Typescript

I. Foundational Concepts:

▼ TypeScript

- **JavaScript:** Dynamic and flexible, but can be prone to errors in large projects.
- **TypeScript:** Adds static typing to JavaScript, providing structure and catching errors early, making it ideal for building complex applications.
- **TypeScript** is a *superset* of JavaScript, meaning any valid JavaScript code is also valid TypeScript code.
- **TypeScript** code needs to be *compiled* into JavaScript before it can be run in a browser or Node.js.
- **TypeScript** is widely used in modern web development, especially with frameworks like Angular and React.

▼ Setting up your environment:

Setting up your TypeScript development environment is straightforward. Here's a step-by-step guide covering the essentials:

1. Install Node.js and npm (Node Package Manager):

- **Why?**
  - Node.js provides the runtime environment for executing JavaScript and TypeScript.
  - npm is used to install and manage packages (including the TypeScript compiler).
- **How?**

- Download the latest LTS (Long Term Support) version from the official Node.js website: nodejs.org.
- The installer includes npm.
- Verify the installation by opening a terminal or command prompt and running:
  - `node -v`
  - `npm -v`

2. Install TypeScript Globally (Optional, but Recommended):

- **Why?**
  - Global installation makes the `tsc` (TypeScript compiler) command available from any directory in your terminal.
- **How?**
  - Open a terminal or command prompt and run:
    - **`npm install -g typescript`**
  - Verify the installation by running:
    - `tsc -v`

3. Create a Project Directory:

- **Why?**
  - To organize your project files.
- **How?**
  - Create a new directory for your project.
  - Navigate to the directory in your terminal.

4. Initialize a `package.json` File:

- **Why?**
  - `package.json` stores project metadata and dependencies.
- **How?**

- In your terminal, run:
  - `npm init -y` (This creates a default `package.json` file)

5. Install TypeScript Locally (Recommended for Project-Specific Configuration):

- **Why?**
  - Local installation allows you to manage TypeScript versions on a per-project basis.
  - It's often preferred for consistency, especially in team environments.
- **How?**
  - In your terminal, run:
    - `npm install typescript --save-dev`
- Now typescript will be in your node_modules folder.

6. Create a `tsconfig.json` File:

- **Why?**
  - `tsconfig.json` configures the TypeScript compiler (tsc).
  - It specifies compiler options, such as target JavaScript version, module system, and strictness settings.
- **How?**
  - In your terminal, run:
    - `npx tsc --init` (if typescript is installed locally)
    - or
    - `tsc --init` (if typescript is installed globally)
  - This creates a default `tsconfig.json` file.
  - Customize the `tsconfig.json` file according to your project's requirements. Some key options to consider:
    - `target`: The target JavaScript version (e.g., "ES2020", "ESNext").
    - `module`: The module system (e.g., "CommonJS", "ESNext").

        - `outDir`: The output directory for compiled JavaScript files.
        - `rootDir`: The root directory of your TypeScript source files.
        - `strict`: Enables strict type checking (recommended).
        - `esModuleInterop`: Enables interoperability between CommonJS and ES Modules.
        - `moduleResolution`: Specifies how modules are resolved. Common options are "node", "nodenext", and "node16".

7. Create a TypeScript Source File:

- **Why?**
    - To write your TypeScript code.
- **How?**
    - Create a `.ts` file in your project directory (e.g., `index.ts`).
    - Add some TypeScript code to the file.

8. Compile TypeScript to JavaScript:

- **Why?**
    - Browsers and Node.js can only execute JavaScript.
- **How?**
    - In your terminal, run:
        - `npx tsc` (if typescript is installed locally)
        - or
        - `tsc` (if typescript is installed globally)
    - This will compile your TypeScript files to JavaScript, generating `.js` files in the `outDir` specified in your `tsconfig.json`.

9. Run the JavaScript File (if applicable):

- **Why?**
    - To execute your compiled JavaScript code.

- **How?**
  - If you're using Node.js, run:
    - `node <output-file>.js` (e.g., `node index.js`)
  - If you're using a browser, open the `.html` file that includes the `.js` file.

10. Integrate with an IDE (Recommended):

- **Why?**
  - IDEs provide features like code completion, error checking, and debugging.
- **Popular Options:**
  - Visual Studio Code (VS Code) (highly recommended)
  - WebStorm
  - Atom (with plugins)
- **Configuration:**
  - Most IDEs automatically detect `tsconfig.json` and provide TypeScript support.

Example `tsconfig.json` (Recommended Starting Point):

JSON

```
{
 "compilerOptions": {
   "target": "ES2020",
   "module": "ESNext",
   "moduleResolution": "NodeNext",
   "outDir": "./dist",
   "rootDir": "./src",
   "strict": true,
   "esModuleInterop": true,
   "skipLibCheck": true,
   "forceConsistentCasingInFileNames": true
 },
 "include": ["src/**/*"]
}
```

This setup provides a solid foundation for TypeScript development. Remember to adjust the `tsconfig.json` options to match your project's specific needs.

▼ **Basic Types:**

1. `number`:

- Represents numeric values, both integers and floating-point numbers.
- **Use:** For any numeric calculations or data.
- **Best Practice:** Use `number` for all numeric values unless you need a very specific integer type.
- **Example:**

TypeScript

```
let age: number = 30;
let price: number = 19.99;
let hex: number = 0xf00d; // Hexadecimal
let binary: number = 0b1010; // Binary
let octal: number = 0o744; // Octal
```

2. `string`:

- Represents textual data.
- **Use:** For any text-based data, such as names, messages, or addresses.
- **Best Practice:** Use `string` for all text values.
- **Example:**

TypeScript

```
let name: string = "Alice";
let message: string = 'Hello, world!';
let templateLiteral: string = `My name is ${name}`;
```

3. `boolean`:

- Represents logical values, either `true` or `false`.
- **Use:** For any conditions or flags.
- **Best Practice:** Use `boolean` for all true/false values.

- **Example:**

TypeScript

```
let isDone: boolean = true;
let hasError: boolean = false;
```

4. `null` **and** `undefined` **:**

- `null` represents an intentional absence of a value.
- `undefined` represents a variable that has been declared but not assigned a value.
- **Use:** To handle cases where a value might be missing.
- **Best Practice:**
  - Enable `strictNullChecks` in your `tsconfig.json` to avoid accidental use of `null` or `undefined`.
  - Use union types (e.g., `string | null`) to explicitly allow `null` or `undefined`.
- **Example:**

TypeScript

```
let value: string | null = null;
let anotherValue: number | undefined;

function processValue(input: string | null) {
  if (input === null) {
    console.log("Input is null");
  } else {
    console.log(input.toUpperCase());
  }
}
```

5. `void` **:**

- Represents the absence of a return value from a function.

- **Use:** For functions that don't return anything (e.g., functions that perform side effects).
- **Best Practice:** Use `void` for functions that don't return a value.
- **Example:**

TypeScript

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

6. `any` **:**

- Represents any type. It essentially disables type checking.
- **Use:** Avoid using `any` whenever possible. It's a last resort when you don't know the type or when migrating from JavaScript.
- **Best Practice:** Avoid `any` as much as possible. It defeats the purpose of TypeScript.
- **Example:**

TypeScript

```
let flexible: any = "Hello";
flexible = 10; // Allowed, but bad practice
```

7. `unknown` **:**

- Represents an unknown type. It's a type-safe alternative to `any`.
- **Use:** When you don't know the type of a value, but you want to perform type checking before using it.
- **Best Practice:** Use `unknown` instead of `any` when you need to handle unknown types.
- **Example:**

TypeScript

```
let unknownValue: unknown = "Hello";

if (typeof unknownValue === "string") {
  console.log(unknownValue.toUpperCase()); // Type guard: TypeScript kn
ows it's a string here
}
```

8. `never` **:**

- Represents a type that never occurs.
- **Use:** For functions that never return (e.g., functions that always throw an error or have an infinite loop).
- **Best Practice:** Use `never` for functions that never return.
- **Example:**

TypeScript

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

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

9. Type Literals:

- Allow you to specify exact values as types.
- **Use:** To restrict a variable to a specific set of values.
- **Best Practice:** Use type literals for constants or when you need to restrict values to a specific set.
- **Example:**

TypeScript

```
let direction: "north" | "south" | "east" | "west" = "north";
let statusCode: 200 | 404 | 500 = 200;
let myStringLiteral: "hello" = "hello"; //only "hello" is allowed.
```

10. Type Inference:

- TypeScript automatically determines the type of a variable based on its value.
- **Use:** To reduce the amount of explicit type annotations.
- **Best Practice:** Rely on type inference when possible, but use explicit type annotations when needed for clarity or to prevent type widening.
- **Example:**

TypeScript

```
let inferredString = "TypeScript"; // Inferred as string
let inferredNumber = 42; // Inferred as number

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

Key Best Practices Summary:

- **Enable** `strictNullChecks`**:** Improves type safety.
- **Avoid** `any`**:** Use `unknown` or specific types instead.
- **Use type guards:** Narrow down types with `typeof`, `instanceof`, or custom type guards.
- **Use type literals:** Restrict values to a specific set.
- **Rely on type inference:** But use explicit types when needed.
- **Use** `never`**:** For functions that never return.
- **Use** `void`**:** For functions that don't return a value.
- **Use union types:** To handle `null` or `undefined` explicitly.

▼ Variables and Constants:

1. `let` (Block-Scoped Variables):

- `let` allows you to declare variables that are *block-scoped*. This means they are only accessible within the block (e.g., inside an `if` statement, `for` loop, or function) where they are defined.
- `let` variables can be reassigned.

Example:

TypeScript

```
function exampleLet() {
  if (true) {
    let x = 10;
    console.log(x); // Output: 10 (x is accessible here)
  }
  // console.log(x); // Error: x is not defined (x is out of scope)

  let y = 20;
  y = 30; // Reassignment is allowed
  console.log(y); // Output: 30
}

exampleLet();
```

Uses:

- Use `let` when you need to reassign a variable's value.
- Use `let` when you need variables that are scoped to a specific block.

2. `const` (Block-Scoped Constants):

- `const` allows you to declare constants (variables whose values cannot be reassigned).
- `const` variables are also block-scoped.
- `const` variables must be initialized when they are declared.

Example:

TypeScript

```
function exampleConst() {
  const PI = 3.14159;
  console.log(PI); // Output: 3.14159
  // PI = 3.14; // Error: Cannot reassign constant variable PI

  if (true) {
    const message = "Hello, const!";
    console.log(message); // Output: Hello, const!
  }
  // console.log(message); // Error: message is not defined (out of scope)

  const person = { name: "Bob", age: 30 };
  person.age = 31; // Allowed: You can modify properties of an object decl
ared with const
  // person = { name: "Alice", age: 25 }; // Error: Cannot reassign the perso
n variable itself
}

exampleConst();
```

Uses:

- Use `const` for values that should not change throughout the program.
- Use `const` for objects and arrays when you want to prevent reassignment of the variable itself (but you can still modify their properties).

3. Scoping Rules:

- **Block Scope:** Variables declared with `let` and `const` are block-scoped, meaning they are only accessible within the block where they are defined. 1. github.com

  1

  github.com

- **Function Scope:** Variables declared with `var` are function-scoped (less common in modern TypeScript, as `let`/`const` are preferred).
- **Global Scope:** Variables declared outside of any function or block are in the global scope.

Example (Scoping):

TypeScript

```
let globalVar = "Global";

function scopeExample() {
  let functionVar = "Function";

  if (true) {
    let blockVar = "Block";
    console.log(globalVar, functionVar, blockVar); // Accessible
  }

  console.log(globalVar, functionVar); // Accessible, blockVar is not
  // console.log(blockVar); // Error: blockVar is not defined
}

scopeExample();
console.log(globalVar); // Accessible
// console.log(functionVar); // Error : functionVar is not defined
```

Best Practices:

- **Prefer `const` by Default:** Use `const` whenever possible to make your code more predictable and prevent accidental reassignment.
- **Use `let` for Reassignable Variables:** Use `let` only when you need to reassign a variable's value.
- **Avoid `var`:** `var` has function-scoping and can lead to unexpected behavior. Use `let` or `const` instead.

- **Declare Variables in the Smallest Possible Scope:** Declare variables as close as possible to where they are used to improve code readability and prevent naming conflicts.
- **Initialize `const` Variables:** Always initialize `const` variables when you declare them.
- **Understand Object/Array Mutability:** Remember that `const` prevents reassignment of the variable itself, not the modification of its properties (for objects and arrays). If you need to deeply freeze an object, there are other methods, but normal const declaration does not accomplish this.
- **Use Descriptive Names:** Give your variables and constants meaningful names to make your code easier to understand.
- **Use strict mode:** "use strict"; at the top of your files will prevent accidental global variables when not using let, const, or var.

▼ **Operators:** Familiarity with JavaScript operators and how they interact with TypeScript types.

1. Arithmetic Operators:

- `+` (addition), (subtraction), (multiplication), `/` (division), `%` (modulo), `*` (exponentiation)
- **TypeScript:** These operators work as expected with `number` types. TypeScript will give you an error if you try to use them with types that are not numbers (unless you use `any`).

TypeScript

```
let a: number = 10;
let b: number = 5;

let sum: number = a + b;
let difference: number = a - b;
let product: number = a * b;
let quotient: number = a / b;
let remainder: number = a % b;
let power: number = a ** b;
```

```
// let error = "string" + 5; // Error: Operator '+' cannot be applied to types 'string' and 'number'.
```

2. Assignment Operators:

- `=`, `+=`, `=`, `=`, `/=`, `%=`, `*=`
- **TypeScript:** The type of the value being assigned must be compatible with the type of the variable.

TypeScript

```
let x: number = 5;
x += 3; // x is now 8
x *= 2; // x is now 16

let str: string = "Hello";
str += " World"; // str is now "Hello World"

// x = "string"; // Error: Type 'string' is not assignable to type 'number'.
```

3. Comparison Operators:

- `==` (loose equality), `===` (strict equality), `!=` (loose inequality), `!==` (strict inequality), `>` (greater than), `<` (less than), `>=` (greater than or equal to), `<=` (less than or equal to) 1. github.com

  1

  github.com

- **TypeScript:** These operators return `boolean` values. TypeScript's strict equality (`===`) and inequality (`!==`) are generally preferred.

TypeScript

```
let num1: number = 10;
let num2: number = "10";
```

```
console.log(num1 == num2);  // true (loose equality)
console.log(num1 === Number(num2)); //true (strict equality)
console.log(num1 === num2); // false (strict equality)
console.log(num1 != num2);  // false (loose inequality)
console.log(num1 !== num2); // true (strict inequality)

console.log(num1 > 5);   // true
console.log(num1 < 20);  // true
```

4. Logical Operators:

- `&&` (logical AND), `||` (logical OR), `!` (logical NOT)
- **TypeScript:** These operators work with `boolean` types.

TypeScript

```
let isTrue: boolean = true;
let isFalse: boolean = false;

console.log(isTrue && !isFalse); // true
console.log(isFalse || isTrue); // true
console.log(!isTrue);          // false
```

5. Bitwise Operators:

- `&` (bitwise AND), `|` (bitwise OR), `^` (bitwise XOR), `~` (bitwise NOT), `<<` (left shift), `>>` (right shift), `>>>` (unsigned right shift)
- **TypeScript:** These operators work with `number` types.

TypeScript

```
let bit1: number = 5; // 0101
let bit2: number = 3; // 0011

console.log(bit1 & bit2); // 0001 (1)
```

```
console.log(bit1 | bit2); // 0111 (7)
console.log(bit1 << 1);  // 1010 (10)
```

6. String Operators:

- `+` (concatenation)
- **TypeScript:** Concatenates `string` types.

TypeScript

```
let firstName: string = "John";
let lastName: string = "Doe";
let fullName: string = firstName + " " + lastName; // "John Doe"
```

7. Type Operators:

- `typeof`, `instanceof` (used in type guards)
- **TypeScript:** Used for type checking and type narrowing.

TypeScript

```
let value: string | number = "Hello";

if (typeof value === "string") {
  console.log(value.toUpperCase()); // TypeScript knows it's a string here
}

class MyClass {}
let myInstance = new MyClass();

if (myInstance instanceof MyClass) {
  console.log("myInstance is an instance of MyClass");
}
```

Best Practices:

- **Use Strict Equality (`===`, `!==`):** Avoid loose equality (`==`, `!=`) whenever possible to prevent unexpected type coercion.

- **Be Mindful of Type Coercion:** Understand how JavaScript's type coercion works, especially when using loose equality or arithmetic operators with different types.
- **Use Type Guards:** When working with union types or `unknown`, use `typeof` or `instanceof` to narrow down the types before performing operations.
- **Leverage TypeScript's Type Checking:** Let TypeScript catch type errors for you. Avoid using `any` if possible.
- **Understand Operator Precedence:** Be aware of operator precedence to avoid unexpected results. Use parentheses to clarify complex expressions.
- **Use Numeric separators:** When working with large numbers, using underscores as separators makes the numbers easier to read. `let largeNumber = 1_000_000;`
- **Use optional chaining and nullish coalescing:** `?.` and `??` are extremely helpful for safely accessing properties of objects that could be null or undefined.

▼ **Control Flow:** `if/else`, `switch`, `for`, `while`, `do...while` loops.

1. `if/else` **Statements:**

- **Use:** To execute different blocks of code based on a condition.
- **Best Practices:**
  - Use strict equality (`===`, `!==`) for comparisons.
  - Keep conditions simple and readable.
  - Use curly braces `{}` to clearly define blocks, even for single-line statements.
  - Use `else if` for multiple conditions.

TypeScript

```
function checkAge(age: number): void {
  if (age >= 18) {
    console.log("You are an adult.");
  } else if (age >= 13) {
```

```
    console.log("You are a teenager.");
  } else {
    console.log("You are a child.");
  }
}

checkAge(20);
checkAge(15);
checkAge(10);
```

2. `switch` Statements:

- **Use:** To execute different blocks of code based on the value of a variable.
- **Best Practices:**
  - Use `break` statements to prevent fall-through.
  - Use a `default` case to handle unexpected values.
  - Consider using enums or type literals for switch values.
  - If you have a lot of conditions, and they are all checking against the same variable, a switch statement may be more readable than a long if/else if chain.

TypeScript

```
enum Color {
  Red,
  Green,
  Blue,
}

function getColorName(color: Color): string {
  switch (color) {
    case Color.Red:
      return "Red";
    case Color.Green:
      return "Green";
```

```
    case Color.Blue:
      return "Blue";
    default:
      return "Unknown color";
  }
}

console.log(getColorName(Color.Green));
console.log(getColorName(3)); // This example will print "Unknown color"
```

3. `for` Loops:

- **Use:** To iterate over a sequence of values.
- **Best Practices:**
  - Use `let` for the loop variable to ensure block scoping.
  - Avoid modifying the loop variable inside the loop body if it will cause unintended side effects.
  - Use `for...of` for iterating over iterable objects (arrays, strings, maps, sets).
  - Use `for...in` for iterating over the keys of an object.

TypeScript

```
function printNumbers(count: number): void {
  for (let i = 0; i < count; i++) {
    console.log(i);
  }
}

printNumbers(5);

let fruits: string[] = ["apple", "banana", "orange"];

for (let fruit of fruits) {
  console.log(fruit);
```

```
}

let person = {name: "John", age: 30};

for (let key in person){
    console.log(key, person[key]);
}
```

4. `while` Loops:

- **Use:** To repeatedly execute a block of code as long as a condition is true.
- **Best Practices:**
  - Ensure that the condition eventually becomes false to prevent infinite loops.
  - Use `while` when you don't know the exact number of iterations.

TypeScript

```
function countdown(start: number): void {
  while (start > 0) {
    console.log(start);
    start--;
  }
}

countdown(3);
```

5. `do...while` Loops:

- **Use:** To repeatedly execute a block of code at least once, and then as long as a condition is true.
- **Best Practices:**
  - Similar to `while` loops, ensure the condition eventually becomes false.
  - Use `do...while` when you need to execute the loop body at least once.

TypeScript

```
function rollDice(): number {
  let roll: number;
  do {
    roll = Math.floor(Math.random() * 6) + 1;
    console.log("Rolled:", roll);
  } while (roll !== 6);
  return roll;
}

rollDice();
```

General Best Practices for Control Flow:

- **Keep Code Readable:** Use meaningful variable names and clear conditions.
- **Avoid Nested Complexity:** Limit the nesting of `if/else` and loop statements to improve readability.
- **Use Early Returns:** Return from functions early to simplify logic and avoid deeply nested `if/else` blocks.
- **Use Type Guards:** When working with union types, use type guards (`typeof`, `instanceof`) to narrow down types before executing code.
- **Use Enums or Type Literals:** For switch statements or when you need to represent a fixed set of values.
- **Use `for...of` when possible:** It is often cleaner than a standard for loop.
- **Use consistent style:** Pick a style, and stick with it.

▼ **Functions:** Defining functions, parameters, return types, arrow functions, function overloading, and rest parameters.

1. Defining Functions (Regular Functions):

- **Use:** To create reusable blocks of code that perform specific tasks.
- **Best Practice:**

- Always specify parameter types and return types for clarity and type safety.
- Use descriptive function names.
- Keep functions focused on a single responsibility.

TypeScript

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

let result: number = add(5, 10);
console.log(result); // Output: 15
```

2. Parameters:

- **Use:** To pass values into functions.
- **Best Practice:**
  - Specify parameter types.
  - Use optional parameters (`?`) when a parameter is not always required.
  - Use default parameters (`= value`) to provide default values.

TypeScript

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

console.log(greet("Alice")); // Output: Hello, Alice!
console.log(greet("Bob", "Good morning")); // Output: Good morning, Bob!
```

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

console.log(calculateArea(5)); // Output: 50
console.log(calculateArea(5, 20)); // Output: 100
```

3. Return Types:

- **Use:** To specify the type of value a function returns.
- **Best Practice:**
  - Always specify return types for clarity and type safety.
  - Use `void` for functions that don't return a value.
  - Let type inference work when possible, but be explicit when it increases readability.

TypeScript

```
function formatName(firstName: string, lastName: string): string {
  return `${lastName}, ${firstName}`;
}

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

4. Arrow Functions:

- **Use:** A concise syntax for defining functions, especially useful for callbacks and short functions.
- **Best Practice:**
  - Use arrow functions for shorter, inline functions.
  - Be aware of the `this` context (arrow functions capture the `this` value of the enclosing scope).

- Use parentheses for parameters when necessary (e.g., multiple parameters or no parameters).

TypeScript

```
const multiply = (a: number, b: number): number ⇒ a * b;
const sayHello = (name: string) ⇒ `Hello, ${name}!`;
const logNumber = (num:number) ⇒ console.log(num);
const noParams = () ⇒ console.log("no params");

console.log(multiply(3, 4)); // Output: 12
console.log(sayHello("Charlie")); // Output: Hello, Charlie!
logNumber(123);
noParams();
```

5. Function Overloading:

- **Use:** To define multiple function signatures with different parameter types or return types.
- **Best Practice:**
  - Use function overloading when you need to handle different parameter combinations.
  - Provide the most specific signatures first.
  - The last signature is the implementation signature, and it must be compatible with all the overload signatures.

TypeScript

```
function addValues(a: number, b: number): number;
function addValues(a: string, b: string): string;
function addValues(a: any, b: any): any {
  return a + b;
}

console.log(addValues(5, 10)); // Output: 15
```

```
console.log(addValues("Hello", " World")); // Output: Hello World
// console.log(addValues(5, "world")); //Error.
```

6. Rest Parameters:

- **Use:** To accept an indefinite number of arguments as an array.
- **Best Practice:**
  - Use rest parameters when you don't know the exact number of arguments.
  - Rest parameters must be the last parameter in the function signature.
  - Type the rest parameter as an array of the expected type.

TypeScript

```
function sumAll(...numbers: number[]): number {
  let total = 0;
  for (let num of numbers) {
    total += num;
  }
  return total;
}

console.log(sumAll(1, 2, 3, 4, 5)); // Output: 15
console.log(sumAll(10, 20)); // Output: 30
```

Key Best Practices Summary:

- **Type Annotations:** Always use type annotations for parameters and return types.
- **Function Naming:** Use descriptive and meaningful function names.
- **Single Responsibility:** Keep functions focused on a single task.
- **Optional Parameters:** Use `?` for optional parameters.
- **Default Parameters:** Use `= value` for default parameter values.

- **Arrow Functions:** Use arrow functions for concise syntax, especially for callbacks.
- **Function Overloading:** Use overloading when you need to handle different parameter combinations.
- **Rest Parameters:** Use rest parameters for an indefinite number of arguments.
- **Readability:** Prioritize readability over brevity.
- **Use strict mode:** "use strict"; at the top of your files will prevent accidental global variables.

▼ **Objects and Interfaces:** Defining interfaces to describe the shape of objects, using interfaces to enforce type contracts. Understanding the difference between interfaces and types.

1. Defining Interfaces:

- **Use:** Interfaces define the structure (shape) of an object. They specify the names and types of properties and methods that an object must have.
- **Best Practice:**
  - Use interfaces to define clear and reusable type contracts.
  - Use descriptive interface names.
  - Use optional properties (`?`) when a property is not always required.
  - Use readonly properties to prevent modification.

TypeScript

```
interface Person {
  name: string;
  age: number;
  email?: string; // Optional property
  readonly id: number; // Readonly property
  greet(): string; // Method signature
}
```

```
let person1: Person = {
  name: "Alice",
  age: 30,
  id: 123,
  greet: function () {
    return `Hello, my name is ${this.name}`;
  },
};

console.log(person1.greet());
// person1.id = 456; // Error: Cannot assign to 'id' because it is a read-only property.
```

2. Using Interfaces to Enforce Type Contracts:

- **Use:** Interfaces ensure that objects conform to a specific structure, preventing type errors.
- **Best Practice:**
    - Use interfaces to enforce type constraints in function parameters and return types.
    - Use interfaces to define the shape of objects used in your application.

TypeScript

```
function printPerson(person: Person): void {
  console.log(`Name: ${person.name}, Age: ${person.age}`);
  if (person.email) {
    console.log(`Email: ${person.email}`);
  }
}

printPerson(person1);

let person2: Person = {
  name: "Bob",
  age: 25,
```

```
  id: 345,
  greet: () => "hi",
};

printPerson(person2);

// let invalidPerson: Person = { name: "Eve" }; // Error: Missing property 'a
ge'
```

3. Understanding the Difference Between Interfaces and Types:

- **Interfaces:**
  - Primarily used to describe the shape of objects.
  - Can be extended using `extends`.
  - Can be merged (declaration merging).
  - Cannot be used to describe primitive types, union types, or tuple types directly.
  - Generally preferred when describing the shape of objects.
- **Types:**
  - More versatile and can describe any type, including primitive types, union types, tuple types, and more.
  - Can be used to create type aliases for complex types.
  - Cannot be merged.
  - Can use union types and intersection types.
  - Can define tuple types.
  - Generally preferred when describing types other than object shapes or when using union or intersection types.

Examples:

TypeScript

```
// Interface extension
interface Employee extends Person {
  employeeId: string;
}

let employee1: Employee = {
  name: "Charlie",
  age: 35,
  id: 567,
  employeeId: "E123",
  greet: () ⇒ "hello",
};

// Interface merging
interface MergeTest {
    prop1: string;
}

interface MergeTest {
    prop2: number;
}

let merged: MergeTest = {
    prop1: "test",
    prop2: 123,
}

// Type alias with union type
type StringOrNumber = string | number;

let value: StringOrNumber = "Hello";
value = 42;

//Type with Tuple
type Point = [number, number];
```

```
let myPoint: Point = [10, 20];

//Type with Intersection
type Name = {
   firstName: string;
}

type Age = {
   age: number;
}

type PersonNameAge = Name & Age;

let personNameAge: PersonNameAge = {
   firstName: "David",
   age: 40
}
```

When to Use Interfaces vs. Types:

- **Use interfaces:** When you need to describe the shape of an object, especially when you might need to extend or merge the interface.
- **Use types:** When you need to describe types other than object shapes, such as primitive types, union types, intersection types, or tuple types, or when you need to create type aliases for complex types.

In many cases, the choice between interfaces and types is a matter of personal preference or team convention. However, understanding their differences will help you make informed decisions about which to use in your projects.

▼ **Arrays:** Working with arrays, array methods, and typed arrays.

1. Working with Arrays:

- **Declaration:**
  - You can declare arrays using the `[]` syntax or the `Array<T>` generic type.

- You can specify the type of elements in the array.

TypeScript

```
// Array of numbers
let numbers: number[] = [1, 2, 3, 4, 5];
let anotherNumbers: Array<number> = [6, 7, 8, 9, 10];

// Array of strings
let fruits: string[] = ["apple", "banana", "orange"];

// Array of mixed types (avoid if possible, use union types instead)
let mixed: (string | number)[] = [1, "two", 3, "four"];

// Array of objects with interfaces
interface Person {
  name: string;
  age: number;
}
let people: Person[] = [{ name: "Alice", age: 30 }, { name: "Bob", age: 25 }];
```

- **Accessing Elements:**
  - Use the index to access elements (zero-based).

TypeScript

```
console.log(numbers[0]); // Output: 1
console.log(fruits[2]); // Output: orange
```

- **Adding and Removing Elements:**
  - `push()`: Adds an element to the end.
  - `pop()`: Removes the last element.
  - `unshift()`: Adds an element to the beginning.
  - `shift()`: Removes the first element.
  - `splice()`: Adds or removes elements at a specific index.

TypeScript

```
numbers.push(6);
console.log(numbers); // Output: [1, 2, 3, 4, 5, 6]

fruits.pop();
console.log(fruits); // Output: ["apple", "banana"]

numbers.unshift(0);
console.log(numbers); // output: [0, 1, 2, 3, 4, 5, 6]
```

```
fruits.shift();
console.log(fruits); // output: ["banana"]

numbers.splice(2, 1, 10); // remove 1 element at index 2, and add 10
console.log(numbers); // output: [0, 1, 10, 3, 4, 5, 6]
```

2. Array Methods:

- TypeScript arrays have many built-in methods, similar to JavaScript.
- TypeScript's type system enhances the safety of these methods.

TypeScript

```
let numbers2: number[] = [1, 2, 3, 4, 5];

// map(): Creates a new array with the results of a function applied to each element.
let doubled: number[] = numbers2.map((num) => num * 2);
console.log(doubled); // Output: [2, 4, 6, 8, 10]

// filter(): Creates a new array with elements that pass a test.
let evenNumbers: number[] = numbers2.filter((num) => num % 2 === 0);
console.log(evenNumbers); // Output: [2, 4]

// reduce(): Reduces the array to a single value.
let sum: number = numbers2.reduce((acc, num) => acc + num, 0);
console.log(sum); // Output: 15

// forEach(): Executes a function for each element.
numbers2.forEach((num) => console.log(num));

// find(): Returns the first element that satisfies a condition.
let found: number | undefined = numbers2.find((num) => num > 3);
console.log(found); // Output: 4

// sort(): sorts an array.
let unsorted: number[] = [5,2,4,1,3];
unsorted.sort((a,b)=> a-b);
console.log(unsorted); //output: [1, 2, 3, 4, 5]

//includes(): checks if an array includes a certain element.
let included: boolean = numbers2.includes(3);
console.log(included); //output: true
```

3. Typed Arrays:

- Typed arrays are a feature of ECMAScript 2015 (ES6) and are supported by TypeScript.
- They are used for working with binary data.

- They provide efficient storage and manipulation of numerical data.
- Types such as `Uint8Array`, `Int16Array`, `Float32Array`, etc.

TypeScript

```
// Uint8Array (array of unsigned 8-bit integers)
let buffer: ArrayBuffer = new ArrayBuffer(8); // 8 bytes
let uint8Array: Uint8Array = new Uint8Array(buffer);

uint8Array[0] = 10;
uint8Array[1] = 20;

console.log(uint8Array); // Output: Uint8Array [ 10, 20, 0, 0, 0, 0, 0, 0 ]

// Float32Array (array of 32-bit floating-point numbers)
let floatBuffer: ArrayBuffer = new ArrayBuffer(12);
let float32Array: Float32Array = new Float32Array(floatBuffer);

float32Array[0] = 3.14;
float32Array[1] = 2.71;

console.log(float32Array); // Output: Float32Array [ 3.140000104904175, 2.7099998474121094, 0 ]
```

Best Practices:

- **Type Annotations:** Always use type annotations for arrays to improve type safety.
- **Immutability (When Possible):** Use methods like `map`, `filter`, and `reduce` to create new arrays rather than modifying existing ones. This promotes immutability and reduces side effects.
- **Use `for...of`:** Use `for...of` loops for iterating over arrays when you don't need the index.
- **Avoid `any[]`:** Avoid using `any[]` if possible. Use specific types or union types instead.
- **Understand Array Methods:** Familiarize yourself with the various array methods to efficiently manipulate arrays.
- **Typed Arrays for Binary Data:** Use typed arrays when working with binary data for performance and efficiency.
- **Use const when appropriate:** If you do not plan to reassign the array variable itself, use const.

- **Use optional chaining:** when working with arrays that might be undefined, optional chaining helps to prevent errors. `myArray?.map(...)`

▼ **Enums:** Using enums to represent a set of named constants.

Enums (enumerations) in TypeScript are a way to define a set of named constants. They make your code more readable and maintainable by providing meaningful names for numeric or string values.

Use:

- **Representing a fixed set of values:** When you have a set of related constants, like days of the week, status codes, or directions.
- **Improving code readability:** By using meaningful names instead of raw numbers or strings.
- **Enhancing type safety:** By ensuring that variables can only hold valid enum values.

Syntax:

TypeScript

```
enum EnumName {
 Member1,
 Member2,
 Member3,
}
```

Types of Enums:

1. **Numeric Enums (Default):**TypeScript
    - If you don't explicitly assign values, enum members are automatically assigned numeric values starting from 0.
    - You can also explicitly assign numeric values.

```
enum Direction {
 Up,    // 0
 Down,  // 1
 Left,  // 2
 Right, // 3
}
```

```
let myDirection: Direction = Direction.Up;
console.log(myDirection); // Output: 0
console.log(Direction[0]); // Output: "Up"

enum StatusCode {
  OK = 200,
  NotFound = 404,
  InternalServerError = 500,
}

let status: StatusCode = StatusCode.NotFound;
console.log(status); // Output: 404
```

2. **String Enums:**TypeScript
    - Each enum member must be explicitly assigned a string value.

```
enum LogLevel {
  Info = "INFO",
  Warning = "WARNING",
  Error = "ERROR",
}

let logLevel: LogLevel = LogLevel.Warning;
console.log(logLevel); // Output: "WARNING"
```

3. **Heterogeneous Enums (Avoid if possible):**TypeScript
    - Enums can contain a mix of numeric and string values, but this is generally discouraged due to potential confusion.

```
enum Mixed {
  A,      // 0
  B = "B",
  C = 10,
  D = "D",
}
```

Uses and Best Practices:

- **Use meaningful names:** Choose names that clearly describe the purpose of the enum members.
- **Use numeric enums for related numeric constants:** When you have a set of numeric constants that represent related values.
- **Use string enums for descriptive string values:** When you need to use string values that are more descriptive than numeric values.

- **Avoid heterogeneous enums:** They can lead to confusion and make your code harder to understand.
- **Use `const enum` for performance:** If you're using numeric enums and don't need the runtime representation of the enum, use `const enum`. This will inline the enum values during compilation, improving performance.TypeScript

```
const enum Color {
 Red,
 Green,
 Blue,
}

let myColor: Color = Color.Red;
// During compilation, Color.Red will be replaced with 0
```

- **Use enums with switch statements:** Enums are often used with `switch` statements to handle different cases based on the enum value.TypeScript

```
enum DayOfWeek {
 Monday,
 Tuesday,
 Wednesday,
 Thursday,
 Friday,
 Saturday,
 Sunday,
}

function isWeekend(day: DayOfWeek): boolean {
 switch (day) {
   case DayOfWeek.Saturday:
   case DayOfWeek.Sunday:
     return true;
   default:
     return false;
 }
}
```

- **Use enums to represent state:** Enums are good for representing the various states of an object.

Enums provide a clear and organized way to manage sets of related constants, making your TypeScript code more readable, maintainable, and type-safe.

▼ **Type Aliases:** Creating type aliases for complex types.

Type aliases in TypeScript are a powerful way to give a name to a type. They don't create a new type; they create a new name that refers to an existing type. This is particularly useful for complex types, making your code more readable and maintainable.

Use:

- **Simplifying Complex Types:** To give a shorter, more meaningful name to a complex type, like a union, intersection, or object type.
- **Improving Readability:** To make your code easier to understand by using descriptive names for types.
- **Enhancing Code Reusability:** To reuse complex types throughout your codebase.

Syntax:

TypeScript

```
type AliasName = Type;
```

Examples:

1. **Union Type Alias:**TypeScript

   ```
   type StringOrNumber = string | number;

   function processInput(input: StringOrNumber): void {
    if (typeof input === "string") {
      console.log(input.toUpperCase());
    } else {
      console.log(input * 2);
    }
   }

   processInput("hello");
   processInput(10);
   ```

   In this example, `StringOrNumber` makes it easier to work with the union type `string | number`.

2. **Object Type Alias:**TypeScript

   ```
   type Point = {
    x: number;
    y: number;
   };

   function calculateDistance(p1: Point, p2: Point): number {
   ```

```
  const deltaX = p2.x - p1.x;
  const deltaY = p2.y - p1.y;
  return Math.sqrt(deltaX ** 2 + deltaY ** 2);
}

const point1: Point = { x: 0, y: 0 };
const point2: Point = { x: 3, y: 4 };

console.log(calculateDistance(point1, point2)); // Output: 5
```

Here, `Point` simplifies the object type `{ x: number; y: number; }`.

3. **Function Type Alias:**TypeScript

```
type StringTransformer = (input: string) => string;

const toUpperCase: StringTransformer = (input) => input.toUpperCase();
const toLowerCase: StringTransformer = (input) => input.toLowerCase();

function applyTransformation(
  input: string,
  transformer: StringTransformer
): string {
  return transformer(input);
}

console.log(applyTransformation("Hello", toUpperCase)); // Output: HELLO
console.log(applyTransformation("WORLD", toLowerCase)); // Output: world
```

`StringTransformer` makes it easier to define and use functions that take a string and return a string.

4. **Tuple Type Alias:**TypeScript

```
type RGB = [number, number, number];

function formatRGB(color: RGB): string {
  return `rgb(${color[0]}, ${color[1]}, ${color[2]})`;
}

const red: RGB = [255, 0, 0];
console.log(formatRGB(red)); // Output: rgb(255, 0, 0)
```

Here, `RGB` simplifies the tuple type `[number, number, number]`.

5. **Intersection Type Alias:**TypeScript

```
type Colorful = {
  color: string;
};

type Circle = {
  radius: number;
```

```
};

type ColorfulCircle = Colorful & Circle;

const myCircle: ColorfulCircle = {
  color: "red",
  radius: 10,
};

console.log(`My circle is ${myCircle.color} and has a radius of ${myCircle.radius}`);
```

This example utilizes an intersection type alias to combine the properties of two types.

Best Practices:

- **Use Descriptive Names:** Choose names that clearly indicate the purpose of the alias.
- **Simplify Complex Types:** Use type aliases to break down complex types into smaller, more manageable parts.
- **Promote Reusability:** Create type aliases for types that are used multiple times throughout your codebase.
- **Improve Readability:** Use type aliases to make your code easier to read and understand.
- **Avoid Overuse:** Don't create type aliases for simple types that are already clear.
- **Consistency:** Keep the use of type aliases consistent throughout a project.
- **Document when needed:** If a type alias is not obvious, add a comment explaining its purpose.

II. Intermediate Concepts:

▼ **Classes:** Defining classes, constructors, properties, methods, inheritance, and access modifiers (public, private, protected).

1. Defining Classes:

- **Purpose:** Classes are blueprints for creating objects. They encapsulate data (properties) and behavior (methods) into a single unit.

- **Syntax:**TypeScript

```
class ClassName {
  // Properties
  // Constructor
  // Methods
}
```

- **Example:**TypeScript

```
class Car {
  make: string;
  model: string;
  year: number;

  constructor(make: string, model: string, year: number) {
    this.make = make;
    this.model = model;
    this.year = year;
  }

  getDescription(): string {
    return `${this.year} ${this.make} ${this.model}`;
  }
}

const myCar = new Car("Toyota", "Camry", 2023);
console.log(myCar.getDescription()); // Output: 2023 Toyota Camry
```

2. Constructors:

- **Purpose:** Constructors are special methods that initialize an object's properties when it's created.
- **Syntax:**TypeScript

```
constructor(parameters) {
  // Initialization logic
}
```

- **Parameter Properties:** A shorthand for defining and initializing properties in the constructor.TypeScript

```
class Point {
  constructor(public x: number, public y: number) {} // Parameter prope
rties

  getDistanceToOrigin(): number {
    return Math.sqrt(this.x ** 2 + this.y ** 2);
  }
}

const point = new Point(3, 4);
console.log(point.x); // Output: 3
console.log(point.getDistanceToOrigin()); // Output: 5
```

3. Properties:

- **Purpose:** Properties store data associated with an object.
- **Types:** Properties must have explicit type annotations or be implicitly typed by their initialization.
- **Readonly:** The `readonly` keyword prevents a property from being modified after initialization.TypeScript

```
class Circle {
  readonly radius: number;
  color: string;

  constructor(radius: number, color: string) {
    this.radius = radius;
    this.color = color;
```

```
  }

  // this.radius = 10; // Error: Cannot assign to 'radius' because it is a re
ad-only property.
}
```

4. Methods:

- **Purpose:** Methods define the behavior of an object.
- **Types:** Method parameters and return types should be explicitly typed.
- `this` **Context:** Inside a method, `this` refers to the current object instance.TypeScript

```
class Counter {
  private count: number = 0;

  increment(): void {
    this.count++;
  }

  getCount(): number {
    return this.count;
  }
}

const counter = new Counter();
counter.increment();
console.log(counter.getCount()); // Output: 1
```

5. Inheritance:

- **Purpose:** Inheritance allows you to create new classes (derived classes) that inherit properties and methods from existing classes (base classes).
- `extends` **Keyword:** Used to specify the base class.
- `super` **Keyword:** Used to call the base class constructor or methods.

- **Method Overriding:** Derived classes can provide their own implementations of base class methods.TypeScript

```
class Animal {
  constructor(public name: string) {}

  makeSound(): void {
    console.log("Generic animal sound");
  }
}

class Dog extends Animal {
  constructor(name: string, public breed: string) {
    super(name);
  }

  override makeSound(): void {
    console.log("Woof!");
  }

  getBreed():string{
      return this.breed;
  }
}

const dog = new Dog("Buddy", "Golden Retriever");
console.log(dog.name); // Output: Buddy
dog.makeSound(); // Output: Woof!
console.log(dog.getBreed()); // Output: Golden Retriever
```

6. Access Modifiers:

- `public`:
  - Accessible from anywhere (default).
- `private`:

- Accessible only within the class.

- `protected`:TypeScript
  - Accessible within the class and its derived classes.

```
class BankAccount {
  public accountNumber: string;
  private balance: number = 0;
  protected ownerName: string;

  constructor(accountNumber: string, ownerName: string) {
    this.accountNumber = accountNumber;
    this.ownerName = ownerName;
  }

  deposit(amount: number): void {
    this.balance += amount;
  }

  private getBalance(): number {
    return this.balance;
  }

  protected getOwnerName(): string{
      return this.ownerName;
  }

  public displayBalance(): void{
      console.log(`Current Balance: ${this.getBalance()}`);
  }
}

class SavingsAccount extends BankAccount {
    constructor(accountNumber: string, ownerName: string){
        super(accountNumber, ownerName);
    }
```

```
    public displayOwner(): void{
        console.log(`Account Owner: ${this.getOwnerName()}`);
    }
}

const account = new SavingsAccount("12345", "Alice");
account.deposit(100);
account.displayBalance(); // Output: Current Balance: 100
account.displayOwner(); //Output: Account Owner: Alice
// account.balance; // Error: Property 'balance' is private
// account.getOwnerName(); // Error: Property 'getOwnerName' is prot
ected
```

Key Concepts and Best Practices:

- **Encapsulation:** Use access modifiers to hide implementation details and control access to properties and methods.
- **Inheritance:** Use inheritance to create reusable code and model "is-a" relationships.
- **Composition:** Favor composition over inheritance when possible for greater flexibility.
- **Interfaces:** Use interfaces to define contracts for classes.
- **Abstract Classes:** Use abstract classes to define base classes that cannot be instantiated.
- **Parameter Properties:** Use parameter properties to simplify constructor definitions.
- `readonly`: Use `readonly` for properties that should not be modified after initialization.
- **Method Overriding:** Use the `override` keyword when overriding methods in derived classes.
- `this` **Context:** Understand how `this` works in methods and arrow functions.

- **Class Design:** Follow the Single Responsibility Principle and other SOLID principles for better class design.

▼ **Generics:** Writing reusable components that can work with a variety of types. Understanding type parameters and constraints.

Generics are a powerful feature in TypeScript that allow you to write reusable components and functions that can work with a variety of types without sacrificing type safety. They enable you to create code that is both flexible and type-safe.

Purpose of Generics:

- **Code Reusability:** Create components and functions that can work with different data types without duplicating code.
- **Type Safety:** Maintain type safety while working with various types.
- **Flexibility:** Adapt to different data structures and types without rewriting code.

1. Type Parameters:

- Type parameters are placeholders for types that will be specified later.
- They are declared using angle brackets (`<T>`).
- `T` is a common convention, but you can use any valid identifier.

Example: Generic Function:

TypeScript

```
function identity<T>(arg: T): T {
  return arg;
}

let result1: string = identity<string>("Hello");
let result2: number = identity<number>(42);
let result3: boolean = identity<boolean>(true);

console.log(result1); // Output: Hello
console.log(result2); // Output: 42
```

```
console.log(result3); // Output: true

// Type inference: you can often omit the explicit type argument
let result4 = identity("TypeScript"); // result4 is inferred as string
```

2. Generic Interfaces:

- Interfaces can also be generic, allowing you to define contracts for various types.

TypeScript

```
interface Pair<T, U> {
  first: T;
  second: U;
}

let pair1: Pair<number, string> = { first: 1, second: "one" };
let pair2: Pair<string, boolean> = { first: "true", second: true };

console.log(pair1);
console.log(pair2);
```

3. Generic Classes:

- Classes can also be generic, allowing you to create classes that work with different types.

TypeScript

```
class Box<T> {
  private value: T;

  constructor(value: T) {
    this.value = value;
  }

  getValue(): T {
```

```
    return this.value;
  }
}

let box1 = new Box<number>(10);
let box2 = new Box<string>("TypeScript");

console.log(box1.getValue()); // Output: 10
console.log(box2.getValue()); // Output: TypeScript
```

4. Type Constraints:

- Type constraints allow you to restrict the types that can be used with a generic type parameter.
- They are defined using the `extends` keyword.

Example: Constraint with Interface:

TypeScript

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length); // Now we know it has a .length propertyreturn a
rg;
  return arg
}

loggingIdentity("Hello"); // Works, string has length
loggingIdentity([1, 2, 3]); // Works, arrays have length
// loggingIdentity(10); // Error: Argument of type 'number' is not assignabl
e to parameter of type 'Lengthwise'.
```

Example: Constraint with Class:

TypeScript

```
class Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}

class Dog extends Animal {
  breed: string;
  constructor(name: string, breed: string) {
    super(name);
    this.breed = breed;
  }
}

function createInstance<T extends Animal>(c: new (name: string, ...args:
any[]) ⇒ T, name: string, ...args: any[]): T {
  return new c(name, ...args);
}

let dog = createInstance(Dog, "Buddy", "Golden Retriever");
console.log(dog.name); // Output: Buddy
console.log(dog.breed); // Output: Golden Retriever
```

5. Default Type Parameters:

- You can provide default types for generic type parameters.

TypeScript

```
function createArray<T = string>(length: number, value: T): T[] {
  let result: T[] = [];
  for (let i = 0; i < length; i++) {
    result[i] = value;
  }
  return result;
```

```
}

let stringArray = createArray(3, "a"); // stringArray is string[]
let numberArray = createArray<number>(3, 10); // numberArray is number
[]
```

Best Practices:

- **Use Descriptive Type Parameters:** Use meaningful names for type parameters to improve readability.
- **Use Constraints:** Use type constraints to enforce specific requirements on types.
- **Prefer Generic Interfaces and Classes:** Use generic interfaces and classes when you need to create reusable components.
- **Use Default Type Parameters:** Use default type parameters when you want to provide default types for generic type parameters.
- **Avoid Excessive Generics:** Don't overuse generics if they don't provide a clear benefit.
- **Keep it simple:** start with simple generic implementations, and expand as needed.
- **Document Generics:** add comments to explain the purpose of generic code.

▼ **Union and Intersection Types:** Combining types using `|` (union) and `&` (intersection).

1. Union Types (`|`)

- **Purpose:**
  - A union type represents a value that can be one of several types. It allows you to specify that a variable or parameter can hold values of multiple different types.
- **Use Cases:**
  - When a function can accept arguments of different types.

- When a variable can hold values of different types at different times.
- When modeling data that can have different shapes.

- **Sample Code:**TypeScript

```
function printId(id: string | number): void {
  if (typeof id === "string") {
    console.log(`ID: ${id.toUpperCase()}`);
  } else {
    console.log(`ID: ${id}`);
  }
}

printId("abc1234");
printId(1234);

type StringOrNumberArray = string | number[];

let myValue: StringOrNumberArray;
myValue = "hello";
myValue = [1, 2, 3];
```

2. Intersection Types (`&`)

- **Purpose:**
  - An intersection type combines multiple types into a single type that has all the properties and methods of all the combined types. It's used to create a type that has the features of multiple types.
- **Use Cases:**
  - When you need to create a type that has the properties of multiple interfaces or types.
  - When you want to combine existing types to create a more specific type.
- **Sample Code:**TypeScript

```
interface Colorful {
  color: string;
}

interface Printable {
  print(): void;
}

type ColorfulPrintable = Colorful & Printable;

const obj: ColorfulPrintable = {
  color: "blue",
  print: () => console.log("Printing..."),
};

console.log(obj.color);
obj.print();
```

3. Best Practices:

- **Use Union Types for Flexibility:**
  - Use union types when a value can be of one of several types. This makes your code more flexible and adaptable to different data inputs.
- **Use Intersection Types for Combining Features:**
  - Use intersection types to combine existing types into a new type that has the properties of all the combined types. This allows you to create more specific and powerful types.
- **Use Type Guards:**
  - When working with union types, use type guards (`typeof`, `instanceof`, custom type guards) to narrow down the type within conditional blocks. This ensures type safety and prevents runtime errors.
- **Discriminated Unions (Tagged Unions):**

    - When working with complex union types, consider using discriminated unions (tagged unions). This involves adding a common property (discriminant) to each type in the union, which allows you to easily narrow down the type at runtime.
- **Use Type Aliases:**
    - Use type aliases to give meaningful names to complex union or intersection types. This makes your code more readable and maintainable.
- **Avoid Excessive Complexity:**
    - Don't create overly complex union or intersection types that are hard to understand. Keep your types as simple as possible while still meeting your needs.
- **Prioritize Readability:**
    - Write type definitions that are easy to read and understand. Use descriptive names for your types and properties.
- **Document Complex Types:**
    - If you have complex union or intersection types, add comments to explain their purpose and usage. This will help other developers (and your future self) understand your code.
- **Use `never` for Impossible Combinations:**
    - If you have a function that should never be called with certain union type combinations, use the `never` type to enforce this.
- **Use Optional Properties Carefully:**
    - When using intersection types, be mindful of optional properties. If one type has an optional property, the intersection type will also have that optional property.
- **Be Aware of Type Narrowing:**
    - TypeScript's type narrowing capabilities are crucial when working with union types. Understand how type guards and other narrowing techniques work.

- **Consider Enums or Literal Types:**
  - When working with discriminated unions, consider using enums or literal types for the discriminant property. This can make your code more readable and maintainable.

▼ **Type Guards:** Using type guards to narrow down types within conditional blocks. `typeof`, `instanceof`, custom type guards.

Type guards are essential in TypeScript for narrowing down the type of a variable within conditional blocks. This allows you to safely perform operations that are specific to a particular type, especially when dealing with union types. Here's a breakdown of `typeof`, `instanceof`, and custom type guards:

1. `typeof` Type Guards:

- **Purpose:**
  - `typeof` is a JavaScript operator that returns a string indicating the type of a value. TypeScript leverages this to narrow down types based on the returned string.
- **Usage:**
  - `typeof` can be used to narrow down types for primitive types like `string`, `number`, `boolean`, `symbol`, `bigint`, and `undefined`.
- **Sample Code:**TypeScript

```
function printValue(value: string | number): void {
  if (typeof value === "string") {
    console.log(value.toUpperCase()); // value is narrowed to string
  } else if (typeof value === "number") {
    console.log(value.toFixed(2)); // value is narrowed to number
  } else {
    console.log("Value is neither string nor number.");
  }
}
```

```
printValue("hello");
printValue(42.123);
```

2. `instanceof` Type Guards:

- **Purpose:**
  - `instanceof` is a JavaScript operator that checks if an object is an instance of a particular class or constructor function. TypeScript uses this to narrow down types based on the result of `instanceof`.
- **Usage:**
  - `instanceof` is used to narrow down types for classes and constructor functions.
- **Sample Code:**TypeScript

```
class Animal {
  move(): void {
    console.log("Moving...");
  }
}

class Dog extends Animal {
  bark(): void {
    console.log("Woof!");
  }
}

function makeAnimalSound(animal: Animal | Dog): void {
  animal.move(); // Common to both Animal and Dog

  if (animal instanceof Dog) {
    animal.bark(); // animal is narrowed to Dog
  }
}
```

```
const myAnimal: Animal = new Animal();
const myDog: Dog = new Dog();

makeAnimalSound(myAnimal);
makeAnimalSound(myDog);
```

3. Custom Type Guards:

- **Purpose:**
  - Custom type guards allow you to define your own functions that perform type narrowing. This is useful when `typeof` and `instanceof` are not sufficient.
- **Usage:**
  - A custom type guard function returns a type predicate, which has the form `variable is Type`.
- **Sample Code:**TypeScript

```
interface Circle {
  kind: "circle";
  radius: number;
}

interface Square {
  kind: "square";
  sideLength: number;
}

type Shape = Circle | Square;

function isCircle(shape: Shape): shape is Circle {
  return shape.kind === "circle";
}

function getArea(shape: Shape): number {
```

```
  if (isCircle(shape)) {
    return Math.PI * shape.radius ** 2; // shape is narrowed to Circle
  } else {
    return shape.sideLength ** 2; // shape is narrowed to Square
  }
}

const myCircle: Circle = { kind: "circle", radius: 5 };
```

▼ **Discriminated Unions:** Using tagged unions to create type-safe variant types.

Discriminated unions, also known as tagged unions or algebraic data types, are a powerful pattern in TypeScript for creating type-safe variant types. They allow you to represent values that can take on different forms, but with a common property (the "discriminant" or "tag") that helps TypeScript narrow down the type at runtime.

Purpose:

- **Type Safety:** Ensure that you're handling all possible variants of a type correctly.
- **Clarity:** Make your code more readable and maintainable by explicitly representing different data shapes.
- **Exhaustiveness Checking:** Allow TypeScript to check if you've handled all possible cases in a `switch` statement or conditional block.

Key Components:

1. **Discriminant (Tag):** A common property that exists in all variants of the union. This property usually has a literal type (e.g., `"circle"`, `"square"`, `"error"`).
2. **Variants:** The different shapes or forms that the union type can take. Each variant has the discriminant property with a unique literal value.
3. **Union Type:** The type that combines all the variants using the `|` (union) operator.

Sample Code:

TypeScript

```
// Define the variants
interface Circle {
  kind: "circle"; // Discriminant
  radius: number;
}

interface Square {
  kind: "square"; // Discriminant
  sideLength: number;
}

interface Rectangle {
    kind: "rectangle"; // Discriminant
    width: number;
    height: number;
}

// Create the union type
type Shape = Circle | Square | Rectangle;

// Function to calculate area based on the shape
function getArea(shape: Shape): number {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.sideLength ** 2;
    case "rectangle":
       return shape.width * shape.height;
    default:
      // This 'default' case is important for exhaustiveness checking.
      // If you add a new shape to the 'Shape' union, TypeScript will
      // warn you if you don't handle it here.
      const _exhaustiveCheck: never = shape;
      return _exhaustiveCheck;
```

```
  }
}

// Example usage
const myCircle: Circle = { kind: "circle", radius: 5 };
const mySquare: Square = { kind: "square", sideLength: 4 };
const myRectangle: Rectangle = {kind: "rectangle", width: 10, height: 5};

console.log(getArea(myCircle)); // Output: 78.53981633974483
console.log(getArea(mySquare)); // Output: 16
console.log(getArea(myRectangle)); // Output: 50
```

Best Practices:

- **Use Literal Types for Discriminants:** Use string or number literal types for the discriminant property. This ensures that the values are known at compile time.
- **Use `switch` Statements:** Use `switch` statements to handle different variants of the union type. This makes your code more readable and maintainable.
- **Exhaustiveness Checking:** Add a `default` case in your `switch` statement and assign the shape to a `never` variable. This allows TypeScript to check if you've handled all possible variants.
- **Use Descriptive Discriminant Names:** Use descriptive names for your discriminant properties (e.g., `kind`, `type`, `status`).
- **Keep Variants Simple:** Keep your variants as simple as possible. Avoid nesting complex objects within variants.
- **Use Type Aliases:** Use type aliases to give meaningful names to your union types.
- **Document Your Unions:** Add comments to explain the purpose and usage of your discriminated unions.
- **Consider Enums:** Enums can also be used as discriminants, especially when the set of possible discriminant values is known and fixed.

- **Avoid Redundant Properties:** Don't add redundant properties to your variants. If a property is common to all variants, move it to a base interface.
- **Use Custom Type Guards:** If you need more complex type narrowing logic, consider using custom type guards.
- **Combine with Generics:** Discriminated unions can be combined with generics to create even more flexible and powerful types.

Sources and related content

▼ **Conditional Types:** Using conditional types to create types that depend on other types.

Conditional types in TypeScript allow you to create a type that depends on the shape of another type. They let you write code that can inspect a type and, based on a condition, choose between one of two different types. Think of them as a type-level `if/else` statement.

Explanation

Conditional types have the form `T extends U ? X : Y`.

- `T extends U`: This is the condition. TypeScript checks if type `T` is assignable to (or can be extended by) type `U`.
- `? X`: If the condition is true, the resulting type is `X`.
- `: Y`: If the condition is false, the resulting type is `Y`.

This is incredibly useful for creating flexible and powerful types, especially when working with generics. A common pattern is to use them in combination with generic type parameters to produce a different return type based on the input type.

Sample Code

Here are a few common use cases for conditional types.

1. The `NonNullable` Type

This example shows how to create a type that excludes `null` and `undefined` from a union type.

TypeScript

```
type MyNonNullable<T> = T extends null | undefined ? never : T;

// T is string | null. It extends null | undefined, so the type is T, which is stri
ng.
type NonNullableString = MyNonNullable<string | null>;
// NonNullableString is now `string`

// T is number. It does not extend null | undefined, so the type is T.
type NonNullableNumber = MyNonNullable<number>;
// NonNullableNumber is now `number`
```

In the example above, `never` is a type that represents a value that can never occur. When used in a conditional type, it effectively removes the matching type from the union.

---

2. The `ReturnType` Type

This is a classic example of a conditional type that extracts the return type of a function type. TypeScript provides a built-in `ReturnType` utility type, but here's how you'd implement it from scratch.

TypeScript

```
type MyReturnType<T> = T extends (...args: any[]) => infer R ? R : any;

function add(a: number, b: number): number {
  return a + b;
}

function greet(name: string): string {
```

```
  return `Hello, ${name}`;
}

// T is the type of the `add` function.
// It extends `(...args: any[]) => infer R`, and `R` is inferred as `number`.
type AddReturnType = MyReturnType<typeof add>;
// AddReturnType is now `number`

// T is the type of the `greet` function.
// It extends `(...args: any[]) => infer R`, and `R` is inferred as `string`.
type GreetReturnType = MyReturnType<typeof greet>;
// GreetReturnType is now `string`
```

In this example, the `infer` keyword is used within the conditional type. It tells TypeScript to "infer" the type of `R` from the function signature and use that inferred type for the return type.

Best Practices

- **Be Descriptive:** Use clear and descriptive names for your generic types. For example, `T` for "Type," but use more specific names like `TMessage` or `TElement` if it adds clarity.
- **Favor Utility Types:** Before creating your own conditional type, check if a built-in utility type already exists. TypeScript's standard library includes `Extract<T, U>`, `Exclude<T, U>`, `NonNullable<T>`, `Parameters<T>`, and `ReturnType<T>`.
- **Use `infer` for Inference:** The `infer` keyword is essential for extracting types from other types (like the return type of a function or the element type of an array). Use it when you need to capture a type to use in the `true` branch of your conditional.
- **Combine with Type Aliases:** Use a `type` alias to give a name to your conditional type. This makes the code more readable and reusable.
- **Keep it Simple:** Avoid creating overly complex conditional types. If a conditional type becomes difficult to read, consider breaking it into multiple, smaller types. This improves readability and maintainability.

- **Document Your Conditional Types:** Because conditional types can be difficult to understand at a glance, be sure to document what your custom conditional type does and how it should be used.

▼ **Mapped Types:** Transforming types based on existing types. `Partial<T>`, `Readonly<T>`, `Pick<T, K>`, `Record<K, T>`.

Mapped types in TypeScript provide a powerful way to create new types by transforming the properties of an existing type. They are a generic feature that iterates over the properties of a type and applies a transformation to each one.

Explanation

The syntax for a mapped type looks similar to an index signature:

TypeScript

```
type MappedType<T> = {
  [P in keyof T]: T[P];
};
```

- `[P in keyof T]`: This part iterates over each property name `P` in the keys of the input type `T`.
- `T[P]`: This gets the type of the property `P` from the original type `T`.

By adding modifiers and transformations, you can create new types with different characteristics.

Utility Types (Sample Code)

TypeScript comes with several built-in mapped types called **utility types**, which are essential for everyday use.

1. `Partial<T>`

- **Purpose:** Makes all properties of a type `T` optional.
- **Use Case:** Creating a type for an object that represents a partial update to an existing object.

TypeScript

```
interface User {
  id: number;
  name: string;
  email: string;
}

// All properties are now optional
type PartialUser = Partial<User>;

let userUpdate: PartialUser = {
  name: "Jane Doe" // Only providing the name property
};
```


2. `Readonly<T>`

- **Purpose:** Makes all properties of a type `T` `readonly`.
- **Use Case:** Ensuring that an object's properties cannot be modified after it's created, promoting immutability.

TypeScript

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

// All properties are now readonly
type ReadonlyPoint = Readonly<Point>;

const myPoint: ReadonlyPoint = {
  x: 10,
  y: 20
};
```

```
// myPoint.x = 30; // Error: Cannot assign to 'x' because it is a read-only pr
operty.
```


3. `Pick<T, K>`

- **Purpose:** Constructs a type by picking a set of properties `K` from a type `T`.
- **Use Case:** Creating a new type with only a subset of properties from a larger interface.

TypeScript

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

// Picks only 'name' and 'price' properties
type ProductSummary = Pick<Product, 'name' | 'price'>;

const productSummary: ProductSummary = {
  name: "Laptop",
  price: 1200
};
```


4. `Record<K, T>`

- **Purpose:** Constructs an object type with keys of type `K` and values of type `T`.
- **Use Case:** Creating a type for a dictionary or a map where all keys and values have specific types.

TypeScript

```
// Keys are strings (e.g., "red", "blue") and values are numbers
type ColorCounts = Record<string, number>;

const counts: ColorCounts = {
  red: 5,
  blue: 10,
  green: 3
};
```

Best Practices

- **Use Built-in Utility Types:** Before creating a custom mapped type, always check if a built-in utility type (like `Partial`, `Readonly`, `Pick`, `Record`, `Omit`, `Exclude`, etc.) already exists.
- **Combine Mapped Types:** Mapped types can be combined to create more complex types. For example, `Partial<Pick<User, 'name' | 'email'>>` would create a type with optional `name` and `email` properties.
- **Enhance with Modifiers:** Use the `readonly` and `?` modifiers to add or remove these properties in a mapped type. For example, `[P in keyof T]-?` removes the optional status of all properties.
- **Document Complex Mapped Types:** If you create a custom mapped type that isn't immediately obvious, add a comment explaining its purpose and how it works.
- **Keep It Simple:** While mapped types are powerful, avoid overly complex transformations that make your code difficult to read.
- **Leverage `keyof` and `typeof`:** Mapped types work best when combined with type operators like `keyof` (to get the keys of a type) and `typeof` (to get the type of a variable).

▼ **Namespaces and Modules:** Organizing code into logical units using namespaces (older) and modules (modern approach). Importing and exporting modules.

Modules and namespaces are both ways to organize code in TypeScript, but they are used for different purposes and represent different eras of JavaScript development. Modules are the modern, standard approach, while namespaces are an older, TypeScript-specific feature that is now less common.

Namespaces (Older Approach)

Namespaces, once called "internal modules," are a way to group related code (like classes, interfaces, and functions) within a global scope to prevent naming conflicts. They create a single global object that acts as a container for your code.

Use and Sample Code

Namespaces are best suited for smaller applications or older codebases where you need to organize code without using a module loader.

TypeScript

```
// greeting.ts
namespace Greetings {
    export interface Greetable {
        greeting: string;
    }

    export class Greeter implements Greetable {
        constructor(public greeting: string) {}
        sayHello() {
            console.log(`Hello, ${this.greeting}!`);
        }
    }
}

// app.ts
// The 'triple-slash directive' is required to include the namespace file
/// <reference path="greeting.ts" />
```

```
let greeter = new Greetings.Greeter("TypeScript");
greeter.sayHello();
```

- `namespace Greetings`: This creates a container object named `Greetings`.
- `export`: The `export` keyword is used to make members of the namespace accessible from outside the container.
- `/// <reference path="..." />`: This is a TypeScript-specific directive that tells the compiler to include the referenced file.

The main drawback of namespaces is that they pollute the global scope with a single container object, and they are not part of the ECMAScript standard.

Modules (Modern Approach)

Modules are the preferred and standard way to organize code in TypeScript and JavaScript. Each file is a module, and any variables, functions, or classes declared in a module are private to that module by default. To make them accessible to other files, you must explicitly **export** them. To use them in another file, you must **import** them.

There are two primary module systems in TypeScript: ES Modules (the modern standard) and CommonJS (used primarily in Node.js).

Importing and Exporting

ES Modules (ESM)

- **Exporting:** Use the `export` keyword for named or default exports.
  - **Named Exports**: `export const PI = 3.14;` or `export class Circle {}`
  - **Default Exports**: `export default class Calculator {}`
- **Importing:** Use the `import` keyword to bring in what you've exported.
  - **Named Imports**: `import { PI, Circle } from './math';`
  - **Default Imports**: `import Calculator from './calculator';`

- **Importing all**: `import * as Math from './math';`

TypeScript

```
// math.ts
export const PI = 3.14159;
export function add(a: number, b: number): number {
   return a + b;
}

// calculator.ts
import { PI, add } from './math';

console.log(PI);
console.log(add(5, 5));
```

CommonJS (CJS)

- **Exporting:** Use `module.exports` or `exports`.
  - **Named Exports**: `exports.add = function() {};`
  - **Default Exports**: `module.exports = { ... };`
- **Importing:** Use the `require()` function.
  - **Importing all**: `const math = require('./math');`
  - **Destructuring**: `const { add } = require('./math');`

TypeScript

```
// math.ts
const add = (a, b) => a + b;
const PI = 3.14159;
module.exports = { add, PI };

// calculator.ts
const { add } = require('./math');
```

```
console.log(add(5, 5));
```

Key Differences

Feature	Namespaces	Modules
Standard	TypeScript-specific	ECMAScript standard
Scope	Global object	File-scoped (private by default)
Usage	Older, smaller apps	Modern, all-size apps
Loading	No module loader required	Requires a module loader or bundler (e.g., webpack, esbuild)
Syntax	`namespace`, `/// <reference>`	`import`, `export`

Export to Sheets

Best Practices

- **Use Modules:** Always use the ES Modules approach (`import`/`export`) for new TypeScript projects. It's the standard for modern JavaScript development.
- **Use a Bundler:** For front-end applications, use a module bundler like webpack, Vite, or esbuild to bundle your modules into a single file for the browser.
- **Avoid Namespaces:** Avoid using namespaces in new code unless you are working with an older codebase that already relies on them.

▼ **Decorators:** Using decorators to add metadata to classes, methods, properties, and parameters.

Decorators are a special kind of declaration that can be attached to a class, method, accessor, property, or parameter. They are functions that add metadata or modify the behavior of the item they decorate at design time.

They are an **experimental feature** in TypeScript and require a specific configuration to be used.

How Decorators Work

A decorator is a function that gets called at runtime with information about the decorated declaration. The function's signature and what it can return depend on what it is decorating.

To enable decorators in your `tsconfig.json`, you need to set the `experimentalDecorators` flag to `true`. You also typically need to enable `emitDecoratorMetadata` if you plan to use a library that relies on it, such as Angular or NestJS.

JSON

```
{
 "compilerOptions": {
   "target": "ES5",
   "experimentalDecorators": true,
   "emitDecoratorMetadata": true
 }
}
```

Class Decorators

A class decorator is a function that takes a single argument: the **constructor** of the class. It can be used to add new properties or methods to the class, or to replace the class with a new one.

Sample Code:

TypeScript

```
function sealed(constructor: Function) {
 Object.seal(constructor);
 Object.seal(constructor.prototype);
}

@sealed
class User {
 constructor(public name: string) {}
}
```

In this example, the `@sealed` decorator makes the `User` class and its prototype non-extensible, preventing new properties from being added.

Method Decorators

A method decorator takes three arguments:

1. The **target** (the class prototype for an instance member, or the constructor function for a static member).

2. The **property key** (the name of the method).
3. The **property descriptor** (a standard JavaScript object that describes the method's configuration, such as `writable` or `enumerable`).

Sample Code:

TypeScript

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

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

 return descriptor;
}

class Calculator {
 @logMethod
 add(a: number, b: number): number {
  return a + b;
 }
}

const calc = new Calculator();
calc.add(2, 3);
```

Here, `@logMethod` wraps the `add` method, logging its arguments and return value.

Property Decorators

A property decorator takes two arguments: the **target** and the **property key**. It cannot directly modify the property's value, but it can be used to add metadata to it.

Sample Code:

TypeScript

```
import 'reflect-metadata';

function format(formatString: string) {
 return Reflect.metadata("format", formatString);
}
```

```
class User {
  @format("email")
  email: string;

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

// Accessing metadata using Reflect.getMetadata
const formatString = Reflect.getMetadata("format", new User("test@example.com"), "email");
console.log(formatString); // Output: email
```

The `@format` decorator here adds metadata about the property's format, which can be read later by a library or framework.

Parameter Decorators

A parameter decorator takes three arguments: the **target**, the **property key** (method name), and the **parameter index**. Like property decorators, it's primarily used for adding metadata.

Sample Code:

TypeScript

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

class Validator {
  validate(@logParameter value: string) {
    // some validation logic
  }
}
```

This decorator logs information about the decorated parameter.

Best Practices

- **Use Frameworks:** Decorators are most commonly used in frameworks like Angular, NestJS, and TypeORM. For general-purpose code, they are often not the best solution due to their experimental nature and potential for making code less readable.

- **Don't Overuse:** Avoid using decorators for simple tasks that can be solved with a function or class. They can add a layer of indirection that makes code harder to debug.
- **Understand `emitDecoratorMetadata`:** If you plan to use decorators with a framework, you will almost certainly need to enable this flag. It emits type information that frameworks use for dependency injection and other features.
- **Be Aware of Changes:** As an experimental feature, the syntax and behavior of decorators could change in future TypeScript versions.

III. Advanced Concepts:

▼ **Advanced Type Manipulation:** Deep dives into conditional types, mapped types, and inference.

TypeScript's type system goes far beyond basic type annotations. Advanced type manipulation techniques like **conditional types**, **mapped types**, and **type inference** allow you to create highly flexible, reusable, and type-safe code. They are the backbone of many of TypeScript's built-in utility types and powerful libraries.

Conditional Types

Conditional types enable you to create a type that depends on a condition, similar to a ternary operator in JavaScript. They have the form `T extends U ? X : Y`.

- `T extends U`: Checks if type `T` is assignable to or a subtype of `U`.
- `? X`: The "true" branch. If the condition is met, the type is `X`.
- `: Y`: The "false" branch. If the condition isn't met, the type is `Y`.

This is a powerful tool for type-level logic.

Sample Code

Let's create a custom `NonNullable` type that removes `null` and `undefined` from a type.

TypeScript

```
type MyNonNullable<T> = T extends null | undefined ? never : T;

// T is string | null. The condition `string | null extends null | undefined` is tr
ue.
type Result1 = MyNonNullable<string | null>; // Result1 is `string`

// T is number. The condition `number extends null | undefined` is false.
type Result2 = MyNonNullable<number>; // Result2 is `number`
```

Mapped Types

Mapped types allow you to create a new type by iterating over the properties of an existing type and transforming them. The syntax is similar to a JavaScript `for...in` loop.

- `[P in keyof T]`: Iterates over each property `P` of the input type `T`.
- `T[P]`: Accesses the type of the property `P` from `T`.

This is used to create utility types like `Partial`, `Readonly`, and `Pick`.

Sample Code

Let's create a custom `Readonly` type that makes all properties of an object `readonly`.

TypeScript

```
type MyReadonly<T> = {
  readonly [P in keyof T]: T[P];
};

interface User {
  id: number;
  name: string;
}

// All properties of ReadonlyUser are now readonly
```

```
type ReadonlyUser = MyReadonly<User>;

const user: ReadonlyUser = { id: 1, name: "Alice" };

// user.id = 2; // Error: Cannot assign to 'id' because it is a read-only prope
rty.
```

Type Inference

Type inference is TypeScript's ability to automatically figure out the type of a variable or expression without an explicit type annotation. It's what makes TypeScript so easy to use. **Inference** with advanced types takes this a step further. The `infer` keyword is used within conditional types to extract a type from another type and assign it to a new type variable.

Sample Code

Let's use `infer` to create a `ReturnType` type that gets the return type of a function.

TypeScript

```
type MyReturnType<T> = T extends (...args: any[]) ⇒ infer R ? R : any;

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

function sum(a: number, b: number): number {
  return a + b;
}

// The 'infer R' keyword captures the return type of the 'greet' function
type GreetReturnType = MyReturnType<typeof greet>; // GreetReturnType
is `string`

// The 'infer R' keyword captures the return type of the 'sum' function
```

```
type SumReturnType = MyReturnType<typeof sum>; // SumReturnType is `number`
```

In this example, the `infer` keyword tells the compiler to "infer" the type of `R` from the return type of the function and assign it to a type variable.

Best Practices

- **Combine for Power:** These techniques are most powerful when combined. Mapped types can use conditional types to conditionally include or transform properties.
- **Favor Built-in Utilities:** Before writing your own, check if a built-in utility type already exists. TypeScript's standard library is extensive and well-tested.
- **Keep it Readable:** While these concepts are advanced, strive for readability. Use descriptive names for your generic type parameters and document complex type transformations.
- **Use `never` for Exhaustiveness:** In conditional types, `never` is the uninhabited type. You can use it to ensure that you've handled all possible cases, which is a key part of discriminated unions.

▼ **Working with the DOM:** Typing the Document Object Model and interacting with the browser.

- **Working with Libraries and Frameworks:** Using TypeScript with React, Angular, Vue, Node.js, etc. Understanding declaration files (`.d.ts`).
- **Performance Optimization:** Writing efficient TypeScript code.
- **Testing TypeScript Code:** Using testing frameworks like Jest or Mocha with TypeScript.
- **Linting and Formatting:** Using tools like ESLint and Prettier to enforce code style and catch errors.
- **Design Patterns in TypeScript:** Applying design patterns using TypeScript's type system.

- **Meta-programming:** Exploring advanced techniques like using decorators for metaprogramming.

▼ Questions