Step00a_json_objects:

JSON (Javascript Object Notation)

1. JSON stands for Javascript Object Notation.
2. JSON is a text-based data format that is used to store and transfer data.
3. // JSON syntax

```
"name": "Vipin",

"age": 21,

"gender": "male",
```

But wait, Is JSON is similar to javaScript objects?

The Answer is No.

1. JavaScript objects can contain functions but JSON not.
2. JavaScript objects can only be used in JavaScript but JSON can be created and used by other programming languages.

JSON Data

1. JSON data consists of key/value pairs similar to JavaScript object properties.
2. The key and values are written in double quotes separated by a :.
3. Example :

// JSON data

```
  "name": "Vipin"
```

4. JSON data requires double quotes for the key.

JSON Object

1. The JSON object is written inside curly braces { }.
2. JSON objects can contain multiple key/value pairs.
3. Example :

```
// JSON object

{ "name": "Vipin", "age": 21 }
```

JSON Array

1. JSON array is written inside square brackets [ ].
2. Example :

```
// JSON array

[ "Vipin", "Ankit", "Raj"]
```

Accessing JSON Data

1. We can access JSON data using the dot notation.
2. Example :

```
// JSON object

const detail = { "name": "Vipin", "age": 21 }

// accessing JSON object

console.log(detail. name); // Vipin
```

1. We can also use square bracket syntax [] to access JSON data.
2. Example :

```
// JSON object

const detail = {

   "name": "Vipin",

   "age": 21
}
```

```
// accessing JSON object

console.log(detail["age"]); // Vipin
```

Use of JSON

1. JSON is the most commonly used format for transmitting data (data interchange) from a server to a client and vice-versa.
2. JSON data are very easy to parse and use.
3. JSON is language independent(We can create and use JSON in other programming languages too).

Step00b_syntax_error:

A syntax error happens when the code doesn't follow the correct structure or rules of the programming language.

Example 1: Missing Quotes

```
let name = Tom; // Missing quotes around 'Tom'
console.log(name);
```

Explanation: Textual values (strings) should be enclosed in quotes. Forgetting quotes can confuse TypeScript and lead to a syntax error.

Example 2: Incorrect Function Definition

```
function greet(name: string) // Missing opening curly brace
  console.log("Hello, " + name);
}
greet("Alice");
```

Explanation: Functions need opening and closing curly braces to define their scope. Missing one causes a syntax error.

Example 3: Misspelled Keyword

```
le num = 7; // Misspelled 'let' keyword
console.log(num);
```

Explanation: Misspelling keywords like 'let' leads to syntax errors because TypeScript doesn't recognize the incorrect spelling.

Step00c_type_error:

A type error in TypeScript occurs when a variable or function is used in a way that's incompatible with its expected type.

Example 1: Incorrect Function Usage

```
let numberValue: number = 10;
console.log(numberValue.toUpperCase()); // Type 'number' doesn't have 'toUpperCase' method
```

Example 2: Incorrect Property Access

```
let person: { name: string, age: number } = { name: "Alia", age: 25 };
console.log(person.job); // Property 'job' does not exist on type '{ name: string; age: number; }'
```

Explanation: The person object doesn't have a property called job. TypeScript alerts about this discrepancy between the expected properties and the actual usage, indicating a type error.

Example 3: Incompatible Function Argument

```
function greet(name: string) {
  return "Assalamualaikum, " + name;
}

let result: string = greet(10); // Argument of type 'number' is not assignable to parameter of
type 'string'
console.log(result);
```

Explanation: The greet function expects a string argument, but a number (10) is provided instead. This mismatch in types triggers a type error.

Step00d_assignability_error:

An assignability error in TypeScript occurs when you try to assign a value to a variable that doesn't match its expected type.

Example 1: Incorrect Value Assignment to String Variable

```
let message: string = "Assalamualaikum World";
message = 6; // Type 'number' is not assignable to type 'string'
console.log(message);
```

Explanation: The variable message is declared as a string, but the assignment of a number (6) to it causes an assignability error since TypeScript expects a string.

Example 2: Assigning Incompatible Object Types

```
let person: { name: string, age: number } = { name: "Rida", age: 23 };
person = { name: "Bob", job: "Engineer" }; // Property 'job' is missing in type '{ name: string; }'
but required in type '{ name: string; age: number; }'
console.log(person);
```

Example 3: Assigning Values to Constants

```
const pi: number = 3.14;
```

```
pi = 3; // Cannot assign to 'pi' because it is a constant
console.log(pi);
```

Explanation: Constants (declared with const) cannot be reassigned. Trying to assign a new value to a constant causes an assignability error.

Step01_strong_typing:

Strong typing in TypeScript refers to the explicit declaration of variable types and enforcing those types strictly.

Example 1: Explicitly Declared Types

```
// Strongly typed syntax
let country: string = "Pakistan";
country = "USA"; // Valid: country is explicitly set to hold strings

let age: number = 9;
let isRaining: boolean = true;
```

Explanation: In this example, each variable (country, age, isRaining) is explicitly declared with its respective type. TypeScript ensures that these variables can only hold values that match their declared type

Example 2: Type Inference

```
// Type inference
let place = "USA";
let temperature = 10.9;
temperature = 22; // Valid: TypeScript infers 'temperature' as a number after reassignment

let isSunny = false;
isSunny = true; // Valid: 'isSunny' is inferred as a boolean
```

Explanation: TypeScript infers the types (string, number, boolean) based on the initially assigned values. It then keeps track of these inferred types for subsequent assignments, ensuring the variables hold compatible values.

Example 3: Benefits of Strong Typing

```
let userName: string = "Alice";
userName = 25; // Error: Type 'number' is not assignable to type 'string'
```

Explanation: Strong typing helps catch potential errors during development. In this case, assigning a number to a variable declared as a string causes a type error, which TypeScript detects before running the code.

step02_const_let:

In TypeScript, const and let are used to declare variables, each with its own characteristics regarding mutability and scope.

In TypeScript, const and let are used to declare variables, each with its own characteristics regarding mutability and scope.

Const: Immutable Variables

```
const pi: number = 3.14;
pi = 3; // Error: Cannot assign to 'pi' because it is a constant
```

Explanation: Variables declared with const are immutable; their values cannot be reassigned once set. They maintain their value throughout their scope.

Let: Mutable Variables

```
let age: number = 25;
age = 30; // Valid: 'let' allows reassignment of values
```

Explanation: Variables declared with let are mutable; their values can be changed or reassigned within their scope.

Block Scope and Function Scope

```
// Block Scope
function greet() {
  if (true) {
    let message = "Assalamualaikum";
    const name = "Zahid";
    console.log(message); // 'message' and 'name' accessible within this block
  }
  console.log(message); // Error: 'message' is not accessible outside the block
  console.log(name); // Error: 'name' is not accessible outside the block
}

// Function Scope
function calculate() {
  let result = 10;
  if (result > 5) {
    let multiplier = 2;
    console.log(result * multiplier); // Accessible within the block
  }
  console.log(multiplier); // Error: 'multiplier' is not accessible outside its block
console.log(result); // Accessible within the block

}
```

- Variables declared with let and const have block-level scope, meaning they are accessible only within the block where they are defined.
- In the examples, message and name are accessible within their respective blocks but not outside them due to block scope.
- Similarly, multiplier is accessible within its block, demonstrating block-scoped variables.

step03a_modules

In TypeScript, modules are used to organize code into separate files and to control the visibility of variables, functions, classes, etc. between different files. The import and export keywords are used to manage the accessibility of these entities between modules.

Exporting in names.ts:

```
// 1st way of export
export let myName: string = 'Rana Ali Zeeshan';

// 2nd way of export
let myAnotherName: string = 'Ahmed';
export { myAnotherName };

// 3rd way of export
let myThirdName: string = 'Sara';
export default myThirdName;
```

Explanation (Export):

1. **1st way of export: myName** is directly exported with **export**. This allows it to be imported using its name from other files.
2. **2nd way of export: myAnotherName** is exported separately using **export { myAnotherName }**. It's not a default export and requires its specific name during import.
3. **3rd way of export: myThirdName** is exported as the default export using **export default**. There can only be one default export per file.

main.ts:

```
// Importing named exports
import { myName as name, myAnotherName } from "./names";

// Importing default export with an optional alias
import newName from "./names";

// Displaying the imported values
```

```
console.log(`${name} ${myAnotherName} ${newName}`);
```

Explanation (Import):

- **Importing named exports: import { myName as name, myAnotherName }** fetches **myName** as **name** and **myAnotherName** as exported from **names.ts**. Names can be aliased as needed.
- **Importing default export: import newName** fetches the default export (**myThirdName**) from **names.ts**. An optional alias can be used for clarity, but it's not required.

Key Points:

- The default export (**export default**) allows you to export one main thing from a file.
- Named exports (**export { ... }**) allow you to export multiple things, and they must be referenced by their names during import.
- The default export doesn't require curly braces **{}** during import, and you can provide an optional alias for clarity.
- Named exports require curly braces during import and must use the exact exported name.

The Default Export (export default):

- **Explanation:** The **export default** statement in TypeScript allows you to export just one main thing from a file. This main thing can be any value, function, class, or object.
- **Usage Example:** Suppose you have a variable **myData** that you want to be the primary export from a file. You'd write **export default myData;**.
- **During Import:** When importing the default export, you don't need to use curly braces **{}**. You can also give it an optional alias, like **import newData from "./data";**.

Named Exports (export { ... }):

- **Explanation:** With named exports in TypeScript (**export { ... }**), you can export multiple values, functions, classes, or objects from a file, each using their specific names.
- **Usage Example:** If you have variables **var1**, **var2**, and **var3** that you want to export, you'd use **export { var1, var2, var3 };**.
- **During Import:** When importing named exports, you need to use curly braces **{}** and refer to the exported names exactly as they are written in the file, like **import { var1, var2 } from "./data";**.

Key Difference During Import:

- **Default Export:** No curly braces are needed, and an alias can be provided.
- **Named Export:** Curly braces are required, and you must use the exact exported names.