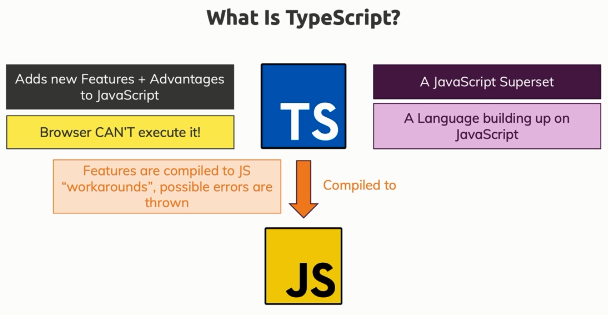
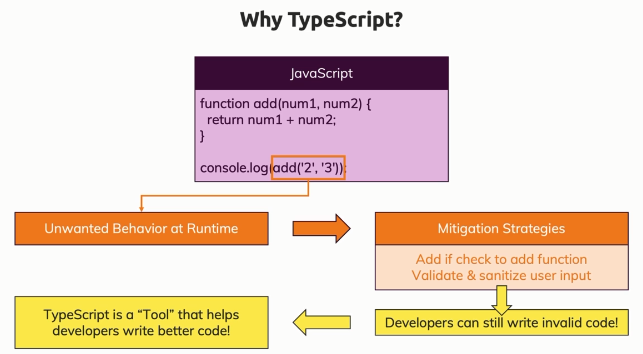
Understanding TypeScript

# Getting Started

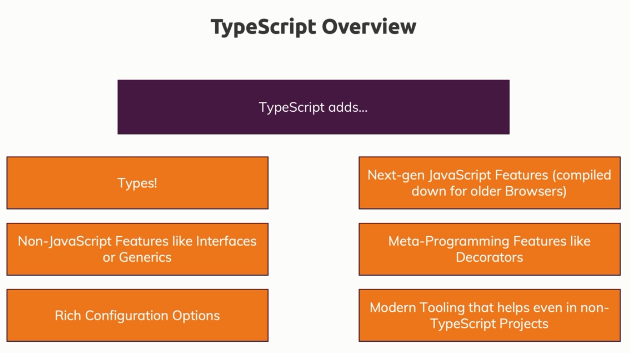
## What Is TypeScript & Why Should You Use It?

* 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.



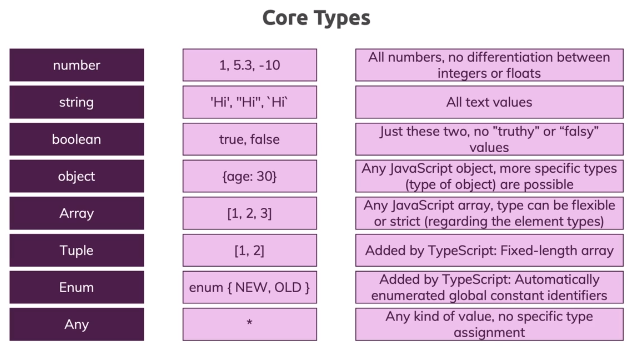


## TypeScript Advantages – Overview



# TypeScript Basics & Basic Types

## Using Types



* 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.

# Tips and Tricks