Functions:
Functions programming mein reusable blocks of code hote hain jo kisi specific task ya operation ko perform karne ke liye likhe jaate hain. Yeh task ho sakta hai kisi calculation, processing, data manipulation, ya koi aur functionality jaise ki output print karna, file se data read karna, etc.

Functions programming mein reusable blocks of code hote hain jo kisi specific task ya operation ko perform karne ke liye likhe jaate hain. Yeh task ho sakta hai kisi calculation, processing, data manipulation, ya koi aur functionality jaise ki output print karna, file se data read karna, etc.

Functions kyun use karte hain?

1. **Reusability:** Functions ko multiple times use karke code ki reusability ko badhaya ja sakta hai. Agar ek specific task ko bar-bar repeat karne ki zarurat ho, toh us task ko ek baar function mein define karke baar-baar us function ko call kiya ja sakta hai.
2. **Modularity:** Functions code ko modular banate hain. Code ko small, manageable, aur maintainable parts mein divide karne mein help karte hain. Har function ek specific task ko perform karta hai, jisse overall code ko samajhna aur maintain karna aasaan ho jaata hai.
3. **Abstraction:** Functions abstract (hiding inner details) details aur complexity ko. Jab aap ek function ko call karte hain, aapko pata nahi hota ki function ke andar kis tarah se kaam ho raha hai. Aapko sirf function ka use karna hota hai aur aapko function ka output milta hai.
4. **Readability:** Functions code ko readable banate hain. Name se hi pata chalta hai ki function kya kaam karta hai, isliye functions ko use karke code ko padhna aur samajhna aasaan ho jaata hai.

Functions programming ka ek important concept hain jo code ko organized aur maintainable banata hai. Isse errors ko find karna, code ko update karna, code ko debug karna aur overall development ko improve karna aasaan ho jaata hai.

Types of Functions:

1. Parameterized Function:

```
// Parameterized Function: Adds two numbers and returns the result
function addNumbers(a: number, b: number): number {
   return a + b;
}

let result1 = addNumbers(5, 3); // Calling the function with specific values
console.log("Result of adding 5 and 3:", result1); // Output: 8
```

```
let result2 = addNumbers(10, 7); // Calling the function again with different values
console.log("Result of adding 10 and 7:", result2); // Output: 17
```

Yeh function addNumbers hai jo do numbers ko add karke result return karta hai. Is function ko alag-alag values ke saath multiple baar call kiya ja sakta hai.

2. Non-parameterized Function:

```
// Non-parameterized Function: Prints a predefined message
function greet(): void {
   console.log("Hello, welcome to TypeScript!"); // Simple print statement
}

greet(); // Calling the function to print the greeting message
```

Yeh function greet hai jo predefined message ko print karta hai. Ismein koi parameter nahi hai aur yeh bas predefined message ko print karta hai jab bhi ise call kiya jaata hai.

Aisy functions me koi calculation b ho sakti hai jis me koi inpurt (params) required na ho.

Function Calling:

Jab aap function ko call karte hain, to program ki execution (control) us function ke line se shuru hoti hai jahaan se aapne function ko call kiya hai. Jab function ka code execute hota hai aur function khatam hota hai, tab control wapas wohi jagah jata hai jahaan se function call hua tha.

```
// Function definition
function greet() {
   console.log("Hello, welcome!"); // Ek greeting message print karen
}

console.log("Start"); // "Start" print karen

greet(); // Function call

console.log("End"); // "End" print karen
```

1. **"Start"** print hone se pehle **console.log("Start")** line execute hoti hai.
2. Phir **greet()** function call hoti hai. Control **greet()** function ke andar jaata hai, jahan **console.log("Hello, welcome!")** line execute hoti hai aur **"Hello, welcome!"** print hota hai.
3. Function ke code execute hone ke baad, control wapas us jagah par jaata hai jahaan se function call hua tha. Yani, **greet()** function ke call ke baad control **console.log("End")** line pe jaata hai aur **"End"** print hota hai.

Function ke andar jo bhi calculations ya operations hote hain, unke baad control wapas function call ke jagah aa jata hai, aur jo bhi function se return hota hai, vo wahan se access kiya ja sakta hai.

```
// Function definition: Adds two numbers and returns the result
function addNumbers(a: number, b: number): number {
   let sum = a + b; // Do numbers ka sum calculate karen
   return sum; // Calculate kiya hua sum return karen
}

let result = addNumbers(5, 3); // Function call karke result variable mein store karen
console.log("Result of adding 5 and 3:", result); // Output: 8
```

1. **addNumbers** function do numbers ko add karke unka sum calculate karta hai.
2. **return** keyword se **sum** variable ka value return kiya jata hai, jo ki **addNumbers** function ke call hote waqt **result** variable mein store ho jaata hai.
3. **console.log** ke through result print hota hai jo function ne calculate kiya aur return kiya hai.

Yahan, function mein jo bhi calculations hote hain, unka result **return** keyword se wapas function call ke jagah par aata hai, jahaan se function ko call kiya gaya tha. Isi tarah se function se aap koi bhi value **return** kar sakte hain aur use function call ke baad apne code mein use kar sakte hain.

Reference:

Jab aap ek function ko call karte hain, to aap actually us function ke "reference" ko use kar rahe hote hain. Reference, asal mein woh jagah ya location hai jahaan function ka code store hota hai. Jab aap function ko call karte hain, aap program ko keh rahe hote hain ki vo code execute ho aur us function ka result le kar aaye.

```
// Function definition: Adds two numbers and returns the result
function addNumbers(a: number, b: number): number {
    let sum = a + b; // Calculate the sum of two numbers
    return sum; // Return the calculated sum
}

let result = addNumbers(5, 3); // Function call
console.log("Result of adding 5 and 3:", result); // Output: 8
```

Yahan addNumbers function ka reference addNumbers(5, 3) ke through use kiya gaya hai. Jab aap addNumbers(5, 3) likhkar function ko call karte hain, program us reference ke saath jata hai jahaan addNumbers ka code likha hua hai. Vahaan se 5 aur 3 ko add karke 8 ka result calculate hota hai aur woh result result variable mein store ho jata hai.

Is tarah se, function call karte waqt aap asal mein us function ke code ka reference use kar rahe hote hain jo aapne pehle define kiya hota hai.

reference ek aisi value hoti hai jo ek variable, function, ya object ke location ya address ko point karti hai. Jab aap kisi function ko call karte hain, aap asal mein us function ke reference ko use kar rahe hote hain jisse program ko pata chalta hai ki woh kis location par hai aur uska code kahaan store hai.

Isi tarah se, agar aap kisi function ko call karte hain, to aap us function ke reference ko use kar rahe hote hain jisse program us function ke code tak pahunch sake aur uska execution ho sake. Jis tarai addNumbers(5, 3) function call ke through aap actually addNumbers function ke code tak pahunch rahe hain aur uska result generate ho raha hai

Type of Params:

Required Parameters:

- Parameters that are defined in the function signature and are required when calling the function. The values are passed in the same order as their corresponding parameters.
- Example:

```
function addNumbers(a: number, b: number): number {
        return a + b;

}


let result = addNumbers(5, 3);
```

Default Parameters:

Default parameters allow initializing function parameters with default values. If a value is not provided during the function call, the default value is used.

```
function greet(name: string = "Guest"): void {
    console.log("Hello, " + name);
}

greet(); // Output: Hello, Guest
greet("John"); // Output: Hello, John
```

Rest Parameters:

Rest parameters allow functions to accept an indefinite number of arguments as an array.

```
function sum(...numbers: number[]): number {
    return numbers.reduce((total, num) => total + num, 0);
}

let total = sum(1, 2, 3, 4, 5);
```

Optional Parameters:

Optional parameters are parameters that may or may not receive a value during the function call.

Any optional parameters for a function must be the last parameters. Placing

an optional parameter before a required parameter would trigger a

TypeScript syntax error:

```
function printMessage(message?: string): void {
    if (message) {
        console.log("Message: " + message);
    } else {
        console.log("No message provided.");
    }
}

printMessage(); // Output: No message provided.
printMessage("Hello!"); // Output: Message: Hello!
```

Note:

function call karte waqt arguments ka sequence function ke definition ke parameters ke sequence ke sath milna chahiye. Iske ilawa, data types bhi wohi honi chahiye jo function ke parameters mein define ki gayi hain.

```
// Function definition
function addNumbers(a: number, b: number): number {
    return a + b;
}

// Function call with correct sequence and data types
let result1 = addNumbers(5, 3);

// Function call with incorrect sequence and data types (will result in an error)
let result2 = addNumbers("Hello", 3); // Error: Argument of type 'string' is not
assignable to parameter of type 'number'
```

Function Return Type:

Explicit Return Type:

An explicit return type is declared explicitly by specifying the type of value that the function will return.

Return Type: The return type of a function is declared using a colon (:) followed by the desired type after the parameter list and before the opening curly brace { of the function body.

```
// Function returning a number
function add(a: number, b: number): number {
    return a + b;
}

let result = add(5, 3); // result will be of type number
console.log(result); // Output: 8
```

Implicit Return Type:

When a return type is not explicitly declared, TypeScript infers the return type based on the value returned by the function.

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

In this case, the return type of the greet function is not explicitly specified. However, TypeScript infers that the function will return a string type value because the return statement returns a string concatenation.

Function Types:

In TypeScript, you can specify the types of functions. This involves defining the types of parameters the function takes and the type of value it returns.

```
let nothingInGivesString: () => string;
let inputAndOutput: (songs: string[], count?: number) => number;
```

- **nothingInGivesString** is a variable that can hold a reference to a function. It's expected to be a function that takes no parameters and returns a string.
- **inputAndOutput** is another variable that can reference a function. This function should take an array of strings (**songs**) as the first parameter and an optional **count** of type **number**. It returns a **number**.

```
// Function matching nothingInGivesString type (no parameters, returns a string)
function getString(): string {
   return "This is a string.";
}

// Function matching inputAndOutput type
function processSongs(songs: string[], count: number = 0): number {
   console.log(`Total songs: ${songs.length}, Count: ${count}`);
   return songs.length + count;
}
```

- **getString()** is a function that returns a string. It matches the **nothingInGivesString** type as it takes no parameters and returns a string.
- **processSongs()** takes an array of strings and an optional count. It logs the total songs and count, then returns the sum of the array's length and the count value. This function matches the **inputAndOutput** type.

By defining function types, TypeScript helps enforce the structure of functions by ensuring that assigned functions match the specified types. If a function structure doesn't align with its defined type, TypeScript will show type-checking errors during compilation.

TypeScript annotations using parentheses.

Function Types and Union Types:

Function Types:

In TypeScript, you can define the types of functions. This includes specifying the types of parameters and the return type of the function.

Union Types:

Union types in TypeScript allow variables to have multiple types. It means a variable can hold values of different types.

```
// Type is a function that returns a union: string | undefined
let returnsStringOrUndefined: () => string | undefined;

// Type is either undefined or a function that returns a string
let maybeReturnsString: (() => string) | undefined;
```

Explanation:

- returnsStringOrUndefined is a variable that holds a function type. This function doesn't take any parameters (() =>) and can return either a string or undefined. This is represented as () => string | undefined.
- maybeReturnsString is another variable. It can either hold a reference to a function or undefined. This function, if present, takes no parameters and returns a string. The | undefined indicates that the variable can also be undefined.

Parentheses ( ) are used in union types to indicate which part of the annotation is related to the function return or the surrounding union type.

- In returnsStringOrUndefined, the parentheses enclose the entire function type (() => string | undefined), indicating that the entire function returns either a string or undefined.
- In maybeReturnsString, the parentheses only enclose the function part (() => string) within the union type, indicating that the union type can be either a function returning a string or undefined.

This notation helps clarify whether the entire union type relates to the function return type or just a part of it within the union.

Parameter Type Inferences:

In TypeScript, the compiler can automatically infer or deduce the types of function parameters based on their context or usage. This helps in scenarios where declaring types for each function parameter could be burdensome.

Explanation with Examples:

Scenario 1: Assigning a Function to a Variable

```
// Declare a variable 'singer' with a function type
let singer: (song: string) => string;

// Assign a function to the 'singer' variable
singer = function (song) {
   // TypeScript infers that 'song' is of type string
   return `Singing: ${song.toUpperCase()}!`; // This is okay
};
```

- Here, singer is declared as a variable that can hold a function. This function is expected to take a parameter named song of type string and return a string.
- When you assign a function to singer, TypeScript automatically infers that the song parameter in the assigned function is of type string. You don't explicitly have to declare it.

Scenario 2: Functions Passed as Arguments

```
const songs = ["Call Me", "Jolene", "The Chain"];

// TypeScript infers that 'song' is of type string and 'index' is of type number
songs.forEach((song, index) => {
   console.log(`${song} is at index ${index}`);
});
```

- In the forEach function, TypeScript infers the types of parameters song and index based on the array being iterated.
- Since songs is an array of strings, TypeScript knows that the song parameter in the callback function is of type string.
- Similarly, since forEach provides the index of the array elements, TypeScript infers that the index parameter is of type number.

Summary:

In both scenarios, TypeScript makes our life easier by automatically figuring out the types of function parameters based on their usage or the context in which they are assigned. This reduces the need for explicit type declarations, making our code more concise and readable.

Function type alias:

Function type aliases in TypeScript provide a way to name a function type, making it easier to reuse and describe the structure of functions.

Example 1: StringToNumber Type Alias

```
// Define a type alias named 'StringToNumber' for a function type
type StringToNumber = (input: string) => number;

// Declare a variable 'stringToNumber' with the 'StringToNumber' type
let stringToNumber: StringToNumber;

// Assign functions to 'stringToNumber' variable
stringToNumber = (input) => input.length; // Ok
stringToNumber = (input) => input.toUpperCase(); // Error: Type 'string' is not assignable to
type 'number'.
```

- Here, StringToNumber is a type alias representing a function that takes a string as input and returns a number.
- TypeScript enforces that any function assigned to stringToNumber adheres to this type.
- The first assignment works as input.length returns a number, aligning with the defined function type.
- The second assignment (input.toUpperCase()) results in an error because toUpperCase() returns a string, not a number as required by the StringToNumber type alias.

Example 2: NumberToString Type Alias Within a Function Parameter

```
// Define a type alias named 'NumberToString' for a function type
type NumberToString = (input: number) => string;

// Function that takes a parameter of type 'NumberToString' alias
function usesNumberToString(numberToString: NumberToString) {
   console.log(`The string is: ${numberToString(1234)}`);
}
```

```
// Function calls using the 'NumberToString' alias
usesNumberToString((input) => `${input}! Hooray!`); // Ok
usesNumberToString((input) => input * 2); // Error: Type 'number' is not assignable to type
'string'.
```

- NumberToString is another type alias representing a function that accepts a number and returns a string.
- The usesNumberToString function takes a parameter (numberToString) of type NumberToString.
- The first function call within usesNumberToString is valid as the provided function returns a string.
- The second function call results in an error because the provided function (input * 2) returns a number, not a string as expected by NumberToString.

Summary:

Type aliases for function types help improve code readability and reusability by providing descriptive names for complex function types. They allow TypeScript to enforce specific function structures across codebases without needing to repeatedly define parameter and return types.

void Return Type:

- void is used as the return type of a function when that function doesn't return any value explicitly.
- It indicates that the function does not produce any meaningful value.

```
function greet(name: string): void {
   console.log(`Hello, ${name}!`);
   // No return statement or return value specified
}
```

- In the greet function, we log a greeting message but don't return any value.
- The void return type signifies that the function doesn't produce a meaningful output or value.

never Return Type:

Think of never as a type representing situations where a function will never return a normal value or complete its execution.

Throwing Errors:

- When a function throws an error, it doesn't return a value but instead halts execution due to the error.

```
function throwError(message: string): never {
    throw new Error(message);
}
```

Infinite Loops:

- Functions that contain infinite loops never finish their execution normally.

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

Conditional Branches that Never Occur:

- A function might have conditional branches that TypeScript knows will never execute due to the logic flow.

```
function neverEndingOperation(x: string): never {
    if (typeof x === "string") {
        // This block will never execute because 'x' is already known to be a string
    } else {
        // This block also won't execute
    }
    // TypeScript knows that this function never finishes normally
    // because both if and else blocks are never executed.
    throw new Error("Unexpected type");
}
```

Summary:

- **never** represents scenarios where a function doesn't return normally due to throwing errors, entering infinite loops, or containing unreachable code branches.
- It's a way for TypeScript to express situations where a function's execution doesn't produce a regular value or doesn't complete.

Function overloading:

Function overloading in TypeScript allows a function to have multiple signatures or definitions. This enables the function to behave differently based on the types or number of arguments it receives.

Example:

Let's say you want a function called add that can add numbers or concatenate strings based on the type of arguments it receives.

```
function add(a: number, b: number): number; // Function signature for adding numbers
function add(a: string, b: string): string; // Function signature for concatenating strings

// Function implementation using overloads
function add(a: number | string, b: number | string): number | string {
    if (typeof a === 'number' && typeof b === 'number') {
        return a + b; // Adding numbers
    } else if (typeof a === 'string' && typeof b === 'string') {
        return a + b; // Concatenating strings
    } else {
        throw new Error('Invalid arguments');
    }
}

// Calling the 'add' function with different types of arguments
console.log(add(5, 3)); // Output: 8 (number)
console.log(add('Hello', ' World')); // Output: Hello World (string)
```

- We define multiple function signatures using the same function name (**add**).
- The function signatures specify different combinations of argument types and return types.
- The actual function **add** comes afterward and implements the logic based on the provided arguments.

When you call the **add** function, TypeScript determines the correct function signature based on the arguments provided and executes the corresponding logic.

Function overloading helps maintain type safety while allowing the flexibility of different argument types or return types for the same function name.

Call-Signature Compatibility:

When you have an overloaded function, the implementation signature (the actual function body) determines the types for parameters and the return type. The implementation signature must be compatible with all the overload signatures.

```
function format(data: string): string; // Ok
function format(data: string, needle: string, haystack: string): string; // Ok
function format(getData: () => string): string;
// The third overload signature is not compatible with the implementation signature.
function format(data: string, needle?: string, haystack?: string) {
   return needle && haystack ? data.replace(needle, haystack) : data;
}
```

- The first two overloaded signatures expect **data** to be a **string**, and they both return a **string**. These are compatible with the implementation signature.
- However, the third overload signature **function format(getData: () => string): string;** expects a function that returns a **string**. This signature isn't compatible with the implementation signature, which accepts **data** as a **string** and optional **needle** and **haystack** parameters.

Summary:

- In TypeScript, for an overloaded function, the actual function body determines the types for parameters and the return type.
- All the overload signatures should be compatible with the implementation signature. If any overload signature conflicts with the types expected in the implementation, TypeScript will raise an error.