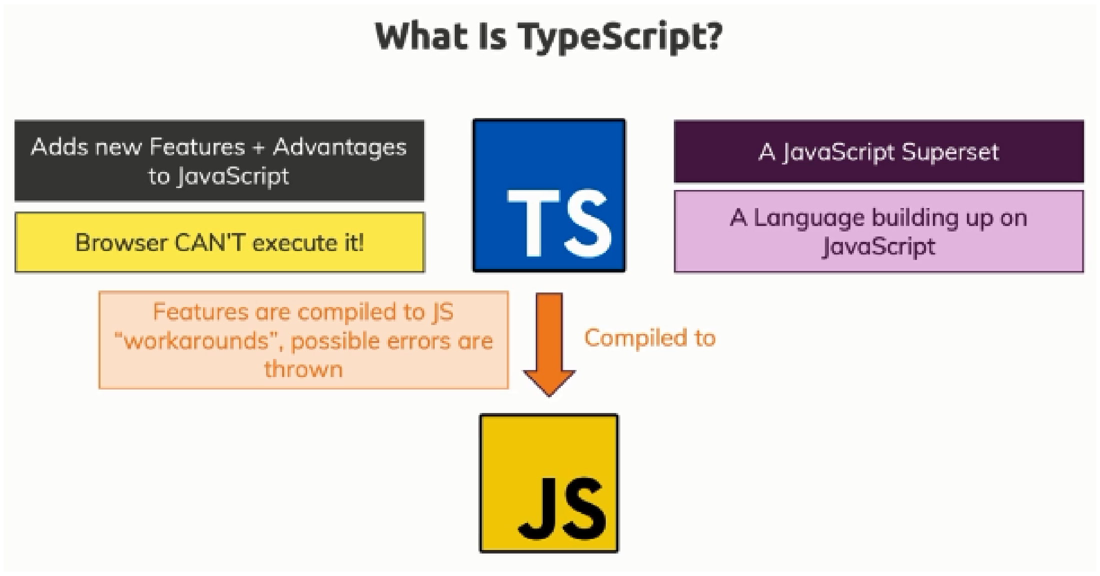
Understanding TypeScript

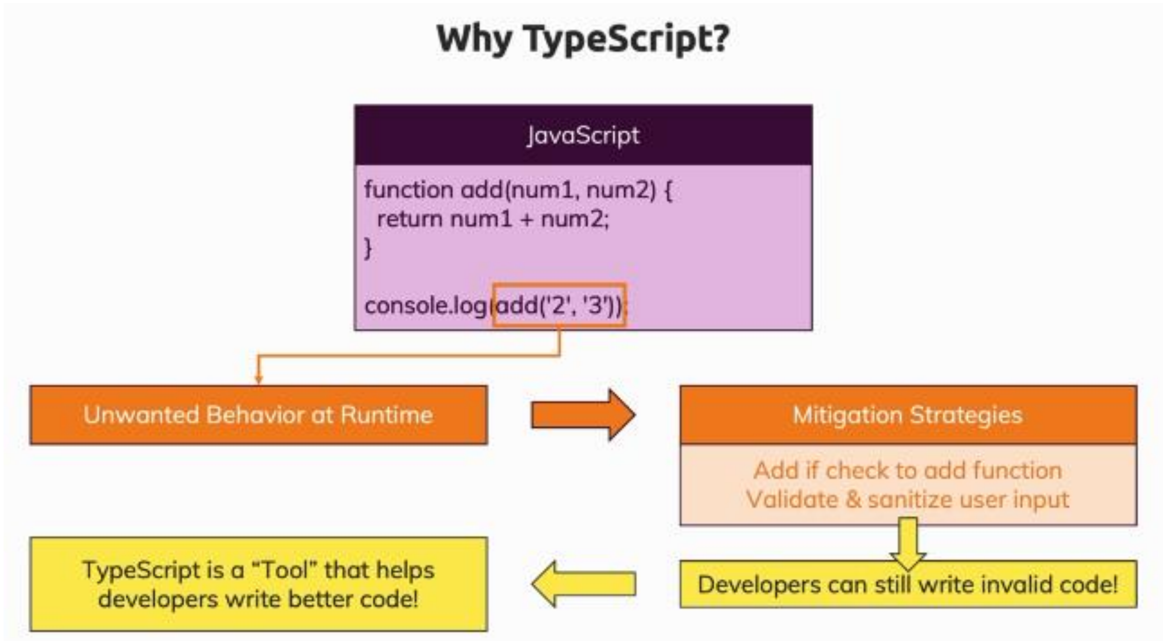
Contents

Getting Started

What Is TypeScript & Why Should You Use It?

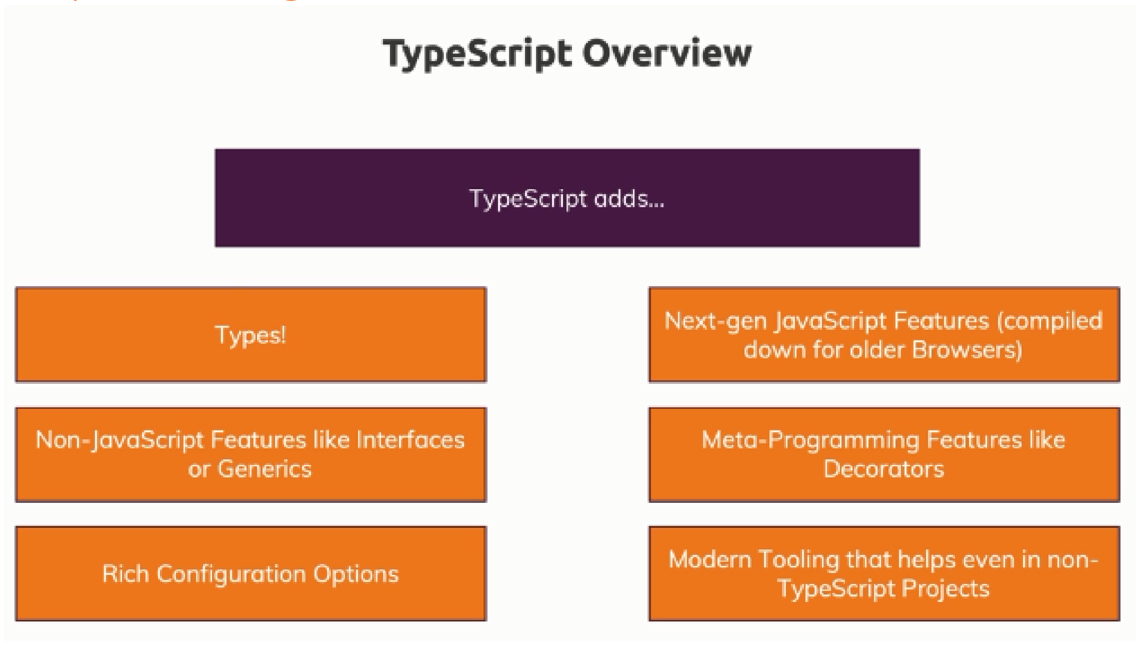
- TypeScript is a programming language, but it's also a tool.
- Of course, it can't add what's not possible at all in the JavaScript language, but it can add new features that simply are easier to use, nicer syntax, things like this.
- It also gives you extra error checking (types) where errors which you would otherwise get as runtime errors can be caught and fixed early during development.


What Is TypeScript?
Adds new Features + Advantages to JavaScript
Browser CAN'T execute it!
TS
A JavaScript Superset
A Language building up on JavaScript
Features are compiled to JS "workarounds", possible errors are thrown
Compiled to
JS


Why TypeScript?
JavaScript
Unwanted Behavior at Runtime
Mitigation Strategies
Add if check to add function
Validate & sanitize user input
Developers can still write invalid code!
TypeScript is a "Tool" that helps developers write better code!

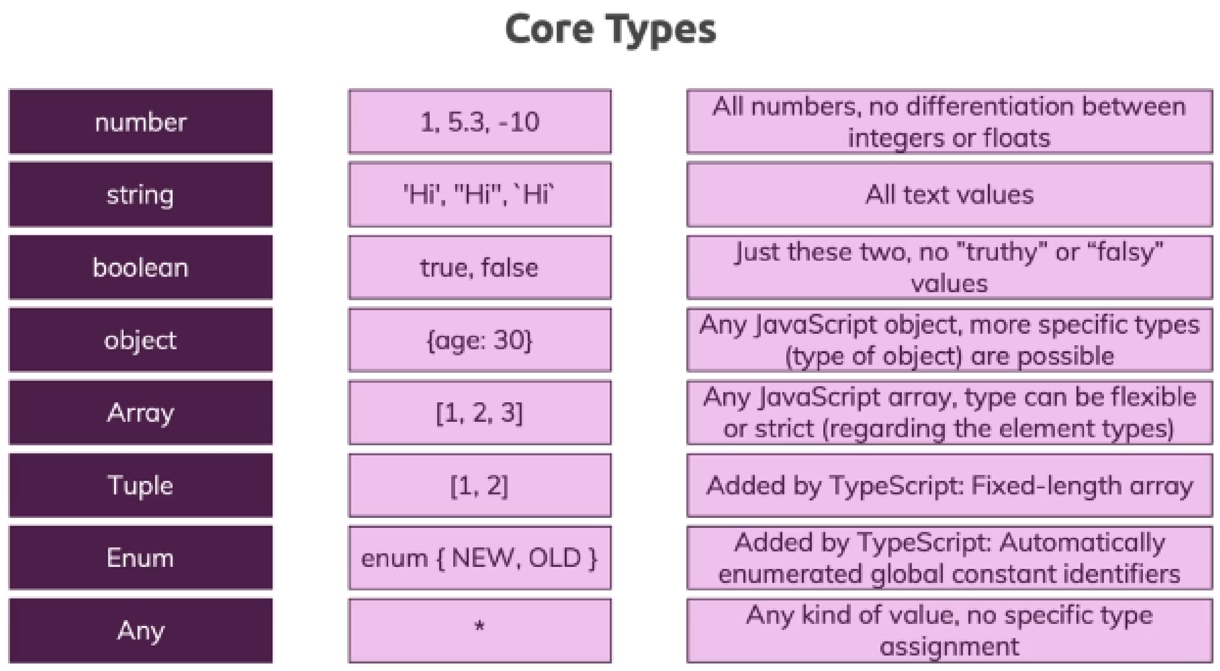
```
function add(num1, num2) {
  return num1 + num2;
}

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

TypeScript Advantages – Overview


TypeScript Overview
TypeScript adds...
Types!
Next-gen JavaScript Features (compiled down for older Browsers)
Non-JavaScript Features like Interfaces or Generics
Meta-Programming Features like Decorators
Rich Configuration Options
Modern Tooling that helps even in non-TypeScript Projects

TypeScript Basics & Basic Types

Using 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
Any	*	Any kind of value, no specific type assignment

- A **tuple** is a special construct which TypeScript understands. In JavaScript that will be a normal array, but during development with typescript we will get errors if we try to update elements in the array with different types.
- If you have a scenario where you need exactly X amount of values in an array and you know the type of each value in advance, then you might want to consider a tuple instead of an array to get even more strictness into your app to be even clearer about the type of data you're working with, the type of data you're expecting.
- **Any** basically means you can store any kind of value in there. Avoid using 'any' as much as possible.
- The **union type** allows either of the mentioned types.
- **Literal type** is kind of union type only, but instead of types, it has actual values.

TypeScript Types vs JavaScript Types

- JavaScript used dynamic types (resolved at runtime).
- TypeScript uses static types (set during development).
- In TypeScript, you work with types like **string** or **number** all the times.
- The **core** primitive types in TypeScript are all **lowercase**!

Type Assignment & Type Inference

- TypeScript has a built in feature, which is called **type inference**. This means that typescript does its best and it does a pretty good job there to understand which type you have in a certain variable or a constant.

Type Aliases

- Type aliases can be used to "create" your own types.
- You're not limited to storing union types though - you can also provide an alias to a (possibly complex) object type.
- For example:
```
type User = { name: string; age: number };
const u1: User = { name: 'Max', age: 30 }; // this works
```
- This allows you to avoid unnecessary repetition and manage types centrally.

Function Types

- Function types are types that describe a function regarding the parameters and the return value type of that function.

The TypeScript Compiler (and its Configuration)

Notes

- tsc CLI Options
  https://www.typescriptlang.org/docs/handbook/compiler-options.html
- What is a tsconfig.json
  https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

Using "Watch Mode"

- E.g. To watch a single file
  ```
  >tsc app.ts --watch
  >tsc app.ts -w
  ```

Compiling the Entire Project / Multiple Files

- What if when we have multiple files?
- Initialize the project as TypeScript project
  ```
  >tsc --init
  ```
- This create `tsconfig.json` file.
- This basically is the indicator for typescript that the project in which this file lies and all sub folders of this folder should be managed by TypeScript.
- Now if you just run `tsc` command without any argument, it will compile all .ts files in this folder and sub-folders and create respective .js file.
- Now this can also be combined with watch mode, so that it will compile all files those change.
  ```
  >tsc --w
  ```

Including & Excluding Files

- “exclude” allows us to exclude specific files from compilation process.
- node_modules is automatically excluded as a default setting.
- “include” allows us to specifically tell TypeScript which files you want to include in the compilation process and anything that's not listed here will not be compiled.
- If we have both “include” and “exclude”, TS will compile include minus exclude.
- “files” is same as “include” with a difference that we cannot provide folders to “files”, it has to be files only.

Setting a Compilation Target

- See settings in "compilerOptions" property in the tsconfig.json file.

Understanding TypeScript Core Libs

- “lib” is an option that allows you to specify which default objects and features typescript knows.
- If “lib” isn't set, then some **defaults** are assumed. The defaults depend on your JavaScript “target”. And for ES6, it by default includes all the features that are globally available in ES6, for example, the map object. So it assumes all these ES6 features which are made available globally in JavaScript, are available in typescript as well. And in addition, it assumes that all DOM APIs are available.

More Configuration & Compilation Options

- “allowJs”, when set to true allows javascript files to be compiled.
- "checkJs", when set to true allows TS to report errors in .js files.

Working with Source Maps

- Source map helps us with debugging and development.
- Setting "sourceMap": true, TS compiler generates corresponding '.map' files which you can use for debugging actual .ts file in the browser.
- Source map files basically act as a bridge, which is understood by modern browsers and developer tools there to connect the JavaScript files to the input files.
- This is super useful in projects because it simplifies debugging.

rootDir and outDir

- The bigger your project gets, the more you might want to organize your files. E.g. a foler to hold all compiled .js files and a src folder which holds all .ts input files.
- By default the compiled .js file sit next to the corresponding .ts file.
- With “outDir”, we can tell the typescript compiler where the created files should be stored. There, it will maintain the folder structure as well.
- With “rootDir”, we can set the input directory, so the folder containing .ts files.
- With outDir amd rootDir set, TS compiler will make sure the input folder structure is replicated in the output folder.

Stop Emitting Files on Compilation Errors

- By default, TS compiler generates .js file even if there are any errors in the .ts files.
- “noEmitOnError”, when set to true, if there is an error in the any .ts file, TS compiler will not generate any .js files.

Strict Compilation

- "strict": true, enables all strict type-checking options.
- All those options are –
  "noImplicitAny": true,
  "strictNullChecks": true,
  "strictFunctionTypes": true,
  "strictBindCallApply": true,
  "strictPropertyInitialization": true,
  "noImplicitThis": true,
  "alwaysStrict": true,

Next-generation JavaScript & TypeScript

"let" and "const"

- Refer – https://stackoverflow.com/questions/762011/whats-the-difference-between-using-let-and-var
- **Scoping rules –**
    - Variables declared by var keyword are scoped to the immediate function body (hence the function scope) while let variables are scoped to the immediate enclosing block denoted by { } (hence the block scope).
- **Hoisting –**
    - 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
    - let variables are not initialized until their definition is evaluated. Accessing them before the initialization results in a ReferenceError.
- **Creating global object property –**
    - At the top level, let, unlike var, does not create a property on the global object.

```
var foo = "Foo";  // globally scoped
let bar = "Bar"; // not allowed to be globally scoped
console.log(window.foo); // Foo
console.log(window.bar); // undefined
```

- **Redeclaration** –
    - In strict mode, var will let you re-declare the same variable in the same scope while let raises a SyntaxError.

```
'use strict';
var foo = "foo1";
var foo = "foo2"; // No problem, 'foo1' is replaced with 'foo2'.

let bar = "bar1";
let bar = "bar2"; // SyntaxError: Identifier 'bar' has already
been declared
```

Arrow Functions

- Benefits –
    - Short syntax
    - If you only have one expression in the function body, you can omit to curly braces and the 'return' keyword and the result of that one expression is then automatically returned.

    - If you have only one argument, you can omit opening/closing brackets. But if you are using it in .ts file and assign type information (e.g. :number), then you need to use opening/closing brackets.

Default Function Parameters

- We can provide default values for function parameters.
- The argument having default value must be the last in the argument list, because it will cause obvious problem while calling the function.

The Spread Operator (...)

- Using spread operator, we call pull out all the elements in a given array.
- So basically, whenever you need a comma separated list of values, you can use the spread operator with given array.
- Spread operator also works with objects. So using spread operator with object, we copy the key-values pairs of that object and we can then create a new object from it (a different copy.)
- E.g.
```
const person1 = {name: ‘Sameer’, age: 30}
const person2 = person;
// here both person1 and person2 are pointing to same object in memory

const person3 = {…person1};
// here person3 is completely different object with same key value
pairs as that of person1
```

Rest Parameters

- You can use spread operator in the place of an argument where you expect list of values.
- You can then call this function with list of comma separated values and then values will be available in the function as an array.
- E.g.
```
const add = (…numbers: number[] ) {
  . . .
}
add(1,2,3); // calling the function
add(1,2,3,4,5,6); // calling the function
```
- This is very useful feature for accepting an unlimited amount of arguments.
- We can use it combined with tuples, where we expect exactly given number of parameters.

- E.g.

  // expecting only 3 values in the array
  ```
  const add = (…numbers: [number, number, number] ) {
    . . .
  }
  add(1,2,3); // This will work
  add(1,2,3,4,5,6); // This will NOT work.
  ```

Array & Object Destructuring

- Destructuring means pulling elements out of the array.
- Array destructuring. E.g.

  ```
  const hobbies = ['Sports', 'Cooking'];

  const [hobby1, hobby2, ...remainingHobbies] = hobbies;
  ```

- Object destructuring. E.g.

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

  const { firstName: userName, age } = person;
  // storing firstName into another variable username, so an alias


  console.log(userName, age, person);
  ```

- The important thing here is for array destructuring, elements are pulled out in order because an array is an ordered list. However in case of object destructuring, the order is not always guaranteed and therefore we don't pull elements out by position, but by key name (mentioned in the object).

How Code Gets Compiled

- TypeScript compiles your code not just from typescript only features to regular JavaScript, but also from modern JavaScript to old JavaScript if you tell TypeScript to do so (via tsconfig.json setting).
- So .js code generated for "target": "es5" will be different than "target": "es6" as TS has to check which features are supported in ES5 vs ES6.

Classes & Interfaces

Classes

- Classes are blueprints for objects.
- Classes allow us to define how objects should look like, which data they should hold, which methods they should have so that we can easily build objects based on these classes.

Compiling a Class to JavaScript

- The generated .js file for a typescript class depends upon the target ES version.
- For details, refer How Code Gets Compiled

Constructor Functions & The "this" Keyword

- A good rule of thumb you can memorize is "this" typically refers to the thing which is calling a method.

"private" and "public" Access Modifiers

- A private property or function is accessible from only inside the class.
- Public is the default access modifier.
- A public property or function is accessible from outside the class as well.

Shorthand Initialization

- E.g.

```
constructor(protected readonly id: string, public name: string) {
  . . .
}
```

// in such case, it is must to use 'public' or private modifier

"readonly" Properties

- If we want to have certain fields not just be private or public, but also shouldn't change after their initialization.
- This adds extra type safety.
- E.g.

```
constructor(protected readonly id: string, public name: string) {
  . . .
}
```

Inheritance

- Whenever you add your own constructor in a class that inherits from another class, you have to add **super** in the inheriting class and you have to execute it like a function super() which calls the constructor of the base class.
- super() must be the first line in the constructor() function.

Overriding Properties & The "protected" Modifier

- "protected" is like private but also accessible in any class that extends this class.

Getters & Setters

- Getters and setters can be great for encapsulating logic and for adding extra logic that should run when you try to read a property or when you try to set up property.
- E.g.

```
class AccountingDepartment extends Department {
  private lastReport: string;

  get mostRecentReport() {
    if (this.lastReport) {
      return this.lastReport;
    }
    throw new Error('No report found.');
  }

  set mostRecentReport(value: string) {
    if (!value) {
      throw new Error('Please pass in a valid value!');
    }
    this.addReport(value);
  }
}

// calling getter and setter functions are properties
accounting.mostRecentReport = 'Year End Report';
console.log(accounting.mostRecentReport);
```

Static Methods & Properties

- Static properties and methods allow you to add properties and methods to classes which are not accessed on an instance of the class but which you access directly on the class.
- This is often used for utility functions that you want to group or map to a class logically or global constants which you always want to store in a class.
- You can't access static members from non-static members of the class.

- To use static properties or methods inside the class, use classname.methodname (or classname.propertyname).
- E.g. Math.pi(), etc. Math class contains lots of utility static methods.
- E.g.

```
class AccountingDepartment extends Department {
  private static instance: AccountingDepartment;

  private constructor(id: string, private reports: string[]) {
    super(id, 'Accounting');
    this.lastReport = reports[0];
  }

  static getInstance() {
    if (AccountingDepartment.instance) {
      return this.instance;
    }
    this.instance = new AccountingDepartment('d2', []);
    return this.instance;
  }
}
```

Abstract Classes

- When you can't provide a general method but you want to **enforce** that, the method exists and then the inheriting classes will need to provide their own implementation because you can't provide a default implementation in the base class.
- E.g.

```
abstract class Department {

  constructor(protected readonly id: string, public name: string) {
  }

  // with (this: Department), we are basically saying that -
  // when describe is executed, "this" inside of describe should always re
fer to an instance that's based on the department class.
  // with this change, we add extra type safety
  abstract describe(this: Department): void;
}
```

- The class must also be marked as abstract if any of its methods is abstract.
- We can have both abstract methods and abstract properties.
- Abstract classes can't be instantiated.

Singletons & Private Constructors

- To ensure and implement singleton pattern, we need to use private constructor.
- E.g.

```
class AccountingDepartment extends Department {

  private static instance: AccountingDepartment;

  private constructor(id: string, private reports: string[]) {
    super(id, 'Accounting');
    this.lastReport = reports[0];
  }

  static getInstance() {
    if (AccountingDepartment.instance) {
      return this.instance;
    }
    this.instance = new AccountingDepartment('d2', []);
    return this.instance;
  }
}
// calling the static method to get singleton instance
const accounting = AccountingDepartment.getInstance();
```

Interfaces

- An interface describes the structure of an object.
- We can use it to describe how an object should look like.
- We can use it as a type to type check for objects that must have this structure.

Using Interfaces with Classes

- An interface and a custom type are not exactly the same.
- Whilst often you can use them interchangeably and you can use the interface instead of a custom type or the other way around.
- One major difference is that **interfaces** can **only** be used to describe the structure of an object. You can use **type** for that as well, but inside of a custom type, you can also store other things like union types and so on. So type is therefore more flexible. But the other side of the coin is that interface is clearer. When you define something as an interface, it is super clear that you want to **define the structure of an object** with that.
- Another difference is we can implement an interface in a class.
- The reason why you often work with interfaces is that interface can be used as a **contract** a class can implement and a class then has to adhere to.

Why Interfaces?

- We can enforce a structure which can be useful if we then have other parts of our code which rely on that structure.

Readonly Interface Properties

- You cannot add public or private or anything like that on an interface, but you can add “readonly” to make it clear that this property in whatever object you build based on this interface must only be set once and is read only thereafter.
- This can be used with “type” as well. Generally its common to use it with interface.

Extending Interfaces

- You can also implement inheritance in interfaces.
- An interface can extend multiple interfaces but a class can only inherit from one class you can't inherit from multiple classes.

Interfaces as Function Types

- Interfaces can also be used to define the structure of a function. So basically as a replacement for the function types.
- E.g.

```
// type AddFn = (a: number, b: number) => number;
interface AddFn {
  // anonymous function
  (a: number, b: number): number;
}

let add: AddFn;

add = (n1: number, n2: number) => {
  return n1 + n2;
};
```

- In this case, using “type” is more common.

Optional Parameters & Properties

- You can also define **optional** properties in interfaces and also in classes.
- You can mark a property or function as optional by adding “?” mark after the property or function name.
- E.g.

```
interface Named {
  readonly name?: string;
  outputName?: string;
}
```

- E.g. Class

```
class Person implements Greetable {
  name?: string;
  age = 30;

  constructor(n?: string) {
    if (n) {
      this.name = n;
    }
  }
}
```

Compiling Interfaces to JavaScript

- There is no translation for interfaces in .js file.
- JavaScript doesn't know about this feature. It's a pure typescript feature only available during development and compilation so you can use it to improve your code helping you to write better code, clearly structured code following clear rules at runtime.
- On TS compilation, the corresponding JS file will have only functions, classes and all other code is compiled. And interfaces are simply dumped.

Advanced Types

Intersection Types

- Intersection types allow us to combine other types.
- We can achieve the same result using multiple interfacing inheritance. However intersection types can be used with any types and not just object type.
- E.g.

```
type Combinable = string | number; // union type
type Numeric = number | boolean;

type Universal = Combinable & Numeric; // intersection type
```

More on Type Guards

- Type guards help us with **union types** because whilst it is nice to have that flexibility, often you need to know which exact type you are getting at runtime.
- Type Guards is just a term that describes the idea or approach of checking if a certain property or method exists before you try to use it, or if you can do something with the type before you try to use it.
- For objects that can be done with **instanceof** or with "**in**". And for other types, you can use **typeof**.
- Refer – 06-advance-types-type-guards

Discriminated Unions

- It's a pattern which you can use when working with union types that makes implementing type guards easier.
- It is available when you work with object types.
- With discriminated union, we have one common property in every object that makes up our union, which describes that object so that we can use this property, that describes this object in our check to have 100 % type safety and understand which properties are available for such an object and which properties are not.
- Refer – 07-advance-types-discriminated-unions

Type Casting

- Typecasting helps you tell TypeScript that some value is of a specific type where TypeScript is not able to detect it on its own.
- Refer – 08-advance-types-type-casting

Index Properties

- Index types is a feature that allows us to create objects which are more flexible regarding the properties they might hold.
- E.g.

```
interface ErrorContainer {
  // { email: 'Not a valid email', username: 'Must start with a character'

  // defining index type
  // here [prop: string] will hold any string as key name and any number o
f such keys e.g. email, username
  // and :string is the value which will hold e.g. 'Not a valid email', 'M
ust start with a character!'
  [prop: string]: string;
}

const errorBag: ErrorContainer = {
  email: 'Not a valid email!',
  username: 'Must start with a capital character!',
};

const errorBag2: ErrorContainer = {
  key1: 'Value 1',
  key2: 'Value 2',
  1: 'number key is interpreted as string',
};
```

- Refer – 09-advance-types-index-properties

Function Overloads

- E.g.

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

const result = add('Max', ' Schwarz');
result.split(' '); // here TS knows ‘result’ is string so can call split()
```

- Function overload can help you in situations where TypeScript would not be able to correctly infer the **return type** on its own.

Optional Chaining

- Optional chaining operator helps us safely access nested properties, a nested objects in our object data. And if the thing in front of the question mark (?) is undefined, it will not access the thing they're after and therefore will not throw a runtime error, but instead it will just not continue.
- Behind the scenes, this basically compiles to an "if" check which checks whatever data exists before it tries to access.

Nullish Coalescing

- Nullish coalescing helps us deal with nullish data.
- Nullish Coalescing operator – ??
- It will check if given variable is null or undefined (not falsy) then use the fallback.
- Refer – 10-advance-types-optional-chaining-nullish-coalescing

Generics

Built-in Generics & What are Generics?

- A generic type is a type which is kind of connected with some other type and is really flexible regarding which exact type that other type is.
- Build in generic types – Array, Promise
  Array knows which data it stores.
  A promise knows which data it returns.
- We get better type safety with generic types.
- They allow you to continue working with your data in a typescript optimal way.

Generic Function

- E.g.

```
function merge<T, U>(objA: T, objB: U) {
  return Object.assign(objA, objB);
}

const mergedObj = merge({ name: 'Max', hobbies: ['Sports'] }, { age: 30 })
;
console.log(mergedObj);
```

Working with Constraints

- For generic types, you can set certain constraints regarding the types your generic types can be based on.
- E.g.

```
function merge<T extends object, U extends object>(objA: T, objB: U) {
  return Object.assign(objA, objB);
}

const mergedObj = merge({ name: 'Max', hobbies: ['Sports'] }, { age: 30 })
;
console.log(mergedObj);
```

The "keyof" Constraint

- This allows us to ensure that we don't make dumb mistakes.
- E.g.

```
// "keyof" constraint
// T is any kind of object. U is any kind of key in T object
function extractAndConvert<T extends object, U extends keyof T>(obj: T, ke
y: U) {
  return 'Value: ' + obj[key];
}

extractAndConvert({ name: 'Max' }, 'name');
```

Generic Classes

- With Generics, we give typescript some extra information that makes a class both a **flexible** and still **strongly typed** class.

Generic Utility Types

- https://www.typescriptlang.org/docs/handbook/utility-types.html

Generic Types vs Union Types

- Union types can be great if you want to have a function, which you can call with one of the allowed types every time you call it.
- Generic types are great if you want to **lock in** a certain type, use the same type throughout the entire class instance you create, use the same type throughout the entire function.

Notes

- Refer - 11-generics

Decorators

- Decorators are a feature which can be very useful for metaprogramming.
- Metaprogramming means that you typically won't use decorator's that often to have a direct impact on the end users of your page, instead decorators are a particularly well suited instrument for writing code, which is then easier to use by other developers.

A First Class Decorator

- A decorator, is in the the end just a function, a function you apply to something, for example, to a class in a certain way.
- Decorators receive arguments. How many arguments - depends on where you use the decorator.
- Decorators execute when your class is defined, not when it is instantiated.
- So the decorator runs when JavaScript finds your class definition, your constructor function definition, not when you use that constructor function to instantiate an object.
- If you add a decorator to a class, the decorator receives one argument – which is the target (so the constructor of the class on which the decorator is applied).

Decorator Factories

- Using decorator factories can give us more power and more possibilities of configuring what the decorator then does internally.
- You can add more than one decorator to a class or anywhere else where you can use a decorator. The order of execution is bottom-up. The bottom decorator is executed first and they're often the decorators above it runs.
- **Creation** of our actual decorator functions happens **in the order in which we specify** the decorator factory functions. But the **execution** of the actual decorator functions then happens **bottom up**, which means the bottommost decorator executes first.

Diving into Property Decorators

- If you add a decorator to a property, the decorator receives two arguments
  - The target of the property.
    - For instance property (property in a class), the target will be the prototype of the object.
    - If we had a static property, target would refer to the constructor function instead.
  - And second argument is the name of the property.
- Property Decorator executes, when your class definition is registered by JavaScript.

Accessor/Method & Parameter Decorators

- If you add a decorator to an accessor (getter, setter functions) or method, the decorator receives three arguments
    - The target of the property.
        - For instance property (property in a class), the target will be the prototype of the object.
        - If we had a static property, target would refer to the constructor function instead.
    - The second argument is the name of the property.
    - And the third argument is property descriptor of type `PropertyDescriptor`.
- If you add a decorator to method paramter, the decorator receives three arguments
    - The target of the property.
        - For instance property (property in a class), the target will be the prototype of the object.
        - If we had a static property, target would refer to the constructor function instead.
    - The second argument is the name of the method in which we use this decorator.
    - And the third argument is position (number) of the argument.

When Do Accessor/Method & Parameter Decorators Execute?

- All executed when you define the class, not when you instantiate object of the class.
- These are not decorators that run at runtime when you call a method or when you work with a property. This is not what they do. Instead, these decorators allow you to do logic behind the scenes additional setup work when a class is defined.

Returning (and changing) a Class in a Class Decorator

- Some decorators, for example, class decorator and all the method decorators, actually are also capable of returning something.
- So you can return something to replace the thing you added the decorator to, e.g. to class with a new class that can implement the old class, but also add it's new custom logic.

Other Decorator Return Types

- Accessor or method decorator can return values. Here we can return new property descriptor.
- Decorators on properties and on parameters also can return something but typescript will ignore it. So return values are not used to be precise.
- More on Property Descriptor – https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

Validation with Decorators

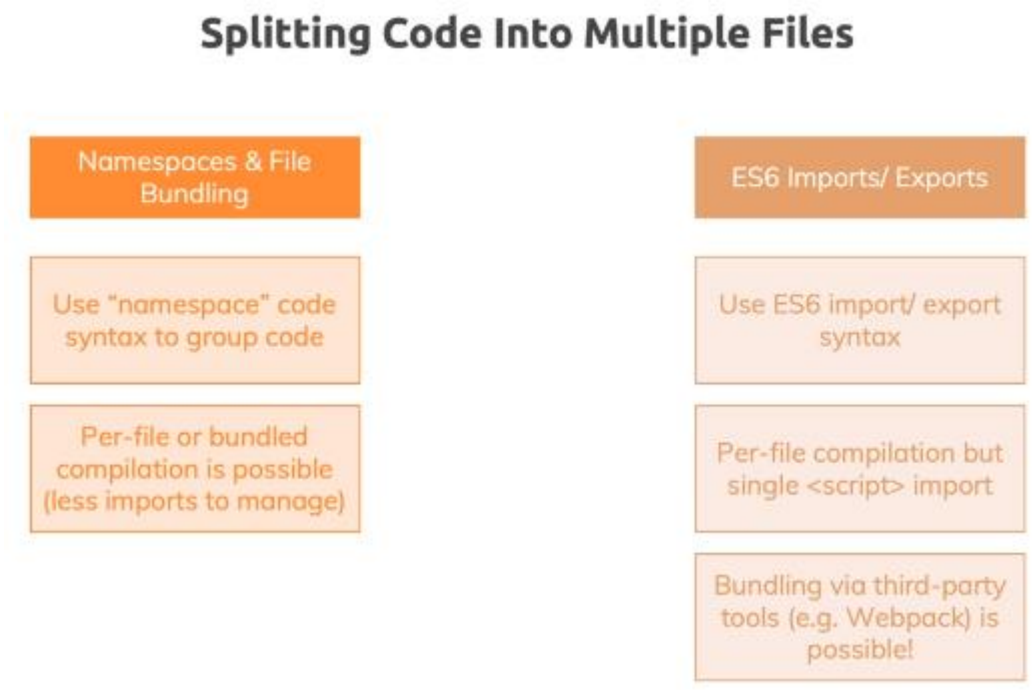
- E.g. https://github.com/typestack/class-validator#readme

Notes

- Many frameworks use decorators for doing all the heavy lifting for developers.
- E.g. Angular, class-validator, NestJS
- Refer – 12-decorators

Modules & Namespaces

Writing Module Code – Your Options



Splitting Code Into Multiple Files

Namespaces & File Bundling	ES6 Imports/ Exports
Use "namespace" code syntax to group code	Use ES6 import/ export syntax
Per-file or bundled compilation is possible (less imports to manage)	Per-file compilation but single <script> import
	Bundling via third-party tools (e.g. Webpack) is possible!

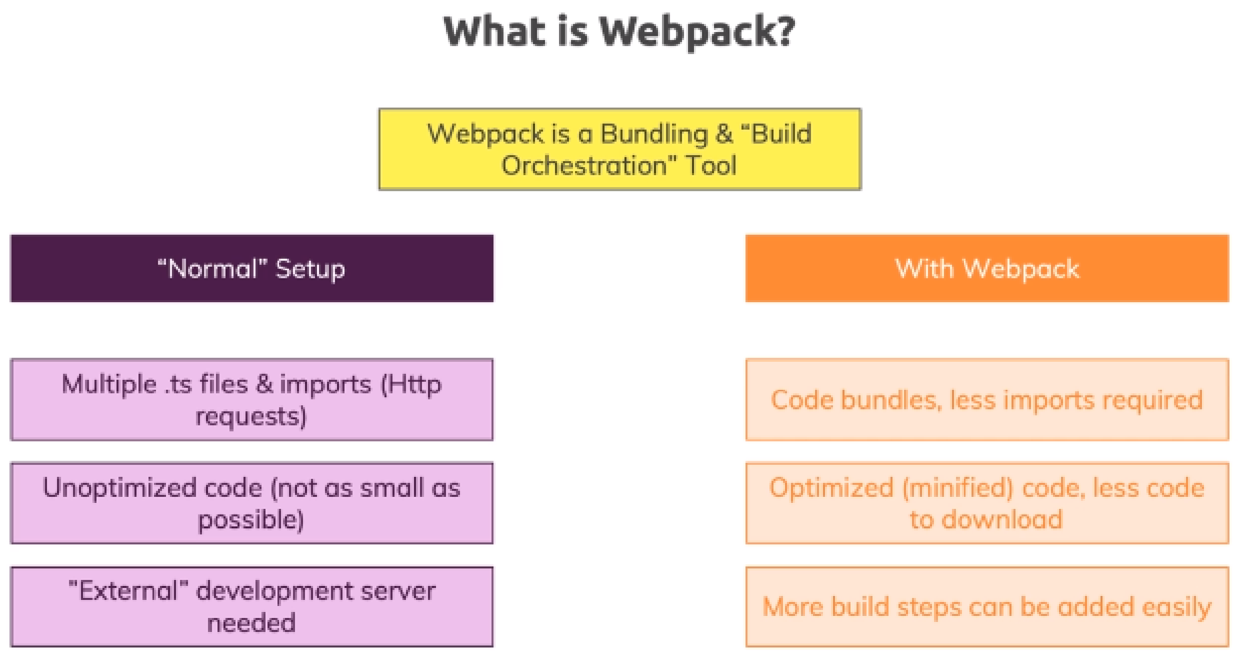
How Does Code In ES Modules Execute?

- The code inside a file runs once when the file is imported for the first time by any other file. If another file then imports that same file again, the code from the imported file does not run again.

Using Webpack with TypeScript

What is Webpack & Why do we need it?

- With ES6 modules, makes our development setup modular so that we have separate organized files. However the disadvantage is when our code runs in browser, the browser has to fetch all those files.
- Fetching multiple files instead of single file has few disadvantages –
  - Browser fires any HTTP requests to fetch those files.
  - Adds delay in overall page load.
  - Every HTP request that needs to be made, takes a little time. It takes time to download the files. But it also requires the browser to set up that request and send it.
- Solution – Webpack
  - Reduce number of http calls to fetch multiple files.
  - Here with webpack, we create a single file from all those module files.
- Webpack is a tool that will help us bundle our files together.
- Webpack is a bundling and build orchestration tool.
- It helps us reduce the amount of HAARP requests by bundling code together so that we can write code split up across multiple files. But webpage then takes all these files and bundled them together and in addition, webpages capable of doing more.
- It will also optimize our code and it also allows us to add more build steps, more build tools, for example, to help us with css files and so on.



What is Webpack?

Webpack is a Bundling & "Build Orchestration" Tool

"Normal" Setup	With Webpack
Multiple .ts files & imports (Http requests)	Code bundles, less imports required
Unoptimized code (not as small as possible)	Optimized (minified) code, less code to download
"External" development server needed	More build steps can be added easily

Installing Webpack & Important Dependencies

- `>npm install --save-dev webpack webpack-cli webpack-dev-server typescript ts-loader`
- `webpack` – It is a tool that allows us to plug in certain functionalities to bundle our code and also to transform our code.
- `webpack-cli` – to run webpage commands in our project.
- `webpack-dev-server` - to have built in development server, which starts webpack under the hood, which watches our files for changes, automatically triggers webpack to recompile when something changes and which then also serves our page.
- `typescript` – Though we install typescript at global level, but it is a good practice to keep it at project level to avoid any breaking changes due to upgrade of global typescript.
- `ts-loader` – is a package to tells webpack how to convert TS code to JavaScript so that webpage is able to do both – compile our code with the help of ts-loader, which in turn uses the TS compiler under the hood. And then webpack also is able to bundle JavaScript files into one bundle code file.

Adding Entry & Output Configuration

- By default webpack is just a bundler.
- For any extra functionality like compiling your TS code, you have to configure webpack.

Adding TypeScript Support with the ts-loader Package

- By default webpack is just a bundler.
- For any extra functionality like compiling your TS code, you have to configure webpack.
- A webpack loader is simply a package to tell webpack how to deal with certain files. E.g. ts-loader tells webpack how to convert TS code to JavaScript.

Finishing the Setup & Adding webpack-dev-server

- `"start": "webpack-dev-server"`
- In the webpack-dev-server mode, the bundle is generated in memory only.

Adding a Production Workflow

- Create separate webpack config file for prod. E.g. webpack.config.prod.js
- Plugin which automatically deletes everything in the dist folder before new output is written.
  >npm install --save-dev clean-webpack-plugin
- `"build:prod": "webpack --config webpack.config.prod.js"`

3rd Party Libraries & TypeScript

- In modern web development, we typically also work with third party libraries. We don't write all the code of our projects on our own instead, typically we utilize third party libraries so that we don't always have to reinvent the wheel on our own, but can add certain functionalities to our projects and then just focus on our core business logic.

Using JavaScript (!) Libraries with TypeScript

- You can install "**types**" of a third party libraries.
- E.g. lodach is pure JS library. In order to get better TS support (e.g. auto completion, etc.), we can install lodash types.
- **.d.ts** files are declaration files, which means they don't contain any actual logic. They contain instructions to typescript. They basically tell typescript how something works and what's included in this package.
- Tn the end, these .d.ts files simply provide a translation from plain JavaScript to typescript. They don't contain any logic that runs, but they define the types a library works with, they define types you use, the types you get back when you call a method and so on.
- If you're working with a third party library written in JavaScript without any typescript code or without built in .d.ts files, which some third party libraries ship with, even if they are written in JavaScript. So if you're working with a library that does not contain built in translation files (.d.ts) and that is not natively written in typescript, then using such a **types** package is the solution (install as dev dependencies).
- These types allows you to use regular normal vanilla JavaScript libraries in a TS project and still get great support, great auto-completion and avoid errors, with the help of the translation types packages.
- The types packages always start with "**@types/**".

Using "declare" as a Last Resort

- What do you do if you have a library where you can't install types.
- So to get proper TS support, we can declare certain variables to type script.
- E.g.
  ```
  declare var GLOBAL: string;
  ```
  This basically tells TS – Don't worry, it will exist. So that TS will not complain.
- It allows you to declare typescript the features or variables where you know that they exist or let typescript know about packages, global variables in general, which TS can't know but you as a developer know that they will be there.

No Types Needed: class-transformer

- https://github.com/typestack/class-transformer

TypeScript-embracing: class-validator

- https://github.com/typestack/class-validator
- This is a package which allows us to add validation rules with the help of some decorators inside of a class. And then whenever we instantiate such a class, we can actually validated for the rules we set up there with the help of decorators

Tips and Tricks

- A good rule of thumb you can memorize is “this” typically refers to the thing which is calling a method.
- Use ES modules instead of TS namespaces.
- Thanks to the source maps, we can debug our original typescript code.
- To use typescript with Node and Express, install third party types – **@types/node**, **@types/express**