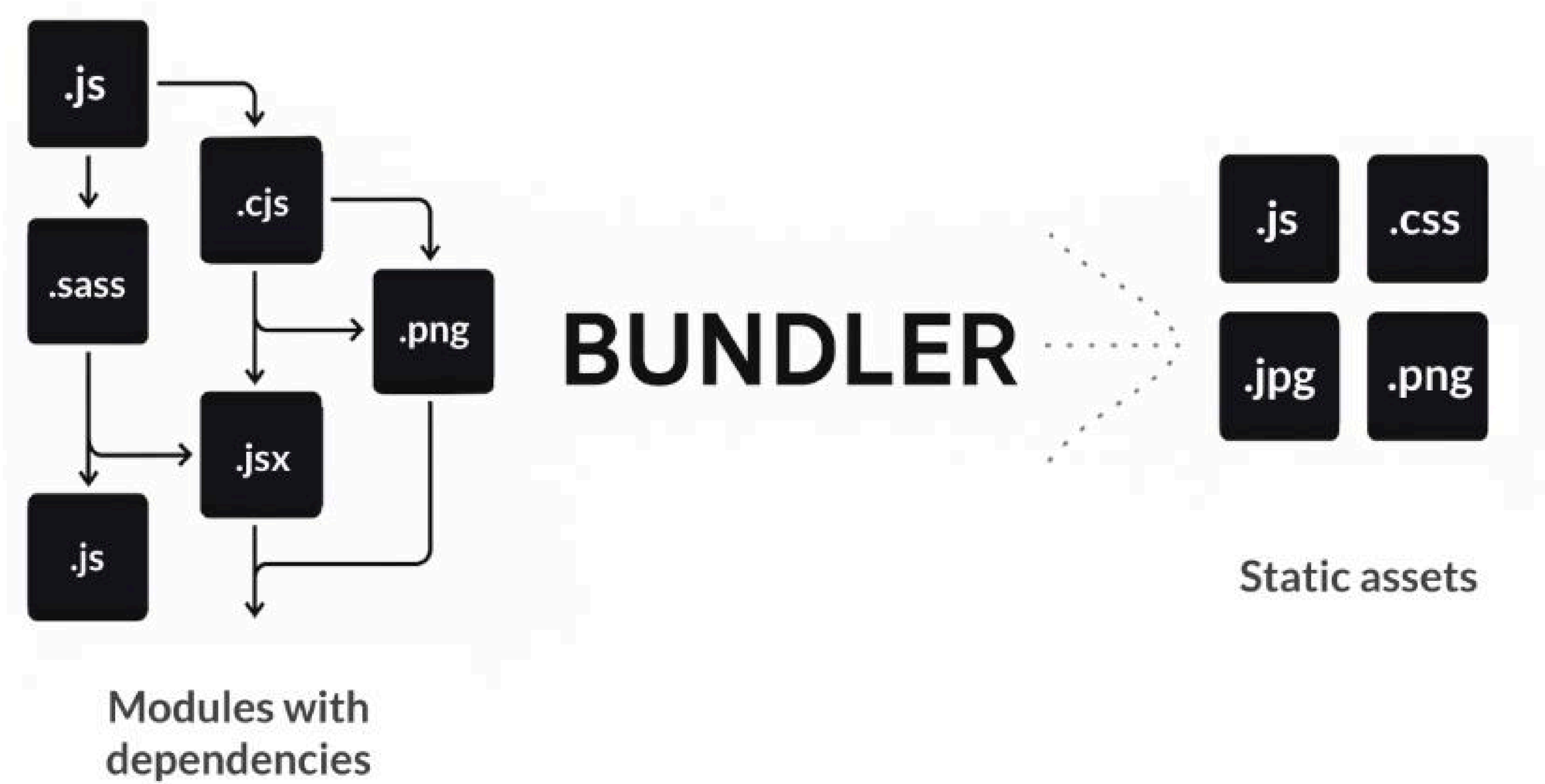
순서 의존성 높아짐

모듈 관계 증가

모든 코드 브라우저 전송

유지보수 불가능

번들링


.js
.cjs
.sass
.png
.jsx
.js
Modules with dependencies
BUNDLER
.js
.css
.jpg
.png
Static assets

번들러를 쓰는 이유

요청 수 감소

로딩 속도 향상

캐싱 최적화

번들링된 파일 하나만 캐시

유지보수성과 배포 효율성

개발할 때는 모듈화 , 배포할 때는 성능 최적화

흐름 정리

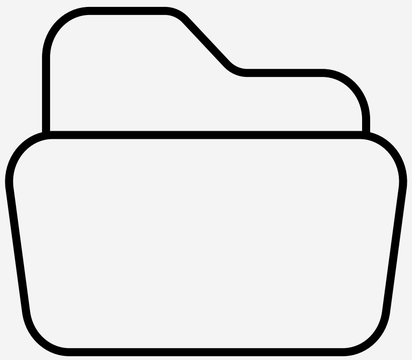
여러 JS 파일들의 의존성 관계 분석

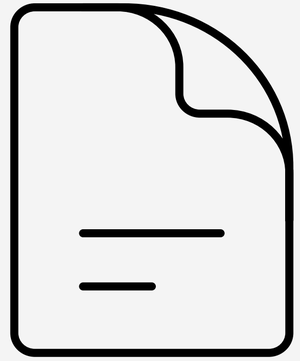
의존성 순서에 맞게 하나로 합치기

```
// index.js
import {MyHeader} from 'MyHeader.js'
import {MyBody} from 'MyBody.js'
```

구체적인 방법 정리

1 모듈 탐색 → 2 의존성 구조 정리 → 3 번들 파일 생성





2. 번들링 최적화 기법
- Tree Shaking

최적화 기법

- **Tree Shaking**: 사용하지 않는 코드(import) 제거
- **Code Spliting**: 한 파일을 여러 개의 작은 파일로 나누기
- **Minification**: 공백 / 주석을 없애서 크기 줄이기

Code Splitting

Code Splitting

```jsx
import React, { Suspense } from "react";

const LazyComponent = React.lazy(() => import("./LazyComponent"));

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  );
}
```

Tree Shaking

Tree shaking

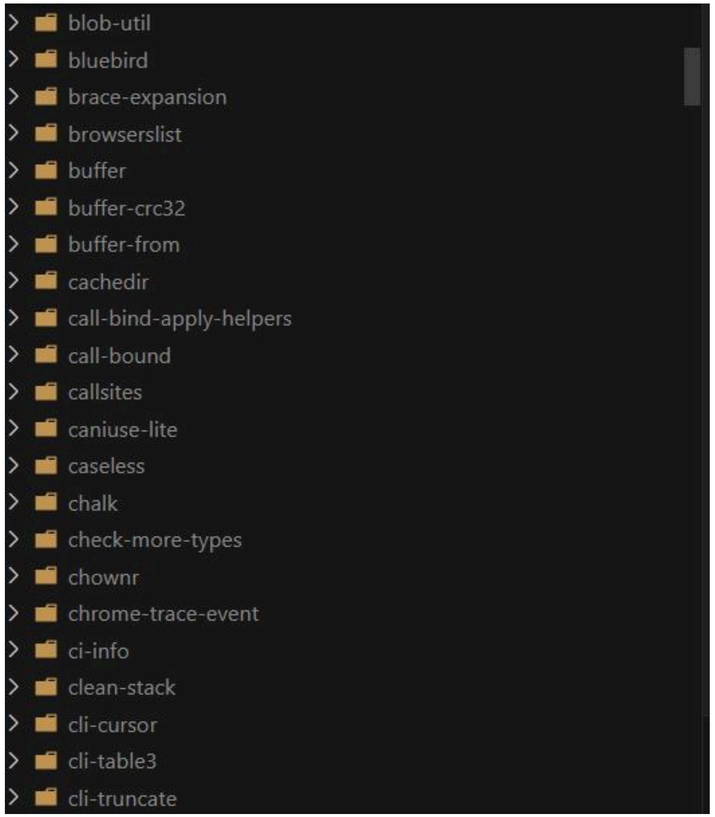
Tree shaking is a term commonly used within a JavaScript context to describe the removal of dead code.

→ 사용되지 않는 코드(dead code)를 제거하기

https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking

Tree Shaking

```
import * as util from '../utilFile';
```



이 많은 걸 전부 다 import?

- ✕ 리소스 낭비 - 번들 파일 크기 커짐
- ✕ 번들 파일 로딩 시간 증가 → 페이지 로딩 속도 증가

Tree Shaking

main.js

```
import { add } from './math.js';


console.log(add(2, 3));
```

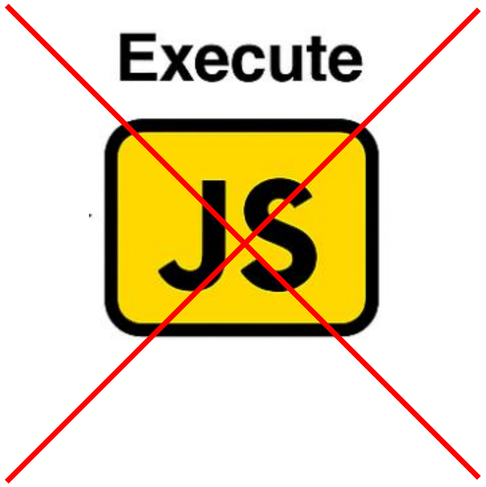
math.js

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


export function subtract(a, b) {
  return a - b;
}
```

3. Tree Shaking 원리 & webpack에서의 고려 사항

정적 분석(Static Analysis)

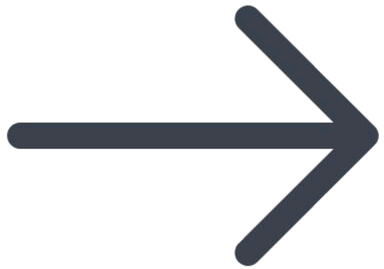
- 프로그램을 실행하지 않고 코드를 분석하는 것



트리셰이킹은 static import를 기반으로 동작

dynamic import

CJS



static import

ESM

정적인 코드

```
// utils.js
export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}

```

```
// main.js (ES6 이후 문법)
import { add } from './math.js';

console.log(add(1, 2));

```

동적인 코드

```
1   const path = './' + moduleName;
2   const mod = require(path);
```

코드 실행 전까지
무슨 모듈을 import할지 알 수 없음

```
1 if (Math.random() > 0.5) {
2   module.exports = { foo: () => {}, bar: () => {}
  };
3 } else {
4     module.exports = () => 'Hello';
5 }
6
```

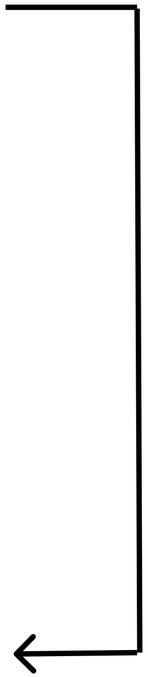
뭐가 export될 건지 알 수 없음

트리셰이킹을 잘쓰려면?

- static import를 유념해두어야함.

만약 Babel을 사용한다면?

ES6 / ES NEXT



ES5

```
{
  "presets": [["@babel/preset-env", { "modules": false }]]
}
```

설정이 필요함!

⇒ import/export를 변환하지 않고 그대로 둠

Side Effect

번들러는 부작용이 없는 코드란 확신이 있을 때만
트리셰이킹한다.

```
1   // package.js
2   {
3     "sideEffects": false
4   }
```

⇒ 이 폴더는 순수함!

정리 - 트리 셰이킹을 제대로 적용하려면

- ES6모듈 구문을 사용해야 한다.
- 컴파일러가 ES모듈을 commonJS 모듈로 변환하지 않도록 해야한다.
- Side Effect를 고려하자.

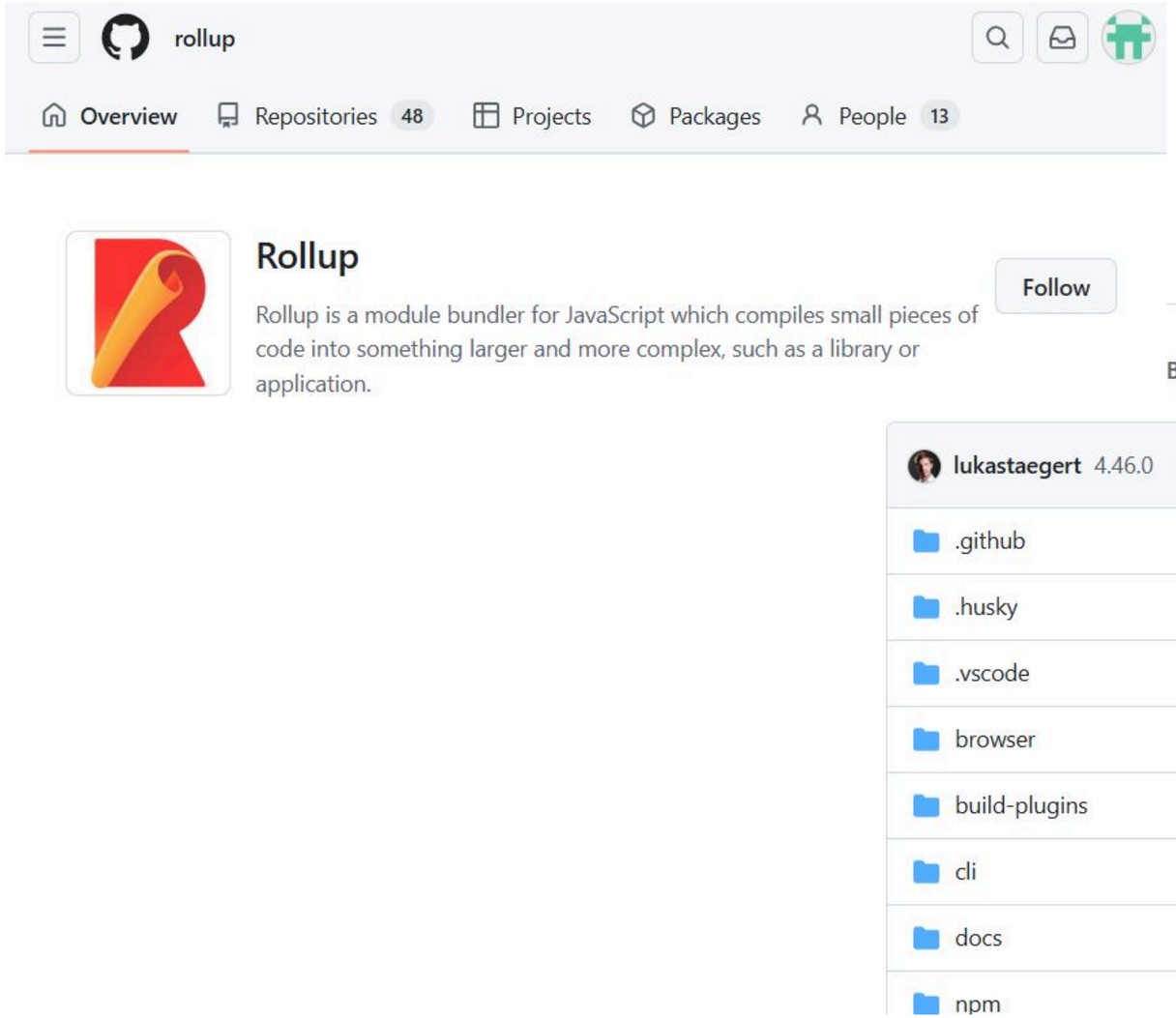
4. Tree Shaking in RollupJS

Rollup

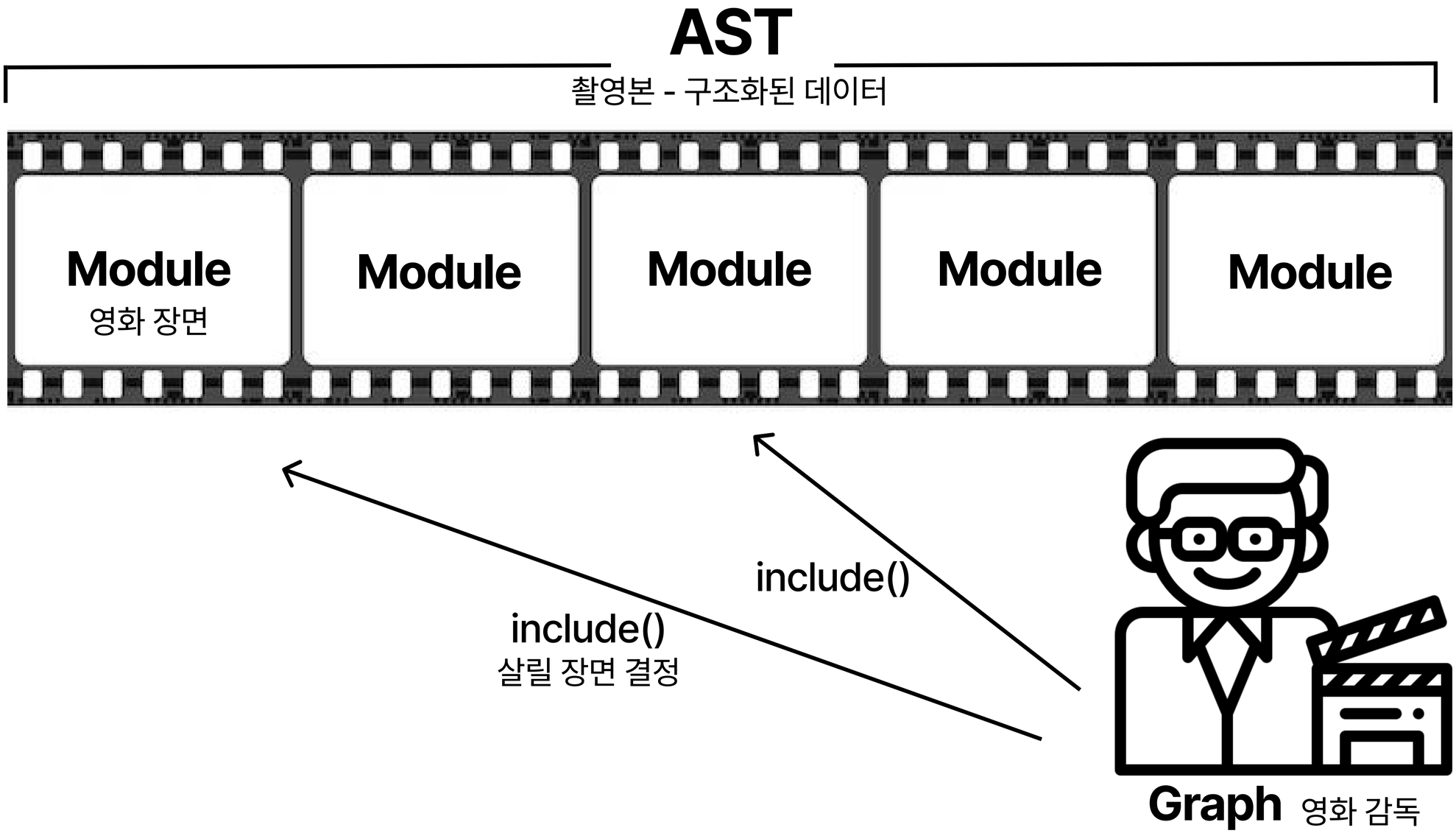
- ESM 기반
- **Tree Shaking에 최적화**

Rollup - Github

https://github.com/rollup/rollup



들어가기 전에


AST
촬영본 - 구조화된 데이터
Module
영화 장면
Module
Module
Module
Module
include()
include()
살릴 장면 결정
Graph
영화 감독

전체적인 흐름 - src/Graph.ts/bulid()

1

모듈 로딩 & AST 생성
- generateModuleGraph()

```
timeStart('generate module graph', 2);
await this.generateModuleGraph();
timeEnd('generate module graph', 2);
```

2

모듈 정렬 & 참조 바인딩
- sortModules()

```
timeStart('sort and bind modules', 2);
this.phase = BuildPhase.ANALYSE;
this.sortModules();
timeEnd('sort and bind modules', 2);
```

3

Tree Shaking 진행
- includeStatements()

```
timeStart('mark included statements', 2);
this.includeStatements();
timeEnd('mark included statements', 2);
```

1

모듈 로딩 & AST 생성
- generateModuleGraph()

```
timeStart('generate module graph', 2);
await this.generateModuleGraph();
timeEnd('generate module graph', 2);
```

2

모듈 정렬 & 참조 바인딩
- sortModules()

```
timeStart('sort and bind modules', 2);
this.phase = BuildPhase.ANALYSE;
this.sortModules();
timeEnd('sort and bind modules', 2);
```

3

Tree Shaking 진행
- includeStatements()

```
timeStart('mark included statements', 2);
this.includeStatements();
timeEnd('mark included statements', 2);
```

1. 모듈 로딩 & AST 생성

```
private async generateModuleGraph(): Promise<void> {
  ({ entryModules: this.entryModules, implicitEntryModules
    await this.moduleLoader.addEntryModules(normalizeEntry
```

src/ModuleLoader.ts

```
this.loadEntryModule(id, true, importer, null)
        ↓
return this.fetchModule(
        ↓
this.addModuleSource(id, importer, module)
        ↓
  await module.setSource(cachedModule);
- else {
  module.updateOptions(sourceDescription);
  await module.setSource(
```

src/Module.ts

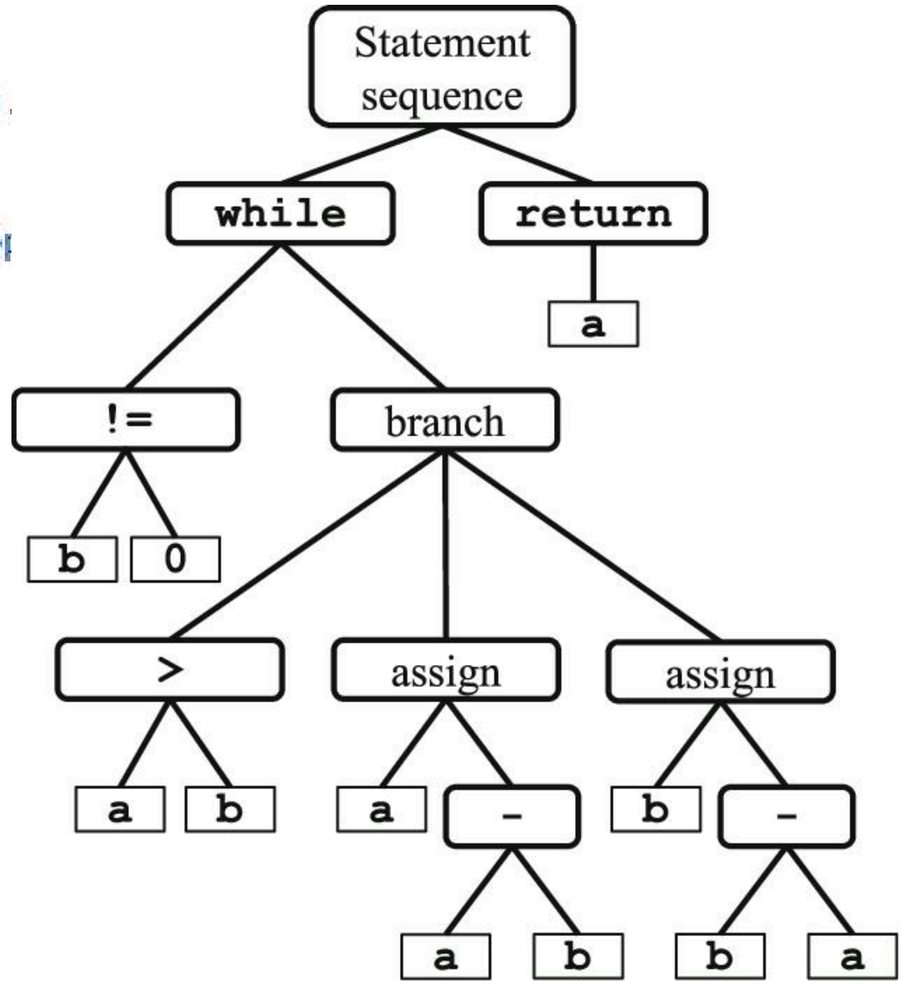
```
async setSource({
  ast,
  code,
  customTransformCache,
  originalCode,
  originalSourcemap,
  resolvedIds,
  sourcemapChain,
  transformDependencies,
  transformFiles,
  ...moduleOptions
}: TransformModuleJSON & {
  resolvedIds?: ResolvedIdMap;
  transformFiles?: EmittedFile[] | undefined;
}): Promise<void> {
  timeStart('generate ast', 3);
  if (code.startsWith('#!')) {
    const shebangEndPosition = code.indexOf('\n');
    this.shebang = code.slice(2, shebangEndPosition);
  }

  this.info.code = code;
  this.originalCode = originalCode;
```

1. 모듈 로딩 & AST 생성 -setSource()

```
timeEnd('generate ast', 3);
const astBuffer = await parseAsync(code, false, this.options
timeStart('generate ast', 3);
this.ast = convertProgram(astBuffer, programParent, this.sco
```

코드 파싱

AST 구성
(Program 객체)



AST : Abstract Syntex Tree

소스 코드를 추상화시켜
트리 형태의 자료구조로 나타낸 것

⇒ 모듈 정보 & 구성(AST)

1. 모듈 로딩 & AST 생성 -fetchModuleDependencies()

```
return this.fetchModule(
```

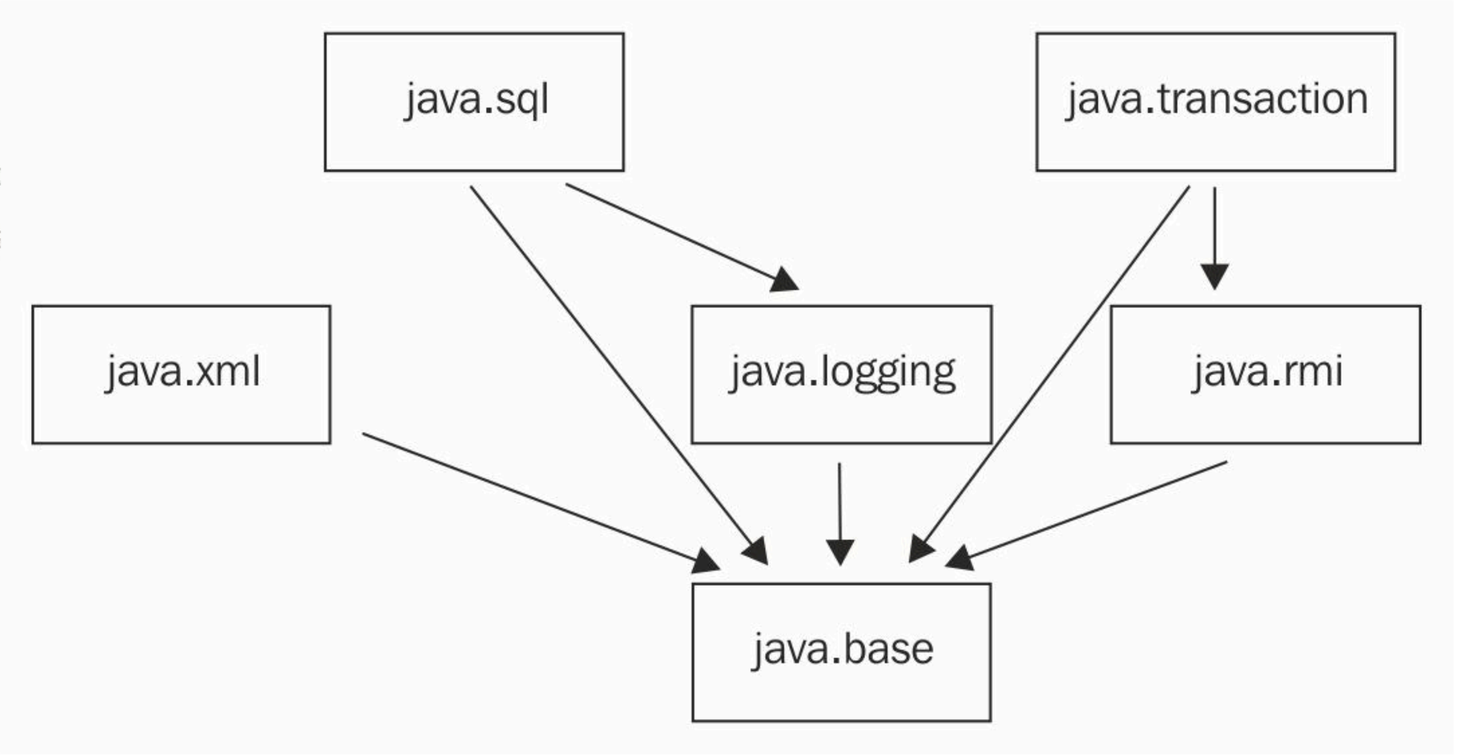
↓

```
if (!isPreload) {
  await this.fetchModuleDependencies(module, ...resolveDependencyPromises);
```

↓

```
await Promise.all([
  this.fetchStaticDependencies(module, resc
  this.fetchDynamicDependencies(module, res
]);
```

의존성 관계 로딩



⇒ 모듈 간 의존 관계

1

모듈 로딩 & AST 생성
- generateModuleGraph()

```
timeStart('generate module graph', 2);
await this.generateModuleGraph();
timeEnd('generate module graph', 2);
```

2

모듈 정렬 & 참조 바인딩
- sortModules()

```
timeStart('sort and bind modules', 2);
this.phase = BuildPhase.ANALYSE;
this.sortModules();
timeEnd('sort and bind modules', 2);
```

3

Tree Shaking 진행
- includeStatements()

```
timeStart('mark included statements', 2);
this.includeStatements();
timeEnd('mark included statements', 2);
```

2. 모듈 정렬 & 참조 바인딩 -sortModules()

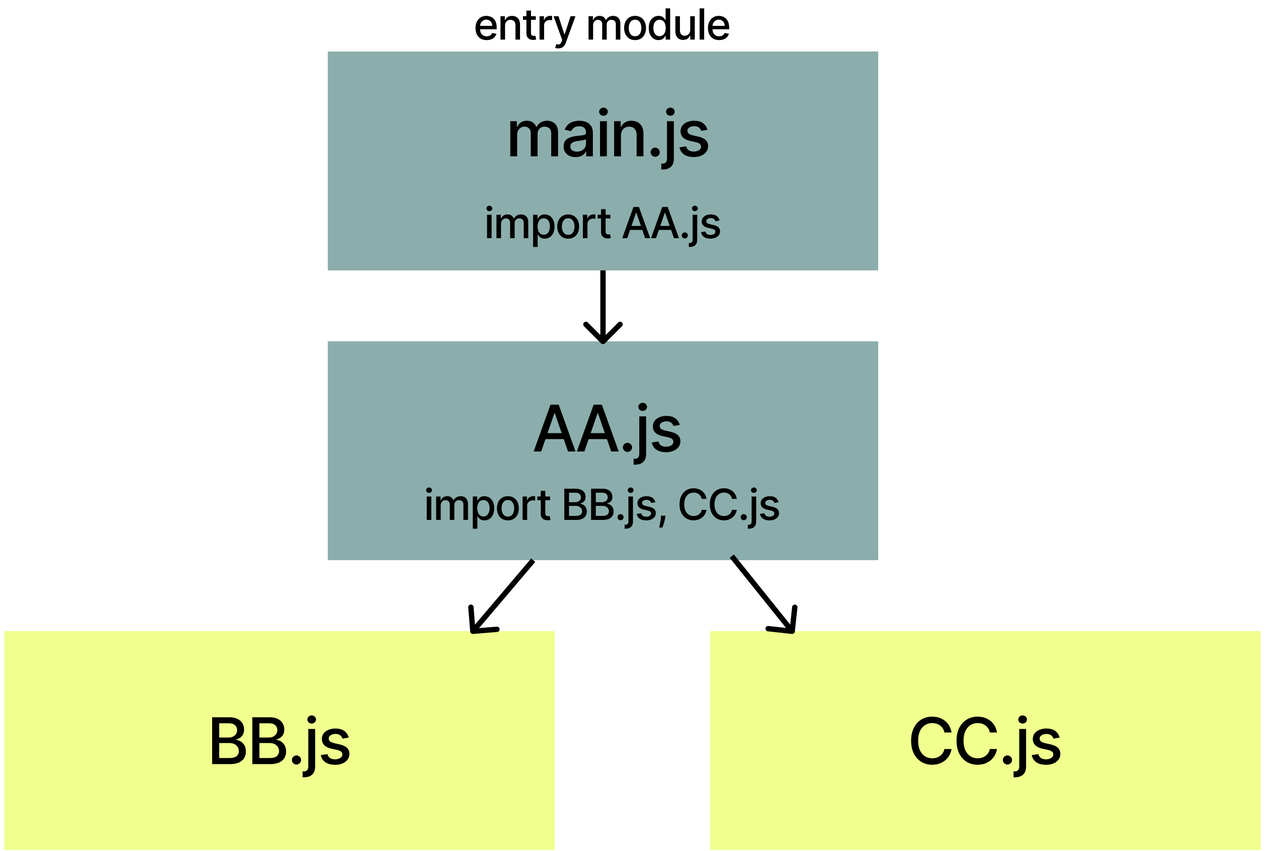
정렬

```
private sortModules(): void {
  const { orderedModules, cyclePaths } = analyseModuleExecution(this.entryModules);
  for (const cyclePath of cyclePaths) {
    this.options.onLog(LOGLEVEL_WARN, logCircularDependency(cyclePath));
  }
  this.modules = orderedModules;
  for (const module of this.modules) {
    module.bindReferences();
  }
  this.warnForMissingExports();
}
```

바인딩

2. 모듈 정렬 & 참조 바인딩 -analyzeModuleExecution()

→ DFS



⇒ 모듈의 실행 순서

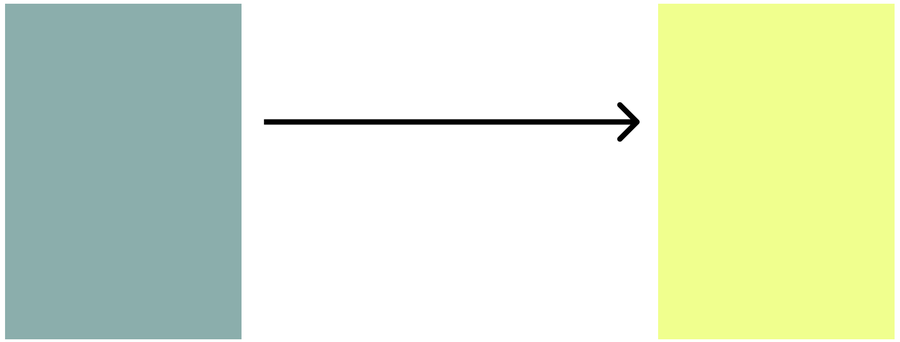
2. 모듈 정렬 & 참조 바인딩 -bindReferences()

```
bindReferences(): void {
  this.ast!.bind();
}
```

→ 구체적으로 어떤 변수/함수를 참조하는지

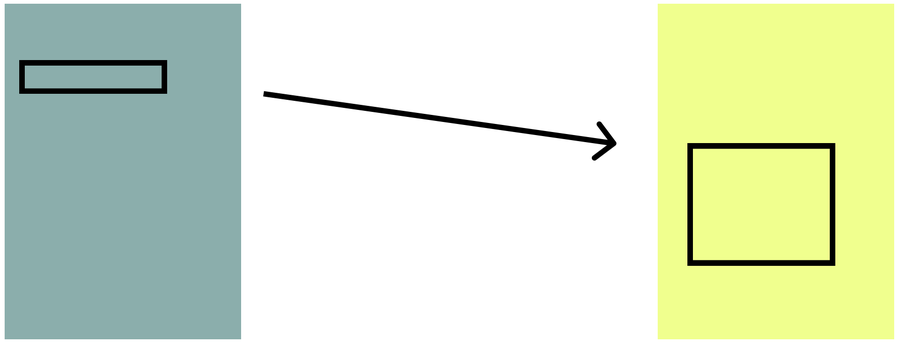
fetchModuleDependencies

모듈-모듈 사이의 의존성



bindReferences

구체적으로 어디에 연결되어 있는지



⇒ 구체적 참조 정보

1

모듈 로딩 & AST 생성
- generateModuleGraph()

```
timeStart('generate module graph', 2);
await this.generateModuleGraph();
timeEnd('generate module graph', 2);
```

2

모듈 정렬 & 참조 바인딩
- sortModules()

```
timeStart('sort and bind modules', 2);
this.phase = BuildPhase.ANALYSE;
this.sortModules();
timeEnd('sort and bind modules', 2);
```

3

Tree Shaking 진행
- includeStatements()

```
timeStart('mark included statements', 2);
this.includeStatements();
timeEnd('mark included statements', 2);
```

3. Tree Shaking 진행

```
private includeStatements(): void {
  const entryModules = [...this.entryModules, ...this.implicitEntryModules];
  for (const module of entryModules) {
    markModuleAndImpureDependenciesAsExecuted(module);
  }

  for (const module of this.modules) {
    if (module.isExecuted) {
      module.hasTreeShakingPassStarted = true;
      if (module.info.moduleSideEffects === 'no-treeshake') {
        module.includeAllInBundle();
      } else {
        module.include();
      }
```

3. Tree Shaking 진행 -include()

src/Modules.ts

```
include(): void {
  const context = createInclusionContext();
  if (this.ast!.shouldBeIncluded(context)) this.ast!.include(context, false);
}
```

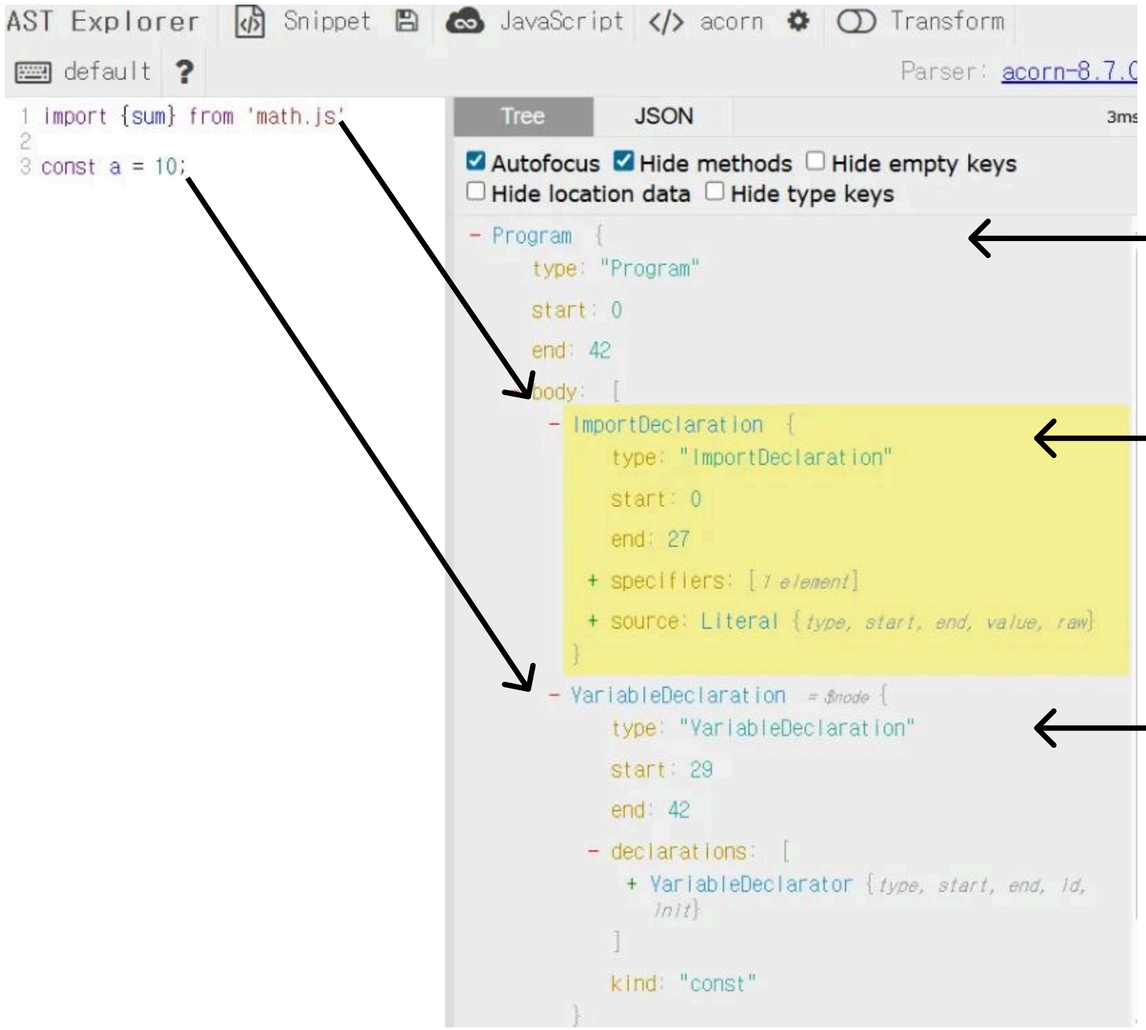
필요한 모듈(노드)면 → include 진행

src/ast/nodes/Program.ts

```
include(context: InclusionContext, includeChildrenRecursively: IncludeChildren): void {
        this.included = true;
        for (const node of this.body) {
                if (includeChildrenRecursively || node.shouldBeIncluded(context)) {
                        node.include(context, includeChildrenRecursively);
                }
        }
}
```

내부 노드들에 대해서 재귀적으로 include 진행

3. Tree Shaking 진행 -include()



include()

include()

include()

https://astexplorer.net/

5. 정리

총정리

번들러의 필요성과 사용 이유

Tree Shaking의 구체적인 원리

Tree Shaking의 구체적인 구현 방식

감사합니다!

Q & A