Module ~~fragments~~ declarations for Stage 2

https://github.com/tc39/proposal-module-declarations

Useful links

- **GitHub repository:**

  https://github.com/tc39/proposal-module-declarations

- **Specification:**

  https://tc39.es/proposal-module-declarations

- **Examples:**

  https://github.com/tc39/proposal-module-declarations/tree/main/examples

Module declarations

```
module math {
  export function add(x, y) { … }
}
//  ↓↓↓↓↓
const math = module {
  export function add(x, y) { … }
};
```

Module declarations are the declaration form of module expressions.

Module declarations

```
module math {
  export function add(x, y) { … }
}
```

```
import { add } from math;
console.log("2 + 3 is", add(2, 3));
```

They can be imported statically, either by the module where they are declared ...

Module declarations

```
module math {
  export function add(x, y) { … }
}

module main {
  import { add } from math;
  console.log("2 + 3 is", add(2, 3));
}
```

They can be imported statically, either by the module where they are declared …

… or by other module declarations or expressions.

Motivation

Bundling

Modern JavaScript applications have thousands of modules, and loading them one by one has performance problems:

- **Loading waterfall**: transitive dependencies get discovered later
- **Per-resource overhead**: one HTTP request per module, or one filesystem access
- **Sub-optimal compression**: compression algorithms work on a per-file basis

Bundlers optimize this by merging multiple modules in a single one

Bundling

Properly emulating the semantics of ES modules is hard

Merge all the top-level scopes

- Conflicting variables need to be renamed
- Need to manually create namespace objects
- Serial execution of modules with top-level await

Wrap every module in a function

- Hard to preserve cross-module hoisting and live bindings
- Runtime cost of a runner to link all the functions together

Bundling

```
// main.js
import { render } from "./ui-library";
import { onClick } from "./events";

onClick("#my-button", () => render());
```

```
// ui-library.js
export function render() { /* … */ }
```

```
// events.js
export function onClick(el, handler) {}
```

```
// main.bundled.js

import { render } from uiLibrary;
import { onClick } from events;

onClick("#my-button", () => render());

module uiLibrary {
  export function render() { /* … */ }
}

module events {
  export function onClick(el, handler)
  { /* … */ }
}
```

Bundling

NOTE: There is a parallel effort for web bundles that can contain generic resources

https://github.com/WICG/bundle-preloading

Module declarations and web bundles can coexist:

- Web bundles are at the HTTP request/response level, while module declarations are within a single JavaScript module
- Web apps have considerably more JavaScript modules than other resources, and module declarations are a solution optimized for that
- Web bundles can contain modules that contain multiple module declarations

Changes since the last presentation

We now have specification text!

https://tc39.es/proposal-module-declarations

- All the major syntax and semantics are defined, but there are some minor TODOs left.

- It depends on the Module expressions proposal, which introduces `Module` objects and expands `import()` to support them.
- It's a diff on top of the modules host hooks refactor (tc39/ecma262#2905), which allows ECMA-262 to better decide when to delegate module loading to the host.

Modules use identifiers instead of URL fragments

```
- module "#math" {
+ module math {
    export function add(x, y) { … }
  }
```

They introduce a binding in the local scope, whose value is a `Module` object.

They are internal to ECMA-262, and do not take part in the host module map.

Module declarations can appear at any level

```
module outer { module nested { … } }
{
  module inBlock { import outer; } // ok
}
import outer;    // ok
import nested;   // linking error
import inBlock; // linking error
```

They are block scoped, and follow common visibility rules: nested scopes can see module declarations from outer scopes.

They are hoisted to the beginning of the block, like strinct-mode function declarations.

Module declarations are private by default, and can be exported

```
module internal { … }

export module numbers {
  export { add } from internal;
}

export default module numbers { … }
```

Previously, every module declaration was automatically public.

The new semantics match bundlers, that can carefully choose which entrypoints to expose.

Module declarations can be imported from other modules

```
- import { sum } from "./math#numbers";

+ import { numbers } from "./math";
+ import { sum } from numbers;
```

As module declarations can be exported, they can also be imported.

Resolution of imported modules happens during the modules loading phase.

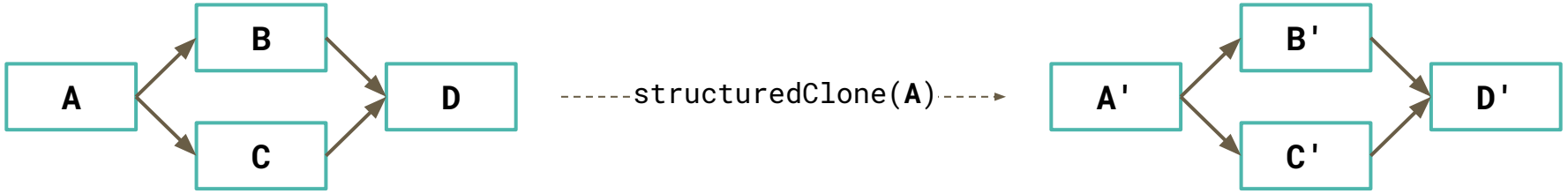
HTML integration

HTML integration

Module declarations inherit all the design decisions from module expressions:

- importing them doesn't go through HTML algorithms
- they inherit `import.meta.url` from the outer module
- they can be structured cloned and passed to workers

When structured cloning, the complete graph of pre-linked declarations is cloned de-duplicating modules imported multiple times.



Host loading hook interaction

```
// main.js
import { A } from "1.js";
import A;
import B;
module B {
  import "2.js";
}

// 1.js
export module A {
  import "3.js"
}
import "4.js"
```

main.LoadRequestedModules():

- **HostLoadImportedModule**("1.js") ⏳
- Mark loading of A as blocked on "1.js"
- InnerModuleLoading(B):
  - **HostLoadImportedModule**("2.js") ⏳

- When "1.js" has been loaded:
  - Process blocked imports:
    - Load(A):
      - **HostLoadImportedModule**("3.js") ⏳
    - InnerModuleLoading("1.js"):
      - **HostLoadImportedModule**("4.js") ⏳

Comments? Questions?

Stage 2?