TypeScript-3

In TypeScript, modules are a way to organize and encapsulate your code into smaller, reusable pieces. Modules help you avoid naming conflicts, manage dependencies, and promote better code organization and maintainability. TypeScript supports two main module systems: CommonJS and ES6 modules.

CommonJS Modules:
CommonJS is a module system commonly used in server-side environments, like Node.js. It uses require to import modules and module.exports to export values.

Example:
Let's say you have two files, math.ts and app.ts.

math.ts:

```
// math.ts
export function add(a: number, b: number): number {
   return a + b;
}
```

app.ts:

```
// app.ts
const math = require('./math'); // Import the module

const result = math.add(3, 4); // Use the exported function
console.log(result);
```

To compile and run this code, you would typically use a tool like ts-node for TypeScript code execution in Node.js. CommonJS is primarily used on the server-side, so it may not work in a browser environment without a bundler like Webpack.

ES6 Modules:
ES6 modules are natively supported by modern browsers and can be used in both front-end and back-end development. They use import and export statements to handle module loading and exporting.

Example:
Let's adapt the previous example using ES6 modules.

math.ts:

```
// math.ts
export function add(a: number, b: number): number {
   return a + b;
}
```

app.ts:

```
// app.ts
import { add } from './math'; // Import the module

const result = add(3, 4); // Use the exported function
console.log(result);
```

To use ES6 modules in a browser, you can include the type="module" attribute in your script tag, like this:

```
<script type="module" src="app.js"></script>
```

This enables ES6 module support in the browser.

Namespace Modules:
TypeScript also supports namespace modules for grouping related code. This is useful for preventing naming conflicts.

Example:

```
// shapes.ts
namespace Shapes {
  export function circleArea(radius: number): number {
    return Math.PI * Math.pow(radius, 2);
  }
}
```

To use this namespace in another file:

```
// app.ts
const area = Shapes.circleArea(5);
console.log(area);
```

It's important to note that using namespaces is less common in modern TypeScript development, and ES6 modules are often preferred for their better support and tooling.

These are the main ways to work with modules in TypeScript. Your choice of module system will depend on your project's environment and requirements.

Triple-slash directives in TypeScript are special comments that provide instructions to the TypeScript compiler (tsc) about how to handle files and dependencies. These directives are typically placed at the top of a TypeScript file and help with various tasks like reference path resolution, module system settings, and more.

Here are the main triple-slash directives in TypeScript, along with examples for each:

/// <reference path="..."/>:
This directive is used to tell the TypeScript compiler about dependencies between files, especially when using out-file compilation.

```
// math.ts
function add(a: number, b: number): number {
  return a + b;
}

// app.ts
/// <reference path="math.ts" />
const result = add(3, 4);
console.log(result);
```

In this example, /// <reference path="math.ts" /> tells the TypeScript compiler that app.ts depends on math.ts.

/// <reference types="..." />:
This directive is used to include type definitions for TypeScript. It is often used to bring in type declarations for libraries that don't provide their own TypeScript type definitions.

```
// app.ts
/// <reference types="node" />
import fs from 'fs';
```
In this case, /// <reference types="node" /> instructs TypeScript to include type definitions for Node.js modules.

/// <reference lib="..." />:
This directive specifies which built-in TypeScript library files should be used. It's often used to customize the available APIs when targeting a specific environment.

```
// tsconfig.json
{
  "compilerOptions": {
    "lib": ["ES6", "DOM"]
  }
}
```
In this example, the lib option in tsconfig.json specifies that TypeScript should include the ECMAScript 6 and DOM libraries.

/// <amd-dependency path="..." />:
This directive is used with Asynchronous Module Definition (AMD) module systems to declare dependencies between AMD modules.

```
// app.ts
/// <amd-dependency path="jquery" />
define(["jquery"], ($) => {
  // Your AMD module code here
});
```
In this example, /// <amd-dependency path="jquery" /> informs the TypeScript compiler about the dependency on the "jquery" AMD module.

/// <amd-module:
This directive is used to specify the module name for AMD modules. It helps when defining AMD modules.

```
// myModule.ts
/// <amd-module name="myModule" />
export function myFunction() {
  // ...
}
```
When defining AMD modules, this directive helps ensure the correct module name is used.

It's important to note that with the advent of modern module systems like CommonJS, ES6 modules, and tools like Webpack, the use of triple-slash directives is becoming less common. Many of these directives are specific to older TypeScript workflows or used in conjunction with module bundlers and declaration files. When working on newer TypeScript projects, consider using modern module resolution and bundling techniques.

It seems there might be some confusion here. "Typed.js" and .d.ts files are two separate concepts in the context of TypeScript.

Typed.js:
"Typed.js" is not directly related to TypeScript. It is a JavaScript library that provides a simple way to create text typing animations, mimicking the effect of someone typing on a keyboard. You can use Typed.js to create dynamic and engaging text animations on your website. To use Typed.js in your TypeScript project, you would typically import it as an external JavaScript library and use TypeScript type annotations to provide type checking. Here's an example of how you might use Typed.js with TypeScript:

First, install Typed.js:

bash
Copy code
```
npm install typed.js
```
Then, in your TypeScript code:

```
import Typed from 'typed.js';

const options: TypedOptions = {
  strings: ['Hello, World!', 'Welcome to TypeScript!'],
  typeSpeed: 50,
};

const element = document.querySelector('.typed-output');
if (element) {
  new Typed(element, options);
}
```
.d.ts Files (Declaration Files):
TypeScript uses declaration files (with the .d.ts extension) to provide type information for JavaScript code that doesn't have TypeScript annotations. These files define the shape of JavaScript code and enable TypeScript to perform type checking and provide better code intelligence.

For libraries like Typed.js that don't have built-in TypeScript support, you can create a declaration file to provide type information. A declaration file typically describes the structure and types for the library. In the case of Typed.js, you might create a typed.js.d.ts file with declarations like this:

```
declare module 'typed.js' {
  class Typed {
    constructor(element: Element, options: TypedOptions);
  }

  interface TypedOptions {
    strings: string[];
    typeSpeed: number;
    // Add more options here as needed.
  }

  export default Typed;
}
```

By creating this declaration file, you allow TypeScript to understand Typed.js and provide type checking when you use it in your TypeScript code, as shown in the previous example.

In summary, "Typed.js" is a JavaScript library for text animations, while .d.ts files are TypeScript declaration files that provide type information for libraries and code that don't have native TypeScript support. When using external JavaScript libraries like Typed.js in a TypeScript project, you can create or find .d.ts files to enable type checking and code assistance.

Partial<Type>:

The Partial<Type> utility type makes all properties of a type optional by creating a new type with the same properties but marked as optional. It's handy when you want to create a new object with some optional properties.
Example 1:

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

type PartialPerson = Partial<Person>;

const partialInfo: PartialPerson = { name: "Alice" }; // age is optional
```
Example 2:

```
interface Product {
  id: number;
  name: string;
  price: number;
}

type PartialProduct = Partial<Product>;

const partialProduct: PartialProduct = { name: "Laptop" }; // id and price are optional
```
Required<Type>:

The Required<Type> utility type makes all properties of a type required by creating a new type with the same properties but without any optionality. This can be useful when you need to ensure all properties are present.
Example 1:

```
interface Configuration {
  host?: string;
  port?: number;
}

type RequiredConfig = Required<Configuration>;

const config: RequiredConfig = { host: "example.com", port: 8080 };
```
Example 2:

```
interface ContactInfo {
  email?: string;
  phone?: string;
}

type CompleteContactInfo = Required<ContactInfo>;

const contact: CompleteContactInfo = { email: "example@example.com", phone: "123-456-7890" };
```

Readonly<Type>:

The Readonly<Type> utility type creates a new type where all properties are marked as readonly, preventing modification of the object after its initialization.

Example 1:

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

type ReadonlyPoint = Readonly<Point>;

const point: ReadonlyPoint = { x: 1, y: 2 };
// point.x = 3; // Error: Cannot assign to 'x' because it is a read-only property.
```

Example 2:

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

type ImmutablePerson = Readonly<Person>;

const person: ImmutablePerson = { name: "Bob", age: 30 };
// person.age = 31; // Error: Cannot assign to 'age' because it is a read-only property.
```

Record<Keys, Type>:

The Record<Keys, Type> utility type creates a new type with keys from Keys mapped to values of type Type. It's useful for generating objects with specific keys and their corresponding types.

Example 1:

```
type Fruit = "apple" | "banana" | "cherry";
type FruitStock = Record<Fruit, number>;

const stock: FruitStock = { apple: 10, banana: 15, cherry: 5 };
```

Example 2:

```
type Days = "Monday" | "Tuesday" | "Wednesday" | "Thursday" | "Friday";
type WorkSchedule = Record<Days, string>;
```

```
const schedule: WorkSchedule = { Monday: "8 AM - 5 PM", Friday: "9 AM - 6 PM" };
```
These TypeScript utility types simplify common type transformations and help ensure type safety in your code. They are powerful tools for working with complex type definitions and improving the robustness of your TypeScript projects.

Certainly, let's continue with the explanation of two more TypeScript utility types and provide two examples for each.

Pick<Type, Keys>:

The Pick<Type, Keys> utility type creates a new type by selecting a subset of properties from the original type. It helps you extract specific properties you're interested in.
Example 1:

```
interface User {
  id: number;
  username: string;
  email: string;
  isAdmin: boolean;
}

type UserSummary = Pick<User, "username" | "email">;

const user: UserSummary = { username: "alice", email: "alice@example.com" };
```
Example 2:

```
interface Product {
  id: number;
  name: string;
  price: number;
}

type ProductInfo = Pick<Product, "name" | "price">;

const product: ProductInfo = { name: "Laptop", price: 999.99 };
```
Omit<Type, Keys>:

The Omit<Type, Keys> utility type creates a new type by excluding specific properties from the original type. It's the opposite of Pick and allows you to remove properties you don't need.
Example 1:

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

type TaskPreview = Omit<Task, "description">;

const task: TaskPreview = { id: 1, title: "Finish project", completed: false };
```
Example 2:

```
interface Employee {
  id: number;
  name: string;
  position: string;
  salary: number;
}

type EmployeeSummary = Omit<Employee, "salary">;

const employee: EmployeeSummary = { id: 101, name: "John Doe", position: "Software Engineer" };
```

Now, let's explore two more TypeScript utility types:

Exclude<Type, ExcludedUnion>:

The Exclude<Type, ExcludedUnion> utility type creates a new type that excludes values from a union type. It's commonly used when you want to remove specific values from a set of values.

Example 1:

```
type Color = "red" | "green" | "blue" | "yellow";
type PrimaryColors = Exclude<Color, "green" | "yellow">;

const primaryColor: PrimaryColors = "red"; // Excludes "green" and "yellow" from the union type.
```

Example 2:

```
type Animal = "dog" | "cat" | "bird" | "fish";
type Mammals = Exclude<Animal, "bird" | "fish">;

const mammal: Mammals = "dog"; // Excludes "bird" and "fish" from the union type.
```

Extract<Type, IncludedUnion>:

The Extract<Type, IncludedUnion> utility type creates a new type that includes only values from a union type that match the specified type. It's the opposite of Exclude.

Example 1:

```
type Fruit = "apple" | "banana" | "cherry" | "kiwi";
type TropicalFruits = Extract<Fruit, "banana" | "kiwi">;

const tropicalFruit: TropicalFruits = "banana"; // Includes only "banana" and "kiwi" from the union type.
```

Example 2:

```
type Vehicle = "car" | "bus" | "bike" | "boat";
type LandVehicles = Extract<Vehicle, "car" | "bike">;

const landVehicle: LandVehicles = "bike"; // Includes only "car" and "bike" from the union type.
```

These utility types in TypeScript are powerful tools for manipulating and working with types effectively, helping you to create more expressive and type-safe code. They are especially useful when working with complex type structures and avoiding common mistakes.