# Detailed TypeScript Notes: From Basic to Advanced

TypeScript is a superset of JavaScript that adds optional static typing to the language. It compiles to plain JavaScript, meaning any valid JavaScript code is also valid TypeScript code.

## 1. Introduction to TypeScript

### What is TypeScript?

TypeScript is an open-source language developed by Microsoft. It extends JavaScript by adding static type definitions, which allows for early detection of errors during development rather than at runtime.

### Why Use TypeScript?

- **Static Type Checking:** Catches errors during compilation, reducing runtime bugs.
- **Improved Readability and Maintainability:** Types make code easier to understand and refactor, especially in large codebases.
- **Enhanced Tooling:** Better IDE support (autocompletion, refactoring, code navigation).
- **Better Collaboration:** Clearer contracts between different parts of the codebase.
- **Scalability:** Ideal for large-scale applications where JavaScript can become unwieldy.

### Setting Up TypeScript

To use TypeScript, you need Node.js and npm (or yarn) installed.

1. **Install TypeScript globally:**
   ```
   npm install -g typescript
   ```
2. **Initialize a TypeScript project:** Create a new directory, navigate into it, and run:
   ```
   tsc --init
   ```
   This command creates a tsconfig.json file, which configures the TypeScript compiler.A basic tsconfig.json might look like this (many options are commented out by default):
   ```
   {
     "compilerOptions": {
       "target": "es2016",                  /* Specify ECMAScript
   target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016',
   'ES2017', 'ES2018', 'ES2019', 'ES2020', 'ES2021', 'ES2022',
   'ESNext'. */
       "module": "commonjs",                /* Specify module code
   generation: 'none', 'commonjs', 'amd', 'system', 'umd', 'es2015',
   'es2020', 'es2022', 'esnext', 'node16', 'nodenext'. */
       "outDir": "./dist",                  /* Specify an output
   folder for all emitted files. */
   ```

```
    "esModuleInterop": true,              /* Emit additional
JavaScript to ease support for importing CommonJS modules. This
enables 'allowSyntheticDefaultImports' for type compatibility. */
    "forceConsistentCasingInFileNames": true, /* Ensure that
casing is consistent across all file paths. */
    "strict": true,                       /* Enable all strict
type-checking options. */
    "skipLibCheck": true                  /* Skip type checking all
.d.ts files. */
  }
}
```

3. **Compile TypeScript:** Create a .ts file (e.g., src/index.ts) and then run:
   ```
   tsc
   ```
   This will compile your TypeScript files into JavaScript files in the outDir specified in tsconfig.json (e.g., dist/index.js).

# 2. Basic Types

TypeScript provides several built-in types to define the kind of values a variable can hold.

## number, string, boolean

These are the fundamental primitive types.

```
// Number
let age: number = 30;
let price: number = 9.99;

// String
let name: string = "Alice";
let greeting: string = `Hello, ${name}!`; // Template literals are
also supported

// Boolean
let isActive: boolean = true;
let hasPermission: boolean = false;
```

## any, unknown, void, null, undefined, never

- **any**: A powerful type that disables type checking for a variable. Use with caution, as it defeats the purpose of TypeScript.
  ```
  let data: any = 10;
  data = "hello";
  data = true;
  data.foo(); // No error at compile time, potential runtime error
  ```

- **unknown**: A safer alternative to any. You must perform type checking before performing operations on an unknown type.
  ```
  let value: unknown = "hello";

  // value.toUpperCase(); // Error: Object is of type 'unknown'.

  if (typeof value === 'string') {
      console.log(value.toUpperCase()); // OK
  }
  ```
- **void**: Used for functions that do not return any value.
  ```
  function logMessage(): void {
      console.log("This function returns nothing.");
  }
  ```
- **null and undefined**: Represent the absence of a value. By default, null and undefined are subtypes of all other types (e.g., number | null). With strictNullChecks enabled in tsconfig.json, they are only assignable to any, unknown, or their respective types.
  ```
  let n: null = null;
  let u: undefined = undefined;

  // With strictNullChecks: true
  let optionalName: string | null = "John";
  optionalName = null; // OK
  // optionalName = undefined; // Error if not explicitly allowed
  ```
- **never**: Represents the type of values that never occur. Used for functions that throw an error or never return (e.g., infinite loops).
  ```
  function throwError(message: string): never {
      throw new Error(message);
  }

  function infiniteLoop(): never {
      while (true) {
          // ...
      }
  }
  ```

## Arrays

Arrays can be typed in two ways:

1. Type[] (preferred)
2. Array<Type> (generic array type)

```
let numbers: number[] = [1, 2, 3, 4];
let names: string[] = ["Alice", "Bob"];
let mixed: (number | string)[] = [1, "two", 3]; // Array with union
types
```

```
let booleanArray: Array<boolean> = [true, false];
```

## Tuples

Tuples allow you to express an array with a fixed number of elements whose types are known, but don't have to be the same.

```
let person: [string, number] = ["John Doe", 30];
// person = [30, "John Doe"]; // Error: Type 'number' is not
assignable to type 'string'.

let rgbColor: [number, number, number, number?] = [255, 0, 0]; //
Optional element
rgbColor = [255, 0, 0, 0.5]; // With optional element
```

## Enums

Enums allow a developer to define a set of named constants.

```
// Numeric Enum (default: starts from 0)
enum Direction {
    Up,      // 0
    Down,    // 1
    Left,    // 2
    Right    // 3
}
let go: Direction = Direction.Up;
console.log(go); // Output: 0
console.log(Direction[0]); // Output: Up

// Numeric Enum with custom starting value
enum StatusCode {
    NotFound = 404,
    Success = 200,
    Accepted = 202,
    BadRequest = 400
}
let status: StatusCode = StatusCode.Success;
console.log(status); // Output: 200

// String Enum
enum UserRole {
    Admin = "ADMIN",
    Editor = "EDITOR",
    Viewer = "VIEWER"
}
let userRole: UserRole = UserRole.Admin;
```

```
console.log(userRole); // Output: ADMIN
```

# 3. Functions

TypeScript allows you to add type annotations to function parameters and their return types.

## Type Annotations for Parameters and Return Types

```
function add(x: number, y: number): number {
    return x + y;
}

let sum: number = add(5, 3); // sum will be 8

function greet(name: string): string {
    return `Hello, ${name}!`;
}

function log(message: string): void {
    console.log(message);
}
```

## Optional and Default Parameters

- **Optional Parameters**: Marked with ?. They must come after all required parameters.
- **Default Parameters**: Provide a default value if the argument is not provided.

```
// Optional parameter
function buildName(firstName: string, lastName?: string): string {
    if (lastName) {
        return firstName + " " + lastName;
    }
    return firstName;
}
console.log(buildName("Bob"));          // Output: Bob
console.log(buildName("Bob", "Smith")); // Output: Bob Smith

// Default parameter
function calculateArea(width: number, height: number = 10): number {
    return width * height;
}
console.log(calculateArea(5));     // Output: 50 (height defaults to
10)
console.log(calculateArea(5, 20)); // Output: 100
```

### Rest Parameters

Allows a function to accept an indefinite number of arguments as an array.

```
function sumAll(message: string, ...numbers: number[]): string {
    let total = numbers.reduce((acc, num) => acc + num, 0);
    return `${message}: ${total}`;
}
console.log(sumAll("Sum of numbers", 1, 2, 3, 4)); // Output: Sum of
numbers: 10
console.log(sumAll("No numbers"));              // Output: No numbers:
0
```

### Function Overloads

Allows you to define multiple function signatures for a single function implementation, providing more precise type checking.

```
function combine(a: number, b: number): number;
function combine(a: string, b: string): string;
function combine(a: any, b: any): any {
    if (typeof a === 'number' && typeof b === 'number') {
        return a + b;
    }
    if (typeof a === 'string' && typeof b === 'string') {
        return a + b;
    }
    throw new Error("Parameters must be both numbers or both
strings.");
}

console.log(combine(10, 20));     // Output: 30
console.log(combine("Hello", "World")); // Output: HelloWorld
// console.log(combine(10, "World")); // Error: No overload matches
this call.
```

# 4. Interfaces and Type Aliases

Both interfaces and type aliases are used to define custom types, but they have some differences.

### Defining Interfaces

Interfaces define the structure of an object, specifying the names and types of its properties and methods.

```
interface User {
    id: number;
```

```
    name: string;
    email?: string; // Optional property
    readonly createdAt: Date; // Readonly property
    greet(message: string): void; // Method signature
}

const user1: User = {
    id: 1,
    name: "Alice",
    createdAt: new Date(),
    greet(message: string) {
        console.log(`${this.name} says: ${message}`);
    }
};

user1.greet("Hello there!");
// user1.createdAt = new Date(); // Error: Cannot assign to
'createdAt' because it is a read-only property.

interface Point {
    x: number;
    y: number;
}

function printPoint(p: Point) {
    console.log(`x: ${p.x}, y: ${p.y}`);
}

printPoint({ x: 10, y: 20 });
```

## Extending Interfaces

Interfaces can extend other interfaces, inheriting their members.

```
interface Person {
    name: string;
    age: number;
}

interface Employee extends Person {
    employeeId: string;
    department: string;
}

const employee1: Employee = {
    name: "Bob",
    age: 40,
    employeeId: "EMP001",
```

```
    department: "Engineering"
};
```

## Implementing Interfaces (with Classes)

Classes can implement interfaces, ensuring they adhere to the interface's structure.

```
interface Shape {
    color: string;
    getArea(): number;
}

class Circle implements Shape {
    constructor(public radius: number, public color: string) {}

    getArea(): number {
        return Math.PI * this.radius * this.radius;
    }
}

class Rectangle implements Shape {
    constructor(public width: number, public height: number, public
color: string) {}

    getArea(): number {
        return this.width * this.height;
    }
}

const myCircle = new Circle(5, "blue");
console.log(myCircle.getArea()); // Output: 78.53981633974483

const myRectangle = new Rectangle(10, 20, "red");
console.log(myRectangle.getArea()); // Output: 200
```

## Type Aliases vs. Interfaces

- **Type Aliases (type)**: Can define primitive types, union types, intersection types, tuples, and more complex object shapes. They can also be used for literal types.
- **Interfaces (interface)**: Primarily used for defining the shape of objects.

**Key Differences:**

- **Declaration Merging:** Interfaces can be "merged" if declared multiple times with the same name (properties are combined). Type aliases cannot.
- **Extending/Implementing:** Interfaces can extend other interfaces and be implemented by classes. Type aliases can be extended using intersection types (&).
- **Readability:** For object shapes, interfaces are often preferred for their clear intent.

```
// Type Alias for a primitive
```

```typescript
type ID = string;
let userId: ID = "abc-123";

// Type Alias for a union type
type StringOrNumber = string | number;
let value: StringOrNumber = "hello";
value = 123;

// Type Alias for an object shape (similar to interface)
type Product = {
    id: number;
    name: string;
    price: number;
};

const product1: Product = { id: 1, name: "Laptop", price: 1200 };

// Type Alias for a function signature
type GreetFunction = (name: string) => string;
const sayHello: GreetFunction = (n) => `Hi, ${n}!`;
console.log(sayHello("Eve"));

// Extending Type Aliases (using intersection)
type BasicAddress = {
    street: string;
    city: string;
};

type FullAddress = BasicAddress & {
    zipCode: string;
    country: string;
};

const address: FullAddress = {
    street: "123 Main St",
    city: "Anytown",
    zipCode: "12345",
    country: "USA"
};

// Interface Declaration Merging Example
interface Settings {
    theme: string;
}

interface Settings {
    fontSize: number;
}
```

```
const appSettings: Settings = {
    theme: "dark",
    fontSize: 16
};
// If `Settings` was a type alias, this would be a duplicate
identifier error.
```

# 5. Classes

TypeScript supports object-oriented programming concepts through classes, similar to ES6 classes but with added type safety.

## Class Members (Properties, Methods)

```
class Greeter {
    // Property with type annotation
    greeting: string;

    // Constructor
    constructor(message: string) {
        this.greeting = message;
    }

    // Method
    greet(): string {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");
console.log(greeter.greet()); // Output: Hello, world
```

## Access Modifiers (public, private, protected)

- **public**: Accessible from anywhere (default).
- **private**: Accessible only within the class where it's defined.
- **protected**: Accessible within the class and by subclasses.

```
class Animal {
    public name: string;
    private age: number;
    protected species: string;

    constructor(name: string, age: number, species: string) {
        this.name = name;
```

```
        this.age = age;
        this.species = species;
    }

    public getAge(): number {
        return this.age;
    }

    protected getSpecies(): string {
        return this.species;
    }
}

class Dog extends Animal {
    constructor(name: string, age: number, breed: string) {
        super(name, age, "Canine"); // Call parent constructor
        this.breed = breed;
    }

    public breed: string;

    public describe(): string {
        // console.log(this.age); // Error: Property 'age' is private
        return `${this.name} is a ${this.breed} of species
${this.getSpecies()} and is ${this.getAge()} years old.`;
    }
}

let myDog = new Dog("Buddy", 5, "Golden Retriever");
console.log(myDog.name);     // Output: Buddy (public)
console.log(myDog.getAge()); // Output: 5 (public method accessing
private property)
// console.log(myDog.age);    // Error: Property 'age' is private
// console.log(myDog.species); // Error: Property 'species' is
protected
console.log(myDog.describe());
```

## Constructors

A special method for creating and initializing objects created within a class. Parameter properties (shorthand for declaring and initializing properties) are very common.

```
class Car {
    // Shorthand for declaring and initializing properties
    constructor(public brand: string, private _year: number) {}

    // Getter for a private property
    get year(): number {
```

```
        return this._year;
    }

    // Setter for a private property
    set year(newYear: number) {
        if (newYear > 1900) { // Basic validation
            this._year = newYear;
        } else {
            console.error("Year must be greater than 1900.");
        }
    }

    drive(): void {
        console.log(`Driving a ${this.brand} from ${this._year}.`);
    }
}

const myCar = new Car("Toyota", 2020);
console.log(myCar.brand); // Output: Toyota
console.log(myCar.year);  // Output: 2020 (using getter)
myCar.year = 2022;
console.log(myCar.year);  // Output: 2022
myCar.year = 1800; // Error message in console
myCar.drive();
```

## Inheritance

Classes can extend other classes, inheriting their properties and methods.

```
class Vehicle {
    constructor(public make: string, public model: string) {}

    start(): void {
        console.log(`${this.make} ${this.model} starting.`);
    }
}

class ElectricCar extends Vehicle {
    constructor(make: string, model: string, public batteryLife:
number) {
        super(make, model); // Call the parent class constructor
    }

    charge(): void {
        console.log(`${this.make} ${this.model} is charging. Battery:
${this.batteryLife}%`);
    }
```

```
    // Override parent method
    start(): void {
        console.log(`Electric ${this.make} ${this.model} silently 
starting.`);
    }
}

const tesla = new ElectricCar("Tesla", "Model 3", 90);
tesla.start(); // Output: Electric Tesla Model 3 silently starting.
tesla.charge();
```

## Abstract Classes

Abstract classes cannot be instantiated directly and often contain abstract methods (methods without an implementation) that must be implemented by concrete subclasses.

```
abstract class Shape2D {
    constructor(public name: string) {}

    abstract getArea(): number; // Abstract method - must be 
implemented by subclasses

    display(): void {
        console.log(`This is a ${this.name} with area: 
${this.getArea()}`);
    }
}

class Circle2D extends Shape2D {
    constructor(name: string, public radius: number) {
        super(name);
    }

    getArea(): number {
        return Math.PI * this.radius * this.radius;
    }
}

class Square2D extends Shape2D {
    constructor(name: string, public side: number) {
        super(name);
    }

    getArea(): number {
        return this.side * this.side;
    }
}
```

```
// const myShape = new Shape2D("Generic"); // Error: Cannot create an
instance of an abstract class.

const circle = new Circle2D("My Circle", 7);
circle.display(); // Output: This is a My Circle with area:
153.93804002589985

const square = new Square2D("My Square", 8);
square.display(); // Output: This is a My Square with area: 64
```

# 6. Generics

Generics provide a way to create reusable components that work with a variety of types rather than a single one. This allows you to write flexible, type-safe code.

## What are Generics?

They allow you to define type parameters that can be used in functions, classes, and interfaces.

## Generic Functions

```
// A generic function that returns the input value
function identity<T>(arg: T): T {
    return arg;
}

let output1 = identity<string>("myString"); // Type of output1 is
string
let output2 = identity<number>(100);       // Type of output2 is
number
let output3 = identity(true);              // Type inference: output3
is boolean

console.log(output1);
console.log(output2);
console.log(output3);

// A generic function that takes an array and returns its first
element
function getFirstElement<T>(arr: T[]): T | undefined {
    return arr.length > 0 ? arr[0] : undefined;
}

let firstString = getFirstElement(["apple", "banana", "cherry"]); //
Type: string
let firstNumber = getFirstElement([10, 20, 30]);                 //
```

```
Type: number
let firstBoolean = getFirstElement([true, false]);                //
Type: boolean
let empty = getFirstElement([]);                                  //
Type: undefined

console.log(firstString);
console.log(firstNumber);
```

### Generic Interfaces

```
interface Box<T> {
    value: T;
}

let stringBox: Box<string> = { value: "Hello" };
let numberBox: Box<number> = { value: 123 };

console.log(stringBox.value);
console.log(numberBox.value);

// Generic interface for a dictionary/map
interface Dictionary<K, V> {
    [key: string]: V; // Key must be string or number for index
signatures
}

let stringDictionary: Dictionary<string, string> = {
    "name": "Alice",
    "city": "New York"
};

let numberDictionary: Dictionary<string, number> = {
    "age": 30,
    "zip": 10001
};

console.log(stringDictionary.name);
console.log(numberDictionary.age);
```

### Generic Classes

```
class GenericList<T> {
    private items: T[] = [];

    addItem(item: T): void {
```

```
        this.items.push(item);
    }

    getItem(index: number): T | undefined {
        return this.items[index];
    }

    getAllItems(): T[] {
        return this.items;
    }
}

let stringList = new GenericList<string>();
stringList.addItem("TypeScript");
stringList.addItem("JavaScript");
console.log(stringList.getItem(0)); // Output: TypeScript

let numberList = new GenericList<number>();
numberList.addItem(10);
numberList.addItem(20);
console.log(numberList.getAllItems()); // Output: [10, 20]
```

## Generic Constraints

Sometimes you want to restrict the types that can be used with a generic. You can do this using extends.

```
interface Lengthwise {
    length: number;
}

// T must have a 'length' property
function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length); // Now we know it has a .length property
    return arg;
}

loggingIdentity({ length: 10, value: 3 }); // OK
// loggingIdentity(3); // Error: Argument of type 'number' is not
assignable to parameter of type 'Lengthwise'.

// Using a type parameter in a generic constraint
function getProperty<T, K extends keyof T>(obj: T, key: K) {
    return obj[key];
}

let myObj = { a: 1, b: 2, c: 3 };
console.log(getProperty(myObj, "a")); // Output: 1
```

```
// console.log(getProperty(myObj, "d")); // Error: Argument of type
'"d"' is not assignable to parameter of type '"a" | "b" | "c"'.
```

# 7. Advanced Types

## Union Types

A variable can be one of several types. Use the | (pipe) symbol.

```
function printId(id: number | string) {
    console.log(`Your ID is: ${id}`);
    // You can use type narrowing to handle different types
    if (typeof id === "string") {
        console.log(id.toUpperCase());
    }
}

printId(101);
printId("202abc");
// printId(true); // Error: Argument of type 'boolean' is not
assignable to parameter of type 'string | number'.
```

## Intersection Types

Combines multiple types into one. A variable of an intersection type must have all properties of all combined types. Use the & (ampersand) symbol.

```
interface Draggable {
    drag(): void;
}

interface Resizable {
    resize(): void;
}

type UIWidget = Draggable & Resizable;

let myWidget: UIWidget = {
    drag() { console.log("Dragging..."); },
    resize() { console.log("Resizing..."); }
};

myWidget.drag();
myWidget.resize();
```

## Type Guards (Type Narrowing)

Mechanisms to narrow down the type of a variable within a certain scope.

- **typeof Type Guard**: For primitive types (string, number, boolean, symbol, bigint, undefined, object, function).

```
function printValue(value: string | number) {
    if (typeof value === 'string') {
        console.log(value.toLowerCase());
    } else {
        console.log(value.toFixed(2));
    }
}
printValue("HELLO");
printValue(123.456);
```

- **instanceof Type Guard**: For classes.

```
class Cat {
    meow() { console.log("Meow!"); }
}
class Dog {
    bark() { console.log("Woof!"); }
}

type Animal = Cat | Dog;

function makeSound(animal: Animal) {
    if (animal instanceof Cat) {
        animal.meow();
    } else {
        animal.bark();
    }
}
makeSound(new Cat());
makeSound(new Dog());
```

- **in Operator Type Guard**: Checks if a property exists on an object.

```
interface Car {
    drive(): void;
}

interface Plane {
    fly(): void;
}

function move(vehicle: Car | Plane) {
    if ("drive" in vehicle) {
        vehicle.drive();
```

```
    } else {
        vehicle.fly();
    }
}

move({ drive: () => console.log("Driving car") });
move({ fly: () => console.log("Flying plane") });
```

- **User-Defined Type Guards (Type Predicates)**: Functions that return a boolean indicating a type. The return type is parameterName is Type.

```
interface Fish { swim(): void; }
interface Bird { fly(): void; }

function isFish(pet: Fish | Bird): pet is Fish {
    return (pet as Fish).swim !== undefined;
}

function getPetAction(pet: Fish | Bird) {
    if (isFish(pet)) {
        pet.swim();
    } else {
        pet.fly();
    }
}

getPetAction({ swim: () => console.log("Fish is swimming") });
getPetAction({ fly: () => console.log("Bird is flying") });
```

## Literal Types

Allows you to define a type that is exactly one specific string, number, or boolean value.

```
type CardinalDirection = "North" | "East" | "South" | "West";

let direction: CardinalDirection;
direction = "North"; // OK
// direction = "Northeast"; // Error: Type '"Northeast"' is not
assignable to type 'CardinalDirection'.

type HTTPMethod = "GET" | "POST" | "PUT" | "DELETE";

function handleRequest(url: string, method: HTTPMethod) {
    console.log(`Handling ${method} request for ${url}`);
}

handleRequest("/api/users", "GET");
```

## Nullable Types (strictNullChecks)

When strictNullChecks is true in tsconfig.json, null and undefined are not assignable to other types by default. You must explicitly allow them using union types.

```
// With "strictNullChecks": true
let username: string = "Alice";
// username = null; // Error: Type 'null' is not assignable to type
'string'.

let optionalUsername: string | null = "Bob";
optionalUsername = null; // OK

let maybeAge: number | undefined;
maybeAge = 25;
maybeAge = undefined; // OK
```

## Non-null Assertion Operator (!)

Tells the compiler that a value is not null or undefined, even if TypeScript's analysis can't prove it. Use with caution, as it bypasses type safety.

```
function greetUser(name: string | null | undefined) {
    // If you are absolutely sure name will not be null/undefined at
this point
    const displayName: string = name!; // Asserting that name is not
null or undefined
    console.log(`Hello, ${displayName.toUpperCase()}`);
}

greetUser("John");
greetUser(null); // This will cause a runtime error if `name` is
accessed without a check
```

## Type Assertions (as keyword, <>)

Tells the compiler to treat a value as a specific type. This does not perform any runtime checks or data conversions.

```
// Using 'as' keyword (preferred in TSX/React)
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;
console.log(strLength);

// Using '<>' (angle-bracket syntax - not allowed in TSX/React)
let anotherValue: any = 123;
let numValue: number = (<number>anotherValue);
console.log(numValue);
```

```
// Common use case: asserting DOM elements
const myCanvas = document.getElementById("myCanvas") as
HTMLCanvasElement;
if (myCanvas) {
    const ctx = myCanvas.getContext("2d");
    // ... now ctx is known to be CanvasRenderingContext2D or null
}
```

# 8. Type Inference

TypeScript's ability to automatically determine the type of a variable, function return, etc., without explicit type annotations.

```
// Variable initialization
let x = 3; // TypeScript infers x: number
// x = "hello"; // Error: Type '"hello"' is not assignable to type
'number'.

let y = [1, 2, 3]; // TypeScript infers y: number[]
// y.push("four"); // Error: Argument of type '"four"' is not
assignable to parameter of type 'number'.

// Function return types
function multiply(a: number, b: number) {
    return a * b; // TypeScript infers return type: number
}

// Object literals
let person = {
    name: "Alice",
    age: 30
};
// TypeScript infers person: { name: string; age: number; }
// person.email = "a@example.com"; // Error: Property 'email' does not
exist on type '{ name: string; age: number; }'.
```

## Contextual Typing

TypeScript can infer types based on the context in which a variable or expression is used.

```
window.addEventListener("click", (event) => {
    // 'event' is contextually typed as MouseEvent
    console.log(event.button);
    // console.log(event.keyCode); // Error: Property 'keyCode' does
not exist on type 'MouseEvent'.
});
```

````
const numbers = [1, 2, 3];
const doubled = numbers.map(num => {
    // 'num' is contextually typed as number
    return num * 2;
});
````

# 9. Declaration Files (.d.ts)

Declaration files describe the shape of JavaScript code (or other non-TypeScript code) so that TypeScript can understand it and provide type checking and autocompletion. They only contain type information, no implementation.

## When to Use Them

- When using a JavaScript library in a TypeScript project.
- When writing a TypeScript library that will be consumed by JavaScript projects.
- When defining ambient declarations for global variables or modules.

## Declaring Variables, Functions, Classes, Modules

**Example: Declaring a global variable and function** globals.d.ts:

````
declare var MY_GLOBAL_VAR: string;
declare function calculateSum(a: number, b: number): number;
```main.ts`:
```typescript
// Assuming MY_GLOBAL_VAR and calculateSum exist in a JS file or are provided by the environment
console.log(MY_GLOBAL_VAR);
let result = calculateSum(10, 20);
console.log(result);
````

**Example: Declaring a module (e.g., for a JS library without types)** my-library.d.ts:

````
declare module "my-library" {
    export function doSomething(input: string): string;
    export class MyClass {
        constructor(name: string);
        getName(): string;
    }
    export interface MyOptions {
        debug?: boolean;
        timeout: number;
    }
}
```app.ts`:
```typescript
````

```
import { doSomething, MyClass, MyOptions } from "my-library";

console.log(doSomething("hello"));

const instance = new MyClass("Test");
console.log(instance.getName());

const options: MyOptions = {
    timeout: 5000
};
```

Many popular JavaScript libraries have their declaration files available through @types/ packages on npm (e.g., npm install @types/react).

# 10. Modules

TypeScript supports ES Modules (import/export) for organizing code into separate files.

## import and export

**math.ts:**

```
export function add(x: number, y: number): number {
    return x + y;
}

export const PI = 3.14159;

export class Calculator {
    multiply(x: number, y: number): number {
        return x * y;
    }
}
```

**app.ts:**

```
import { add, PI, Calculator } from "./math"; // Relative path

console.log(add(5, 7)); // Output: 12
console.log(PI);       // Output: 3.14159

const calc = new Calculator();
console.log(calc.multiply(4, 6)); // Output: 24
```

## Default Exports, Named Exports

**user.ts:**

```
// Default export (only one per module)
```

```
export default class UserProfile {
    constructor(public name: string, public email: string) {}

    getInfo(): string {
        return `Name: ${this.name}, Email: ${this.email}`;
    }
}

// Named export
export const ADMIN_ROLE = "admin";
```

**main.ts:**

```
import UserProfile, { ADMIN_ROLE } from "./user"; // No curly braces
for default export

const user = new UserProfile("Jane Doe", "jane@example.com");
console.log(user.getInfo());

console.log(`Admin Role: ${ADMIN_ROLE}`);
```

# 11. Decorators (Experimental)

Decorators are a special kind of declaration that can be attached to classes, methods, accessors, properties, or parameters. They use the form @expression. To enable decorators, you need to set "experimentalDecorators": true and "emitDecoratorMetadata": true in your tsconfig.json.

```
{
  "compilerOptions": {
    // ... other options
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

## Class Decorators

Applied to the class constructor.

```
function sealed(constructor: Function) {
    Object.seal(constructor);
    Object.seal(constructor.prototype);
    console.log("Class has been sealed!");
}

@sealed
class BugReport {
```

```
    type = "report";
    title: string;

    constructor(title: string) {
        this.title = title;
    }
}

// const report = new BugReport("Fix bug");
// console.log(report.title);

// You can't add new properties to a sealed class
// (new BugReport("test") as any).newProp = "value"; // Error in
strict mode, runtime error otherwise
```

## Method Decorators

Applied to a method.

```
function logMethod(target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
    const originalMethod = descriptor.value;

    descriptor.value = function(...args: any[]) {
        console.log(`Calling method: ${propertyKey} with args:
${JSON.stringify(args)}`);
        const result = originalMethod.apply(this, args);
        console.log(`Method ${propertyKey} returned: ${result}`);
        return result;
    };

    return descriptor;
}

class UserService {
    @logMethod
    getUser(id: number, name: string): string {
        return `User with ID: ${id}, Name: ${name}`;
    }
}

const userService = new UserService();
userService.getUser(1, "Alice");
```

## Property Decorators

Applied to a property.

```
function format(formatString: string) {
    return function (target: any, propertyKey: string) {
        let value: string;

        const getter = function() {
            return value;
        };

        const setter = function(newVal: string) {
            value = formatString.replace("%s", newVal);
        };

        Object.defineProperty(target, propertyKey, {
            get: getter,
            set: setter,
            enumerable: true,
            configurable: true,
        });
    };
}

class Message {
    @format("Hello, %s!")
    greeting: string;

    constructor(text: string) {
        this.greeting = text;
    }
}

const msg = new Message("World");
console.log(msg.greeting); // Output: Hello, World!
```

## Parameter Decorators

Applied to a parameter in a method or constructor.

```
function logParameter(target: any, propertyKey: string | symbol,
parameterIndex: number) {
    console.log(`Parameter at index ${parameterIndex} of method
${String(propertyKey)} was decorated.`);
}

class Example {
    greet(@logParameter name: string, @logParameter age: number) {
        console.log(`Hello, ${name}! You are ${age} years old.`);
    }
}
```

```
const ex = new Example();
ex.greet("Bob", 25);
```

# 12. Utility Types

TypeScript provides several built-in utility types to facilitate common type transformations.

## Partial<T>

Constructs a type with all properties of T set to optional.

```
interface Todo {
    title: string;
    description: string;
    completed: boolean;
}

type PartialTodo = Partial<Todo>;

const todoUpdate: PartialTodo = {
    description: "Learn TypeScript",
    completed: true
};
// All properties are optional now
```

## Required<T>

Constructs a type consisting of all properties of T set to required.

```
interface UserInfo {
    id: number;
    name?: string;
    email?: string;
}

type FullUserInfo = Required<UserInfo>;

const newUser: FullUserInfo = {
    id: 1,
    name: "John Doe",
    email: "john@example.com"
};
// newUser.name = undefined; // Error: Type 'undefined' is not
assignable to type 'string'.
```

## Readonly<T>

Constructs a type with all properties of T set to readonly.

```
interface Point3D {
    x: number;
    y: number;
    z: number;
}

type ReadonlyPoint = Readonly<Point3D>;

const origin: ReadonlyPoint = { x: 0, y: 0, z: 0 };
// origin.x = 10; // Error: Cannot assign to 'x' because it is a read-only property.
```

## Pick<T, K>

Constructs a type by picking the set of properties K from T.

```
interface ProductDetails {
    id: number;
    name: string;
    price: number;
    description: string;
    category: string;
}

type ProductSummary = Pick<ProductDetails, "id" | "name" | "price">;

const laptopSummary: ProductSummary = {
    id: 101,
    name: "Gaming Laptop",
    price: 1500
};
```

## Omit<T, K>

Constructs a type by omitting the set of properties K from T.

```
interface Task {
    id: string;
    title: string;
    description: string;
    dueDate: Date;
    completed: boolean;
}
```

```
type NewTask = Omit<Task, "id" | "completed">; // For creating a new
task

const myNewTask: NewTask = {
    title: "Write documentation",
    description: "Complete the TypeScript notes.",
    dueDate: new Date()
};
```

### Exclude<T, U>

Constructs a type by excluding from T all union members that are assignable to U.

```
type AllColors = "red" | "green" | "blue" | "yellow" | "purple";
type PrimaryColors = "red" | "blue";

type NonPrimaryColors = Exclude<AllColors, PrimaryColors>; // "green"
| "yellow" | "purple"

let color: NonPrimaryColors = "green";
// color = "red"; // Error: Type '"red"' is not assignable to type
'"green" | "yellow" | "purple"'.
```

### Extract<T, U>

Constructs a type by extracting from T all union members that are assignable to U.

```
type MixedData = string | number | boolean | string[];
type StringsOnly = Extract<MixedData, string | string[]>; // string |
string[]

let data1: StringsOnly = "hello";
let data2: StringsOnly = ["a", "b"];
// let data3: StringsOnly = 123; // Error
```

### NonNullable<T>

Constructs a type by excluding null and undefined from T.

```
type NullableString = string | null | undefined;
type NonNullableString = NonNullable<NullableString>; // string

let text: NonNullableString = "some text";
// text = null; // Error
```

### Parameters<T>

Constructs a tuple type from the types of a function's parameters.

```
function greetPerson(name: string, age: number, isActive: boolean):
string {
    return `Hello ${name}, you are ${age} years old. Active:
${isActive}`;
}

type GreetParams = Parameters<typeof greetPerson>; // [name: string,
age: number, isActive: boolean]

const params: GreetParams = ["Charlie", 40, true];
// greetPerson(...params); // This would work
```

### ReturnType<T>

Constructs a type consisting of the return type of function T.

```
type GreetReturn = ReturnType<typeof greetPerson>; // string

let greetingResult: GreetReturn = "Welcome!";
// greetingResult = 123; // Error
```

### InstanceType<T>

Constructs a type consisting of the instance type of a constructor function type T.

```
class MyClass {
    constructor(public value: number) {}
    getValue() { return this.value; }
}

type MyClassInstance = InstanceType<typeof MyClass>; // MyClass

const instance: MyClassInstance = new MyClass(100);
console.log(instance.getValue());
```

# 13. Conditional Types

Conditional types allow you to express non-uniform type mappings. They take the form T extends U ? X : Y.

```
type IsString<T> = T extends string ? "Yes" : "No";

type Check1 = IsString<string>;  // "Yes"
type Check2 = IsString<number>;  // "No"
type Check3 = IsString<boolean>; // "No"
```

```
// Example: Get a property type if it's a string, otherwise never
type StringProperty<T, K extends keyof T> = T[K] extends string ? T[K]
: never;

interface UserProfile {
    name: string;
    age: number;
    email: string;
    isAdmin: boolean;
}

type UserNameType = StringProperty<UserProfile, "name">;    // string
type UserAgeType = StringProperty<UserProfile, "age">;      // never
type UserEmailType = StringProperty<UserProfile, "email">;  // string
type UserAdminType = StringProperty<UserProfile, "isAdmin">; // never
```

### infer keyword

The infer keyword is used within conditional types to "infer" a type that can then be used in the true branch of the conditional type.

```
// Infer the element type of an array
type ElementType<T> = T extends (infer U)[] ? U : T;

type ArrayElement = ElementType<string[]>; // string
type NonArrayElement = ElementType<number>;  // number

// Infer the return type of a function (similar to ReturnType utility
type)
type GetReturnType<T> = T extends (...args: any[]) => infer R ? R :
any;

type Func1Return = GetReturnType<() => string>;       // string
type Func2Return = GetReturnType<(x: number) => number[]>; // number[]
type NonFuncReturn = GetReturnType<boolean>;          // any
```

## 14. Mapped Types

Mapped types allow you to create new types by transforming the properties of an existing type. They iterate over the keys of a type using a [P in K] syntax.

```
// Make all properties of an object readonly
type ReadonlyProperties<T> = {
    readonly [P in keyof T]: T[P];
};

interface Coordinates {
```

```
    x: number;
    y: number;
}

type ImmutableCoordinates = ReadonlyProperties<Coordinates>;
// type ImmutableCoordinates = { readonly x: number; readonly y:
number; }

const coords: ImmutableCoordinates = { x: 10, y: 20 };
// coords.x = 30; // Error: Cannot assign to 'x' because it is a
read-only property.

// Make all properties optional
type OptionalProperties<T> = {
    [P in keyof T]?: T[P];
};

type PartialCoords = OptionalProperties<Coordinates>;
// type PartialCoords = { x?: number; y?: number; }

// Remove 'readonly' or '?' modifiers
type Mutable<T> = {
    -readonly [P in keyof T]: T[P]; // Remove readonly
};

type RequiredProps<T> = {
    [P in keyof T]-?: T[P]; // Remove optional
};

type MyReadonlyAndOptional = Readonly<Partial<Coordinates>>;
type MyMutableAndRequired =
RequiredProps<Mutable<MyReadonlyAndOptional>>;
// MyMutableAndRequired is equivalent to Coordinates

// Mapping to a new type (e.g., converting all properties to booleans)
type Flags<T> = {
    [P in keyof T]: boolean;
};

type UserFlags = Flags<UserProfile>;
/*
type UserFlags = {
    name: boolean;
    age: boolean;
    email: boolean;
    isAdmin: boolean;
}
*/
```

```
const userFlags: UserFlags = {
    name: true,
    age: false,
    email: true,
    isAdmin: true
};
```

# 15. Type Predicates

Type predicates are functions that return a boolean and inform the TypeScript compiler about the type of a variable if the function returns true. They use the parameterName is Type syntax in their return type.

```
interface Car {
    brand: string;
    drive(): void;
}

interface Bicycle {
    gears: number;
    pedal(): void;
}

function isCar(vehicle: Car | Bicycle): vehicle is Car {
    return (vehicle as Car).drive !== undefined;
}

function getVehicleDetails(vehicle: Car | Bicycle) {
    if (isCar(vehicle)) {
        console.log(`This is a car of brand: ${vehicle.brand}`);
        vehicle.drive();
    } else {
        console.log(`This is a bicycle with ${vehicle.gears} gears`);
        vehicle.pedal();
    }
}

getVehicleDetails({ brand: "Toyota", drive: () =>
console.log("Vroom!") });
getVehicleDetails({ gears: 21, pedal: () => console.log("Pedaling...")
});
```

# 16. ESNext Features & TypeScript

TypeScript aims to support upcoming JavaScript features as they become standardized.

## Async/Await

TypeScript fully supports async/await for asynchronous operations, providing type safety for Promises.

```
async function fetchData(): Promise<string> {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve("Data fetched successfully!");
        }, 1000);
    });
}

async function processData() {
    try {
        const data: string = await fetchData(); // Type inference for
'data'
        console.log(data);
    } catch (error) {
        // 'error' is of type 'unknown' by default with strict mode
        if (error instanceof Error) {
            console.error("Error fetching data:", error.message);
        } else {
            console.error("Unknown error:", error);
        }
    }
}

processData();
```

## Iterators/Generators

TypeScript provides types for iterators and generators.

```
function* idGenerator(): IterableIterator<number> {
    let id = 0;
    while (true) {
        yield id++;
    }
}

const generator = idGenerator();
console.log(generator.next().value); // 0
console.log(generator.next().value); // 1
console.log(generator.next().value); // 2

// Using for...of with iterators
for (const num of [1, 2, 3]) {
```

```
    console.log(num);
}
```

This comprehensive guide should give you a solid foundation in TypeScript, from its core concepts to more advanced type manipulations and features. Remember to practice by writing code and experimenting with different types and scenarios!