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.

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

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

  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
- 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

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

- Useful for things like a dropdown menu

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 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

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
    }
  }
    - 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**
    - property names need be preceded by and 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;]
- **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( ) { }

    defaultMethod( ) { }

    abstract method( )
  }
    - Abstract classes function as blueprints for concrete classes to implement
    - Abstract class constructors can never be invoked
    - 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
    - For some reason, the concrete class must invoke the abstract class's constructor even though it doesn't construct anything in its own constructor?