

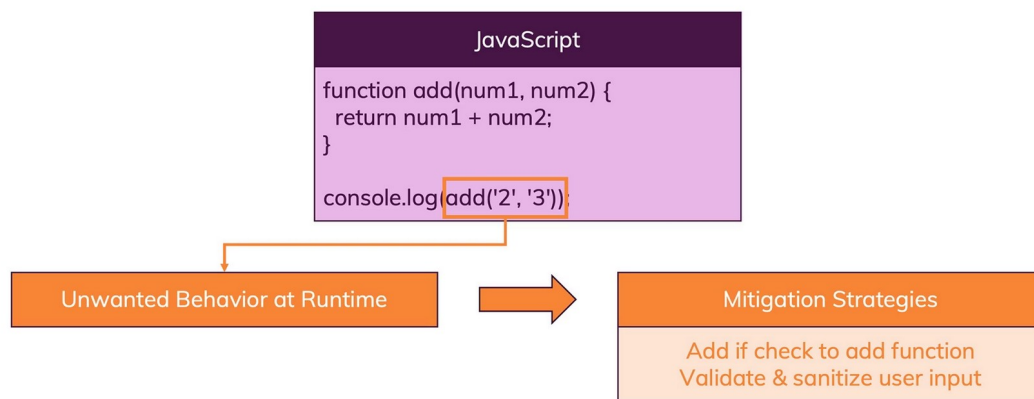
What is TypeScript

TypeScript is a super set of JavaScript.

TypeScript builds on top of JavaScript. First, you write the TypeScript code. Then, you compile the TypeScript code into plain JavaScript code.

TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

Why TypeScript?



Installation

TypeScript can be installed through three installation routes depending on how you intend to use it:

- NPM module
- NuGet package
- Visual Studio Extension

If you are using Node.js, you want the npm version.

```
sudo npm install -g typescript
```

Install lite server

Lightweight development only node server that serves a web app, opens it in the browser, refreshes when html or javascript change, injects CSS changes using sockets.

```
npm init
```

```
npm i --save-dev lite-server
```

package.json

```
"scripts": {  
  "start": "lite-server"  
}
```

Types

Core Types

number	1, 5.3, -10	All numbers, no differentiation between integers or floats
string	'Hi', "Hi", `Hi`	All text values
boolean	true, false	Just these two, no "truthy" or "falsy" values
object	{age: 30}	Any JavaScript object, more specific types (type of object) are possible
Array	[1, 2, 3]	Any JavaScript array, type can be flexible or strict (regarding the element types)
Tuple	[1, 2]	Added by TypeScript: Fixed-length array
Enum	enum { NEW, OLD }	Added by TypeScript: Automatically enumerated global constant identifiers

```
/*  
!: That's the non-null assertion operator.  
It is a way to tell the compiler "this expression cannot  
be null or undefined here,  
so don't complain about the possibility of it being null  
or undefined."  
Sometimes the type checker is unable to make that  
determination itself.  
But it's better to use if (number) {} and do your stuff  
there.  
*/  
let number = document.getElementById('number')! as  
HTMLInputElement  
  
function print (price: number, message: string) {  
  return message + price  
}
```

<https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html#non-null-assertion-operator>

! suppress errors like this: `TS2531: Object is possibly 'null'.`

You can compile file with `tsc script.ts` command

If we don't pass correct type to `print()` function; compiler complains about that but it doesn't stop compilation

`error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.`

Type Inference

In TypeScript, there are several places where type inference is used to provide type information when there is no explicit type annotation. For example, in this code

```
let x = 3 // let x: number
```

When a type inference is made from several expressions, the types of those expressions are used to calculate a “best common type”

```
let x = [0, 1, null]; // let x: (number | null)[ ]
```

```
let price: number; // When we don't initialize variable immediately it's a good practice to declare the type
```

Tuple

```
let person: {  
  name: string,  
  age: number,  
  hobbies: string[],  
  role: [number, string] // Tuple: Fixed Length & type array  
} = {  
  name: 'Twitch',  
  age: 20,  
  hobbies: ['Reading', 'Hiking'],  
  role: [2, 'Admin']  
}
```

```
}

// Type inference feature also can detect it. because we
// initialize variable. So it's redundant and can get rid of
// that
```

Enum

```
enum Role { ADMIN, AUTHOR, USER }
// It starts from 0; if you wanna change it simply assign
// any value

// enum Role { ADMIN = 'ADMIN', AUTHOR = 2, USER }

let person = {
  role: Role.ADMIN
}
```

Union Types

```
function combine(a: number | string, b: number | string) {
  // We can suppress typescript errors with annotation
  // mentioned below but probably if you accept multiple types,
  // you end up with multiple if statement to checkout type of
  // parameter and do something different based on that.
  // More information in 'Type Guard' and 'Function Overload'
  // section

  // @ts-ignore
  return a + b;
}
```

Literals

```
function increment(size: 'bigInt' | 'smallInt') {
  // Only these two strings are valid for ts
}
```

Alias

```
type sizes = 'bigInt' | 'smallInt'
```

```
function increment(size: sizes) {  
  //  
}
```

Return Type

```
function increment(size: sizes): void {  
  //  
}
```

Function Type

```
let concatenate: Function  
concatenate = combine // We can store function pointers  
inside variable  
  
// It tells ts that 'concatenate' will be a function but  
that's not right and we need to be precise about  
parameters and return type.  
  
let concatenate: (a: number, b: number) => number
```

Unknown

```
let input: unknown // It's better than 'any' and is more  
restricted
```

Never

```
function throwError(message: string, code: number): never  
{  
  throw { message, code: 500 }  
}  
  
// We can use void but never is best practice here,  
because it crashes the code and never produce any result.
```

Configuration

Watch Mode

```
tsc app.ts --watch // or -w
```

But what about when we are working with multiple entry files?

```
tsc --init // Initialize the project
```

It creates `tsconfig.json` file. Right now you can run `tsc` command and it will compile all of the `.ts` files at once.

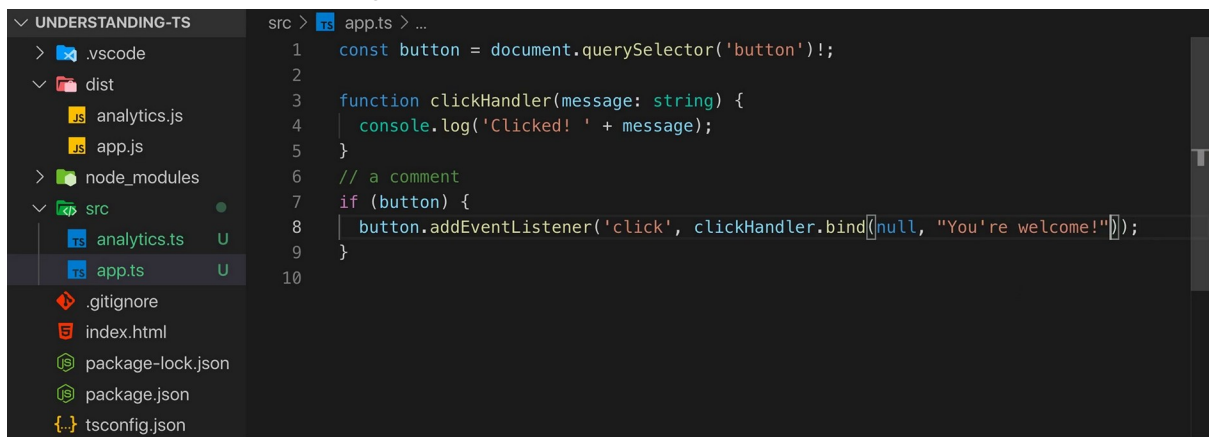
`tsconfig.json`

`tsconfig.json`

```
{
  "compilerOptions": {
    "target": "es5"
    // transpile the code to that version.
    // For example 'let' and 'const' will be converted to 'var'.
    // because they are not supported in es5
    "lib": [
      "DOM",
      "ES6",
      "DOM.Iterable",
      "ScriptHost"
    ],
    // Exact default lib's settings for es6
    // That's the reason why document, Map() etc is recognized in
    // ts file
    "sourceMap": true
    // The JavaScript sources executed by the browser are
    // often transformed in some way from the original sources
    // created by a developer.
    // In these situations, it's much easier to debug the
    // original source, rather than the source in the transformed
    // state that the browser has downloaded. A source map is a
    // file that maps from the transformed source to the original
    // source, enabling the browser to reconstruct the original
    // source and present the reconstructed original in the
    // debugger.
    "outDir": "./dist"
```

```
"rootDir": "./src"
"removeComments": true
"noEmitOnError": true
// Don't create output file if have compiler's issues.
"noImplicitAny": true
// for example in `function log(data) {}`
// because parameter 'data' implicitly has an 'any' type.
"strictNullChecks": true
"strictBindCallApply": true
"noUnusedLocals": true
"noUnusedParameters": true
"noImplicitReturns": true
// Something like 'return statement is missing' in PHP
// If you have at least one case that your function does
// return something then you have to make sure you return
// something in all cases
"jsx": "preserve" // React .jsx files
"allowJs": true // Get errors from .js files
"checkJs": true // Enable error reporting in type-
checked JavaScript files
"declaration": true
"declarationMap": true
// Generate .d.ts files (It's an advanced topic for
// library maintainers)
"noEmit": true
// Ts compiler will check your files and report
// potential errors without actually creating output file.
},
"exclude": [
  "node_modules" // Tell compiler to don't touch ts
  files inside node_modules folder
],
"include": [
  "src" // Opposite behavior of exclude
],
"files": [
  "app.ts"
]
}
```

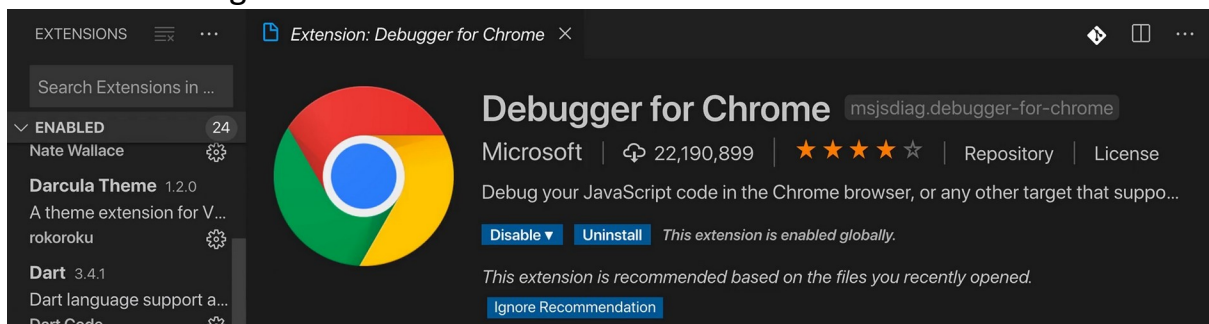
strictBindCallApply



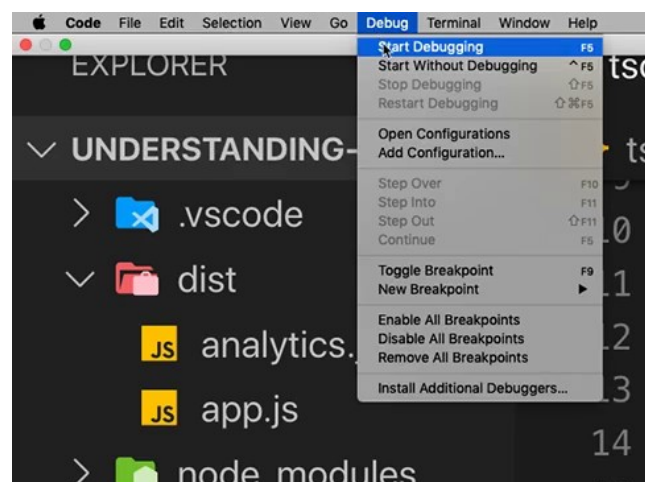
```
src > app.ts > ...
1  const button = document.querySelector('button')!;
2
3  function clickHandler(message: string) {
4    console.log('Clicked! ' + message);
5  }
6  // a comment
7  if (button) {
8    button.addEventListener('click', clickHandler.bind(null, "You're welcome!"));
9  }
10
```

Debugging in VS Code

Install following extension:



Make sure "SourceMap" is enable, set your breakpoints then click on "Start Debugging"



Next-generation JavaScript & TypeScript

<http://kangax.github.io/compat-table/es6/>

let and const

<https://stackoverflow.com/a/11444416/5827477>

`var` and `let` are both used for variable declaration in JavaScript but the difference between them is that `var` is function scoped and `let` is block scoped.

While variables declared with `var` keyword are **hoisted** (initialized with undefined before the code is run) which means they are accessible in their enclosing scope even before they are declared

Arrow function

```
function add(a, b) {  
    return a + b  
}  
  
// Using arrow function:  
const add = (a, b) => {  
    return a + b  
}  
  
// Shorter syntax if you have one expression:  
const add = (a, b) => a + b  
  
// May remove parenthesis if you have one parameter:  
// Must use () => if we have any.  
const print = message => console.log(message)
```

Default Function Parameter

```
const add = (a: number, b: number = 1) => {  
    return a + b  
}
```

Spread Operator

```
const hobbies = ['Football', 'Hiking']  
const leasure = ['Study']  
  
leasure.push(hobbies); // => ['Study', ['Football',  
'Hiking']] ] and that's not what we want.
```

```
// You may use forEach loop and push them one by one. But
WHY!!!
leasure.push(...hobbies); // => ['Study', 'Football',
'Hiking' ]

// Or
const leasure = ['Study', ...hobbies]
```

You may wonder how we could push to a constant?

Arrays are objects and objects are referenced values, when we push, we change memory but not the address.

```
const person = {
  name: 'Max',
  age: 30
}

const copy = person // We are copying the pointer at this
'person' object in memory into to 'copy' const
```

In order to really copy object:

```
const copy = { ...person}
```

Rest Parameter

```
// It will merge all incoming arguments into an array
function add (...numbers: number[]) {
  return numbers.reduce((result, value) => {
    return result + value
  }, 0)
}
```

Object Destructuring

```
const [hobby1, hobby2, ...remaining] = hobbies;
const { name, age } = person;
```

Classes and Interfaces

Classes

```
class Department {
    private readonly id: number
    // 'Readonly' properties can be accessed outside the class
    // but their value can not be changed after initialization
    // (they either need to be initialized at declaration or
    // inside constructor)

    constructor(id: number, private name: string) {
        this.id = id
    }
    // For each parameter which has access modifier
    // a property of the same name is created
    // and the value will store there.

    // Method
    describe(): void {
        console.log('Department: ' + this.name)
    }
}
```

It will transpile to **Constructor Function** in oldest versions of JS

```
var Department = (function () {
    function Department(n) {
        this.name = n
    }
    return Department
})();
```

this keyword

```
const department = new Department('IT')
department.describe() // IT

const office = { describe: department.describe }
office.describe() // undefined

// Because it's looking for property called 'name' inside
// object
// You just need to add this property like
// const office = { name: 'DUMMY', describe:
// department.describe }
```

Inheritance

```
class ITDepartment extends Department {  
    constructor(id: number) {  
        super(id, 'IT');  
        // This will call constructor of base class from inside  
        // the subclass  
        // Equivalent to parent::construct() in PHP  
    }  
}
```

Setters and Getters

```
get thumbnail() {  
    // department.thumbnail will trigger that  
}  
  
set thumbnail(value) {  
    // department.thumbnail = 'something' will trigger  
    // that  
}  
  
// Like accessor and mutator in Laravel
```

Static methods and properties

```
static key = 'DEPARTMENTS'  
  
// We can get it (or call it) without creating new  
// instance of class by Department.key
```

Abstract

Abstract classes are mainly for inheritance where other classes may derive from them. We cannot create an instance of an abstract class.

An abstract class typically includes one or more abstract methods or property declarations. The class which extends the abstract class must define all the abstract methods.

```
abstract class Office {  
    abstract describe(): void;  
}
```

Singleton

```
class Container {
  private static instance: Container
  private constructor() {}

  static getInstance() {

    // Because we are in an static method; We can access to
    // any other 'static' stuff using 'this'
    if (this.instance) {
      return this.instance
    }

    this.instance = new Container()
    return this.instance
  }
}
```

Interfaces

Interface is a structure that defines the contract in your application. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface.

```
interface Person {
  name: string,
  age?: number // equivalent to number|undefined
  greet?(message: string): void
  // ? optional props or methods
}

// It's equivalent to: type Person = { ... }

let max: Person;

max = {
  name: 'Max',
  age: 30,
  greet(message: string) {
    console.log(message)
  }
}
```

Interface has no implementation details at all whereas abstract classes can be mixture of both.

```
class Max implements Person {  
    // ...  
}
```

A class can implement multiple interfaces but only inherit from one class.

An interface can **extends** one or more interfaces.

We can't use access modifiers inside interfaces but **readonly** is allowed.

Interface for Functions

```
interface Mathematics {  
    (a: number, b: number): number  
}  
  
let math: Mathematics  
  
math = (a: number, b: number) => a + b
```

Advanced Types

Intersection Types

You may combine multiple types or interfaces using &

```
type Model = {  
    name: string  
};  
  
type SoftDeletes = {  
    deleted_at: Date  
};  
  
type User = Model & SoftDeletes  
  
let user: User = {  
    name: 'Max',  
    deleted_at: new Date  
}
```

Equivalent to:

```
interface Model {
  name: string
}

interface SoftDeletes {
  deleted_at: Date
}

interface User extends Model, SoftDeletes {}
```

```
type Combinable = string | number
type Numeric = number | boolean
type Universal = Combinable & Numeric // number
// In case of union type that is basically the types they
// have in common
// In case of object type is simply the combination of
// these object properties
```

Type Guards

Union types are useful for modeling situations when values can overlap in the types they can take on. What happens when we need to know specifically whether we have?

- for concrete types (number, string etc.) use `typeof`
- for objects use `<property> in object`
- for classes use `instanceof`

Discriminated Unions

We can't use `instanceof` for interfaces and need to share one common property which describe them.

```
interface Horse {
  type: 'horse'
}

switch (animal.type)
  case 'horse':
    //
```

Type Casting

```
// 1)
const input =
<HTMLInputElement>document.getElementById('name');

// 2)
const input = document.getElementById('name') as
HTMLInputElement;

// 3)
const input = document.getElementById('name')

if (input) {
    let name = (input as HTMLInputElement).value
}
```

Index Properties

Use this feature when you don't know exact properties' name and count

```
interface ErrorBag {
    [key: string]: string
}
```

<https://stackoverflow.com/a/44441178/5827477>

`Record<Keys, Type>` is a Utility type in typescript. It is a much cleaner alternative for key-value pairs where property-names are not known. It's worth noting that **`Record<Keys, Type>`** is a named alias to **`{[k: Keys]: Type}`** where Keys and Type are generics.

```
type ErrorBag = Record<string, string[]>
```

```
▼ {message: "The given data was invalid.",...}
  ▼ errors: {name: ["The name field is required."], slug: ["The slug field is required."],...}
    ► locations: ["The locations field is required."]
    ► name: ["The name field is required."]
    ► slug: ["The slug field is required."]
    message: "The given data was invalid."
```


Function Overloads

If using union types for your function's parameters, output type might be different due to internal behavior

Best practice to write all of possibilities above function:

```
type Combinable = string | number

function combine(a: number, b: number): number;
function combine(a: string, b: string): string;
function combine(a: number, b: string): string;
function combine(a: string, b: number): string;
function combine(a: Combinable, b: Combinable) {
  if (typeof a === 'string' || typeof b === 'string') {
    return a.toString() + b.toString()
  }
  return a + b;
}
```

Optional Chaining

```
let job = user?.job?.title
// Traditional way
let job = user.job && user.job.title
```

Null Coalescing

```
let result = input ?? 'DEFAULT' // If input is undefined
or null
```

Generic Types

let's start with identity function. The identity function is a function that will return back whatever is passed in. You can think of this in a similar way to the *echo* command.

Without generics, we would either have to give the identity function a specific type.

```
function identity(a: number): number {  
    return a;  
}
```

So what about other types like string?

we could describe the identity function using the `any` type:

```
function identity(a: any): any {  
    return a;  
}  
  
const i = identity(2)
```

TypeScript doesn't know the type of `i` variable (it's `any`) and we may call string methods like `i.split()` and the compiler doesn't complain about that.

While using `any` is certainly generic in that it will cause the function to accept any and all types for the type of arg, we actually are losing the information about what that type was when the function returns. If we passed in a number, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned. Here, we will use a type variable, a special kind of variable that works on types rather than values.

```
function identity<Type>(a: Type): Type {  
    return a;  
}
```

Built-in Generics

```
const names: Array<string> = ['Twitch', 'Thresh']  
// OR  
// const names: string[] = [...]
```

```
const promise: Promise<number> = new Promise((resolve) =>  
{  
    setTimeout(() => {
```

```
    resolve(10)
  }, 2000)
})
```

Generic Functions

```
function merge(objA: object, objB: object) {
  return Object.assign(objA, objB)
}

const merged = merge({name: 'Twitch'}, {age: 20})
merged.age // TS2339: Property 'age' does not exist on
type 'object'.
```

We can always use type casting but it's cumbersome

```
const merged = merge({name: 'Twitch'}, {age: 20}) as {
  name: string, age: number }
```

```
function merge<T, U>(a: T, b: U) { // Typescript
  automatically returns intersection of T&U
  return Object.assign(a, b)
}

const obj = merge({name: 'Max'}, {age: 30})

// OR
// merge<{ name: string }, { age: number }>({name: 'Max'},
// {age: 30})
```

Constraint

The problem is that we can still pass number as argument.

```
merge({name: 'Max'}, 30)
```

To fixed that we are going to use constraints:

```
function merge<T extends object, U extends object>(a: T,
b: U) {
    // ...
}
```

keyof

In the example below we have to make sure given key exists in object:

```
function get(item: object, key: string) {
    return 'Value: ' + item[key]
    // No index signature with a parameter of type
    // 'string' was found on type '{}'
```

Here we ensure that second parameter must be any key on the object:

```
function get<T extends object, U extends keyof T>(item: T,
key: U) {
    return 'Value: ' + item[key]
}
```

Generic class

```
class DataStorage {
    private data: string[] = [];

    add(item: string) {
        this.data.push(item);
    }
}

const textStorage = new DataStorage()
textStorage.add('Twitch')
```

The class declared above only support string types. what about numbers, objects etc?

```
class DataStorage<T> {
```

```

    private data: T[] = [];

    add(item: T) {
        this.data.push(item);
    }

    get() {
        return this.data;
    }
}

const textStorage = new DataStorage<string>()
//textStorage.add(2); // TS2345: Argument of type '2' is
//not assignable to parameter of type 'string'.
textStorage.add('Twitch')

```

Partial Generic

```

interface Goal {
    title: string,
    completed: boolean
}

function createGoal(title: string, completed: boolean):
Goal {
    return {
        title: title,
        completed: completed
    }
}

```

That's fine but in reality we may want to do some extra work before assigning parameters.

```

function createGoal(title: string, completed: boolean):
Goal {
    let goal: Goal = {} // Type '{}' is missing the
    following properties from type 'Goal': title, completed
    goal.title = title;
    goal.completed = completed
}

```

That's where partials become handy dandy

```
function createGoal(title: string, completed: boolean):
Goal {
    let goal: Partial<Goal> = {}
    goal.title = title
    goal.completed = completed

    return goal as Goal; // we need to cast it otherwise
it returns Partial<Goal> not Goal
}
```

Readonly

```
const names: Readonly<string[]> = ['Max', 'Alex'];
```

Decorators

Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members.

To enable experimental support for decorators, you must enable the `experimentalDecorators` compiler option either on the command line or in your `tsconfig.json`

Class Decorators

A Class Decorator is declared just before a class declaration. The class decorator is applied to the constructor of the class and can be used to observe, modify, or replace a class definition.

```
// Some programmers start their decorator functions with
uppercase letter.
function Logger(constructor: Function) {
    console.log('Logger...')
}
```

```
// Decorators execute when your class is defined not when
it's instantiated.
@Logger
class Person {
  name = 'Max'
  constructor() {
    console.log('Instantiate Person class ...')
  }
}
```

If you wanna pass arguments, you should use decorator factories.

Decorator Factories

In this example we will build simple render mechanism that angular uses.

<https://angular.io/start#template-syntax>

```
function Component(properties: { selector: string }) {
  /*
    We may add an underscore as a name, which
    basically signals to typescript that I won't use it but I
    have to specify it though. function(_: any)
  */
  return function (constructor: any) {
    const element =
document.getElementById(properties['selector']) as
HTMLElement

    const component = new constructor;

    if (element) {
      element.innerHTML = component.template();
    }
  }
}

@Component({
  selector: 'app'
})
class ProductComponent {
  template() {
    return '<h1>Product Page</h1>'
  }
}
```

```
}  
}
```

Multiple Decorators

```
function Logger(text: string) {  
    console.log('Logger Factory')  
    return function(constructor: any) {  
        console.log('Logger');  
    }  
}  
  
function Component(properties: { selector: string }) {  
    console.log('Component Factory')  
    return function (constructor: any) {  
        console.log('Component')  
    }  
}  
  
@Logger('Logging')  
@Component({  
    selector: 'app'  
})  
  
class ProductComponent {  
    template() {  
        return '<h1>Product Page</h1>'  
    }  
}
```

Bottom most decorator executes first. so output will be:

```
Logger Factory  
Component Factory  
Component  
Logger
```

Method Decorators

<https://www.typescriptlang.org/docs/handbook/decorators.html#method-decorators>


```

class ProjectForm {
  constructor() {
    // Select some elements ...

    this.configure();
  }

  private configure() {
    this.form.addEventListener("submit", this.submit);
  }

  private submit(e: Event) {
    e.preventDefault();

    // When attaching a handler function to an element using
    // addEventListener(), the value of 'this' inside the handler
    // will be a reference to the element. It will be the same as
    // the value of the currentTarget property of the event
    // argument that is passed to the handler.

    console.log(this.title.value);
    // e.currentTarget === this
  }
}

```

<https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener>

Solution:

```

private configure() {
  this.form.addEventListener("submit",
    this.submit.bind(this));
}

```

But let's do it with decorators.

```
function autobind(target: any, propertyKey: string,
descriptor: PropertyDescriptor) {
  return {
    configurable: true,
    get() {
      return descriptor.value.bind(this)
    }
  }
}
```

```
@autobind
private configure() {
  this.form.addEventListener("submit", this.submit);
}
```

Accessor Decorators

<https://www.typescriptlang.org/docs/handbook/decorators.html#accessor-decorators>

Property Decorators

<https://www.typescriptlang.org/docs/handbook/decorators.html#property-decorators>

Parameter Decorators

<https://www.typescriptlang.org/docs/handbook/decorators.html#parameter-decorators>

Build drag and drop

<https://github.com/AmirRezaM75/typescript-drag-and-drop>

Modules & Namespaces

Namespaces

<https://www.typescriptlang.org/docs/handbook/namespaces.html>

In the ECMAScript specification draft, *internal modules* were removed around September 2013, but TypeScript kept the idea under a different name.

Namespaces allow the developer to create separate organization units that can be used to hold multiple values, like properties, classes, types, and interfaces.

As we add more stuff, we're going to want to have some kind of organization scheme so that we can keep track of our types and not worry about name collisions with other objects.

/src/app.ts

```
namespace App {  
    // If you want to use your class, interfaces... outside of  
    // your namespace, you would have to first export them to be  
    // available externally  
    export interface Draggable {  
        //  
    }  
}  
  
// You are now able to access it outside of the namespace  
// by using its fully qualified name. In this case,  
App.Draggable  
  
// ....
```

As our application grows, we'll want to split the code across multiple files to make it easier to maintain.

<https://www.typescriptlang.org/docs/handbook/triple-slash-directives.html>

Triple-slash directives are single-line comments containing a single XML tag and they are only valid at the top of their containing file. The contents of the comment are used as compiler directives

/src/app.ts

```
/// <reference path="drag.ts" />  
namespace App {  
    // Just like with interfaces, namespaces in  
    // TypeScript also allow for declaration merging. This means  
    // that multiple declarations of the same namespace will be  
    // merged into a single declaration.
```

```
}
```

Once there are multiple files involved, we'll need to make sure all of the compiled code gets loaded.

We can use concatenated output using the `outFile` option to compile all of the input files into a single JavaScript output file:

`tsconfig.json`

```
"module": "amd",  
"outFile": "./dist/bundle.js",
```

ES6 Modules

`tsconfig.json`

```
"module": "es2015",
```

`index.html`

```
<script type="module" src="dist/app.js"></script>
```

`src/interfaces/draggable.ts`

```
export interface Draggable {  
  //  
}
```

`src/components/project-item.ts`

```
import { Draggable } from "../interfaces/draggable.js";  
  
// Don't forget .js extension otherwise you'll receive  
404. We can get rid of that when using webpack.
```

Each module runs once for entire application not per import.

Default Export

Along with named exports, the system also lets a module have a default export. This is useful when you want to export a single value or to have a fallback value for your module

You can have multiple named exports per module but only one default export.

```
export default abstract class Component
```

```
import Component from './base.js';  
  
// Default exports are not named, so you can import them  
as anything you like
```

Although it's not something that is used too often, a module can have both named exports and a default export, if you wish.

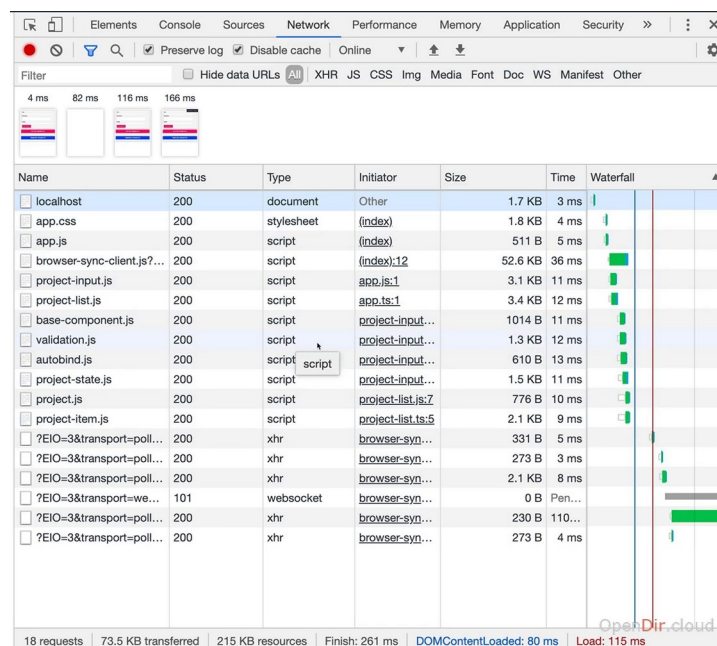
Import *

```
import * as Types from 'types.ts'
```

```
Types.Draggable
```

Webpack

One of the most effective ways to reduce the number of HTTP requests is to combine all JavaScript resources into one.



Name	Status	Type	Initiator	Size	Time	Waterfall
localhost	200	document	Other	1.7 KB	3 ms	
app.css	200	stylesheet	(index)	1.8 KB	4 ms	
app.js	200	script	(index)	511 B	5 ms	
browser-sync-client.js?	200	script	(index):12	52.6 KB	36 ms	
project-input.js	200	script	app.js:1	3.1 KB	11 ms	
project-list.js	200	script	app.js:1	3.4 KB	12 ms	
base-component.js	200	script	project-input...	1014 B	11 ms	
validation.js	200	script	project-input...	1.3 KB	12 ms	
autobind.js	200	script	project-input...	610 B	13 ms	
project-state.js	200	script	project-input...	1.5 KB	11 ms	
project.js	200	script	project-list.js:7	776 B	10 ms	
project-item.js	200	script	project-list.js:5	2.1 KB	9 ms	
?EIO=3&transport=poll...	200	xhr	browser-syn...	331 B	5 ms	
?EIO=3&transport=poll...	200	xhr	browser-syn...	273 B	3 ms	
?EIO=3&transport=poll...	200	xhr	browser-syn...	2.1 KB	8 ms	
?EIO=3&transport=we...	101	websocket	browser-syn...	0 B	Pen...	
?EIO=3&transport=poll...	200	xhr	browser-syn...	230 B	110...	
?EIO=3&transport=poll...	200	xhr	browser-syn...	273 B	4 ms	

18 requests | 73.5 KB transferred | 215 KB resources | Finish: 261 ms | DOMContentLoaded: 80 ms | Load: 115 ms

```
npm install --save-dev webpack webpack-cli webpack-dev-server typescript ts-loader
```

`tsconfig.json`

```
// "rootDir": "./src",  
"sourceMap": true,  
// "outFile": "./",  
// "outDir": "./dist",
```

Remove `.js` extension from all imports.

<https://docs.w3cub.com/webpack~1/webpack-dev-server>

`webpack.config.js`

```
const path = require('path');  
  
module.exports = {  
  mode: 'development',  
  entry: './src/app.ts',  
  output: {  
    filename: 'bundle.js',  
    path: path.resolve(__dirname, 'dist'), // We need  
    absolute path  
    publicPath: '/dist/', // slashes are important  
  },  
  devtool: 'inline-source-map',  
  module: {  
    rules: [  
      {  
        test: /\.ts$/,  
        use: 'ts-loader',  
        exclude: /(node_modules)/,  
      }  
    ]  
  },  
  resolve: {  
    extensions: ['.ts', '.js']  
  },  
  devServer: {  
    static: {  
      directory: path.join(__dirname, '/'),  
    },  
    port: 9000,  
  },  
}
```

package.json

```
"scripts": {  
  "start": "webpack-dev-server",  
  "build": "webpack --config webpack.config.prod.js",  
},
```

Third Party Libraries

Some packages like *Loadash* doesn't support typescript officially but luckily you may find type declaration in [this repository](#). Otherwise you have to use `declare var something: string` on yourself.