Life of an ESM in Node.js

Joyee Cheung

Agenda

- What happens when Node.js is asked to import a module
- How does ESM loading differ in Node.js, bundlers and browsers
- New Node.js/transpilers/language features in different stages of ESM loading
- Some common misunderstanding about how ESM works

- Useful if you are building JavaScript tooling/framework, want to maximize your module loading performance, or need to debug mysterious errors
  - AI can get confused – they get trained on out-of-date information on the Internet!

About me

- Igalia & Bloomberg
- Node.js TSC & V8 committer
- Contributing to some module loading improvements in Node.js recently
- @joyeecheung on GitHub
- Find the slides of this talk at

Which one hits first when we run node `main.mjs`?

```
// main.mjs
import './throws-error.mjs';             // A: module with `throw new Error(...)`
import { notExported } from './mod.mjs'; // B: module that doesn't export notExported
import './non-existent.mjs';             // C: module that doesn't exist on disk
import './contains-syntax-errors.mjs';   // D: module that contains a syntax error
```

Which one hits first when we run node `main.mjs`?

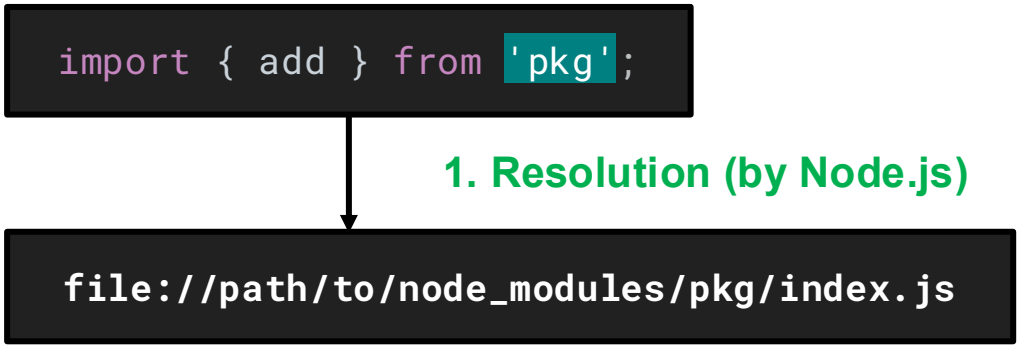
```
// main.mjs
import './throws-error.mjs';
import { notExported } from './mod.mjs';
import './non-existent.mjs';              // 🔥
import './contains-syntax-errors.mjs';
```

```
$ node main.mjs
# Error [ERR_MODULE_NOT_FOUND]: Cannot find module 'non-existent.mjs' imported from test.mjs
```

What would Node.js do when it sees import ‘pkg’

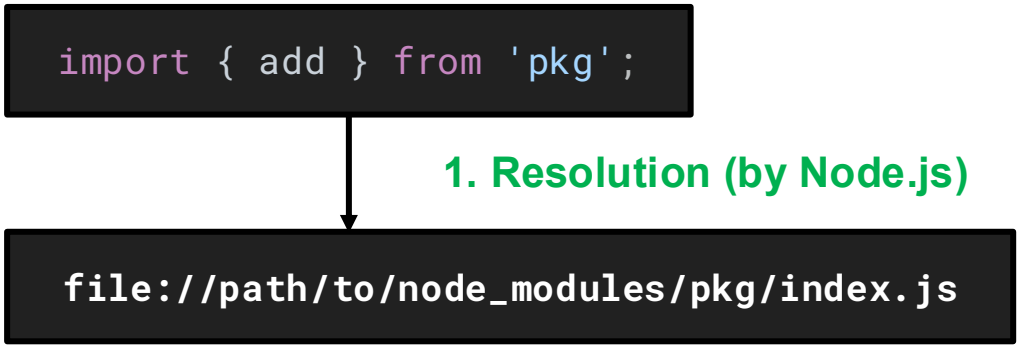
```
import { add } from 'pkg';
```

1. Resolution: module request to URL



Resolution: resolving a module request to a URL.

1. Resolution: module request to URL



If the module can't be resolved e.g. doesn't exist on disk, Node.js throws an error

```
import './non-existent.mjs'; 🔥
```

1. Resolution: environment differences

Node.js

If the specifier is not a path, looks into `node_modules`

```
import { add } from 'pkg'; // Look up path/to/node_modules/pkg/index.js
```

Consults import/export conditions in `package.json`

```
"exports": "entry.js", // Can lead to path/to/pkg/entry.js
```

If the specifier is a path, it must be exact; does not probe directories or omitted extensions for ESM (only done for CommonJS)

```
import { foo } from './dir';           // ERR_UNSUPPORTED_DIR_IMPORT
import { bar } from './extensionless'; // ERR_MODULE_NOT_FOUND
```

1. Resolution: environment differences

Bundlers (when bundling for Node.js)

If the specifier is not a path, look into `node_modules`

```
import { add } from 'pkg'; // Look up path/to/node_modules/pkg/index.js
```

Consult import/export conditions in `package.json` (and have bundler-specific conditions)

```
"exports": "entry.js", // Can lead to path/to/pkg/entry.js
```

If the specifier is a path, most bundlers **DO** probe directories or omitted extensions for ESM (faux-ESM)

```
import { foo } from './dir'; // resolve to ./dir/index.js
import { bar } from './extensionless'; // resolve to ./extensionless.js
```

1. Resolution: environment differences

Browsers

No concept of `node_modules` or `package.json`

```
import { add } from 'pkg'; // Failed to resolve module specifier "pkg"
```

If the specifier is a path, it must be exact

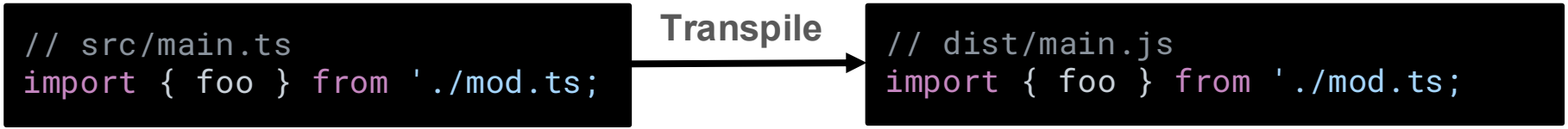
```
import { foo } from './dir';           // 404
import { bar } from './extensionless'; // 404
```

Supports import maps

```
<script type="importmap">...</script>
```

1. Resolution: Tip for Node.js TypeScript interop

- Node.js supports TypeScript via type-stripping from v22
  - Just type erasure, no transform e.g. converting ESM to CommonJS
- For optimal performance and compatibility
  - Use type stripping *at development time*
  - Transpile to JavaScript before distributing or running in environments where startup time is important.



1. Resolution: Tip for Node.js TypeScript interop

- When transpling TypeScript to JavaScript for distribution using e.g. tsc, typically `src/**/*.ts` are emitted as `dist/**/*.js`.
- Relative import to `.ts` then point to wrong files after transpilation.

```
- src/
  - main.ts
  - mod.ts
```

```
- dist/
  - main.js
  - mod.js
```

```ts
// src/main.ts
import { foo } from './mod.ts;
```

Transpile

```ts
// dist/main.js
import { foo } from './mod.ts';
```

`mod.ts` is not in the `./dist` directory 💣

1. Resolution: Tip for Node.js TypeScript interop

- Some users solve this by omitting the extensions and rely on bundlers to convert ESM to CommonJS
- Bundlers probe .ts for extensionless imports at build time resolution, Node.js ComomnJS probe .js for extensionless requires at run time resolution

- src/
  - main.ts
  - mod.ts

- dist/
  - main.js
  - mod.js

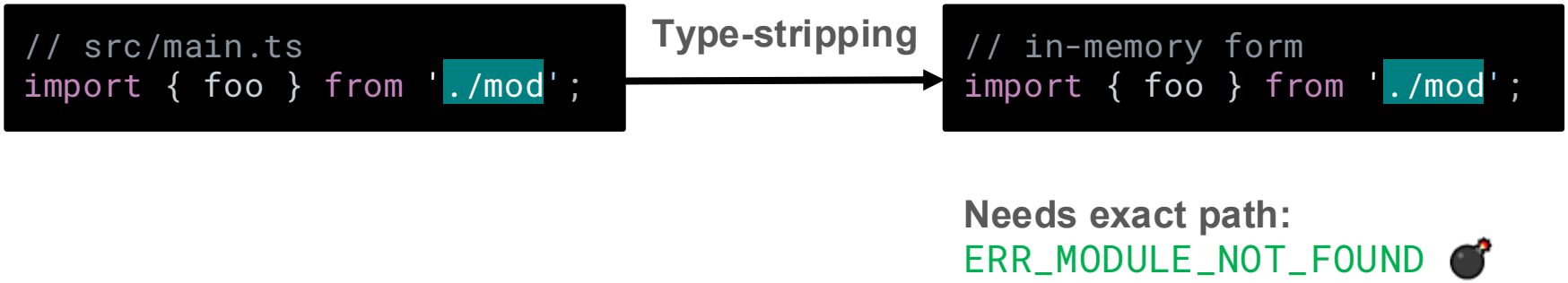
```ts
// src/main.ts
import { foo } from './mod';
```

Transpile

```ts
// dist/main.js
const $mod = require('./mod');
const foo = $mod.foo;
```

1. Resolution: Tip for Node.js TypeScript interop

- But extensionless import doesn't work in Node.js with type-stripping, which leaves ESM as ESM, and ESM in Node.js doesn't probe .js – it requires exact path



1. Resolution: Tip for Node.js TypeScript interop

- Use `rewriteRelativeImportExtensions` in `tsconfig.json` during transpilation
- tsc (>= 5.7) and compatible transpilers would rewrite `.ts` to `.js` in the module specifier during transpilation.

```
// tsconfig.json
{
  "rewriteRelativeImportExtensions": true // rewrites .ts to .js
}
```

- When run by Node.js in development, `.ts` loads `.ts` in ESM
- When transpiled to be run in Node.js, `.js` loads (transpiled) `.js` in ESM

```
// src/main.ts
import { foo } from './mod.ts';
```

Transpile

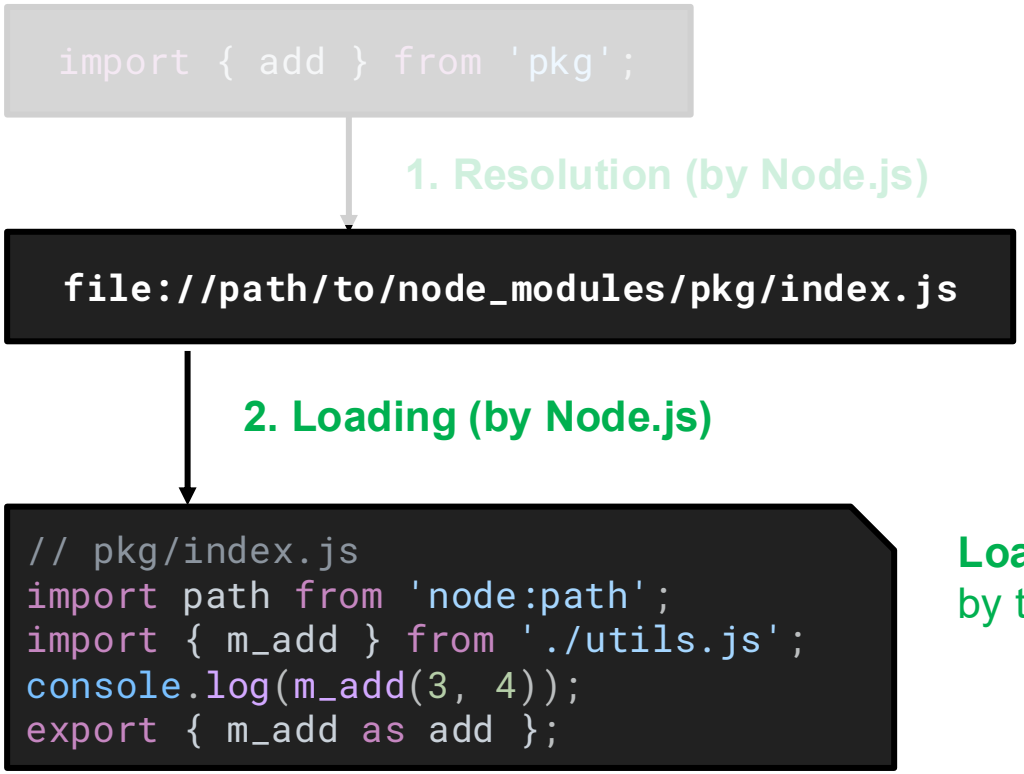
```
// dist/main.js
import { foo } from './mod.js';
```

1. Resolution: Node.js module customization hooks

```
import module from 'node:module';
// Node.js >= 22, release candidate
module.registerHooks({
  resolve(specifier, context, nextResolve) { // Resolve hook
    // Redirect `import ... From '.../db.js'` to
    // `import ... from 'db.mock.js'` for testing purposes
    if (specifier.endsWith('db.js')) {
      const replaced = specifier.replace(/db\.js$/, 'db.mock.js');
     return nextResolve(replaced, context);
    }

    // No customization needed, pass through
    return nextResolve(specifier, context);
  }
});
```

2. Loading: URL to source code



Loading: fetching the source code identified by the resolved URL

2. Loading: URL to source code

```
import { add } from 'pkg';
```

1. Resolution (by Node.js)

```
file://path/to/node_modules/pkg/index.js
```

2. Loading (by Node.js)

```
// pkg/index.js
import path from 'node:path';
import { m_add } from './utils.js';
console.log(m_add(3, 4));
export { m_add as add };
```

If e.g. the file is not readable due to lack of permissions, Node.js can throw here

```
import './no-read-access.mjs'; 🔥
```

2. Loading: Environment differences

	In-thread loading?	Async / streaming?	Origin policy
Node.js	In-thread (except in deprecated paths)	Synchronous (from fs/memory, to reduce races)	
Bundlers (bundling for Node.js)	Varies: some in-thread, some off-thread	Varies: some async, some sync	
Browser	Off-thread	Off-thread & async (over network)	

2. Loading: Environment differences

	In-thread loading?	Async / streaming?	Origin policy
Node.js	In-thread (except in deprecated paths)	Synchronous (from fs/memory, to reduce races)	Opaque origin (only `file://` & `data://` & `node:`); relies on file system permissions
Bundlers (bundling for Node.js)	Varies: some in-thread, some off-thread	Varies: some async, some sync	No concept of origin
Browser	Off-thread	Streamed & async (over network)	Same-origin policy + CORS

2. Loading: Node.js module customization hooks

```
import module from 'node:module';
// Node.js >=22, release candidate, runs in-thread and synchronously
module.registerHooks({
  load(url, context, nextLoad) { // Load hook
    // If the URL resolves to a custom protocol, return preset custom
    // source code.
    if (url.startsWith('custom-protocol:')) {
      return { source: customSource, format: 'module', shortCircuit: true }
    }

    // No customization needed, pass through
    return nextLoad(url, context, context);
  }
});
```

3. Parsing

```
import { add } from 'pkg';
```

1. Resolution (by Node.js)

```
file://path/to/node_modules/pkg/index.js
```

2. Loading (by Node.js)

```
// pkg/index.js
import path from 'node:path';
import { m_add } from './utils.js';
console.log(m_add(3, 4));
export { m_add as add };
```

Checks syntax errors, produces AST (at least for top level code)

```
import './contains-syntax-errors.mjs';
```

AST + metadata

3. Parsing (Node.js driving V8)

3. Parsing: Environment differences

	Module format detection	In thread/streamed?
Node.js	• Check extensions (`.mjs`/`.cjs` etc.) • Check `package.json` "`type`" field • file syntax (since v20, adds a cost but improves UX)	
Bundlers (bundling for Node.js)	• Check extensions (`.mjs`/`.cjs` etc.) • Check `package.json` "`type`" field • Some check file syntax	
Browser	• `<script type=..>` tag • "`type`" passed to workers • Transitive modules assumed to be ESM	

3. Parsing: Environment differences

	Module format detection	In thread/streamed?
Node.js	• Check extensions (`.mjs`/`.cjs` etc.) • Check `package.json` "`type`" field • file syntax (since v20, adds a cost but improves UX)	In thread, not streamed (need to interop with JS hooks and existing patching)
Bundlers (bundling for Node.js)	• Check extensions (`.mjs`/`.cjs` etc.) • Check `package.json` "`type`" field • Some check file syntax	Depends on the bundler
Browser	• `<script type=..>` tag • "`type`" passed to workers • Transitive modules assumed to be ESM	Off thread and streamed

4. Compilation

```
import { add } from 'pkg';
```

1. Resolution (by Node.js)

```
file://path/to/node_modules/pkg/index.js
```

2. Loading (by Node.js)

```
// pkg/index.js
import path from 'node:path';
import { m_add } from './utils.js';
console.log(m_add(3, 4));
export { m_add as add };
```

3. Parsing (by V8)

AST + metadata

4. Compilation (by V8)

Bytecode + metadata

Module main.js

Compilation: generates bytecode

4. Compilation: Environment differences

Browsers

- Bytecode cached to disk, reused when the same module is loaded again without changes & browser is not updated
- Like parsing, compilation in browsers is typically off-thread
- Inner functions in a module are usually only *preparsed* (to look for syntax errors); their bytecode is not compiled until the function is invoked for the first time

```
const constant = 1; // fully parsed, bytecode emitted when module is compiled
function add() { // only pre-parsed, not compiled until first invocation.
  return a + b;
}
export { add, constant };
```

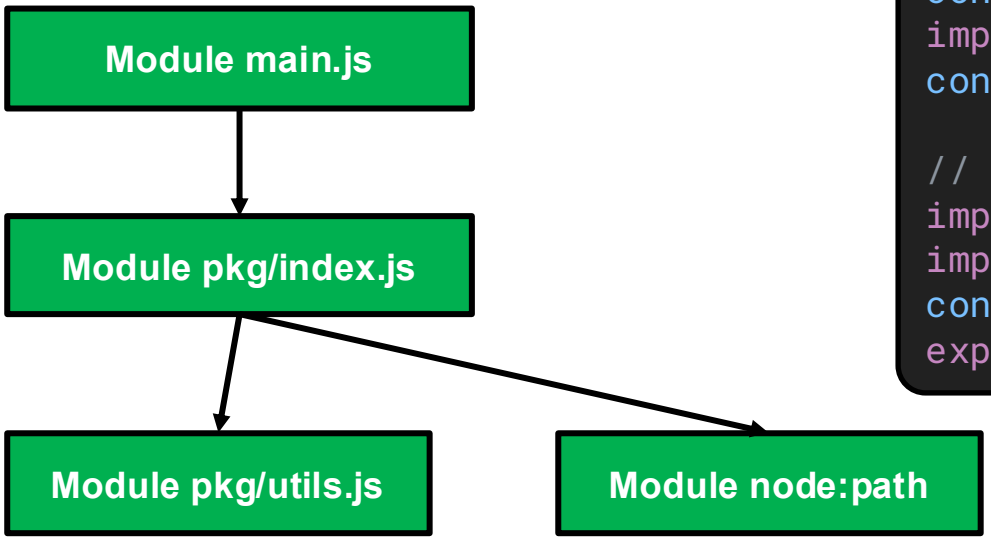
4. Compilation: Environment differences

Node.js

- On-disk compilation cache supported since v22, currently opt-in
  - `NODE_COMPILE_CACHE=dir / module.enableCompileCache()`
  - Only reused when module is not changed & run in the same Node.js version
  - Node.js Single-Executable Applications can “bundle” the bytecode via “useCodeCache”
- Like parsing, compilation in Node.js happens in-thread
- Inner functions are also only *preparsed*
  - May be configurable in the future.

```
const constant = 1; // fully parsed, bytecode emitted when module is compiled
function add() { // only pre-parsed, not compiled until first invocation.
  return a + b;
}
export { add, constant };
```

5. Handling dependencies



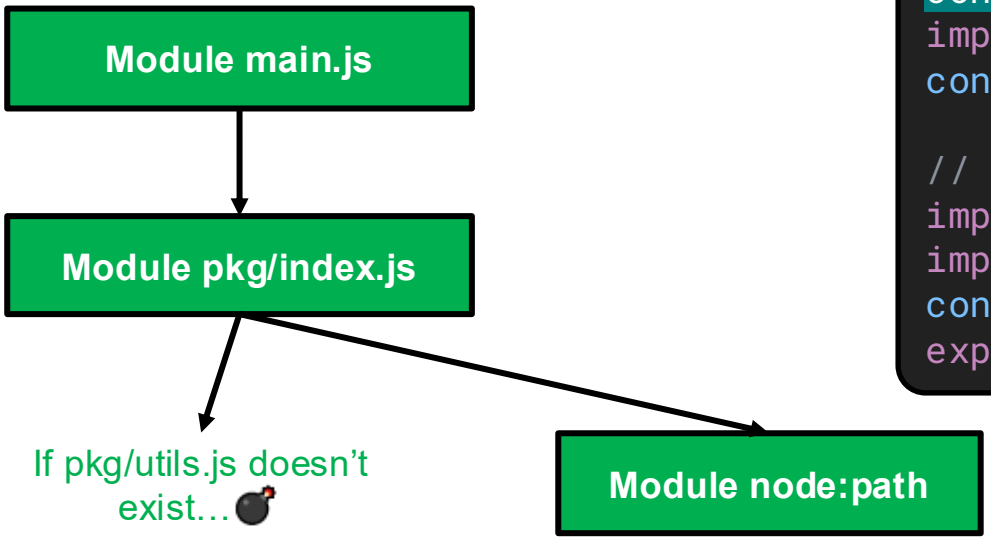

```
// main.js
console.log('main');
import { add } from 'pkg';
console.log(add(1, 2));

// pkg/index.js
import path from 'node:path';
import { m_add } from './utils.js';
console.log(m_add(3, 4));
export { m_add as add };
```

5. Handling dependencies (by Node.js)

Iterate through the module requests, resolve, load, parse and compile the dependency graph transitively.

5. Handling dependencies



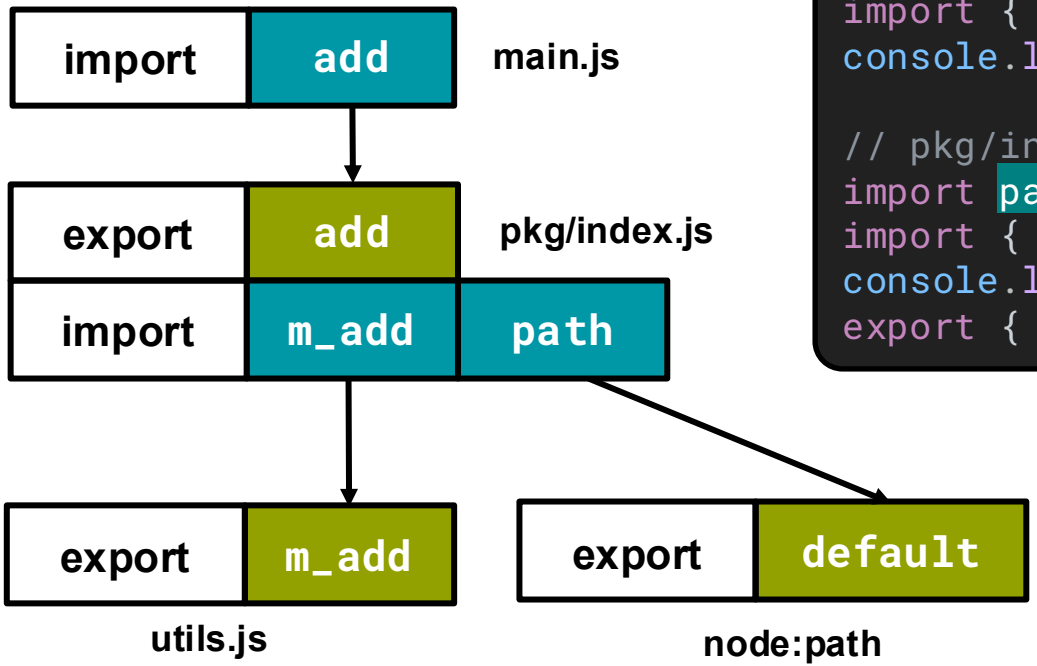

```
// main.js
console.log('main'); // won't execute
import { add } from 'pkg';
console.log(add(1, 2));

// pkg/index.js
import path from 'node:path';
import { m_add } from './utils.js'; 💣
console.log(m_add(3, 4));
export { m_add as add };
```

5. Handling dependencies (by Node.js)

- If any of the transitive dependencies can't be resolved, loaded or parsed, Node.js will throw an error before any module code in the graph is executed
- Import statements are hoisted per the language design. They will be handled first even if they are placed after the executable code.

6. Instantiation



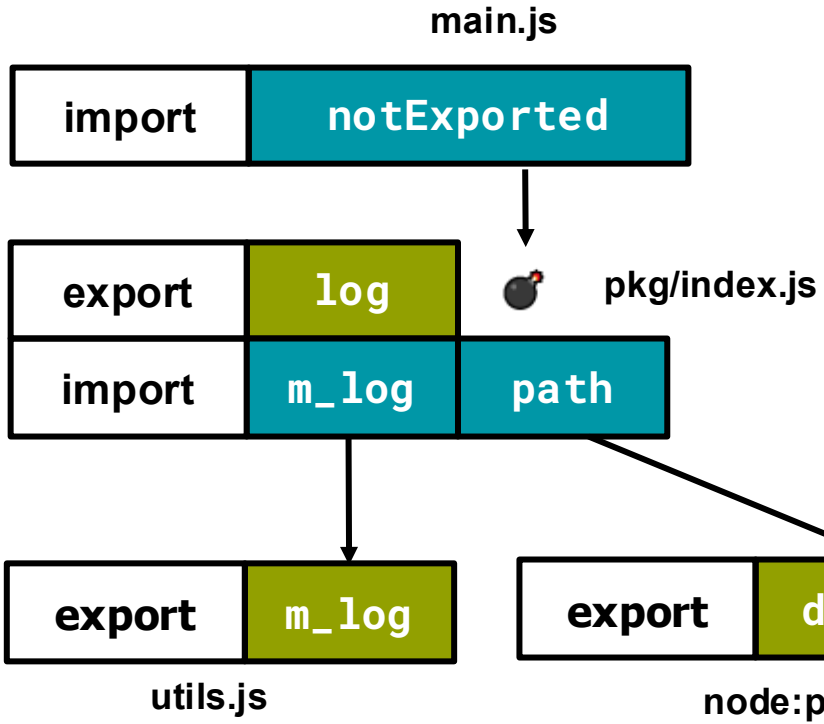

```
// main.js
console.log('main');
import { add } from 'pkg';
console.log(add(1, 2));

// pkg/index.js
import path from 'node:path';
import { m_add } from './utils.js';
console.log(m_add(3, 4));
export { m_add as add };
```

- Try to match **import/exports** and checks **mismatches**
- This has been renamed to “Linking” in ECMA262, but still called “Instantiation” in Node.js/V8

6. Instantiation (by V8)

6. Instantiation




```
// main.js
console.log('main'); // won't execute
import { notExported } from 'pkg';
```

- If import/exports are not matched, V8 throws an error
- Again, this happens *before* execution

6. Instantiation (by V8)

Customizing parsing -> instantiation

- More special in ESM, cannot share primitives with CommonJS via `module.registerHooks()`
- `vm.Modules` API (`--experimental-vm-modules`)
  - Used by a few testing frameworks for lighter weight isolation
  - Has several bugs and caveats stemmed from design issues, experimental since 2017 (e.g. this was why e.g. Jest ESM support is still experimental now)
  - Stalled until recent renewed momentum from contributors
  - No entity "owns" the development of Node.js – if no contributor volunteers to make progress, progress won't happen by itself

Customizing parsing -> instantiation: revamped API

- Allows registering a more powerful loader in the execution context
- Avoid issues in memory management, aborting, error reporting, etc. from the design

```
const loader = new SourceTextModuleLoader({ // Draft API!!
  getRequestedModules(requests, context, parent) {
    // Customize dependency handling
    return requests.map((request) => {
      const deps = SourceTextModule(
        source, { url, /* parsing & compilation options */ });
      const subDeps = this.getRequestedModules(deps.requests);
      deps.link(subDeps);
      return deps;
    });
  }
});
const [root  = loader.getRequestedModules([{ url: rootURL }]);
root.instantiate(/* instantiation options */);
```

Check out the discussion issue!

7. Evaluation

```
// main.js
console.log('main'); // won't execute
import './throws-error.mjs';

// throws-error.mjs
throw new Error('child error');
```

7. Evaluation (by V8)

- Dependencies are executed *before* dependents
- If dependencies throw an error at the top level, dependents won't be executed.

7. Evaluation

```
// main.js
// Node.js can do something before the entire module graph is run
import { add } from 'pkg';
// Node.js can't do any thing between the code in pkg is run and this line
console.log(add(1, 2));

// pkg/index.js
import path from 'node:path';
import { m_add } from './utils.js';
console.log(m_add(3, 4));
export { m_add as add };
```

7. Evaluation (by V8)

- Node.js only gets to kick off the evaluation of the root module. V8 controls the evaluation of the transitive submodules.
- This is why there's no ESM evaluate hook

7. Evaluation: Surprises

- ESM namespaces reflect mutations from within the exporter, but cannot be mutated from the importer.
- The mutability is mandated by the language and controlled by JS engine.
- The embedder is considered external to it and cannot mutate the namespace either

```
// mod.mjs
export let count = 0;

export function inc() {
  count++;
}
```

```
// main.mjs
import * as ns from "./mod.js";
console.log(ns.count); // 0
ns.count = 5;
console.log(ns.count); // 0
ns.inc();
console.log(ns.count); // 1
```

7. Evaluation: `import.meta`

- Lazily initialized the first-time `import.meta` is accessed
- In HTML standard, available in most environments
  - `import.meta.url`
  - `import.meta.resolve()`
- WinterTC (ECMA TC55) maintains a `import.meta` registry
- Node.js:
  - >= 20.11.0 : `import.meta.filename, import.meta.dirname`
  - >= 22.18.0 : `import.meta.main (boolean)`
  - AI still likes to derive them from `import.meta.url` or just struggles with entry point detection..

Evaluation: New language features

Source Phase Import (stage 3) – user-controlled WASM evaluation since Node.js 24

```
import source wasmModule from './add.wasm';
// synchronous instantiation, can customize imports passed to wasm
const instance = new WebAssembly.Instance(wasmModule, imports);
```

Deferred Evaluation (WIP, stage 3, requires `--js-defer-import-eval`)

```
import defer * as ns from './heavy.js';
// Initialization in heavy.js will not be run until first property access
function rarelyUsedAPI() {
  return ns.expensivelyInitializedThing; // Trigger heavy.js evaluation
}
```

Which one hits first when we run node `main.mjs`?

```
// main.mjs
import './throws-error.mjs';              // A
import { notExported } from './mod.mjs'; // B
import './non-existent.mjs';              // C: Resolution Error, hits 1st 🔥
import './contains-syntax-errors.mjs';    // D
```

```
$ node main.mjs

# Error [ERR_MODULE_NOT_FOUND]: Cannot find module 'non-existent.mjs' imported from test.mjs
```

Which one hits first when we run node `main.mjs`?

```
// main.mjs
import './throws-error.mjs';              // A
import { notExported } from './mod.mjs'; // B
import './existent.mjs';                  // Fixed
import './contains-syntax-errors.mjs';    // D: Parsing Error, 2nd 🔥
```

```
$ node main.mjs
# SyntaxError: ...
```

Which one hits first when we run node `main.mjs`?

```
// main.mjs
import './throws-error.mjs';              // A
import { notExported } from './mod.mjs'; // B: Instantiation Error,3rd 🔥
import './existent.mjs';                  // Fixed
import './no-syntax-errors.mjs';          // Fixed
```

```
$ node main.mjs

# SyntaxError: The requested module './mod.mjs' does not provide an export named
'notExported'
```

Which one hits first when we run node `main.mjs`?

```
// main.mjs
import './throws-error.mjs';              // A: Evaluation Error, 4th 🔥
import { exported } from './mod.mjs';     // Fixed
import './existent.mjs';                  // Fixed
import './no-syntax-errors.mjs';          // Fixed
```

```
$ node main.mjs
# Error: error thrown by throws-error.mjs
```

What keeps ESM alive?

The language specification asks the engine/embedder to ensure idempotency of module loading.

- If this operation is called multiple times with two (*referrer*, *moduleRequest*) pairs such that:
  - the first *referrer* is the same as the second *referrer*;
  - ModuleRequestsEqual(the first *moduleRequest*, the second *moduleRequest*) is **true**;

  and it performs FinishLoadingImportedModule(*referrer*, *moduleRequest*, *payload*, *result*) where *result* is a normal completion, then it must perform FinishLoadingImportedModule(*referrer*, *moduleRequest*, *payload*, *result*) with the same *result* each time.

What keeps ESM alive?

```mjs
// mod.mjs
export let count = 0;
export function inc() {
  count++;
}
```

```mjs
// main.js
{
  const a = await import("./mod.js");
  console.log(a.count); // 0
  a.inc();
  console.log(a.count); // 1
}

// Even after a long time when a is completely gone
{
  const a = await import("./mod.js");
  console.log(a.count); // 1: a must still be alive!
}
```

- Node.js maintains various internal caches to ensure the language requirement is met
- Other circumstantial references: inter-module references, V8 compilation caches

What keeps ESM alive?

- These internal references make tools/frameworks prone to leaks when implementing ESM hot reloading in Node.js
- WIP: `module.clearCache()` API
  - Not clearing all the caches yet, but will make unloading of ESM easier in the future.

What keeps ESM alive?

- Delegate the idempotency requirement to user land for tools/frameworks
- Use cache busting URLs with search parameters like in browsers: if the cleared module is never loaded again, it's practically never violating the language requirement

```
{
  const a = await import('./mod.js?v=1'); // To meet spec requirements, use different URL
  console.log(a.count); // 0
  a.inc();
  console.log(a.count); // 1
}

module.clearCache('./mod.js?v=1', {  // Draft API!!
  parentURL: import.meta.url, resolver: 'import',
}); // The first module will be unloaded. Users need to make sure it won't be loaded again.

{
  const a = await import('./mod.js?v=2');
  console.log(a.count); // 0: a is loaded again
}
```

Thanks