TypeScript

Wednesday, May 15, 2024 2:46 PM

https://www.typescripttutorial.net/

Setup

- Install Node.js
- In CLI:
  npm install -g typescript
  npm install -g ts-node
  - ts-node is used to skip the transpilation step. You can go straight to *ts-node <filepath to .ts file>* to run it
- Develop in VS Code
  - Recommended to add Live Server extension
  - With this extension, you can run the webpage and whenever you transpile the code, it'll automatically update in the browser instead of having to refresh the browser

TypeScript is a superset of JavaScript

- A .ts file is transpiled into a .js file with:
  tsc <filepath to .ts file>
- This then creates a .js file that can be run with Node.js
- The goal of TS is to have code errors pop up at or before compile time rather than run time
- VS Code's hover tooltips should be more descriptive when the type is known and offer a helpful list of available methods and properties when applicable

Better Objects

- When trying to reference an object property it is easy to misspell it
  - An error like this in JS shows up as undefined during runtime and can someimes go unnoticed

```
interface <interface_name>{
        <property_name>[: property_type],
        etc.
};
```

- This syntax is one of the improvements TS introduces: interfaces
- When misspelling an interface's property (thereby referencing a non-existent property), VS Code will highlight the error

Better functions

- Because JS parameters are non-typed, it is easy to feed in arguments in the wrong order
- You can specify what data type a function accepts and returns:
  function <functionName>(<varName>[: varType], etc.)[: return_type]
  - The return_type can be an Object or Interface

Data Types

- If an entity is not given a type, it is implicitly *any* type

Primitive

- string
- number
- bigint
- boolean
- null
- undefined
- symbol

Objects

- functions
- arrays
- classes
- etc.

any Type

- Allows a variable to take on any type at any time
- Effectively, opts the variable out of TS's safety checks

Type Unions

varNm: type1 | type2

- For when you need the flexibility for a variable to take on multiple different types of values, but not all types

Type Guards

if (typeof A === "dataType") { }
if (A instanceof Object) { }

- For when code accepts multiple different types
- Conditionally executes appropriate code
- When the conditions narrow down the type options to 1 choice left, TS knows that the else clause will be for that last choice

if ("propertyName" in Object) { }

- The in conditional operate checks if the given object has the given member

Custom Type Guards for Objects

function typeGuardName(input: any): input is Type { }

- A user-defined function that outputs whether the input is of the given type
- May be useful for making type checking more forgiving than always requiring an exact match

Type Intersections

type TypeNm = TypeA & TypeB;

- Combines types together with an AND relationship rather than an OR relationship
- All variables of a type intersection must have the members of all of the types in the intersection
- Note that if the types in the intersection share a member of the same name, that member must be the same type across all of them

Type Alias

type <typeNm> = type1 | type2

- When a type union has a large number of options and is repeated throughout code, it would be easier to specify a type
- Whenever the type union is required, you can specify the type alias instead of the entire type list again

String Literal Union Types
type <typeNm>: '<stringLiteral01>' | '<stringLiteral02>'

- Limits a string variable to only take on the literal values given
- Used like an enum. Not sure this is ever necessary

Type Casting / Narrowing / Assertion
____ as dataType
<dataType>______

- Either of these syntaxes can be used to explicitly convert a variable from one type to another
    - Likely only works when downcasting from a broad type to a more specific subtype
    - Can be used to downcast a Parent into a more specific subclass
- When a function returns a type union, meaning it might return 1 type or another under differing conditions, we will need to type cast the output in order to use it
  fn( ) as dataType;
  <dataType>fn( );
    - This is not "type casting". It just tells the TS compiler how to treat the output according to its type

enum Type
https://www.typescripttutorial.net/typescript-tutorial/typescript-enum/
enum <name> {
val1,
val2
};

- Useful for things like a dropdown menu

Declaration Syntax

- *js* = Basic JavaScript tool

Basic

- *<js_declarator> <varNm>[: varType][= value];*

Array

- *<js_declarator> <arrNm>[: arrType][ ] = [<arrData>];*
- Note that the assignment value is optional

Object

- *<js_declarator> <objNm>: {*
  <propNm>[: propType];
  };

Function
<js_declarator> <fnNm> = function (<paramNm>[: paramType])[: returnType] { <fnBody>; }

- Typing with Arrow Notation:
  (p1: type, p2: type): returnType => { functionBody; }

- All functions are children of JS's Object prototype and are treated as objects in TS
    - This means every function can use methods inherited from Object
- The returnType is optional, but I disagree with removing clarity
    - If not specified, the inferred return type will be *void*
- *void* can be used for returnType to indicate a function has no return
    - A return can still be specified in functions implementing a *function type*, but it will be ignored and not returned out
- All parameters are required by default in TS and the number of arguments given must always match the number of parameters
    - You must still account for *null* or *undefined* as arguments to required parameters
- Optional parameters are defined by adding '?' to the end of their names:
  function (<paramNm>?[: paramType])
    - Optional parameters must always be declared after any required parameters
    - Avoid making parameters for **Callback Functions** optional as it can lead to function bodies that try to operate on *undefined*
- Default values:
  function(<paramNm> = <defaultVal>)
    - Function like optional parameters if placed after all required parameters:
      If the user does not provide an argument or passes in undefined, the default value is used
    - Functions like required parameters if placed before any required parameter:
      The user will be required to pass in some value or undefined
- Rest parameter:
  function (...<paramNm>[: paramType][ ])
    - Allows the user to pass in 0 to as many arguments as they wish
    - A function can only ever have 1 of these and they must be placed at the end of the parameter list
    - Inside the function body, the arguments will be in an array that goes by *paramNm*

  Sometimes TS will raise an error when giving a spread argument to a built-in JS function
  Math.atan2(...args); // raises error
    - This can be solved by turning the argument into a Tuple first:
      const args = [<elementList>] as const;
- Complications of the *this* keyword:
  https://www.typescriptlang.org/docs/handbook/2/functions.html#declaring-this-in-a-function
    - It is very easy to confuse contexts in both JavaScript and TS
    - The safest practice seems to be to explicitly declare *this* and its expected type in the function header:
      function (this: <thisType>)
- Overloads
  function name(p1: type, p2: type): returnType01;
  function name(p1: type, p2: type, p3: type): returnType02;
  function name(p1: any, p2: type, p3?: type): any {}
  https://www.typescriptlang.org/docs/handbook/2/functions.html#function-overloads
    - Naming multiple functions the same but with differing parameter lists causes that name to be overloaded
    - TS will call the function that first matches the user's given arguments
        - Because of this, overloaded functions should be defined in order of most specific to least specific
    - At a minimum, 2 signatures need to be declared above the function definition
        - The function definition's signature must have *any* as its types
        - If the number of parameters differs between overloads, the appropriate number of definition signature parameters must be made optional
- Type parameters for Generic Functions
  (The syntax examples of this bullet point are literal, with no <> to signal a required component or [ ] to signal an optional component. This is because this syntax uses these brackets.)
  Allows a function to adapt to the type of its input

function fnNm<T1, T2, etc.>(param1: T1, param2: T2[ ], etc.): T1 | undefined { }
  - The types are optional;
    T1... can be inferred;
    This is basically equivalent to not declaring a type if used in this way
  - param1 is given the type of T1
    - T1 is also made the return type of the function, else it's undefined
    - Any of the given types can be made the return type
    - Note that when you make the return type variable, you open yourself up to crashes if your code uses non-existent member functions downstream

Call a Generic Function
fnNm<dataType>(args);

Type Constraints
function fnNm<T1 extends { typeProperty: expectedPropertyReturnType }> ( ) { }
  - If the T1 type doesn't have the typeProperty property or method with a type/return type of expectedPropertyReturnType, then it won't be allowed to be used
  - Useful for ensuring a generic function only receives input it is capable of handling
  - Without a type constraint, a generic function will fail silently when given bad input

function fnNm<T1, T2 extends keyof T1>(a: T1, b: T2) { return a[b]; }
  - Type constraints must be used as above with the keyword *keyof* when using one variable to index another
  - This also applies when trying to access Object members (think dictionary keys). If the key doesn't exist in the dictionary, an error will be raised.

- Accomodating JS's Parameter Destructuring with typed parameters
  https://www.typescriptlang.org/docs/handbook/2/functions.html#parameter-destructuring
  - The type list should follow the argument "object" as if it were a regular parameter
  - Alternatively, you can declare a named *type* and use it

class Declarations
```
class classNm {
    constructor(accessModifier initVal: propType) {
        this.property = initVal;
    }

    method() { }
}
```

- **class Access Modifiers**
  All class components can have their access limited like in Java.

  Use these keywords before a declaration to assign them:
  private
  - Component can only be accessed within this class

  protected
  - Component can be accessed by this class and its subclasses (classes defined within another class)

  public (default)

  readonly
  - Causes a property to be immutable

- **Inheritance**
```
class Child extends Parent {
    constructor( ) {
        // The keyword super refers to the parent's components
        super(); // Must init parent

        // Child property initialization can happen both before and after the parent's init
    }
}
```

  Overriding Inherited Methods
  - Override a parent's method by naming it the same in the child
  - You can use the parent's overridden method with *super.method()*

- **Getters and Setters**
  - To enable these special methods, property names need to be preceded by an underscore: *_property*
    - They must also be declared outside of the constructor signature
      If inside the contructor, they will be treated as a normal property
  - *accessModifier get/set property( ) { }*
    - The keywords *get* and *set* are used to define the associated functions for _properties
  - These special methods can be invoked with the dot operator instead of with the method name:
    class.property[= val;]

  Abstract Getters and Setters
  - Both the getter and setter of a property must either implemented or abstract.
    You cannot have an abstract getter and a concrete setter.
    You can get around this by implementing both, possibly with dummy code, and then overriding it in the concrete classes.

- **static**
  accessModifier static name
  - This declaration keyword causes the property to share its value across all instances of the class
  - Any property that will be made static must be declared outside of the constructor's signature
  - Note that when trying to access a static property, the class itself must be referenced, not the instance:
    instance.staticProperty // Wrong
    class.staticProperty // Correct
    - This applies wherever you reference it
    - Making the static property a Getter/Setter property with all of its necessary trappings allows you to reference it using the instances
      - The only time the class must be referenced is when you are referencing the static property from within the Getter/Setter
  - Methods can be declared static as well to allow you to use them without having to instantiate the class
    - The variables inside these methods do not retain their values across calls

- **abstract**
  abstract class name {
  constructor( ) { } // Optional

  defaultMethod( ) { }

  abstract method( )
  }
    - Abstract classes function as blueprints for concrete classes to implement
    - Abstract class constructors can never be invoked directly
        - Abstract class constructors can be omitted
        - For some reason, the concrete class must invoke the abstract class's constructor even though it doesn't do anything in its own constructor?
            - It is baked into TypeScript that any child class *must* invoke its parent's constructor, whether it is implemented or not
    - Default methods can be implemented for concrete classes to inherit the functionality of
    - Abstract methods aren't implemented. They are there to require that the concrete class implement them
        - Note how we can control what data type these abstract methods must return
- **Generic class**
  class Name<TypeName> { }
    - Much like generic functions, allows the variables within the class to assume the type the user designates
    - How is this different from just using an *any* typed variable?
        - The *any* type does not allow for the type safety features of TS

interface
interface CustomTypeName {
propertyName: propertyType;

optionalPropertyName?: propertyType;

readonly constantPropertyName: propertyType;
}
- TS's interfaces can function as advanced type filters
- Do the property name of objects need to match the property names in the interface?
    - Yes, they must match the names exactly
- The number of properties an object has does not have to exactly match the number an interface has
    - It only needs to have all the properties the interface calls for
    - It can optionally also have other properties

Like Java, interfaces can also be *implemented* by classes to show that they guarantee certain things about them:
class MyClass implements MyInterface { }
- When a class implements an interface, it will be required to possess the properties and methods that the interface calls for
- Note that any members required by an interface can't have an access modifier
    - All interface-mandated members are public

function interfaces
- *interface FunctionInterface {(*
  parameterName: parameterType,
  optionalParameterName?: parameterType,
  ...restParameterName: parameterType[ ]
  ): returnType
  }

  let fn: FunctionInterface = function ( ): returnType { };

- What's mind boggling about this tool is that the function assigned to a variable typed as the FunctionInterface is under no obligation to have the paramNameAndTypeList
    - Note that if the parameters aren't named by the function to be assigned, that function can't reference them in its body
        - This sort of reveals why the names don't matter.; it's so that we can call the arguments whatever we want to in the function body.
    - Also note that only the function to be assigned can institute default values for the parameters
        - A function interface cannot dictate default values
    - Even if it does supply one, the parameter names don't have to match
        - Only the parameters' types, order, and number are required to match the interface
    - Only the returnType is required to match
- When calling a function of FunctionInterface type
    - VS Code will list the parameters of the interface in its parameter list popup
        - The parameter names of the function assigned to the variable name don't matter at all
    - The caller *must* give all the arguments required by the interface
        - It does not matter if the function itself didn't call for them
- In essence, the only things guaranteed by a function interface are that:
    - The user will be forced to supply the required arguments and the function will accept exactly that many arguments
    - The function will return the given data type

Extending functionality with interfaces
interface InterfaceA extends InterfaceB, ClassC, ... { }
- By using the extend keyword, we can add more members to an interface
    - This is done because editing the original interface would break all the classes that currently implement it. In the case of a few classes, this is fine, but when hundreds of classes implement an interface it would require too much work to edit the original
- By extending a class, the interface inherits all of the class's members (including private and protected ones)
    - The interface also becomes locked to that class and its subclasses. No other classes can implement the interface.
    - The reasoning for this is less clear. If you're going to extend a class in such a way that it affects it and all of its subclasses, then why not just add the members directly to that class?
        - The reason might be when you only want a subset of the eligible classes to implement the added feature. Implementing an extension interface is optional while implementing an inherited method is not.

Generic interfaces

interface InterfaceName<T1, T2> {
property: T1

method(input: T1): T2;
}

interface IndexTypeName<T1> {
// A list can implement this interface and have its index be something other than a number
[indexName: dataType]: T1
}

- How is this any more useful than an enum or dictionary?
  - tbd
- Note that an index type is only that. You can't add properties or methods to its interface.

Differences between abstract classes and interfaces

interface	abstract
Can only define structure	Provides common functionality in addition to structure
Downstream implementors can use multiple interfaces	Downstream implementors can only inherit from 1 abstract class

- Use interfaces when you only want to describe what all the classes that implement it will contain
- Use abstract classes when you also want to provide a common way for those classes to do certain things

Modular code

- Just like in JS, TS supports the keywords *export* and *import*
- This means we can have separate .ts files (modules) export their code for other modules to use

export _______

- Exports whatever is defined next

__someDeclaration__
export { someDeclaration as alias }

- Makes the given entity available for import by others under the name, *alias*

export _____ from 'module';

- Exports the given entities from a previously imported module
- This just seems like a convenience so that the final main module doesn't have to import every related module. It can just import the final intermediary module.

export default defaultEntity;
import defaultEntity from 'module';

- A default export allows you to import from the given module without { }'s.
  This is different from the "named exports" shown above.
- A module can only have 1 default export
- There does not seem to be any benefit from using default exports if we still have to name the entity in the import anyways. It would've been better if this enabled "import 'module'; ".

import { entity as alias, etc. } from './relativePathToModule';
*import * from 'module';*
import type { typeName } from 'module';

- Syntax for importing modular code
- Note that the file extension is optional
- Like usual, an asterisk signifies "import all"
- "import type" is specific language for importing types
  - This used to not be an option before TS version 3.8
  - Not sure why it was added now, as it seems the import functions just the same

Automating builds using tsc, node, & other modules

https://www.typescripttutorial.net/typescript-tutorial/nodejs-typescript/

- Follow this tutorial to see an example of how to automate building and running a TS app