Typescript

Thursday, December 12, 2024 8:35 PM

What is Typescript?
TypeScript is a **strongly typed superset of JavaScript** developed and maintained by Microsoft.TypeScript is a free, open-source programming language that adds static typing to JavaScript.

Why use typescript?
1. Static Typing
- TypeScript allows you to define types for variables, function parameters, and return values. This helps catch errors **at compile time** rather than runtime.
- Example:
```
function add(a: number, b: number): number {
   return a + b; // Errors if non-numbers are passed.
}
```
- **Benefit**: Fewer runtime bugs and better code predictability.

2. Improved Developer Experience
- IDEs like VS Code offer **better autocompletion, refactoring, and error checking** for TypeScript compared to JavaScript.
- Example:
```
let user = { name: "Alice", age: 25 };
user. // Suggests `name` and `age` in the dropdown.
```

3. Scalability
- TypeScript's type system and interfaces make it easier to manage large codebases, as they act like a contract for how objects, functions, and modules interact.
- **Benefit**: Ensures consistent and maintainable code for long-term projects.

4. Early Error Detection
- TypeScript catches many common errors (like typos, incorrect function usage, or missing properties) before code is executed, saving debugging time.
- Example:
```
let num: number = "42"; // Error: Type 'string' is not assignable to type 'number'.
```

5. Enhanced Readability and Self-Documentation
- Types serve as documentation for your code, making it easier for other developers to understand your intent without needing extra comments.
- Example:
```
function sendEmail(recipient: string, content: string): void {
   // Logic here
}
```
- **Benefit**: Improves collaboration, especially in team environments.

6. Modern JavaScript Features with Backward Compatibility
- TypeScript supports the latest JavaScript (e.g., ES6+) and provides transpilation for older browsers or environments.
- Example:
```
class Person {
   constructor(public name: string, private age: number) {}
}
```

7. Code Refactoring Made Easy
- With its type system, TypeScript enables safe and reliable refactoring, helping you confidently rename variables, extract functions, or restructure code.

8. Interfaces and Generics
- These allow for defining complex data structures and writing reusable, type-safe components.
- Example (Generics):
```
function identity<T>(arg: T): T {
   return arg;
}
```

9. Ecosystem and Tooling
- TypeScript works seamlessly with popular frameworks like React, Angular, and Vue, and it integrates well into modern build tools like

Webpack, Vite, and Babel.

10. Migration Path

- TypeScript is a superset of JavaScript, so existing JavaScript code can be incrementally converted to TypeScript. This gradual adoption makes it ideal for large projects.

index.ts:

```
let a : number = 1;
console.log(a);
```

Compilation : tsc index.ts
                    node index.js

How to create tsconfig.json file?
tsc --init

Use of tsconfig.json file:

The tsconfig.json file is a configuration file used in **TypeScript** projects. It specifies the **compiler options**, file inclusion/exclusion rules, and other settings for the TypeScript compiler (tsc). By defining these settings, tsconfig.json ensures consistent behavior across the project when compiling TypeScript code to JavaScript.

Commonly Used Compiler Options:

- **target**: Specifies the JavaScript version for output (e.g., ES5, ES6, ES2020).
- **module**: Determines the module system (e.g., CommonJS, ESNext).
- **strict**: Enables strict type-checking (e.g., no implicit any, strict null checks).
- **outDir**: Sets the output directory for compiled JavaScript files.
- **rootDir**: Specifies the root folder for TypeScript source files.
- **sourceMap**: Generates source maps to debug the original TypeScript code in browsers.
- **lib**: Includes specific TypeScript libraries (e.g., DOM, ES2020).
- **baseUrl** and **paths**: Configure module resolution for cleaner imports.

What is type annotation?

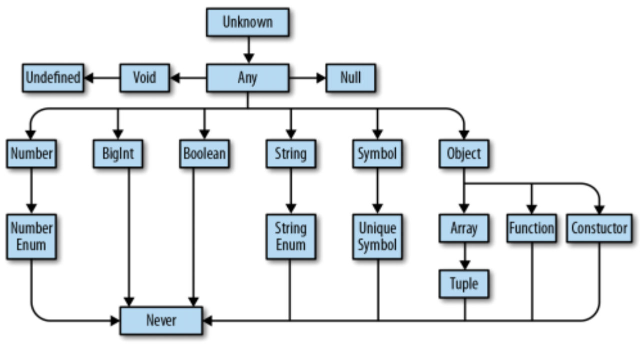
Type annotation is a way of explicitly specifying the type of variable,function parameter, or function return.

Syntax :
For Variable :
Let/const/var variable_name : type = value;

For Function :

Function fun_name(a:number):number{
}
, …….etc

Types in typescript :



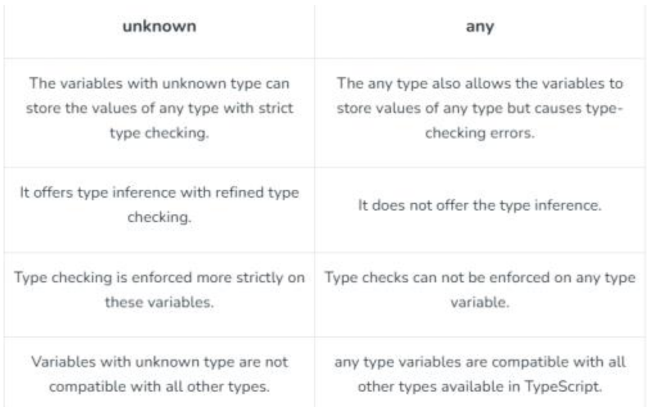
Advantage :

1. It helps the typescript compiler to enforce type checking and provide type safety during development.
2. What operation could we performed on that variable or values.

~~~markdown
## **1. Basic Types**
### **Primitive Types**
These are the basic types in TypeScript, similar to JavaScript.
| Type       | Description                                  | Example                       |
|-----------|---------------------------------------------|------------------------------|
| `string`  | Represents textual data                     | `let name: string = "Alice";`|
| `number`  | Represents numeric values                   | `let age: number = 25;`      |
| `boolean` | Represents true/false values                | `let isAdmin: boolean = true;`|
| `null`    | Represents the absence of a value           | `let empty: null = null;`    |
| `undefined` | Represents an uninitialized value         | `let u: undefined = undefined;`|
| `symbol`  | Represents unique values (ES6 feature)      | `let id: symbol = Symbol();` |
| `bigint`  | Represents very large integers (ES2020)     | `let big: bigint = 123n;`    |
---
## **2. Special Types**
### **`any`**
Disables type checking and allows any value.
```typescript
let value: any = "Hello";
value = 42; // No error
```
### **`unknown`**
A safer alternative to `any`. You must perform type checks before using it.
```typescript
let input: unknown = "Hello";
if (typeof input === "string") {
  console.log(input.toUpperCase()); // Safe
}
```
### **`void`**
Represents the absence of a return value, often used for functions.
```typescript
function logMessage(message: string): void {
  console.log(message);
}
```
### **`never`**
Represents a value that never occurs, such as in functions that throw errors or never terminate.
```typescript
function throwError(message: string): never {
  throw new Error(message);
}
```
---
## **3. Object Types**
### **Object**
Represents non-primitive types (objects, arrays, functions).
```typescript
let obj: object = { name: "Alice", age: 25 };
```
### **Array**
Specify arrays using `type[]` or `Array<type>`.
```typescript
let numbers: number[] = [1, 2, 3];
let names: Array<string> = ["Alice", "Bob"];
```
### **Tuple**
Represents a fixed-length array with specific types.
```typescript
let tuple: [string, number] = ["Alice", 25];
```
---
## **4. Union and Intersection Types**
### **Union (`|`)**
A variable can have one of several types.
```typescript
let id: number | string = 101; // Can be a number or a string
```
### **Intersection (`&`)**
Combines multiple types into one.
```typescript
~~~

~~~
type Address = { city: string };
type Contact = { email: string };
type User = Address & Contact;
let user: User = { city: "New York", email: "user@example.com" };
```
---
## **5. Literal Types**
Allows specific values as types.
```typescript
let direction: "up" | "down" | "left" | "right" = "up";
```
---
## **6. Enums**
Define a set of named constants.
### Numeric Enum:
```typescript
enum Direction {
  Up = 1,
  Down,
  Left,
  Right,
}
let dir: Direction = Direction.Up;
```
### String Enum:
```typescript
enum Status {
  Success = "SUCCESS",
  Error = "ERROR",
}
let status: Status = Status.Success;
```
---
## **7. Type Aliases**
Create custom types with the `type` keyword.
```typescript
type Point = { x: number; y: number };
let point: Point = { x: 10, y: 20 };
```
---
## **8. Interfaces**
Define the shape of an object.
```typescript
interface User {
  name: string;
  age: number;
}
let user: User = { name: "Alice", age: 25 };
```
---
## **9. Function Types**
### Function Parameters and Return Types:
```typescript
function greet(name: string): string {
  return `Hello, ${name}`;
}
```
### Anonymous Function:
```typescript
let add: (a: number, b: number) => number = (x, y) => x + y;
```
---
## **10. Advanced Types**
### **Type Assertions**
Override TypeScript's inferred type.
```typescript
let value: any = "Hello, TypeScript!";
let strLength: number = (value as string).length;
```
### **Nullable Types**
Use `strictNullChecks` to prevent assigning `null` or `undefined` unless explicitly stated.
```typescript
let name: string | null = null;
```
### **Keyof**
Get the keys of a type as a union of strings.
~~~

~~~
```typescript
type User = { id: number; name: string };
type UserKeys = keyof User; // "id" | "name"
```
### **Mapped Types**
Create types dynamically based on another type.
```typescript
type ReadOnly<T> = { readonly [K in keyof T]: T[K] };
```
---
## **11. Generics**
Write reusable components with generic types.
```typescript
function identity<T>(value: T): T {
  return value;
}
let num = identity<number>(42);
let str = identity<string>("Hello");
```
---
TypeScript's type system is flexible and powerful. Would you like to dive deeper into any specific type or feature?
~~~



unknown	any
The variables with unknown type can store the values of any type with strict type checking.	The any type also allows the variables to store values of any type but causes type-checking errors.
It offers type inference with refined type checking.	It does not offer the type inference.
Type checking is enforced more strictly on these variables.	Type checks can not be enforced on any type variable.
Variables with unknown type are not compatible with all other types.	any type variables are compatible with all other types available in TypeScript.

What is function?

```
// Function is block of code that perform specific task
// Normal Function
function greet(a:string)
{
    console.log(a);
}
greet("Hello Ram");
// Function with return type
function greet1(a:string):string{
    return a;
}
console.log(greet1("Hello Krishna"));
// Arrow function
const app = (a:string) :string =>{
    return a;
}
console.log(app("Hello"));
// Optional Parameter
function sample(a?:number)
{
    console.log(a);
}
sample(1);
```

What is type inference?

Type inference in typescript refers to the ability of the typescript compiler to automatically determine and assign types to variables,expression and function return values based on their usage and context in the code.

Advantages of Type Inference

- **Reduces Boilerplate**: Less need for repetitive type annotations.
- **Improves Readability**: Makes code cleaner and easier to maintain.
- **Ensures Type Safety**: TypeScript still enforces type rules even without explicit annotations.

Default parameter : Default parameters allow you to assign a **default value** to a parameter.
Optional Parameter : Optional parameters allow you to define parameters that **may or may not be provided** by the caller.(?)

Example of Default :

```
function calculateArea(width: number, height: number = 10): number {
  return width * height;
}

console.log(calculateArea(5)); // 50 (height defaults to 10)
console.log(calculateArea(5, 20)); // 100 (height is explicitly set to 20)
console.log(calculateArea(5, undefined)); // 50 (height defaults to 10)
```

Example of Optional parameter:

```
function calculateArea(width: number, height?: number): number {
  if (height === undefined) {
    return width * width; // Assume a square if height is not provided.
  }
  return width * height;
}

console.log(calculateArea(5)); // 25 (assumes a square)
console.log(calculateArea(5, 10)); // 50
```

Array in typescript?

an **array** is a collection of elements that are of the same type (or a union of types).

```
// Using square bracket
let arr : number[] = [1,2,3,4,5];
console.log(arr);
// Using Array Constructor
let arr1 : number[] = new Array(1,2,3,4,5,6);
console.log(arr1);
// Using Array.of
let arr2: number[] = Array.of(1, 2, 3, 4);
console.log(arr2);
// Different types of values
let arr3 : (number|string)[] = [1,2,"Ram",'a'];
console.log(arr3);
console.log(arr2[1]);
console.log(arr2.length);
arr.push(10);
console.log(arr);
arr.splice(0,2,100,200)
console.log(arr);
arr.pop()
console.log(arr);
```

```
// Iterate:
for(let i :number = 0;i<arr.length;i++)
{
    console.log(arr[i]);
}
for(let n in arr)
{
    console.log(n);
}
for(let a of arr3)
{
    console.log(a);

}


// map
let arr4 = arr.map((ele:number,index:number)=>{
    return ele*5;
})
console.log(arr4);
// filter
let arr5 = arr2.filter((ele:number,ind:number)=>{
    if (ind % 2==0) {
        return ele * 2
    }
})
console.log(arr5);
// reduce
let sum = arr2.reduce((ele:number,curr:number)=>{
    ele+=ele+curr;
    return ele;
})
console.log(sum);
```

Method :

1. Mutating Array Methods (Methods that modify the original array)

Method	**Description**	**Example**
push()	Adds one or more elements to the end of an array.	arr.push(4) → [1, 2, 3, 4]
pop()	Removes the last element from an array.	arr.pop() → [1, 2, 3]
shift()	Removes the first element from an array.	arr.shift() → [2, 3, 4]
unshift()	Adds one or more elements to the beginning of an array.	arr.unshift(0) → [0, 1, 2, 3, 4]
splice()	Changes the contents of an array by removing or replacing elements.	arr.splice(2, 1, 10) → [1, 2, 10, 4]
reverse()	Reverses the order of the elements in the array.	arr.reverse() → [4, 3, 2, 1]
sort()	Sorts the elements of the array.	arr.sort() → [1, 2, 3, 4]
fill()	Changes all elements in an array to a static value from a start index to an end index.	arr.fill(0, 1, 3) → [1, 0, 0, 4]
copyWithin()	Shallow copies a portion of the array to another location within the array.	arr.copyWithin(0, 2) → [3, 4, 3, 4]

2. Non-Mutating Array Methods (Methods that do not modify the original array)

Method	**Description**	**Example**
concat()	Merges two or more arrays into a new array.	arr.concat([5, 6]) → [1, 2, 3, 4, 5, 6]
join()	Joins all elements of the array into a string.	arr.join('-') → "1-2-3-4"
slice()	Returns a shallow copy of a portion of the array.	arr.slice(1, 3) → [2, 3]
indexOf()	Returns the first index at which a given element is found.	arr.indexOf(3) → 2
lastIndexOf()	Returns the last index at which a given element is found.	arr.lastIndexOf(3) → 2
includes()	Checks if a certain element exists in the array.	arr.includes(2) → true

find()	Returns the first element that satisfies a provided condition.	arr.find(x => x > 2) → 3
findIndex()	Returns the index of the first element that satisfies a condition.	arr.findIndex(x => x > 2) → 2
map()	Creates a new array populated with the results of a provided function.	arr.map(x => x * 2) → [2, 4, 6, 8]
filter()	Creates a new array with all elements that pass the test implemented by the provided function.	arr.filter(x => x > 2) → [3, 4]
reduce()	Executes a reducer function on each element of the array to reduce it to a single value.	arr.reduce((acc, x) => acc + x, 0) → 10
reduceRight()	Similar to reduce(), but starts from the last element.	arr.reduceRight((acc, x) => acc + x, 0) → 10
some()	Tests if at least one element in the array satisfies the condition.	arr.some(x => x > 2) → true
every()	Tests if every element in the array satisfies the condition.	arr.every(x => x > 0) → true
forEach()	Executes a provided function once for each array element.	arr.forEach(x => console.log(x))
flat()	Flattens a nested array into a single array.	arr.flat() → [1, 2, 3, 4]
flatMap()	Maps each element using a mapping function, and then flattens the result into a new array.	arr.flatMap(x => [x, x]) → [1, 1, 2, 2, 3, 3, 4, 4]

3. Array Testing Methods

Method	**Description**	**Example**
isArray()	Checks if a value is an array.	Array.isArray([1, 2, 3]) → true

4. Other Methods

Method	**Description**	**Example**
from()	Creates a new array from an iterable (e.g., a string or a Set).	Array.from('hello') → ['h', 'e', 'l', 'l', 'o']
keys()	Returns a new Array Iterator object that contains the keys of the array.	arr.keys() → Array Iterator {0, 1, 2}
values()	Returns a new Array Iterator object that contains the values of the array.	arr.values() → Array Iterator {1, 2, 3}
entries()	Returns a new Array Iterator object that contains the key-value pairs of the array.	arr.entries() → Array Iterator { [0, 1], [1, 2], [2, 3] }

5. Array Buffer Methods (for Typed Arrays)

Method	**Description**	**Example**
buffer	Returns the ArrayBuffer object that stores the elements of the typed array.	typedArray.buffer
byteLength	Returns the length (in bytes) of the array buffer.	typedArray.byteLength
byteOffset	Returns the offset (in bytes) of the array buffer.	typedArray.byteOffset