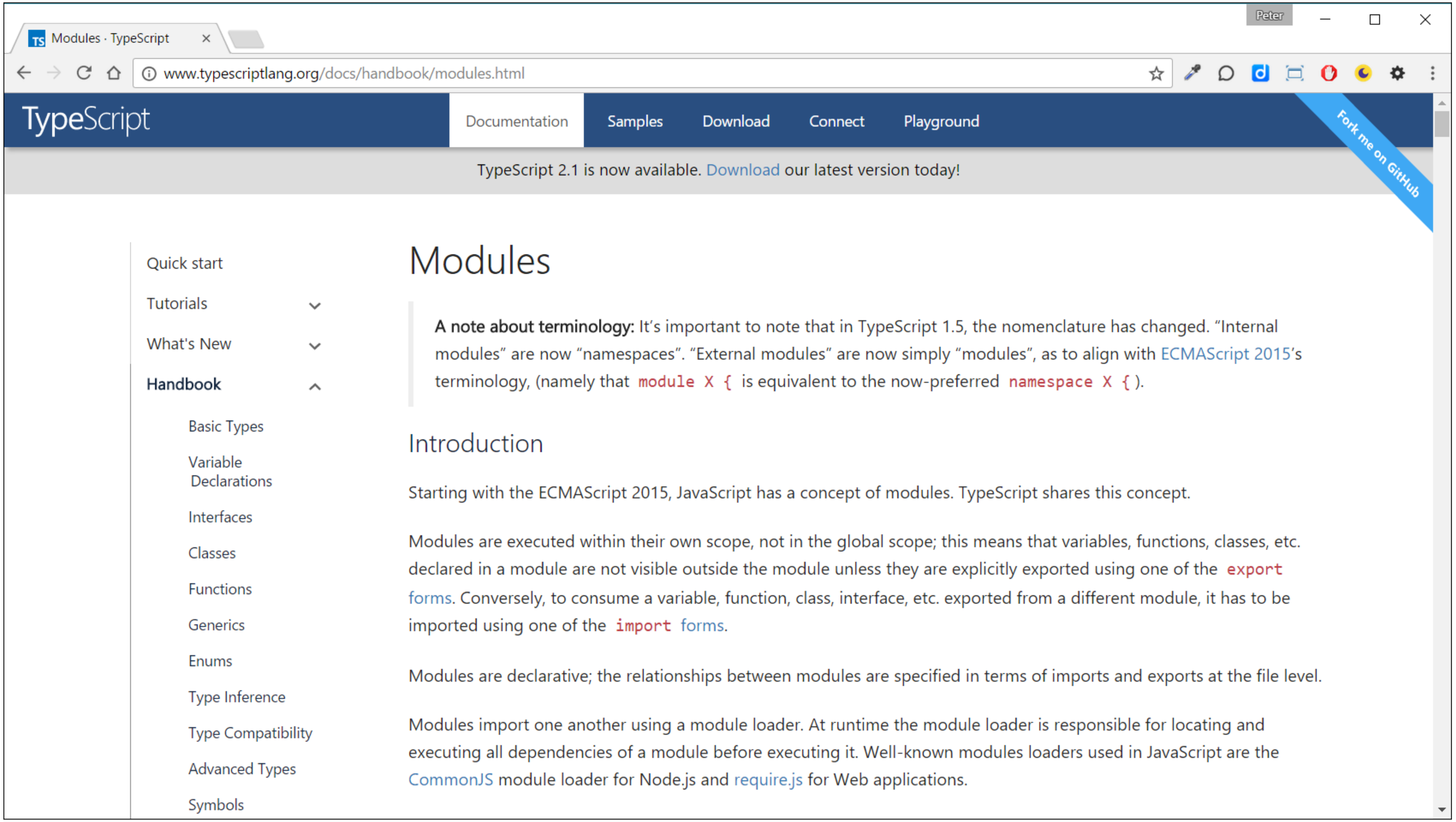
Training TypeScript

TypeScript Modules and Namespaces

Peter Kassenaar

info@kassenaar.com



TypeScript 2.1 is now available. Download our latest version today!

Modules

> **A note about terminology:** It's important to note that in TypeScript 1.5, the nomenclature has changed. "Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with ECMAScript 2015's terminology, (namely that `module X {` is equivalent to the now-preferred `namespace X {`).

Introduction

Starting with the ECMAScript 2015, JavaScript has a concept of modules. TypeScript shares this concept.

Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the `export` forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the `import` forms.

Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.

Modules import one another using a module loader. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it. Well-known modules loaders used in JavaScript are the CommonJS module loader for Node.js and require.js for Web applications.

http://www.typescriptlang.org/docs/handbook/

Modules vs. Namespaces

- Mostly a semantic difference, all about *terminology*
- Commonly used as:
  - Modules: a file.
  - Namespace: grouping related code together with the `namespace` keyword
    - Namespaces can span multiple files!
    - Contents of the file are concatenated in the same namespace
- Example: folder `../20-modules`

On Modules

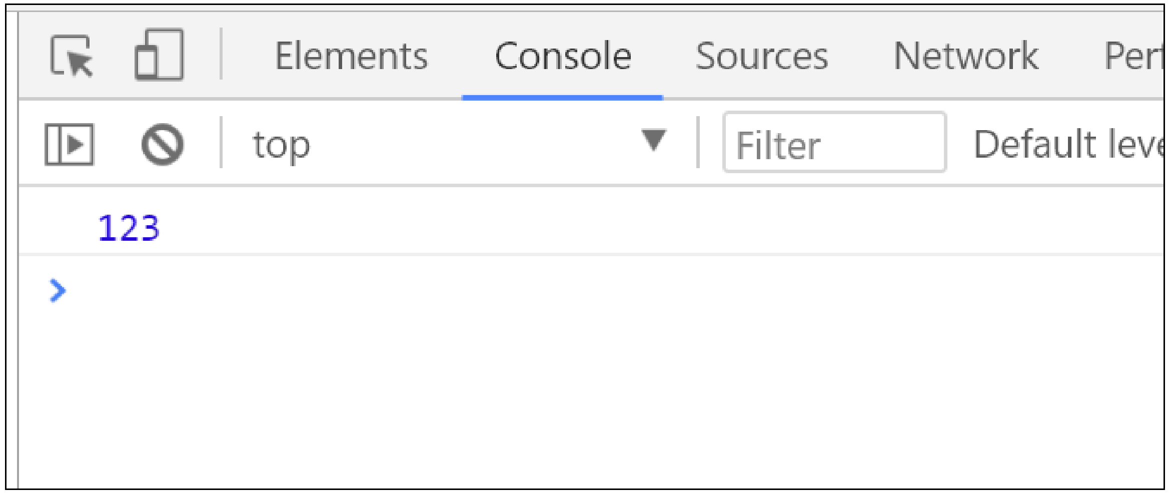
- Modules are executed within *their own scope*, not in the global scope.
  - When a file has in `import` or `export` keyword, it is considered a module
- Modules are *declarative*
  - Relations between modules are specified in the paths in `import/export` statements
- Modules import one another using a *module loader*
  - CommonJS for Node.js
  - Require.js for webapplications

Global Module

By default, TypeScript uses a global namespace (!)

```
// foo.ts
let foo = 123;
```

```
// bar.ts – valid!
var bar = foo;
console.log(bar);
```



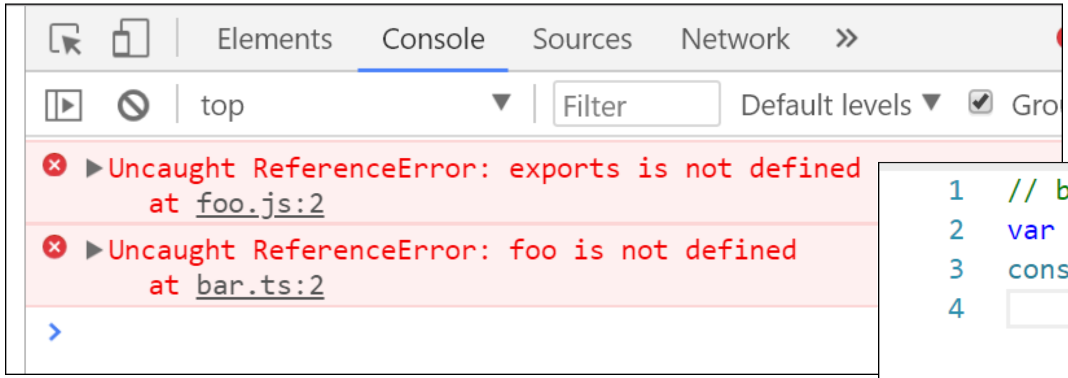
```
123
>
```

File Modules aka *External Modules*

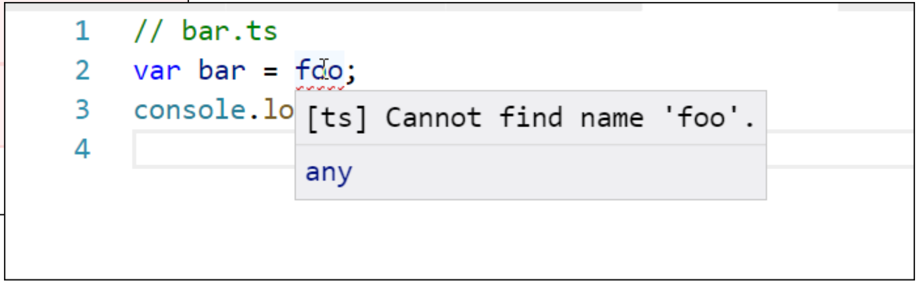
With `import` or `export` keyword, TypeScript (or the module loader) creates a *local scope* within that file

```
// foo.ts
export let foo = 123;
```

```
// bar.ts
var bar = foo;
console.log(bar);
```



```
Uncaught ReferenceError: exports is not defined
    at foo.js:2
Uncaught ReferenceError: foo is not defined
    at bar.ts:2
```



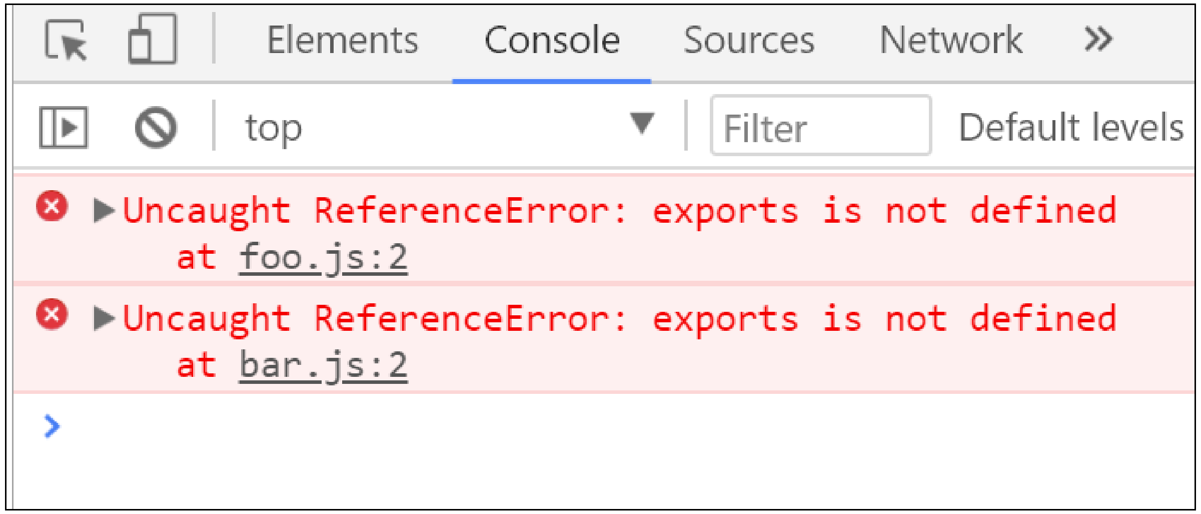
```
// bar.ts
var bar = foo;
console.lo
```

```
[ts] Cannot find name 'foo'.
any
```

Modules need to be imported

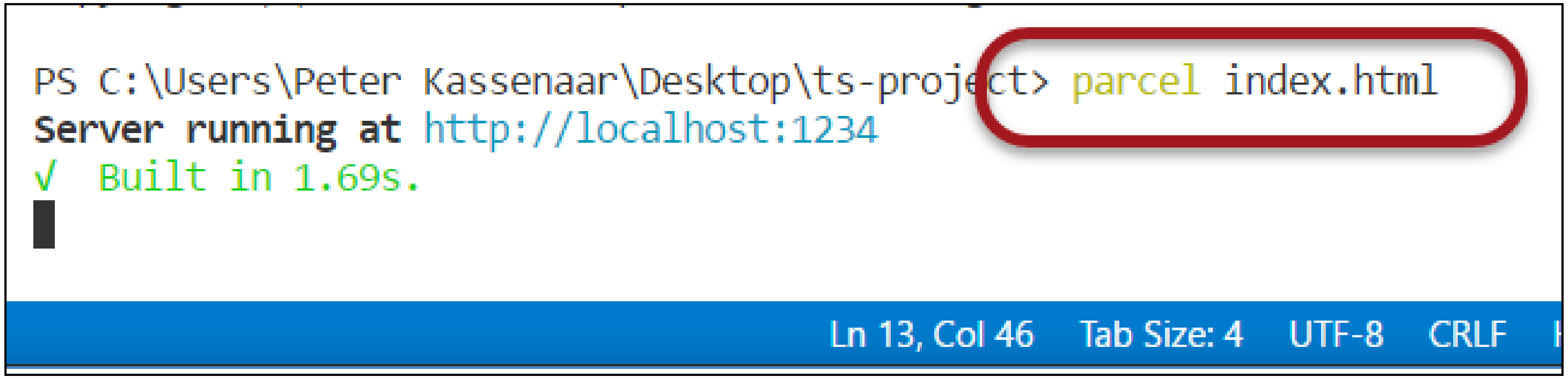
```
// bar.ts
import { foo } from './foo';
var bar = foo;
console.log(bar);
```

…now we need a module loader, like Parcel or WebPack



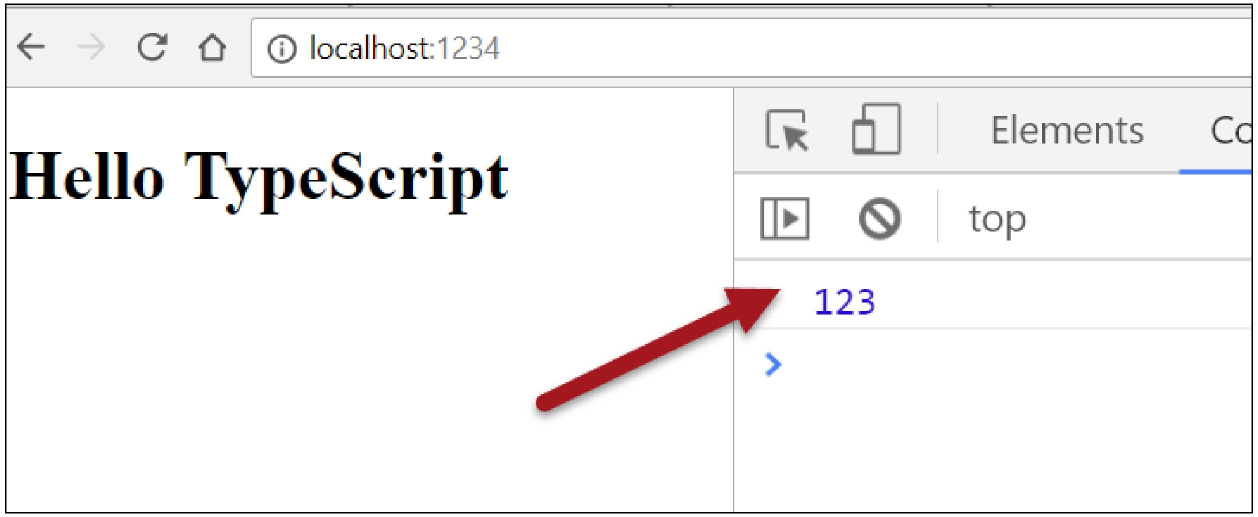
```
Uncaught ReferenceError: exports is not defined
    at foo.js:2
Uncaught ReferenceError: exports is not defined
    at bar.js:2
```

Using Parcel

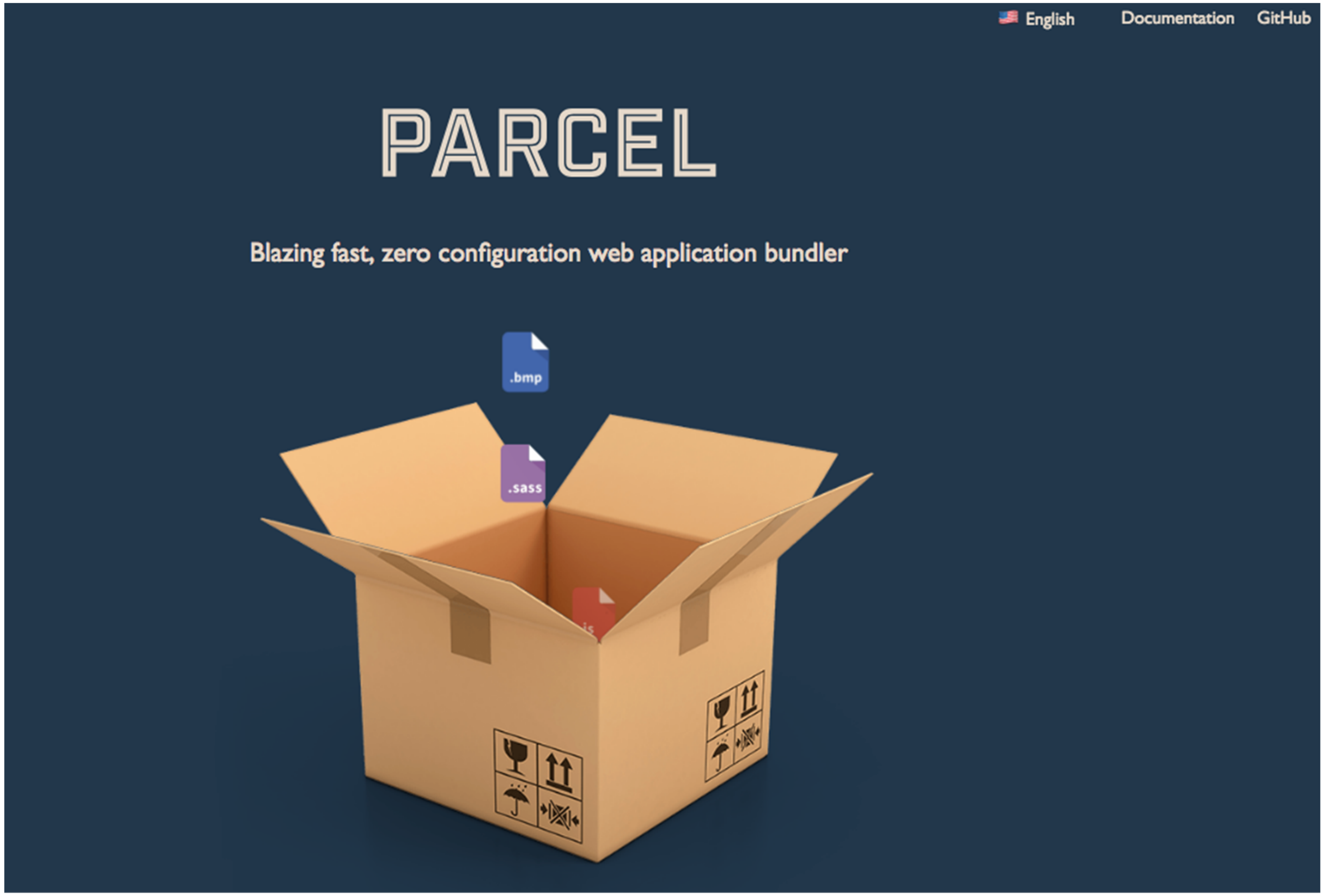


```
PS C:\Users\Peter Kassenaar\Desktop\ts-project> parcel index.html
Server running at http://localhost:1234
√  Built in 1.69s.
```

```html
<body>
    <h1>Hello TypeScript</h1>
    <script src="bar.js"></script>
</body>
```



https://parceljs.org/



```
npm install parcel-bundler -g
```

“Using an import in `bar.ts` not only allows you to bring in stuff from other files, but also marks `bar.ts` as a module and therefore, declarations in `bar.ts` don't pollute the global namespace either.”

External Modules

- On CommonJS, AMD, UMD and others...
- Lots of module systems.
  - Unclear!
  - Inconsistent
  - TypeScript generates different JavaScript, based upon the `module` option!
- Different kinds of modules
  - `AMD` – do not use anymore. Was browser only.
  - `SystemJS` – outdated. Superseded by ES Modules
  - `ES Modules` – work in progress. Not widely supported yet
  - `commonjs`: use this one in `tsconfig.json`.

```
"compilerOptions": {
    "module": "commonjs",
    …
}
```

ES Module Syntax

Using the `import` and `export` keywords

```
// foo.ts
export let foo = 123;
export type someType = {
  foo: string;
};
```

```
// bar.ts
import { foo, someType } from './foo';
```

```
// bar.ts - import all with alias
import * as foo from './foo';
var bar = foo.foo;
```

Export default

Export using `export default`

- before a variable (no `let` / `const` / `var` needed)
- before a function
- before a class

Modules can have only one (1) `default export`!

```
// some var
let someVar: number = 123
export default someVar;
```

```
// bar.ts - import default exported variable
import someCustomName from './foo'; // === 123
```

globals.d.ts

- NOT recommended
- Used for putting interfaces/types in the global namespace to have them available everywhere in your project.
- Better approach: use file modules instead, as discussed before.

```typescript
// globals.d.ts
interface globalPerson {
   name: string;
}
```

```typescript
// bar.ts
// Interface globalPerson lives in global namespace.
// No import required.
let person: globalPerson = {
   name: 'Peter '
};
```

Namespaces

- Namespaces are a convenient syntax around the common IIFE-pattern used in JavaScript:

```
var Utils;
(function (Utils) {
  function utility() {
    return 123;
  }
})(Utils || (Utils = {}));
```

Using the namespace keyword

In TypeScript mainly used to group related functions. Like:

```typescript
namespace Utils {
  export function log(msg: string) {
    console.log(msg);
  }
  export function error(msg: string) {
    console.error(msg);
  }
  //.. other stuff
}
```

```typescript
// Using the Utils namespace - no import required
Utils.log('This is a logging message');
Utils.error('This is a logging message');
```

Verdict

- Don’t use namespaces, unless you have to.
- Use external (file based) modules instead.

Workshop

- Create new files, like the `foo.ts/bar.ts` examples in this presentation
  - Make sure functions are available in the global namespace by default.
- Create a module, using the `import/export` keywords
  - Make sure functions and variables are NOT available in global namespace anymore
  - Install Parcel JS
  - Use WebPack or Parcel as a Module Loader – make it work!
- Create a `globals.d.ts` file with some data
  - make sure it is available everywhere
- Create a namespace, spanning multiple files (this can be done!).
  - Make sure it works by using it in `bar.ts`