https://github.com/john-smilga/typescript-course

https://www.youtube.com/watch?v=JHEB7RhJG1Y&list=LL&index=11

This project stands as an in-depth guide to TypeScript, meticulously covering its fundamental basics and progressing to its more advanced concepts. It starts with basic setup instructions for creating a TypeScript project using Vite and progresses through a series of tutorials covering various TypeScript features and best practices. Key topics include type annotations, type inference, practical applications of type annotation, union types, handling of "any", "unknown", and "never" types, arrays, and objects fundamentals, challenges to reinforce learning, and functions with their complexities.

The project also delves into advanced TypeScript features such as generics, fetching data with TypeScript, working with the Zod library for data validation, understanding TypeScript declaration files, and class-based programming with TypeScript. Each tutorial is designed to provide hands-on experience with TypeScript, helping learners understand how to apply TypeScript features in real-world scenarios effectively.

Overall, the project is an in-depth TypeScript learning resource, ideal for developers who wish to gain a thorough understanding of TypeScript, from basic to advanced levels, through practical examples and challenges.

Install

```
npm create vite@latest typescript -- --template vanilla-ts
```

Setup

- create src/tutorial.ts
- import tutorial in src/main.ts

```
import "./tutorial.ts";
```

- write code in tutorial
- create README.md
- copy from final

Type Annotations

TypeScript Type Annotations allow developers to specify the types of variables, function parameters, return types, and object properties.

```
let awesomeName: string = "shakeAndBake";
awesomeName = "something";
awesomeName = awesomeName.toUpperCase();
// awesomeName = 20;
```

```
console.log(awesomeName);

let amount: number = 12;
amount = 12 - 1;
// amount = 'pants';

let isAwesome: boolean = true;
isAwesome = false;
// isAwesome = 'shakeAndBake';
```

Type Inference

The typescript compiler can infer the type of the variable based on the literal value that is assigned when it is defined. Just make sure you are working in the typescript file 😄

```
let awesomeName = "shakeAndBake";
awesomeName = "something";
awesomeName = awesomeName.toUpperCase();
// awesomeName = 20;
```

Challenge

- Create a variable of type string and try to invoke a string method on it.
- Create a variable of type number and try to perform a mathematical operation on it.
- Create a variable of type boolean and try to perform a logical operation on it.
- Try to assign a value of a different type to each of these variables and observe the TypeScript compiler's response.
- You can use type annotation or inference

```
// 1. String
let greeting: string = "Hello, TypeScript!";
greeting = greeting.toUpperCase(); // This should work fine

// 2. Number
let age: number = 25;
age = age + 5; // This should work fine

// 3. Boolean
let isAdult: boolean = age >= 18;
isAdult = !isAdult; // This should work fine

// 4. Assigning different types
// Uncommenting any of these will result in a TypeScript error
// greeting = 10; // Error: Type 'number' is not assignable to type 'string'
// age = 'thirty'; // Error: Type 'string' is not assignable to type 'number'
// isAdult = 'yes'; // Error: Type 'string' is not assignable to type 'boolean'
```

Setup Info

- even with error you can run the project but you won't be able to build it "npm run build"

Union Type

In TypeScript, a Union Type allows a variable to hold a value of multiple, distinct types, specified using the | operator. It can also be used to specify that a variable can hold one of several specific values. More examples are coming up.

```
let tax: number | string = 10;
tax = 100;
tax = "$10";

// fancy name - literal value type
let requestStatus: "pending" | "success" | "error" = "pending";
requestStatus = "success";
requestStatus = "error";
// requestStatus = 'random';
```

Type - "any"

In TypeScript, the "any" type is a powerful way to work with existing JavaScript, allowing you to opt-out of type-checking and let the values pass through compile-time checks. It means a variable declared with the any type can hold a value of any type. Later will also cover - "unknown" and "never"

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

Practical Application of Type Annotation

```
const books = ["1984", "Brave New World", "Fahrenheit 451"];

let foundBook: string | undefined;

for (let book of books) {
    if (book === "1984") {
        foundBook = book;
        foundBook = foundBook.toUpperCase();
        break;
    }
}

console.log(foundBook?.length);
```

The reason to explicitly type foundBook as string | undefined is to make it clear to anyone reading the code (including your future self) that foundBook might be undefined at runtime. This is a good practice in TypeScript because it helps prevent bugs related to undefined values.

Challenge

- Create a variable orderStatus of type 'processing' | 'shipped' | 'delivered' and assign it the value 'processing'. Then, try to assign it the values 'shipped' and 'delivered'.
- Create a variable discount of type number | string and assign it the value 20. Then, try to assign it the value '20%'.

```
// 1. Order Status
let orderStatus: "processing" | "shipped" | "delivered" = "processing";
orderStatus = "shipped";
orderStatus = "delivered";
// orderStatus = 'cancelled'; // This will result in a TypeScript error

// 2. Discount
let discount: number | string = 20;
discount = "20%";
// discount = true; // This will result in a TypeScript error
```

Arrays - Fundamentals

In TypeScript, arrays are used to store multiple values in a single variable. You can define the type of elements that an array can hold using type annotations.

```
let prices: number[] = [100, 75, 42];

let fruit: string[] = ["apple", "orange"];
// fruit.push(1);
// let people: string[] = ['bobo', 'peter', 1];
//
// be careful "[]" means literally empty array
// let randomValues: [] = [1];

// be careful with inferred array types
// let names = ['peter', 'susan'];
// let names = ['peter', 'susan', 1];
let array: (string | boolean)[] = ["apple", true, "orange", false];
```

Challenge

- Create an array temperatures of type number[] and assign it some values. Then, try to add a string value to it.
- Create an array colors of type string[] and assign it some values. Then, try to add a boolean value to it.

- Create an array mixedArray of type (number | string)[] and assign it some values. Then, try to add a boolean value to it.

```
// 1. Temperatures
let temperatures: number[] = [20, 25, 30];
// temperatures.push('hot'); // This will result in a TypeScript error

// 2. Colors
let colors: string[] = ["red", "green", "blue"];
// colors.push(true); // This will result in a TypeScript error

// 3. Mixed Array
let mixedArray: (number | string)[] = [1, "two", 3];
// mixedArray.push(true); // This will result in a TypeScript error
```

Objects - Fundamentals

In TypeScript, an object is a collection of key-value pairs with specified types for each key, providing static type checking for properties.

```
let car: { brand: string; year: number } = { brand: "toyota", year: 2020 };
car.brand = "ford";
// car.color = 'blue';

let car1: { brand: string; year: number } = { brand: "audi", year: 2021 };
// let car2: { brand: string; year: number } = { brand: 'audi' };

let book = { title: "book", cost: 20 };
let pen = { title: "pen", cost: 5 };
let notebook = { title: "notebook" };

let items: { readonly title: string; cost?: number }[] = [book, pen, notebook];

items[0].title = "new book"; // Error: Cannot assign to 'title' because it is a
read-only property
```

Challenge

- Create an object bike of type { brand: string, year: number } and assign it some values. Then, try to assign a string to the year property.
- Create an object laptop of type { brand: string, year: number } and try to assign an object with missing year property to it.
- Create an array products of type { title: string, price?: number }[] and assign it some values. Then, try to add an object with a price property of type string to it.

```
// 1. Bike
let bike: { brand: string; year: number } = { brand: "Yamaha", year: 2010 };
```

```
// bike.year = 'old'; // This will result in a TypeScript error

// 2. Laptop
let laptop: { brand: string; year: number } = { brand: "Dell", year: 2020 };
// let laptop2: { brand: string, year: number } = { brand: 'HP' }; // This will
result in a TypeScript error

// 3. Products
let product1 = { title: "Shirt", price: 20 };
let product2 = { title: "Pants" };
let products: { title: string; price?: number }[] = [product1, product2];
// products.push({ title: 'Shoes', price: 'expensive' }); // This will result in a
TypeScript error
```

Functions - Fundamentals

In TypeScript, functions can have typed parameters and return values, which provides static type checking and autocompletion support.

```
function sayHi(name: string) {
    console.log(`Hello there ${name.toUpperCase()}!!!`);
}

sayHi("john");
// sayHi(3)
// sayHi('peter', 'random');

function calculateDiscount(price: number): number {
    // price.toUpperCase();
    const hasDiscount = true;
    if (hasDiscount) {
        return price;
        // return 'Discount Applied';
    }
    return price * 0.9;
}

const finalPrice = calculateDiscount(200);
console.log(finalPrice);

// "any" example
function addThree(number: any) {
    let anotherNumber: number = 3;
    return number + anotherNumber;
}
const result = addThree(2);
const someValue = result;

// run time error
someValue.myMethod();
```

Challenge

- Create a new array of names.
- Write a new function that checks if a name is in your array. This function should take a name as a parameter and return a boolean.
- Use this function to check if various names are in your array and log the results.

```
const names: string[] = ["John", "Jane", "Jim", "Jill"];

function isNameInList(name: string): boolean {
    return names.includes(name);
}

let nameToCheck: string = "Jane";
if (isNameInList(nameToCheck)) {
    console.log(`${nameToCheck} is in the list.`);
} else {
    console.log(`${nameToCheck} is not in the list.`);
}
```

Functions - Optional and Default Parameters

In TypeScript, a default parameter value is an alternative to an optional parameter. When you provide a default value for a parameter, you're essentially making it optional because you're specifying a value that the function will use if no argument is provided for that parameter.

However, there's a key difference between a parameter with a default value and an optional parameter. If a parameter has a default value, and you call the function without providing an argument for that parameter, the function will use the default value. But if a parameter is optional (indicated with a ?), and you call the function without providing an argument for that parameter, the value of the parameter inside the function will be undefined.

- a function with optional parameters must work when they are not supplied

```
function calculatePrice(price: number, discount?: number) {
    return price - (discount || 0);
}

let priceAfterDiscount = calculatePrice(100, 20);
console.log(priceAfterDiscount); // Output: 80

let priceWithoutDiscount = calculatePrice(300);
console.log(priceWithoutDiscount); // Output: 300

function calculateScore(initialScore: number, penaltyPoints: number = 0) {
    return initialScore - penaltyPoints;
}

let scoreAfterPenalty = calculateScore(100, 20);
```

```
console.log(scoreAfterPenalty); // Output: 80

let scoreWithoutPenalty = calculateScore(300);
console.log(scoreWithoutPenalty); // Output: 300
```

Function - rest parameter

In JavaScript, a rest parameter is denoted by three dots (...) before the parameter's name and allows a function to accept any number of arguments. These arguments are collected into an array, which can be accessed within the function.

```
function sum(message: string, ...numbers: number[]): string {
    const doubled = numbers.map((num) => num * 2);
    console.log(doubled);

    let total = numbers.reduce((previous, current) => {
        return previous + current;
    }, 0);
    return `${message} ${total}`;
}

let result = sum("The total is:", 1, 2, 3, 4, 5); // result will be "The total is:
15"
```

Functions - "void" return type

In TypeScript, void is a special type that represents the absence of a value. When used as a function return type, void indicates that the function does not return a value.

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

logMessage("Hello, TypeScript!"); // Output: Hello, TypeScript!
```

It's important to note that in TypeScript, a function that is declared with a void return type can still return a value, but the value will be ignored.For example, the following code is valid TypeScript:

```
function logMessage(message: string): void {
    console.log(message);
    return "This value is ignored";
}

logMessage("Hello, TypeScript!"); // Output: Hello, TypeScript!
```

Functions - Using Union Types as Function Parameters

Challenge

Your task is to create a function named processInput that accepts a parameter of a union type string | number. The function should behave as follows:

- If the input is of type number, the function should multiply the number by 2 and log the result to the console.
- If the input is of type string, the function should convert the string to uppercase and log the result to the console.

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

processInput(10); // Output: 20
processInput("Hello"); // Output: HELLO
```

In this example, the processInput function takes a parameter input that can be either a string or a number. Inside the function, we use a type guard (typeof input === 'number') to check the type of input at runtime. If input is a number, we double it. If input is a string, we convert it to uppercase.

Functions - Using Objects as Function Parameters

```
function createEmployee({ id }: { id: number }): {
    id: number;
    isActive: boolean;
} {
    return { id, isActive: id % 2 === 0 };
}

const first = createEmployee({ id: 1 });
const second = createEmployee({ id: 2 });
console.log(first, second);

// alternative
function createStudent(student: { id: number; name: string }) {
    console.log(`Welcome to the course ${student.name.toUpperCase()}!!!`);
}

const newStudent = {
    id: 5,
    name: "anna",
};
```

```
createStudent(newStudent);
```

Gotcha - Excess Property Checks

```
function createStudent(student: { id: number; name: string }) {
    console.log(`Welcome to the course ${student.name.toUpperCase()}!!!`);
}

const newStudent = {
    id: 5,
    name: "anna",
    email: "anna@gmail.com",
};

createStudent(newStudent);
createStudent({ id: 1, name: "bob", email: "bob@gmail.com" });
```

TypeScript only performs excess property checks on object literals where they're used, not on references to them.

In TypeScript, when you pass an object literal (like { id: 1, name: 'bob', email: 'bob@gmail.com' }) directly to a function or assign it to a variable with a specified type, TypeScript checks that the object only contains known properties. This is done to catch common errors.

However, when you pass newStudent to createStudent, TypeScript doesn't complain about the email property. This is because newStudent is not an object literal at the point where it's passed to createStudent.

Challenge

Your task is to create a function named processData that accepts two parameters:

- The first parameter, input, should be a union type that can be either a string or a number.
- The second parameter, config, should be an object with a reverse property of type boolean, by default it "reverse" should be false

The function should behave as follows:

- If input is of type number, the function should return the square of the number.
- If input is of type string, the function should return the string in uppercase.
- If the reverse property on the config object is true, and input is a string, the function should return the reversed string in uppercase.

```
function processData(
    input: string | number,
    config: { reverse: boolean } = { reverse: false }
): string | number {
    if (typeof input === "number") {
```

```
        return input * input;
    } else {
        return config.reverse
            ? input.toUpperCase().split("").reverse().join("")
            : input.toUpperCase();
    }
}

console.log(processData(10)); // Output: 100
console.log(processData("Hello")); // Output: HELLO
console.log(processData("Hello", { reverse: true })); // Output: OLLEH
```

Type Alias

A type alias in TypeScript is a new name or shorthand for an existing type, making it easier to reuse complex types. However, it's important to note that it doesn't create a new unique type - it's just an alias.All the same rules apply to the aliased type, including the ability to mark properties as optional or readonly.

```
const john: { id: number; name: string; isActive: boolean } = {
    id: 1,
    name: "john",
    isActive: true,
};
const susan: { id: number; name: string; isActive: boolean } = {
    id: 1,
    name: "susan",
    isActive: false,
};

function createUser(user: { id: number; name: string; isActive: boolean }): {
    id: number;
    name: string;
    isActive: boolean;
} {
    console.log(`Hello there ${user.name.toUpperCase()} !!!`);

    return user;
}
```

```
type User = { id: number; name: string; isActive: boolean };

const john: User = {
    id: 1,
    name: "john",
    isActive: true,
};
const susan: User = {
    id: 1,
    name: "susan",
```

```
    isActive: false,
};

function createUser(user: User): User {
    console.log(`Hello there ${user.name.toUpperCase()} !!!`);
    return user;
}

type StringOrNumber = string | number; // Type alias for string | number

let value: StringOrNumber;
value = "hello"; // This is valid
value = 123; // This is also valid

type Theme = "light" | "dark"; // Type alias for theme

let theme: Theme;
theme = "light"; // This is valid
theme = "dark"; // This is also valid

// Function that accepts the Theme type alias
function setTheme(t: Theme) {
    theme = t;
}

setTheme("dark"); // This will set the theme to 'dark'
```

Challenge

- Define the Employee type: Create a type Employee with properties id (number), name (string), and department (string).
- Define the Manager type: Create a type Manager with properties id (number), name (string), and employees (an array of Employee).
- Create a Union Type: Define a type Staff that is a union of Employee and Manager.
- Create the printStaffDetails function: This function should accept a parameter of type Staff. Inside the function, use a type guard to check if the 'employees' property exists in the passed object. If it does, print a message indicating that the person is a manager and the number of employees they manage. If it doesn't, print a message indicating that the person is an employee and the department they belong to.
- Create Employee and Manager objects: Create two Employee objects. One named alice and second named steve. Also create a Manager object named bob who manages alice and steve.
- Test the function: Call the printStaffDetails function with alice and bob as arguments and verify the output.

```
type Employee = { id: number; name: string; department: string };
type Manager = { id: number; name: string; employees: Employee[] };
```

```
type Staff = Employee | Manager;

function printStaffDetails(staff: Staff) {
    if ("employees" in staff) {
        console.log(
            `${staff.name} is a manager of ${staff.employees.length} employees.`
        );
    } else {
        console.log(
            `${staff.name} is an employee in the ${staff.department} department.`
        );
    }
}

const alice: Employee = { id: 1, name: "Alice", department: "Sales" };
const steve: Employee = { id: 1, name: "Steve", department: "HR" };
const bob: Manager = { id: 2, name: "Bob", employees: [alice, steve] };

printStaffDetails(alice); // Outputs: Alice is an employee in the Sales
department.
printStaffDetails(bob);
```

Intersection Types

In TypeScript, an intersection type (TypeA & TypeB) is a way of combining multiple types into one. This means that an object of an intersection type will have all the properties of TypeA and all the properties of TypeB. It's a way of creating a new type that merges the properties of existing types.

```
type Book = { id: number; name: string; price: number };
type DiscountedBook = Book & { discount: number };
const book1: Book = {
    id: 2,
    name: "How to Cook a Dragon",
    price: 15,
};

const book2: Book = {
    id: 3,
    name: "The Secret Life of Unicorns",
    price: 18,
};

const discountedBook: DiscountedBook = {
    id: 4,
    name: "Gnomes vs. Goblins: The Ultimate Guide",
    price: 25,
    discount: 0.15,
};
```

Type Alias - Computed Properties

Computed properties in JavaScript are a feature that allows you to dynamically create property keys on objects. This is done by wrapping an expression in square brackets [] that computes the property name when creating an object.

```
const propName = "age";

type Animal = {
    [propName]: number;
};

let tiger: Animal = { [propName]: 5 };
```

Interface - Fundamentals

- allow to setup shape for objects (only objects)

```
interface Book {
    readonly isbn: number;
    title: string;
    author: string;
    genre?: string;
}

const deepWork: Book = {
    isbn: 9781455586691,
    title: "Deep Work",
    author: "Cal Newport",
    genre: "Self-help",
};

deepWork.title = "New Title"; // allowed
// deepWork.isbn = 654321; // not allowed
```

Interface - Methods

```
interface Book {
    readonly isbn: number;
    title: string;
    author: string;
    genre?: string;
    // method
    printAuthor(): void;
    printTitle(message: string): string;
}
```

```
const deepWork: Book = {
    isbn: 9781455586691,
    title: "Deep Work",
    author: "Cal Newport",
    genre: "Self-help",
    printAuthor() {
        console.log(this.author);
    },
    printTitle(message) {
        return `${this.title} ${message}`;
    },
};
deepWork.printAuthor();
const result = deepWork.printTitle("is an awesome book");
console.log(result);
```

Interface - Methods (more options)

It's generally a good practice to match the structure of the interface and the implementing object or class as closely as possible. This makes the code easier to understand and maintain. So, if printAuthor is defined as a method in the Book interface, it would be more consistent to implement it as a method in the deepWork object.

```
interface Book {
    readonly isbn: number;
    title: string;
    author: string;
    genre?: string;
    // method
    printAuthor(): void;
    printTitle(message: string): string;
    printSomething: (someValue: number) => number;
}

const deepWork: Book = {
    isbn: 9781455586691,
    title: "Deep Work",
    author: "Cal Newport",
    genre: "Self-help",
    printAuthor() {
        console.log(this.author);
    },
    printTitle(message) {
        return `${this.title} ${message}`;
    },
    // first option
    // printSomething: function (someValue) {
    //   return someValue;
    // },
    // second option
    printSomething: (someValue) => {
```

```
        // "this" gotcha
        console.log(deepWork.author);
        return someValue;
    },
    // third option
    // printSomething(someValue) {
    //   return someValue;
    // },
    // alternative
    // printAuthor: () => {
    //   console.log(deepWork.author);
    // },
};
console.log(deepWork.printSomething(34));

deepWork.printAuthor();
const result = deepWork.printTitle("is an awesome book");
console.log(result);
```

Challenge

- Start by defining an interface Computer using the interface keyword. This will serve as a blueprint for objects that will be of this type.
- Inside the interface, define the properties that the object should have. In this case, we have id, brand, ram, and storage.
- Use the readonly keyword before the id property to indicate that it cannot be changed once it's set.
- Use the ? after the storage property to indicate that this property is optional and may not exist on all objects of this type.
- Also inside the interface, define any methods that the object should have. In this case, we have upgradeRam, which is a function that takes a number and returns a number.
- Now that we have our interface, we can create an object that adheres to this interface. This object should have all the properties defined in the interface (except for optional ones, which are... optional), and the methods should be implemented.
- Finally, we can use our object. We can call its upgradeRam method to increase its RAM.

```
interface Computer {
    readonly id: number; // cannot be changed once initialized
    brand: string;
    ram: number;
    upgradeRam(increase: number): number;
    storage?: number; // optional property
}

const laptop: Computer = {
    id: 1,
    brand: "random brand",
    ram: 8, // in GB
    upgradeRam(amount: number) {
        this.ram += amount;
```

```
        return this.ram;
    },
};

laptop.storage = 256; // assigning value to optional property

console.log(laptop.upgradeRam(4)); // upgrades RAM by 4GB
console.log(laptop);
```

Interface - Merging, Extend, TypeGuard

```
interface Person {
    name: string;
    getDetails(): string;
}

interface DogOwner {
    dogName: string;
    getDogDetails(): string;
}

// Merging (reopening) an interface in TypeScript is a process where you declare
an interface with the same name more than once, and TypeScript will merge their
members.

// Merging the interface
interface Person {
    age: number;
}

// Usage
const person: Person = {
    name: "John",
    age: 30,
    getDetails() {
        return `Name: ${this.name}, Age: ${this.age}`;
    },
};

// Extending an interface in TypeScript is a way to create a new interface that
inherits the properties and methods of an existing interface. You use the extends
keyword to do this. When you extend an interface, the new interface will have all
the members of the base interface, plus any new members that you add.

// Extending the interface
interface Employee extends Person {
    employeeId: number;
}

const employee: Employee = {
    name: "jane",
```

```
    age: 28,
    employeeId: 123,
    getDetails() {
        return `Name: ${this.name}, Age: ${this.age}, Employee ID:
${this.employeeId}`;
    },
};

// Interface multiple inheritance
interface Manager extends Person, DogOwner {
    managePeople(): void;
}

const manager: Manager = {
    name: "Bob",
    age: 35,
    dogName: "Rex",
    getDetails() {
        return `Name: ${this.name}, Age: ${this.age}`;
    },
    getDogDetails() {
        return `Dog Name: ${this.dogName}`;
    },
    managePeople() {
        console.log("Managing people...");
    },
};
```

Challenge - Part 1

- Define the Person interface Start by defining a Person interface with a name property of type string.
- Define the DogOwner interface Next, define a DogOwner interface that extends Person and adds a dogName property of type string.
- Define the Manager interface Then, define a Manager interface that extends Person and adds two methods: managePeople and delegateTasks. Both methods should have a return type of void.
- Define the getEmployee function Now, define a function called getEmployee that returns a Person, DogOwner, or Manager. Inside this function, generate a random number and use it to decide which type of object to return. If the number is less than 0.33, return a Person. If it's less than 0.66, return a DogOwner. Otherwise, return a Manager.
- Finally, create a variable called employee that can be a Person, DogOwner, or Manager, and assign it the return value of getEmployee. Then, log employee to the console.

```
interface Person {
    name: string;
}

interface DogOwner extends Person {
    dogName: string;
}
```

```
interface Manager extends Person {
    managePeople(): void;
    delegateTasks(): void;
}

const employee: Person | DogOwner | Manager = getEmployee();
console.log(employee);

function getEmployee(): Person | DogOwner | Manager {
    const random = Math.random();

    if (random < 0.33) {
        return {
            name: "john",
        };
    } else if (random < 0.66) {
        return {
            name: "sarah",
            dogName: "Rex",
        };
    } else {
        return {
            name: "bob",
            managePeople: () => console.log("Managing people..."),
            delegateTasks: () => console.log("Delegating tasks..."),
        };
    }
}
```

Challenge - Part 2

A type predicate in TypeScript is a special kind of return type for a function that not only returns a boolean, but also asserts that the argument is of a specific type if the function returns true. It's typically used in user-defined type guard functions to narrow down the type of a variable within a certain scope. The syntax is arg is Type, where arg is the function argument and Type is the type you're checking for.

- Define the isManager function Define a function called isManager that takes an object of type Person | DogOwner | Manager and returns a boolean. This function should check if the managePeople method exists on the object, and return true if it does and false if it doesn't. The return type of this function should be a type predicate: obj is Manager.
- Run your code to see if it works as expected. If employee is a Manager, you should see the output of the delegateTasks method in the console. If employee is a Person or DogOwner, there should be no output.

```
// function isManager(obj: Person | DogOwner | Manager): boolean {
//   return 'managePeople' in obj;
// }

function isManager(obj: Person | DogOwner | Manager): obj is Manager {
    return "managePeople" in obj;
```

```
}

if (isManager(employee)) {
    employee.delegateTasks();
}
```

Interface vs Type Alias

A type alias is a way to give a name to a type. It can represent primitive types, union types, intersection types, tuples, and any other types. Once defined, the alias can be used anywhere in place of the actual type.

```
type Person = {
    name: string;
    age: number;
};

let john: Person = { name: "John", age: 30 };
```

Interface

An interface is a way to define a contract for a certain structure of an object.

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

let john: Person = { name: "John", age: 30 };
```

Key Differences

- Type aliases can represent primitive types, union types, intersection types, tuples, etc., while interfaces are primarily used to represent the shape of an object.

```
// Type alias for a primitive type
type Score = number;
type NumberOrString = number | string;
// Type alias for literal types
type Direction = "up" | "down" | "left" | "right";

// Using the type aliases
let gameScore: Score = 100;
let move: Direction = "up";
```

- Interfaces can be merged using declaration merging. If you define an interface with the same name more than once, TypeScript will merge their definitions. Type aliases can't be merged in this way.

- Interfaces can be implemented by classes, while type aliases cannot.
- Type aliases can use computed properties, while interfaces cannot.

```
interface Person {
    name: string;
    greet(): void;
}

class Employee implements Person {
    name: string;

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

    greet() {
        console.log(`Hello, my name is ${this.name}`);
    }
}

let john = new Employee("John");
john.greet(); // Outputs: Hello, my name is John
```

```
const propName = "age";

type Animal = {
    [propName]: number;
};

let tiger: Animal = { [propName]: 5 };
```

Tuples

In TypeScript, a Tuple is a special type that allows you to create an array where the type of a fixed number of elements is known, but need not be the same - in other words it's an array with fixed length and ordered with fixed types. This is useful when you want to group different types of values together.

Tuples are useful when you want to return multiple values from a function.

By default, tuples in TypeScript are not read-only. This means you can modify the values of the elements in the tuple.However, TypeScript does provide a way to make tuples read-only using the readonly keyword.

```
let person: [string, number] = ["john", 25];
console.log(person[0]); // Outputs: john
console.log(person[1]); // Outputs: 25

let john: [string, number?] = ["john"];
```

```
function getPerson(): [string, number] {
    return ["john", 25];
}

let randomPerson = getPerson();
console.log(randomPerson[0]); // Outputs: john
console.log(randomPerson[1]);

// let susan: [string, number] = ['susan', 25];
// susan[0] = 'bob';
// susan.push('some random value');

let susan: readonly [string, number] = ["susan", 25];
// susan[0] = 'bob';
// susan.push('some random value');
console.log(susan);
```

Enums

Enums in TypeScript allow us to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases.

```
enum ServerResponseStatus {
    Success = 200,
    Error = "Error",
}

interface ServerResponse {
    result: ServerResponseStatus;
    data: string[];
}

function getServerResponse(): ServerResponse {
    return {
        result: ServerResponseStatus.Success,
        data: ["first item", "second item"],
    };
}

const response: ServerResponse = getServerResponse();
console.log(response);
```

Enums - Gotcha : Reverse Mapping

In a numeric enum, TypeScript creates a reverse mapping from the numeric values to the enum member names. This means that if you assign a numeric value to an enum member, you can use that numeric value anywhere the enum type is expected.

In a string enum, TypeScript does not create a reverse mapping. This means that if you assign a string value to an enum member, you cannot use that string value anywhere the enum type is expected. You must use the enum member itself.

```
enum ServerResponseStatus {
    Success = "Success",
    Error = "Error",
}

Object.values(ServerResponseStatus).forEach((value) => {
    console.log(value);
});
```

```
enum ServerResponseStatus {
    Success = 200,
    Error = 500,
}

Object.values(ServerResponseStatus).forEach((value) => {
    if (typeof value === "number") {
        console.log(value);
    }
});
```

```
enum NumericEnum {
    Member = 1,
}

enum StringEnum {
    Member = "Value",
}

let numericEnumValue: NumericEnum = 1; // This is allowed
console.log(numericEnumValue); // 1

let stringEnumValue: StringEnum = "Value"; // This is not allowed
```

```
enum ServerResponseStatus {
    Success = "Success",
    Error = "Error",
}

function getServerResponse(): ServerResponse {
    return {
        // result: ServerResponseStatus.Success,
```

```
        // this will not fly with string enum but ok with number
        result: "Success",
        data: ["first item", "second item"],
    };
}
```

Challenge

- Define an enum named UserRole with members Admin, Manager, and Employee.
- Define a type alias named User with properties id (number), name (string), role (UserRole), and contact (a tuple with two elements: email as string and phone as string).
- Define a function named createUser that takes a User object as its parameter and returns a User object.
- Call the createUser function with an object that matches the User type, store the result in a variable, and log the variable to the console.

```
// Define an enum named UserRole
enum UserRole {
    Admin,
    Manager,
    Employee,
}

// Define a type alias named User
type User = {
    id: number;
    name: string;
    role: UserRole;
    contact: [string, string]; // Tuple: [email, phone]
};

// Define a function named createUser
function createUser(user: User): User {
    return user;
}

// Call the createUser function
const user: User = createUser({
    id: 1,
    name: "John Doe",
    role: UserRole.Admin,
    contact: ["john.doe@example.com", "123-456-7890"],
});

console.log(user);
```

Type Assertion

Type assertion in TypeScript is a way to tell the compiler what the type of an existing variable is. This is especially useful when you know more about the type of a variable than TypeScript does.

```
let someValue: any = "This is a string";

// Using type assertion to treat 'someValue' as a string
let strLength: number = (someValue as string).length;

type Bird = {
    name: string;
};

// Assume we have a JSON string from an API or local file
let birdString = '{"name": "Eagle"}';
let dogString = '{"breed": "Poodle"}';

//

// Parse the JSON string into an object
let birdObject = JSON.parse(birdString);
let dogObject = JSON.parse(dogString);

// We're sure that the jsonObject is actually a Bird
let bird = birdObject as Bird;
let dog = dogObject as Bird;

console.log(bird.name);
console.log(dog.name);

enum Status {
    Pending = "pending",
    Declined = "declined",
}

type User = {
    name: string;
    status: Status;
};
// save Status.Pending in the DB as a string
// retrieve string from the DB
const statusValue = "pending";

const user: User = { name: "john", status: statusValue as Status };
```

Type - 'unknown'

The unknown type in TypeScript is a type-safe counterpart of the any type. It's like saying that a variable could be anything, but we need to perform some type-checking before we can use it.

"error instanceof Error" checks if the error object is an instance of the Error class.

```
let unknownValue: unknown;
```

```
// Assign different types of values to unknownValue
unknownValue = "Hello World"; // OK
unknownValue = [1, 2, 3]; // OK
unknownValue = 42.3344556; // OK

// unknownValue.toFixed( ); // Error: Object is of type 'unknown'

// Now, let's try to use unknownValue
if (typeof unknownValue === "number") {
    // TypeScript knows that unknownValue is a string in this block
    console.log(unknownValue.toFixed(2)); // OK
}

function runSomeCode() {
    const random = Math.random();
    if (random < 0.5) {
        throw new Error("Something went wrong");
    } else {
        throw "some error";
    }
}

try {
    runSomeCode();
} catch (error) {
    if (error instanceof Error) {
        console.log(error.message);
    } else {
        console.log(error);
        console.log("there was an error....");
    }
}
```

Type - "never"

In TypeScript, never is a type that represents the type of values that never occur.you can't assign any value to a variable of type never. TypeScript will give a compile error if there are any unhandled cases, helping ensure that all cases are handled.

```
// let someValue: never = 0;

type Theme = "light" | "dark";

function checkTheme(theme: Theme) {
    if (theme === "light") {
        console.log("light theme");
        return;
    }
    if (theme === "dark") {
        console.log("dark theme");
        return;
```

```
    }
    theme;
    // theme is of type never, because it can never have a value that is not
'light' or 'dark'.
}
```

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

function getColorName(color: Color) {
    switch (color) {
        case Color.Red:
            return "Red";
        case Color.Blue:
            return "Blue";
        default:
            // at build time
            let unexpectedColor: never = color;
            // at runtime
            throw new Error(`Unexpected color value: ${unexpectedColor}`);
    }
}

console.log(getColorName(Color.Red)); // Red
console.log(getColorName(Color.Blue)); // Blue
// console.log(getColorName(Color.Green)); // Green
```

Modules - Global Scope "Gotcha"

If your TypeScript files aren't modules (i.e., they don't have any import or export statements), they're treated as scripts in the global scope. In this case, declaring the same variable in two different files would cause a conflict.

tutorial.ts

```
let name = "shakeAdnBake";

const susan = "susan";

export let something = "something";
```

actions.ts

```
const susan = "susan";

export const something = "something";
```

tsconfig.json

```
"moduleDetection": "force",
```

- output

tsconfig.json

```
"module": "ESNext",
```

Modules - Imports/Exports (including types)

```
export function sayHello(name: string): void {
    console.log(`Hello ${name}!`);
}

export let person = "susan";

export type Student = {
    name: string;
    age: number;
};

const newStudent: Student = {
    name: "peter",
    age: 24,
};

export default newStudent;
```

```
import newStudent, { sayHello, person, type Student } from "./actions";

sayHello("TypeScript");
console.log(person);
console.log(newStudent);

const anotherStudent: Student = {
    name: "bob",
    age: 23,
};
```

```
console.log(anotherStudent);
```

Modules - Javascript Files

When you set "allowJs": true in your tsconfig.json, TypeScript will process JavaScript files and can infer types to a certain extent based on the structure and usage of your JavaScript code.

However, TypeScript's ability to infer types from JavaScript is not as robust as when working with TypeScript files. For example, it might not be able to infer complex types or types that depend on runtime behavior.

- create example.js
- export someValue, import in tutorial

```
  "allowJs": true,
```

Type Guarding

Type guarding is a term in TypeScript that refers to the ability to narrow down the type of an object within a certain scope. This is usually done using conditional statements that check the type of an object.

In the context of TypeScript, a type guard is some expression that performs a runtime check that guarantees the type in some scope.

Challenge - "typeof" guard

- starter code

```
type ValueType = string | number | boolean;

let value: ValueType;
const random = Math.random();
value = random < 0.33 ? "Hello" : random < 0.66 ? 123.456 : true;
```

- Define the function checkValue that takes one parameter value of type ValueType.
- Inside the function, use an if statement to check if value is of type string. If it is, log value converted to lowercase and then return from the function.
- If value is not a string, use another if statement to check if value is of type number. If it is, log value formatted to two decimal places and then return from the function.
- If value is neither a string nor a number, it must be a boolean. Log the string "boolean: " followed by the boolean value.
- Finally, call the checkValue function with value as the argument.

```
function checkValue(value: ValueType) {
    if (typeof value === "string") {
```

```
        console.log(value.toLowerCase());
        return;
    }
    if (typeof value === "number") {
        console.log(value.toFixed(2));
        return;
    }
    console.log(`boolean: ${value}`);
}

checkValue(value);
```

Challenge - Equality Narrowing

In TypeScript, equality narrowing is a form of type narrowing that occurs when you use equality checks like === or !== in your code

- starter code

```
type Dog = { type: "dog"; name: string; bark: () => void };
type Cat = { type: "cat"; name: string; meow: () => void };
type Animal = Dog | Cat;
```

- Define a function named makeSound that takes one parameter animal of type Animal.
- Inside the function, use an if statement to check if animal.type is 'dog'.
- If animal.type is 'dog', TypeScript knows that animal is a Dog in this block. In this case, call the bark method of animal.
- If animal.type is not 'dog', TypeScript knows that animal is a Cat in the else block. In this case, call the meow method of animal.
- Now you can call the makeSound function with an Animal as the argument. The function will call the appropriate method (bark or meow) depending on the type of the animal.

```
function makeSound(animal: Animal) {
    if (animal.type === "dog") {
        // TypeScript knows that `animal` is a Dog in this block
        animal.bark();
    } else {
        // TypeScript knows that `animal` is a Cat in this block
        animal.meow();
    }
}
```

Challenge - check for property

The "in" operator in TypeScript is used to narrow down the type of a variable when used in a conditional statement.It checks if a property or method exists on an object. If it exists, TypeScript will narrow the type to

the one that has this property.

- starter code

```
type Dog = { type: "dog"; name: string; bark: () => void };
type Cat = { type: "cat"; name: string; meow: () => void };
type Animal = Dog | Cat;
```

- Define a function named makeSound that takes one parameter animal of type Animal.
- Inside the function, use an if statement with the in operator to check if the bark method exists on the animal object.
- If the bark method exists on animal, TypeScript knows that animal is a Dog in this block. In this case, call the bark method of animal.
- If the bark method does not exist on animal, TypeScript knows that animal is a Cat in the else block. In this case, call the meow method of animal.
- Now you can call the makeSound function with an Animal as the argument. The function will call the appropriate method (bark or meow) depending on the type of the animal.

```
function makeSound(animal: Animal) {
    if ("bark" in animal) {
        // TypeScript knows that `animal` is a Dog in this block
        animal.bark();
    } else {
        // TypeScript knows that `animal` is a Cat in this block
        animal.meow();
    }
}
```

Challenge - "Truthy"/"Falsy" guard

In TypeScript, "Truthy"/"Falsy" guard is a simple check for a truthy or falsy value

- Define a function named printLength that takes one parameter str which can be of type string, null, or undefined.
- Inside the function, use an if statement to check if str is truthy. In JavaScript and TypeScript, a truthy value is a value that is considered true when encountered in a Boolean context. All values are truthy unless they are defined as falsy (i.e., except for false, 0, -0, 0n, "", null, undefined, and NaN).
- If str is truthy, it means it's a string (since null and undefined are falsy). In this case, log the length of str using the length property of the string.
- If str is not truthy (i.e., it's either null or undefined), log the string 'No string provided'.
- Now you can call the printLength function with a string, null, or undefined as the argument. The function will print the length of the string if a string is provided, or 'No string provided' otherwise.

```
function printLength(str: string | null | undefined) {
    if (str) {
        // In this block, TypeScript knows that `str` is a string
        // because `null` and `undefined` are falsy values.
        console.log(str.length);
    } else {
        console.log("No string provided");
    }
}

printLength("Hello"); // Outputs: 5
printLength(null); // Outputs: No string provided
printLength(undefined); // Outputs: No string provided
```

Challenge - "instanceof" type guard

The instanceof type guard is a way in TypeScript to check the specific class or constructor function of an object at runtime. It returns true if the object is an instance of the class or created by the constructor function, and false otherwise.

```
try {
    // Some code that may throw an error
    throw new Error("This is an error");
} catch (error) {
    if (error instanceof Error) {
        console.log("Caught an Error object: " + error.message);
    } else {
        console.log("Caught an unknown error");
    }
}
```

- Start by defining the function using the function keyword followed by the function name, in this case checkInput.
- Define the function's parameter. The function takes one parameter, input, which can be of type Date or string. This is denoted by input: Date | string.
- Inside the function, use an if statement to check if the input is an instance of Date. This is done using the instanceof operator.
- If the input is an instance of Date, return the year part of the date as a string. This is done by calling the getFullYear method on the input and then converting it to a string using the toString method.
- If the input is not an instance of Date (which means it must be a string), return the input as it is.
- After defining the function, you can use it by calling it with either a Date or a string as the argument. The function will return the year part of the date if a Date is passed, or the original string if a string is passed.
- You can store the return value of the function in a variable and then log it to the console to see the result.

```
function checkInput(input: Date | string): string {
    if (input instanceof Date) {
        return input.getFullYear().toString();
    }
    return input;
}

const year = checkInput(new Date());
const random = checkInput("2020-05-05");
console.log(year);
console.log(random);
```

Challenge - Type Predicate

A type predicate is a function whose return type is a special kind of type that can be used to narrow down types within conditional blocks.

- starter code

```
type Student = {
    name: string;
    study: () => void;
};

type User = {
    name: string;
    login: () => void;
};

type Person = Student | User;

const randomPerson = (): Person => {
    return Math.random() > 0.5
        ? { name: "john", study: () => console.log("Studying") }
        : { name: "mary", login: () => console.log("Logging in") };
};

const person = randomPerson();
```

- Define the Person and Student types. Student should have a study method and Person should have a login method.
- Create a function named isStudent that takes a parameter person of type Person.
- In the function signature, specify a return type that is a type predicate: person is Student.
- In the function body, use a type assertion to treat person as a Student, and check if the study - method is not undefined. This will return true if person is a Student, and false otherwise.
- Use the isStudent function in an if statement with person as the argument.
- In the if block (where isStudent(person) is true), call the study method on person. TypeScript knows that person is a Student in this block, so this is safe.

- In the else block (where isStudent(person) is false), call the login method on person. This is safe because if person is not a Student, it must be a Person, and all Person objects have a login method.

```
function isStudent(person: Person): person is Student {
    // return 'study' in person;
    return (person as Student).study !== undefined;
}

// Usage

if (isStudent(person)) {
    person.study(); // This is safe because TypeScript knows that 'person' is a
Student.
} else {
    person.login();
}
```

Optional - type "never" gotcha

```
type Student = {
    name: string;
    study: () => void;
};

type User = {
    name: string;
    login: () => void;
};

type Person = Student | User;

const person: Person = {
    name: "anna",
    study: () => console.log("Studying"),
    // login: () => console.log('Logging in'),
};
// person;
function isStudent(person: Person): person is Student {
    // return 'study' in person;
    return (person as Student).study !== undefined;
}

// Usage

if (isStudent(person)) {
    person.study(); // This is safe because TypeScript knows that 'person' is a
Student.
} else {
    // in this case person is type "never"
```

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

Challenge - Discriminated Unions and exhaustive check using the never type

A discriminated union in TypeScript is a type that can be one of several different types, each identified by a unique literal property (the discriminator), allowing for type-safe handling of each possible variant.

- starter code

```
type IncrementAction = {
    amount: number;
    timestamp: number;
    user: string;
};

type DecrementAction = {
    amount: number;
    timestamp: number;
    user: string;
};

type Action = IncrementAction | DecrementAction;
```

- Write a reducer function that takes the current state and an action, and returns the new state. The reducer function should use a switch statement or if-else chain on the type property of the action to handle each action type differently.
- In the default case of the switch statement or the final else clause, perform an exhaustive check by assigning the action to a variable of type never. If there are any action types that haven't been handled, TypeScript will give a compile error.
- Implement the logic for each action type in the reducer function. This typically involves returning a new state based on the current state and the properties of the action.
- Use the reducer function in your application to handle actions and update the state.

```
type IncrementAction = {
    type: "increment";
    amount: number;
    timestamp: number;
    user: string;
};

type DecrementAction = {
    type: "decrement";
    amount: number;
```

```
    timestamp: number;
    user: string;
};

type Action = IncrementAction | DecrementAction;

function reducer(state: number, action: Action): number {
    switch (action.type) {
        case "increment":
            return state + action.amount;
        case "decrement":
            return state - action.amount;

        default:
            const unexpectedAction: never = action;
            throw new Error(`Unexpected action: ${unexpectedAction}`);
    }
}

const newState = reducer(15, {
    user: "john",
    type: "increment",
    amount: 5,
    timestamp: 123456,
});
```

Generics - Fundamentals

Generics in TypeScript are a way to create reusable code components that work with a variety of types as opposed to a single one.

In other words, generics allow you to write a function or a class that can work with any data type. You can think of generics as a kind of variable for types.

```
// In TypeScript, you can declare an array using two syntaxes:

// let array1: string[] = ['Apple', 'Banana', 'Mango'];
// let array2: number[] = [1, 2, 3];
// let array3: boolean[] = [true, false, true];

let array1: Array<string> = ["Apple", "Banana", "Mango"];
let array2: Array<number> = [1, 2, 3];
let array3: Array<boolean> = [true, false, true];
```

Generics - Create Generic Function and Interface

```
//
function createString(arg: string): string {
    return arg;
```

```
}
function createNumber(arg: number): number {
    return arg;
}

// Define a generic function
function genericFunction<T>(arg: T): T {
    return arg;
}

const someStringValue = genericFunction<string>("Hello World");
const someNumberValue = genericFunction<number>(2);

// Define a generic interface
interface GenericInterface<T> {
    value: T;
    getValue: () => T;
}

const genericString: GenericInterface<string> = {
    value: "Hello World",
    getValue() {
        return this.value;
    },
};
```

Generics - Promise Example

```
// A Promise in JavaScript (and thus TypeScript) is an object representing the
eventual completion or failure of an asynchronous operation.

async function someFunc(): Promise<string> {
    return "Hello World";
}

const result = someFunc();
```

Generics - Generate Array

```
// generate an array of strings
function generateStringArray(length: number, value: string): string[] {
    let result: string[] = [];
    result = Array(length).fill(value);
    return result;
}

console.log(generateStringArray(3, "hello"));

function createArray<T>(length: number, value: T): Array<T> {
```

```
    let result: T[] = [];
    result = Array(length).fill(value);
    return result;
}

let arrayStrings = createArray<string>(3, "hello"); // ["hello", "hello", "hello"]
let arrayNumbers = createArray<number>(4, 100); // [100, 100, 100, 100]

console.log(arrayStrings);
console.log(arrayNumbers);
```

Generics - Part 5

```
function pair<T, U>(param1: T, param2: U): [T, U] {
    return [param1, param2];
}

// Usage
let result = pair<number, string>(123, "Hello");
console.log(result); // Output: [123, "Hello"]
```

Generics - Inferred Type and Type Constraints

```
function pair<T, U>(param1: T, param2: U): [T, U] {
    return [param1, param2];
}

// Usage
let result = pair(123, "Hello");

//  const [name,setName] = useState('')
//  const [products,setProducts] = useState<Product[]>([])

// type constraint on the generic type T, generic type can be either a number or a
string.

function processValue<T extends number | string>(value: T): T {
    console.log(value);
}

processValue("hello");
processValue(12);
processValue(true);
```

Generics - Type Constraints 2

```
type Car = {
    brand: string;
    model: string;
};

const car: Car = {
    brand: "ford",
    model: "mustang",
};

type Product = {
    name: string;
    price: number;
};

const product: Product = {
    name: "shoes",
    price: 1.99,
};

type Student = {
    name: string;
    age: number;
};

const student: Student = {
    name: "peter",
    age: 20,
};

// T extends Student is a type constraint on the generic type T. It means that the
type T can be any type, but it must be a subtype of Student or Student itself. In
other words, T must have at least the same properties and methods that Student
has.

// function printName<T extends Student>(input: T): void {
//   console.log(input.name);
// }

// printName(student);

// function printName<T extends Student | Product>(input: T): void {
//   console.log(input.name);
// }

// printName(product);

// The extends { name: string } part is a type constraint on T. It means that T
can be any type, but it must be an object that has at least a name property of
type string.
// In other words, T must have at least the same properties and methods that {
name: string } has.
function printName<T extends { name: string }>(input: T): void {
```

```
    console.log(input.name);
}

printName(student);
printName(product);
```

Generics - Default Value

```
interface StoreData<T = any> {
    data: T[];
}

const storeNumbers: StoreData<number> = {
    data: [1, 2, 3, 4],
};

const randomStuff: StoreData = {
    data: ["random", 1],
};
```

```
// data is located in the data property

const { data } = axios.get(someUrl);

axios.get<{ name: string }[]>(someUrl);

export class Axios {
    get<T = any, R = AxiosResponse<T>, D = any>(
        url: string,
        config?: AxiosRequestConfig<D>
    ): Promise<R>;
}

export interface AxiosResponse<T = any, D = any> {
    data: T;
    status: number;
    statusText: string;
    headers: RawAxiosResponseHeaders | AxiosResponseHeaders;
    config: InternalAxiosRequestConfig<D>;
    request?: any;
}
```

Fetch Data

- typically axios and React Query (or both 🚀🚀🚀)
- we won't set any state values

```
const url = "https://www.course-api.com/react-tours-project";

async function fetchData(url: string) {
    try {
        const response = await fetch(url);

        // Check if the request was successful.
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }

        const data = await response.json();
        return data;
    } catch (error) {
        const errMsg =
            error instanceof Error ? error.message : "there was an error...";
        console.error(errMsg);
        // throw error;
        return [];
    }
}

const tours = await fetchData(url);
tours.map((tour: any) => {
    console.log(tour.name);
});

// return empty array
// throw error in catch block
// we are not setting state values in this function
```

Challenge - Fetch Data

- setup type and provide correct return type

```
const url = "https://www.scourse-api.com/react-tours-project";

// Define a type for the data you're fetching.
type Tour = {
    id: string;
    name: string;
    info: string;
    image: string;
    price: string;
    // Add more fields as necessary.
};

async function fetchData(url: string): Promise<Tour[]> {
    try {
        const response = await fetch(url);
```

```
        // Check if the request was successful.
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }

        const data: Tour[] = await response.json();
        console.log(data);
        return data;
    } catch (error) {
        const errMsg =
            error instanceof Error ? error.message : "there was an error...";
        console.error(errMsg);

        // throw error;
        return [];
    }
}

const tours = await fetchData(url);
tours.map((tour) => {
    console.log(tour.name);
});
```

ZOD Library

- Tour Data "Gotcha"

```
npm install zod
```

- Zod
- Error Handling in Zod

```
import { z } from "zod";
const url = "https://www.course-api.com/react-tours-project";

const tourSchema = z.object({
    id: z.string(),
    name: z.string(),
    info: z.string(),
    image: z.string(),
    price: z.string(),
    somethign: z.string(),
});

// extract the inferred type
type Tour = z.infer<typeof tourSchema>;

async function fetchData(url: string): Promise<Tour[]> {
```

```
    try {
        const response = await fetch(url);

        // Check if the request was successful.
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }

        const rawData: Tour[] = await response.json();
        const result = tourSchema.array().safeParse(rawData);
        if (!result.success) {
            throw new Error(`Invalid data: ${result.error}`);
        }
        return result.data;
    } catch (error) {
        const errMsg =
            error instanceof Error ? error.message : "there was an error...";
        console.log(errMsg);

        // throw error;
        return [];
    }
}

const tours = await fetchData(url);
tours.map((tour) => {
    console.log(tour.name);
});
```

Typescript Declaration File

In TypeScript, .d.ts files, also known as declaration files,consist of type definitions, and are used to provide types for JavaScript code. They allow TypeScript to understand the shape and types of objects, functions, classes, etc., in JavaScript libraries, enabling type checking and autocompletion in TypeScript code that uses these libraries.

- create types.ts
- export RandomType

tsconfig.json

- `lib`: Set to `["ES2020", "DOM", "DOM.Iterable"]`. This specifies the library files to be included in the compilation. Specify a set of bundled library declaration files that describe the target runtime environment.
- libraries

DefinitelyTyped

- password hashing library

```
npm i bcryptjs
```

```
npm install --save-dev @types/bcryptjs
```

tsconfig.json Configuration

tsconfig

This project's TypeScript configuration is defined in the `tsconfig.json` file. Here's a breakdown of the configuration options:

- `include`: Set to `["src"]`. This tells TypeScript to only convert files in the `src` directory.
- `target`: Set to `ES2020`. This is the JavaScript version that the TypeScript code will be compiled to.
- `useDefineForClassFields`: Set to `true`. This enables the use of the `define` semantics for initializing class fields.
- `module`: Set to `ESNext`. This is the module system for the compiled code.
- `lib`: Set to `["ES2020", "DOM", "DOM.Iterable"]`. This specifies the library files to be included in the compilation.
- `skipLibCheck`: Set to `true`. This makes TypeScript skip type checking of declaration files (`*.d.ts`).
- `moduleResolution`: Set to `bundler`. This sets the strategy TypeScript uses to resolve modules.
- `allowImportingTsExtensions`: Set to `true`. This allows importing of TypeScript files from JavaScript files.
- `resolveJsonModule`: Set to `true`. This allows importing of `.json` modules from TypeScript files.
- `isolatedModules`: Set to `true`. This ensures that each file can be safely transpiled without relying on other import/export files.
- `noEmit`: Set to `true`. This tells TypeScript to not emit any output files (`*.js` and `*.d.ts` files) after compilation.
- `strict`: Set to `true`. This enables all strict type-checking options.
- `noUnusedLocals`: Set to `true`. This reports an error when local variables are declared but never used.
- `noUnusedParameters`: Set to `true`. This reports an error when function parameters are declared but never used.
- `noFallthroughCasesInSwitch`: Set to `true`. This reports an error for fall through cases in switch statements.

Classes - Intro

Classes in JavaScript are a blueprint for creating objects. They encapsulate data with code to manipulate that data. Classes in JavaScript support inheritance and can be used to create more complex data structures.

A constructor in a class is a special method that gets called when you create a new instance of the class. It's often used to set the initial state of the object.

```
class Book {
    title: string;
    author: string;
    constructor(title: string, author: string) {
        this.title = title;
        this.author = author;
    }
}

const deepWork = new Book("deep work ", "cal newport");
```

Classes - Instance Property / Default Property

The checkedOut property in Book class is an instance property (or member variable). It's not specifically set in the constructor, so it could also be referred to as a default property or a property with a default value.

```
class Book {
    title: string;
    author: string;
    checkedOut: boolean = false;
    constructor(title: string, author: string) {
        this.title = title;
        this.author = author;
    }
}

const deepWork = new Book("deep work ", "cal newport");
deepWork.checkedOut = true;
// deepWork.checkedOut = 'something else';
```

Classes - ReadOnly Modifier

- readonly modifier

```
class Book {
    readonly title: string;
    author: string;
    checkedOut: boolean = false;
    constructor(title: string, author: string) {
```

```
        this.title = title;
        this.author = author;
    }
}

const deepWork = new Book("deep work ", "cal newport");

deepWork.title = "something else";
```

Classes - Private and Public Modifiers

- private and public modifiers

```
class Book {
    public readonly title: string;
    public author: string;
    private checkedOut: boolean = false;
    constructor(title: string, author: string) {
        this.title = title;
        this.author = author;
    }
    public checkOut() {
        this.checkedOut = this.toggleCheckedOutStatus();
    }
    public isCheckedOut() {
        return this.checkedOut;
    }
    private toggleCheckedOutStatus() {
        return !this.checkedOut;
    }
}

const deepWork = new Book("Deep Work", "Cal Newport");
deepWork.checkOut();
console.log(deepWork.isCheckedOut()); // true
// deepWork.toggleCheckedOutStatus(); // Error: Property 'toggleCheckedOutStatus'
is private and only accessible within class 'Book'.
```

Classes - Shorthand Syntax

In TypeScript, if you want to use the shorthand for creating and initializing class properties in the constructor, you need to use public, private, or protected access modifiers.

```
class Book {
    private checkedOut: boolean = false;
    constructor(public readonly title: string, public author: string) {}
}
```

Classes - Getters and Setters

Getters and setters are special methods in a class that allow you to control how a property is accessed and modified.They are used like properties, not methods, so you don't use parentheses to call them.

```
class Book {
    private checkedOut: boolean = false;
    constructor(public readonly title: string, public author: string) {}
    get info() {
        return `${this.title} by ${this.author}`;
    }

    private set checkOut(checkedOut: boolean) {
        this.checkedOut = checkedOut;
    }
    get checkOut() {
        return this.checkedOut;
    }
    public get someInfo() {
        this.checkOut = true;
        return `${this.title} by ${this.author}`;
    }
}

const deepWork = new Book("deep work", "cal newport");
console.log(deepWork.info);
// deepWork.checkOut = true;
console.log(deepWork.someInfo);
console.log(deepWork.checkOut);
```

Classes - Implement Interface

In TypeScript, an interface is a way to define a contract for a certain structure of an object. This contract can then be used by a class to ensure it adheres to the structure defined by the interface.

When a class implements an interface, it is essentially promising that it will provide all the properties and methods defined in the interface. If it does not, TypeScript will throw an error at compile time.

```
interface IPerson {
    name: string;
    age: number;
    greet(): void;
}

class Person implements IPerson {
    constructor(public name: string, public age: number) {}

    greet() {
        console.log(
            `Hello, my name is ${this.name} and I'm ${this.age} years old.`
```

```
        );
    }
}

const hipster = new Person("shakeAndBake", 100);
hipster.greet();
```

Tasks - Setup

- create tasks.html (root) and src/tasks.ts
- optional : change href in main.ts
- optional css
    - create tasks.css (copy from final or end of the README)
    - setup link in tasks.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Tasks</title>
        <link rel="stylesheet" href="src/tasks.css" />
    </head>
    <body>
        <main>
            <h2>Tasks</h2>
            <form class="form">
                <input type="text" class="form-input" />
                <button type="submit" class="btn">add task</button>
            </form>
            <ul class="list"></ul>
            <button class="test-btn">click me</button>
        </main>
        <script type="module" src="src/tasks.ts"></script>
    </body>
</html>
```

Tasks - Part 2

```
const btn = document.querySelector(".btn");

btn?.addEventListener("click", () => {
    console.log("something");
});

if (btn) {
    // do something
}
```

The ! operator in TypeScript is officially known as the Non-null assertion operator. It is used to assert that its preceding expression is not null or undefined.

```
const btn = document.querySelector(".btn")!;

btn.addEventListener("click", () => {
    console.log("something");
});
```

Element is the most general base class from which all element objects (i.e. objects that represent elements) in a Document inherit. It only has methods and properties common to all kinds of elements. More specific classes inherit from Element.

```
const btn = document.querySelector<HTMLButtonElement>(".selector")!;

btn.disabled = true;

const btn = document.querySelector(".selector")! as HTMLButtonElement;

btn.disabled = true;
```

Tasks - Part 3

```
const taskForm = document.querySelector<HTMLFormElement>(".form");
const formInput = document.querySelector<HTMLInputElement>(".form-input");
const taskListElement = document.querySelector<HTMLUListElement>(".list");

// task type
type Task = {
    description: string;
    isCompleted: boolean;
};

const tasks: Task[] = [];
```

Tasks - Part 4

```
taskForm?.addEventListener("submit", (event) => {
    event.preventDefault();
    const taskDescription = formInput?.value;
    if (taskDescription) {
        // add task to list
        // render tasks
```

```
        // update local storage

        formInput.value = "";
        return;
    }
    alert("Please enter a task description");
});
```

- event gotcha

```
function createTask(event: SubmitEvent) {
    event.preventDefault();
    const taskDescription = formInput?.value;
    if (taskDescription) {
        // add task to list
        // render tasks
        // update local storage

        formInput.value = "";
        return;
    }
    alert("Please enter a task description");
}

taskForm?.addEventListener("submit", createTask);
```

Tasks - Part 5

```
taskForm?.addEventListener("submit", (event) => {
    event.preventDefault();
    const taskDescription = formInput?.value;
    if (taskDescription) {
        const task: Task = {
            description: taskDescription,
            isCompleted: false,
        };
        // add task to list
        addTask(task);
        // render tasks

        // update local storage

        formInput.value = "";
        return;
    }
    alert("Please enter a task description");
});

function addTask(task: Task): void {
```

```
    tasks.push(task);
    // console.log(tasks);
}
```

Tasks - Part 6

```
function renderTask(task: Task): void {
    const taskElement = document.createElement("li");
    taskElement.textContent = task.description;
    taskListElement?.appendChild(taskElement);
}
```

```
// add task to list
addTask(task);
// render task
renderTask(task);
```

Tasks - Part 7

```
// Retrieve tasks from localStorage
const tasks: Task[] = loadTasks();

// Load tasks from localStorage
function loadTasks(): Task[] {
    const storedTasks = localStorage.getItem("tasks");
    return storedTasks ? JSON.parse(storedTasks) : [];
}

// tasks.forEach((task) => renderTask(task));
tasks.forEach(renderTask);

// Update tasks in localStorage
function updateStorage(): void {
    localStorage.setItem("tasks", JSON.stringify(tasks));
}
```

```
// add task to list
addTask(task);
// render task
renderTask(task);
// update local storage
updateStorage();
```

Tasks - Part 8

```
function renderTask(task: Task): void {
    const taskElement = document.createElement("li");
    taskElement.textContent = task.description;
    // checkbox
    const taskCheckbox = document.createElement("input");
    taskCheckbox.type = "checkbox";
    taskCheckbox.checked = task.isCompleted;

    taskElement.appendChild(taskCheckbox);
    taskListElement?.appendChild(taskElement);
}
```

Tasks - Part 9

```
function renderTask(task: Task): void {
    const taskElement = document.createElement("li");
    taskElement.textContent = task.description;
    // checkbox
    const taskCheckbox = document.createElement("input");
    taskCheckbox.type = "checkbox";
    taskCheckbox.checked = task.isCompleted;
    // toggle checkbox
    taskCheckbox.addEventListener("change", () => {
        task.isCompleted = !task.isCompleted;
        updateStorage();
    });

    taskElement.appendChild(taskCheckbox);
    taskListElement?.appendChild(taskElement);
}
```

Tasks - CSS

tasks.css

```
/* ============= GLOBAL CSS =============== */

*,
::after,
::before {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}
```

```
html {
    font-size: 100%;
} /*16px*/

:root {
    /* colors */
    --primary-100: #e2e0ff;
    --primary-200: #c1beff;
    --primary-300: #a29dff;
    --primary-400: #837dff;
    --primary-500: #645cff;
    --primary-600: #504acc;
    --primary-700: #3c3799;
    --primary-800: #282566;
    --primary-900: #141233;

    /* grey */
    --grey-50: #f8fafc;
    --grey-100: #f1f5f9;
    --grey-200: #e2e8f0;
    --grey-300: #cbd5e1;
    --grey-400: #94a3b8;
    --grey-500: #64748b;
    --grey-600: #475569;
    --grey-700: #334155;
    --grey-800: #1e293b;
    --grey-900: #0f172a;
    /* rest of the colors */
    --black: #222;
    --white: #fff;
    --red-light: #f8d7da;
    --red-dark: #842029;
    --green-light: #d1e7dd;
    --green-dark: #0f5132;

    --small-text: 0.875rem;
    --extra-small-text: 0.7em;
    /* rest of the vars */

    --border-radius: 0.25rem;
    --letter-spacing: 1px;
    --transition: 0.3s ease-in-out all;
    --max-width: 1120px;
    --fixed-width: 600px;
    --view-width: 90vw;
    /* box shadow*/
    --shadow-1: 0 1px 3px 0 rgba(0, 0, 0, 0.1), 0 1px 2px 0 rgba(0, 0, 0, 0.06);
    --shadow-2: 0 4px 6px -1px rgba(0, 0, 0, 0.1), 0 2px 4px -1px rgba(0, 0, 0,
0.06);
    --shadow-3: 0 10px 15px -3px rgba(0, 0, 0, 0.1), 0 4px 6px -2px rgba(0, 0, 0,
0.05);
    --shadow-4: 0 20px 25px -5px rgba(0, 0, 0, 0.1), 0 10px 10px -5px rgba(0, 0,
0, 0.04);
    /* DARK MODE */
```

```
    --background-color: var(--grey-50);
    --text-color: var(--grey-900);
}

body {
    background: var(--background-color);
    font-family: system-ui, -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto,
        Oxygen, Ubuntu, Cantarell, "Open Sans", "Helvetica Neue", sans-serif;
    font-weight: 400;
    line-height: 1;
    color: var(--text-color);
}
p {
    margin: 0;
}
h1,
h2,
h3,
h4,
h5 {
    margin: 0;
    font-weight: 400;
    line-height: 1;
    text-transform: capitalize;
    letter-spacing: var(--letter-spacing);
}

h1 {
    font-size: clamp(2rem, 5vw, 5rem); /* Large heading */
}

h2 {
    font-size: clamp(1.5rem, 3vw, 3rem); /* Medium heading */
}

h3 {
    font-size: clamp(1.25rem, 2.5vw, 2.5rem); /* Small heading */
}

h4 {
    font-size: clamp(1rem, 2vw, 2rem); /* Extra small heading */
}

h5 {
    font-size: clamp(0.875rem, 1.5vw, 1.5rem); /* Tiny heading */
}

small,
.text-small {
    font-size: var(--small-text);
}

a {
    text-decoration: none;
```

```
}
ul {
    list-style-type: none;
    padding: 0;
}

.img {
    width: 100%;
    display: block;
    object-fit: cover;
}
/* buttons */

.btn {
    cursor: pointer;
    color: var(--white);
    background: var(--primary-500);
    border: transparent;
    letter-spacing: var(--letter-spacing);
    box-shadow: var(--shadow-1);
    transition: var(--transition);
    text-transform: capitalize;
    display: inline-block;
}
.btn:hover {
    background: var(--primary-700);
    box-shadow: var(--shadow-3);
}
.btn-hipster {
    color: var(--primary-500);
    background: var(--primary-200);
}
.btn-hipster:hover {
    color: var(--primary-200);
    background: var(--primary-700);
}
.btn-block {
    width: 100%;
}

/* alerts */
.alert {
    padding: 0.375rem 0.75rem;
    margin-bottom: 1rem;
    border-color: transparent;
    border-radius: var(--border-radius);
}

.alert-danger {
    color: var(--red-dark);
    background: var(--red-light);
}
.alert-success {
    color: var(--green-dark);
```

```
    background: var(--green-light);
}
/* form */

.form {
    background: var(--white);
    border-radius: var(--border-radius);
    box-shadow: var(--shadow-2);
    padding: 2rem 2.5rem;
    margin-bottom: 2rem;
}

.form-input {
    width: 100%;
    padding: 0.375rem 0.75rem;
    border-top-left-radius: var(--border-radius);
    border-bottom-left-radius: var(--border-radius);
    background: var(--background-color);
    border: 1px solid var(--grey-200);
}

main {
    padding: 5rem 0;
    min-height: 100vh;
    width: 90vw;
    max-width: 500px;
    margin: 0 auto;
}

/* title */
main h2 {
    text-align: center;
    margin-bottom: 2rem;
}

.form {
    display: grid;
    grid-template-columns: 1fr 100px;
}

.form button {
    border-top-right-radius: var(--border-radius);
    border-bottom-right-radius: var(--border-radius);
}

.list li {
    display: flex;
    justify-content: space-between;
    align-items: center;
    padding: 0.75rem 1.25rem;
    margin-bottom: 0.5rem;
    background: var(--white);
    border-radius: var(--border-radius);
```

```
    box-shadow: var(--shadow-1);
}
```