week 9.2 notes

Week 9.2

Introduction to Typescript

In this lecture, Harkirat offers a brief introduction to `TypeScript`, covering language classifications, the importance of strong typing, and an overview of TypeScript's execution. The lecture includes insights into the TypeScript compiler, implementing basic types, and understanding the distinctions between `Interfaces` and `Types`

Introduction to TypescriptTypes of Languages1] Loosely Typed LanguagesC++ Code (Doesn't Work ❌)2] Strongly Typed LanguagesJavaScript Code (Does Work ✅)TypescriptWhy TypescriptWhat TypescriptHow TypescriptExecution of TypeScript CodeTypeScript Compiler (tsc)Setting up a Typescript Nodejs ApplicationBasic Types in TypescriptProblems and Code Implementation1] Hello World Greeting2] Sum Function3] Age Verification Function4] Delayed Function ExecutionThe tsconfig.json File in TypeScript1] Target Option in tsconfig.json:2] rootDir:3] outDir4] noImplicitAny5] removeCommentsInterfacesUnderstanding InterfacesAssignment 1Assignment 2Implementing InterfacesTypesFeaturesInterfaces vs TypesMajor Differences1. Declaration Syntax:2. Extension and Merging:3. Declaration vs. Implementation:Other DifferencesWhen to Use WhichExamples

Types of Languages

1] Loosely Typed Languages

1. **Runtime Type Association:** Data types are associated with values at runtime. Unlike strongly typed languages, type information is not strictly bound during compilation but rather at the time of execution.
2. **Dynamic Type Changes:** Variables can change types during execution, offering more adaptability. This flexibility allows for a dynamic approach to variable assignments and operations.

3. **Runtime Error Discovery:** Type errors may be discovered during runtime, potentially leading to unexpected behaviors. This characteristic provides more freedom but requires careful handling.
4. **Examples of Loosely Typed Languages: J**avaScript, Python, Ruby

C++ Code (Doesn't Work ❌)

```
#include <iostream>

int main() {
  int number = 10;
  number = "text";  // Error: Cannot assign a string to an in
teger variable
  return 0;
}
```

Explanation:

- C++ is a statically-typed language, meaning variable types must be declared and are enforced at compile-time.
- In the given code, `number` is declared as an integer (`int`), and attempting to assign a string ("text") to it results in a compile-time error.
- The type mismatch between the declared type and the assigned value leads to a compilation failure.

2] Strongly Typed Languages

1. **Compile-Time Enforcement:** The data type of a variable is strictly enforced during compilation. This means that the compiler checks and ensures that variables are used in a way that is consistent with their types at compile time.
2. **Type Safety:** The compiler or interpreter guarantees that operations are performed only on compatible types. This ensures that type-related errors are caught early in the development process.

3. **Early Error Detection:** Type errors are identified and addressed at compile-time, providing early feedback to developers. This leads to increased reliability and reduces the likelihood of runtime errors.
4. **Examples of Strongly Typed Languages:** Java, C#, TypeScript

JavaScript Code (Does Work ✅)

```
function main() {
  let number = 10;
  number = "text";  // Valid: JavaScript allows dynamic typin
g
  return number;
}
```

Explanation:

- JavaScript is a dynamically-typed language, allowing variables to change types during runtime.
- In the provided JavaScript code, `number` is initially assigned the value `10` (a number), and later, it is assigned the value `"text"` (a string).
- JavaScript allows this flexibility, and the code executes without type-related errors.

Considerations:

- Statically-typed languages like C++ provide early error detection during compilation, ensuring type consistency.
- Dynamically-typed languages like JavaScript offer flexibility but may require careful handling to avoid unexpected runtime errors.

> The choice between strongly typed and loosely typed languages depends on project requirements, developer preferences, and the balance between early error detection and flexibility during development. Each type has its

> advantages and considerations, influencing their suitability for specific use cases.

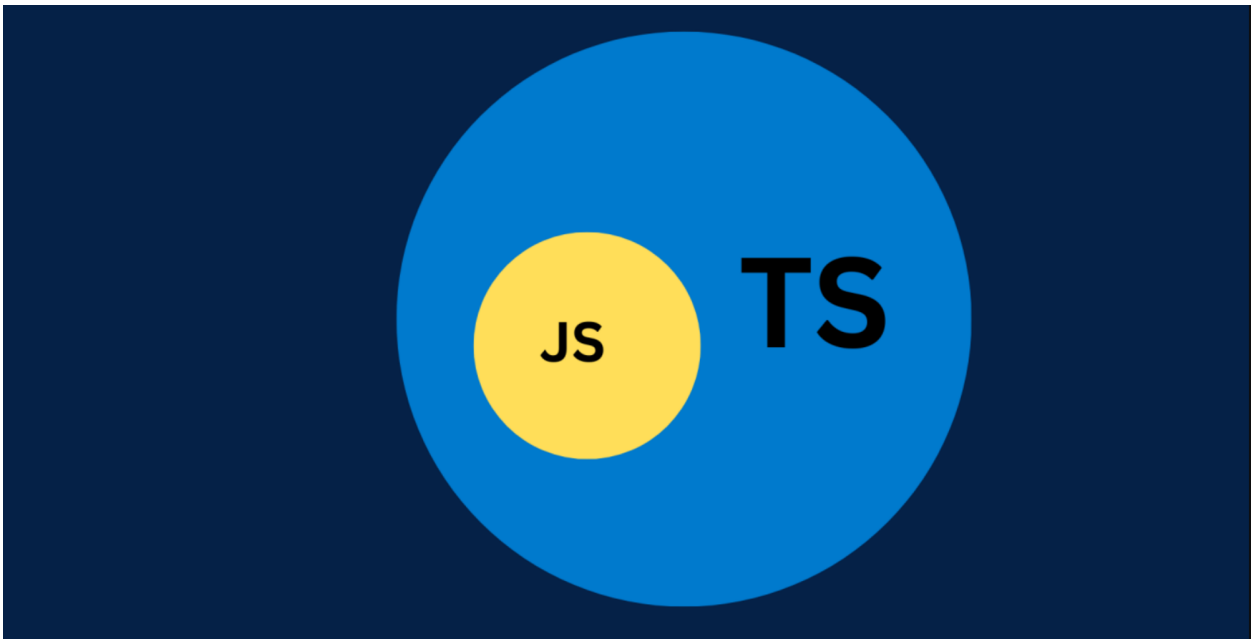
Typescript

Why Typescript

JavaScript is a powerful and widely used programming language, but it has a dynamic typing system, which means variable types are determined at runtime. While dynamic typing provides flexibility, it can lead to runtime errors that are challenging to catch during development.

What Typescript

In response to these challenges, Microsoft introduced TypeScript, a superset of JavaScript that adds static typing to the language. TypeScript is designed to address some of the limitations of JavaScript by providing developers with a more robust type system.



How Typescript

1. **Static Typing:**

- TypeScript introduces static typing, allowing developers to declare the types of variables, parameters, and return values at compile-time.
- Static typing helps catch potential errors during development, offering a level of code safety that may not be achievable in pure JavaScript.

2. **Compatibility with JavaScript:**
    - TypeScript is a superset of JavaScript, meaning that any valid JavaScript code is also valid TypeScript code.
    - Developers can gradually adopt TypeScript in existing JavaScript projects without the need for a full rewrite.
3. **Tooling Support:**
    - TypeScript comes with a rich set of tools and features for development, including code editors (like Visual Studio Code) with built-in TypeScript support.
    - The TypeScript compiler (tsc) translates TypeScript code into plain JavaScript, allowing it to run in any JavaScript environment.
4. **Enhanced IDE Experience:**
    - IDEs (Integrated Development Environments) that support TypeScript offer improved code navigation, autocompletion, and better refactoring capabilities.
    - TypeScript's type information enhances the overall development experience.
5. **Interfaces and Type Declarations:**
    - TypeScript introduces concepts like interfaces and type declarations, enabling developers to define clear contracts for their code.
    - Interfaces help document the shape of objects, making it easier to understand and maintain the code.
6. **Compilation:**
    - TypeScript code is transpiled to JavaScript during the compilation process, ensuring that the resulting code is compatible with various JavaScript environments and browsers.

> Overall, TypeScript provides developers with the benefits of static typing while preserving the flexibility and features of JavaScript. It has gained popularity in large-scale applications and projects where maintaining code quality and catching errors early are crucial.

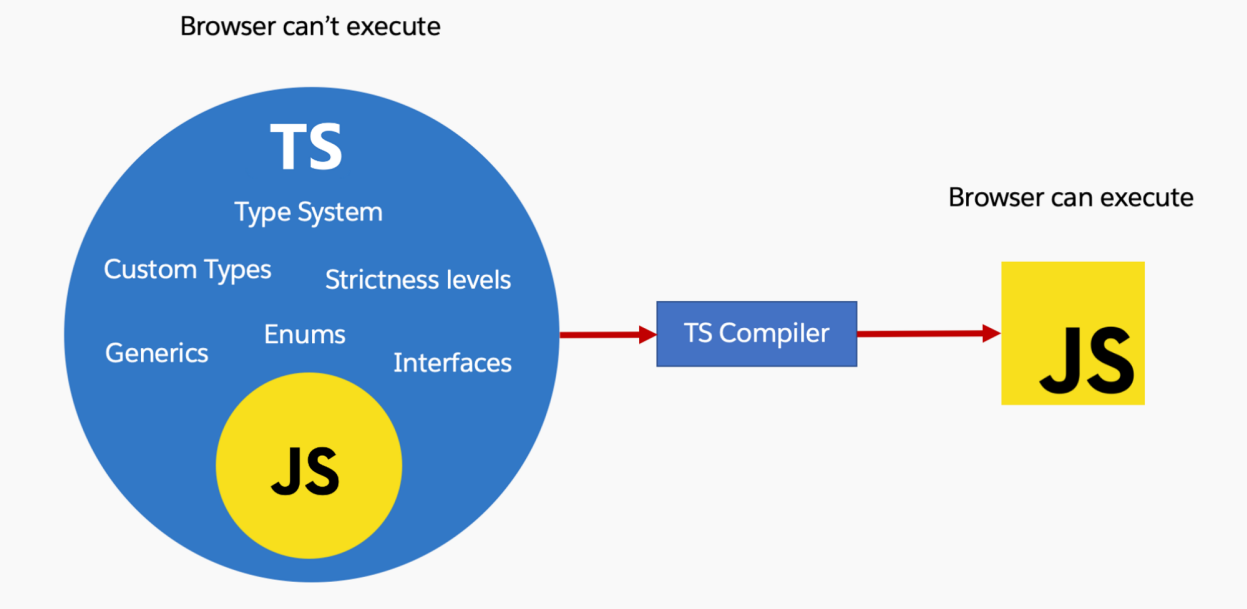
Execution of TypeScript Code

TypeScript code doesn't run natively in browsers or JavaScript environments. Instead, it undergoes a compilation process to generate equivalent JavaScript code. Here's an overview of how TypeScript code is executed:

1. **Writing TypeScript Code:**
    - Developers write TypeScript code using `.ts` or `.tsx` files, employing TypeScript's syntax with features like static typing, interfaces, and type annotations.
2. **TypeScript Compiler (tsc):**
    - The TypeScript Compiler (`tsc`) is a command-line tool that processes TypeScript code.
    - Developers run `tsc` to initiate the compilation process.
3. **Compilation Process:**
    - The TypeScript Compiler parses and analyzes the TypeScript code, checking for syntax errors and type-related issues.
    - It generates equivalent JavaScript code, typically in one or more `.js` or `.jsx` files.
4. **Generated JavaScript Code:**
    - The output JavaScript code closely resembles the original TypeScript code but lacks TypeScript-specific constructs like type annotations.
    - TypeScript features that aren't present in JavaScript (e.g., interfaces) are often transpiled or emitted in a way that doesn't affect runtime behavior.
5. **JavaScript Execution:**

- The generated JavaScript code can now be executed by any JavaScript runtime or browser.
- Developers can include the resulting JavaScript files in HTML documents or use them in Node.js environments.

6. **Runtime Environment:**
   - In the chosen runtime environment, the JavaScript code is interpreted or compiled by the JavaScript engine (e.g., V8 in Chrome, SpiderMonkey in Firefox).
   - Just-in-time (JIT) compilation or interpretation occurs to convert the code into machine code that the computer's processor can execute.
7. **Interacting with the DOM (Browser Environments):**
   - In browser environments, the JavaScript code, generated from TypeScript, may interact with the Document Object Model (DOM) to manipulate web page structure and behavior.



TypeScript Compiler (tsc)

- The TypeScript Compiler (`tsc`) is responsible for transpiling TypeScript code into JavaScript.

- It is a part of the official TypeScript distribution and can be installed using tools like npm.
- Developers run `tsc` from the command line, specifying the TypeScript file(s) they want to compile.
- Configuration for the compilation process can be provided via a `tsconfig.json` file.
- The compiler performs type checking, emits JavaScript files, and allows customization of compilation options.

> In summary, TypeScript code is transformed into JavaScript through the TypeScript Compiler (tsc). This compilation process ensures that TypeScript's features are compatible with existing JavaScript environments, enabling developers to benefit from static typing during development while still producing standard JavaScript for execution.

In addition to the TypeScript Compiler (`tsc`), several alternative tools have gained popularity for their efficiency, speed, and additional features when transpiling TypeScript to JavaScript. Here are a couple of noteworthy ones:

1. esbuild: a highly performant JavaScript bundler and minifier, but it also supports TypeScript.
2. **swc (Speedy Web Compiler): a fast and low-level JavaScript/TypeScript compiler.**

Setting up a Typescript Nodejs Application

Let's walk through the process of setting up a simple TypeScript Node.js application locally on your machine.

Step 1 - Install TypeScript Globally:

```
npm install -g typescript
```

This command installs TypeScript globally on your machine, allowing you to use the `tsc` command anywhere.

Step 2 - Initialize a Node.js Project with TypeScript:

```
mkdir node-app
cd node-app
npm init -y
npx tsc --init
```

These commands create a new directory (`node-app`), initialize a Node.js project with default settings (`npm init -y`), and then generate a `tsconfig.json` file using `npx tsc --init`.

Step 3 - Create a TypeScript File (a.ts):

```
// a.ts
const x: number = 1;
console.log(x);
```

Step 4 - Compile the TypeScript File to JavaScript:

```
tsc -b
```

The `-b` flag tells TypeScript to build the project based on the configuration in `tsconfig.json`. This generates a JavaScript file (`index.js`) from the TypeScript source (`a.ts`).

Step 5 - Explore the Generated JavaScript File (`index.js`):

```
// index.js
const x = 1;
console.log(x);
```

Note that the generated JavaScript file doesn't include TypeScript-specific code. It's standard JavaScript without types.

Step 6 - Attempt to Assign a String to a Number:

```
// a.ts
let x: number = 1;
x = "harkirat";
console.log(x);
```

Step 7 - Try Compiling the Code Again:

```
tsc -b
```

Upon compiling, TypeScript detects the type error (`x` being assigned a string) and reports it in the console. Additionally, no `index.js` file is generated due to the type error.

> This example illustrates one of TypeScript's key benefits: catching type errors at compile time. By providing static typing, TypeScript enhances code reliability and helps identify potential issues before runtime. This is particularly valuable in large codebases where early error detection can save time and prevent bugs.

Basic Types in Typescript

In TypeScript, basic types serve as the building blocks for defining the data types of variables. Here's an overview of some fundamental types provided by TypeScript:

1. **Number:**
    - Represents numeric values.

- Example:

```
let age: number = 25;
```

2. **String:**
   - Represents textual data (sequences of characters).
   - Example:

```
let name: string = "John";
```

3. **Boolean:**
   - Represents true or false values.
   - Example:

```
let isStudent: boolean = true;
```

4. **Null:**
   - Represents the absence of a value.
   - Example:

```
let myVar: null = null;
```

5. **Undefined:**
   - Represents a variable that has been declared but not assigned a value.
   - Example:

```

```

```
let myVar: undefined = undefined;
```

Problems and Code Implementation

1] Hello World Greeting

Objective:

Learn how to give types to function arguments in TypeScript.

Task:
Write a TypeScript function named
`greet` that takes a user's first name as an argument and logs a greeting message to the console.

Function Signature:

```
function greet(firstName: string): void {
    // Implementation goes here
}
```

Solution:

```
function greet(firstName: string): void {
    console.log("Hello " + firstName);
}

// Example Usage
greet("harkirat");
```

Explanation:

1. **Function Definition (`function greet(firstName: string): void`):**
    - The `greet` function is declared with a parameter named `firstName`.
    - `: string` indicates that the `firstName` parameter must be of type string.

- `: void` specifies that the function does not return any value.

2. **Function Body (`console.log("Hello " + firstName);`):**
   - Inside the function body, a `console.log` statement prints a greeting message to the console.
   - The message includes the provided `firstName` parameter.
3. **Function Invocation (`greet("harkirat");`):**
   - The function is called with the argument `"harkirat"`.
   - The provided argument must be a string, aligning with the specified type in the function definition.

> This example demonstrates the basic usage of TypeScript types in function parameters, ensuring that the expected data type is enforced and catching errors related to type mismatches during development.

2] Sum Function

Objective:

Learn how to assign a return type to a function in TypeScript.

Task:

Write a TypeScript function named `sum` that takes two numbers as arguments and returns their sum. Additionally, invoke the function with an example.

Function Signature:

```
function sum(a: number, b: number): number {
    // Implementation goes here
}
```

Solution:

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

// Example Usage
console.log(sum(2, 3));
```

Explanation:

1. **Function Definition (`function sum(a: number, b: number): number`):**
    - The `sum` function is declared with two parameters, `a` and `b`, both of type number.
    - `: number` indicates that the function returns a value of type number.
2. **Function Body (`return a + b;`):**
    - Inside the function body, the sum of `a` and `b` is calculated using the `+` operator.
    - The result is then returned.
3. **Function Invocation (`console.log(sum(2, 3));`):**
    - The function is called with the arguments `2` and `3`.
    - The result is logged to the console using `console.log`.

> This example showcases how to specify the return type of a function in TypeScript, ensuring that the function returns the expected data type. In this case, the sum function returns a number.

3] Age Verification Function

Objective:

Understand Type Inference in TypeScript.

Task:
Write a TypeScript function named
`isLegal` that takes an `age` as a parameter and returns `true` if the user is 18 or older, and `false` otherwise. Also, invoke the function with an example.

Function Signature:

```
function isLegal(age: number): boolean {
    // Implementation goes here
}
```

Solution:

```
function isLegal(age: number): boolean {
    if (age > 18) {
        return true;
    } else {
        return false;
    }
}

// Example Usage
console.log(isLegal(22)); // Output: true
```

Explanation:

1. **Function Definition (`function isLegal(age: number): boolean`):**
    - The `isLegal` function is declared with a parameter `age` of type number.
    - `: boolean` indicates that the function returns a boolean value.
2. **Function Body (`if (age > 18) {...}`):**
    - Inside the function body, an `if` statement checks if the provided `age` is greater than 18.
    - If true, the function returns `true`; otherwise, it returns `false`.

3. **Function Invocation (`console.log(isLegal(22));`):**
    - The function is called with the argument `22`.
    - The result (`true`) is logged to the console using `console.log`.

> This example demonstrates how TypeScript's type inference can be leveraged. The return type (boolean) is implicitly inferred based on the conditions within the function. The isLegal function is designed to return a boolean value indicating whether the provided age is 18 or older.

4] Delayed Function Execution

Objective:

Learn to work with functions as parameters in TypeScript.

Task:
Write a TypeScript function named `delayedCall` that takes another function (`fn`) as input and executes it after a delay of 1 second. Also, invoke the `delayedCall` function with an example.

Function Signature:

```
function delayedCall(fn: () => void): void {
    // Implementation goes here
}
```

Solution:

```
function delayedCall(fn: () => void): void {
    setTimeout(fn, 1000);
}

// Example Usage
```

```
delayedCall(function() {
    console.log("hi there");
});
```

Explanation:

1. **Function Definition (`function delayedCall(fn: () => void): void`):**
    - The `delayedCall` function is declared with a parameter `fn` of type function that takes no arguments and returns `void`.
    - `: void` indicates that the function doesn't return any value.
2. **Function Body (`setTimeout(fn, 1000);`):**
    - Inside the function body, `setTimeout` is used to delay the execution of the provided function (`fn`) by 1000 milliseconds (1 second).
3. **Function Invocation (`delayedCall(function() {...});`):**
    - The `delayedCall` function is invoked with an anonymous function as an argument.
    - The provided function logs "hi there" to the console after a 1-second delay.

> This example illustrates how TypeScript handles functions as first-class citizens, allowing them to be passed as arguments to other functions. The delayedCall function provides a way to execute a given function after a specified delay.

The tsconfig.json File in TypeScript

The `tsconfig.json` file in TypeScript is a configuration file that provides settings for the TypeScript compiler (`tsc`). It allows you to customize various aspects of the compilation process and define how your TypeScript code should be transpiled into JavaScript.

Below are a bunch of options that you can change to change the compilation process in the `tsconfig.json` file:

1] Target Option in `tsconfig.json`:

The `target` option in a `tsconfig.json` file specifies the ECMAScript target version to which the TypeScript compiler (`tsc`) will compile the TypeScript code. It allows you to define the lowest version of ECMAScript that your code should be compatible with. Here's an explanation and example usage:

- **ES5 (ECMAScript 5):**
  - When `target` is set to `"es5"`, the TypeScript compiler generates code compatible with ECMAScript 5, which is widely supported across browsers.
  - Example:

```
{
  "compilerOptions": {
    "target": "es5",
    // Other options...
  }
}
```

  - TypeScript Code:

```
const greet = (name: string) => `Hello, ${name}!`;
```

  - Output:

```
"use strict";
var greet = function (name) { return "Hello, ".concat(n
ame, "!"); };
```

- **ES2020 (ECMAScript 2020):**

- When `target` is set to `"es2020"`, the TypeScript compiler generates code compatible with ECMAScript 2020, incorporating the latest features.
- Example:

```
{
  "compilerOptions": {
    "target": "es2020",
    // Other options...
  }
}
```

- TypeScript Code:

```
const greet = (name: string) => `Hello, ${name}!`;
```

- Output:

```
"use strict";
const greet = (name) => `Hello, ${name}!`;
```

> By setting the target option, you ensure that the generated JavaScript code adheres to the specified ECMAScript version, allowing you to control the level of compatibility and take advantage of the features available in newer ECMAScript versions.

2] `rootDir:`

- The `rootDir` option in a `tsconfig.json` file specifies the root directory where the TypeScript compiler (`tsc`) should look for `.ts` files.

- It is considered a good practice to set `rootDir` to the source folder (`src`), indicating the starting point for TypeScript file discovery.
- Example:

```
{
  "compilerOptions": {
    "rootDir": "src",
    // Other options...
  }
}
```

3] `outDir`

- The `outDir` option defines the output directory where the TypeScript compiler will place the generated `.js` files.
- It determines the structure of the output directory relative to the `rootDir`.
- Example:

```
{
  "compilerOptions": {
    "outDir": "dist",
    // Other options...
  }
}
```

> If rootDir is set to "src" and outDir is set to "dist", the compiled files will be placed in the dist folder, mirroring the structure of the src folder.

4] `noImplicitAny`

- The `noImplicitAny` option in a `tsconfig.json` file determines whether TypeScript should issue an error when it encounters a variable with an implicit `any` type.
- **Enabled (`"noImplicitAny": true`):**

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    // Other options...
  }
}
```

```
// Compilation Error: Implicit any type
const greet = (name) => `Hello, ${name}!`;
```

- **Disabled (`"noImplicitAny": false`):**

```
{
  "compilerOptions": {
    "noImplicitAny": false,
    // Other options...
  }
}
```

No error will be issued for implicit `any` types.

5] `removeComments`

- The `removeComments` option in a `tsconfig.json` file determines whether comments should be included in the final JavaScript output.
- **Enabled (`"removeComments": true`):**

```
{
  "compilerOptions": {
    "removeComments": true,
```

```
    // Other options...
  }
}
```

Comments will be stripped from the generated JavaScript files.

- **Disabled (`"removeComments": false`):**

```
{
  "compilerOptions": {
    "removeComments": false,
    // Other options...
  }
}
```

Comments will be retained in the generated JavaScript files.

> These options provide flexibility and control over the compilation process, allowing you to structure your project and handle type-related scenarios according to your preferences.

Interfaces

In TypeScript, an interface is a way to define a contract for the shape of an object. It allows you to specify the expected properties, their types, and whether they are optional or required. Interfaces are powerful tools for enforcing a specific structure in your code.

Understanding Interfaces

Suppose you have an object representing a user:

```
const user = {
  firstName: "harkirat",
  lastName: "singh",
```

```
  email: "email@gmail.com",
  age: 21,
};
```

To assign a type to the `user` object using an interface, you can create an interface named `User`:

```
interface User {
  firstName: string;
  lastName: string;
  email: string;
  age: number;
}
```

Now, you can explicitly specify that the `user` object adheres to the `User` interface:

```
const user: User = {
  firstName: "harkirat",
  lastName: "singh",
  email: "email@gmail.com",
  age: 21,
};
```

Explanation:

1. **Interface Declaration (`interface User`):**
   - The `interface` keyword is used to declare an interface named `User`.
   - Inside the interface, you define the expected properties (`firstName`, `lastName`, `email`, `age`) along with their types.
2. **Assigning Type to Object (`const user: User = { /* ... */ };`):**
   - By stating `const user: User`, you are explicitly indicating that the `user` object must adhere to the structure defined by the `User` interface.

- If the `user` object deviates from the defined structure or misses any required property, TypeScript will raise a compilation error.

Assignment 1

Problem: Create a function `isLegal` that returns true or false if a user is above 18. It takes a user as an input.

Solution:

```
// Define an interface to specify the structure of a user obj
ect
interface User {
  firstName: string;
  lastName: string;
  email: string;
  age: number;
}

// Create a function 'isLegal' that checks if a user is above
18
function isLegal(user: User): boolean {
    // Check if the user's age is greater than 18
    if (user.age > 18) {
        return true; // Return true if the user is legal
    } else {
        return false; // Return false if the user is not lega
l
    }
}
```

Code Explanation:

- An interface "User" is defined to enforce the structure of a user object with properties: firstName, lastName, email, and age.

- The function "isLegal" takes a user object as input and checks if the user's age is greater than 18.
- It returns true if the user is legal (age > 18) and false otherwise.

Assignment 2

Problem: Create a React component that takes todos as an input and renders them.

Solution:

```
// Define an interface to specify the structure of a todo obj
ect
interface TodoType {
  title: string;
  description: string;
  done: boolean;
}

// Define the input prop for the Todo component
interface TodoInput {
  todo: TodoType;
}

// Create a React component 'Todo' that takes a 'todo' prop a
nd renders it
function Todo({ todo }: TodoInput): JSX.Element {
  return (
    <div>
      <h1>{todo.title}</h1>
      <h2>{todo.description}</h2>
      {/* Additional rendering logic can be added for other p
roperties */}
    </div>
```

```
  );
}
```

Code Explanation:

- An interface "TodoType" is defined to specify the structure of a todo with properties: title, description, and done.
- An interface "TodoInput" is defined to specify the input prop for the Todo component.
- The React component "Todo" takes a prop "todo" of type "TodoType" and renders its properties (title and description).

Implementing Interfaces

In TypeScript, you can implement interfaces using classes. This provides a way to define a blueprint for the structure and behavior of a class. Let's take an example:

Assume you have a `Person` interface:

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

Now, you can create a class that adheres to this interface:

```
class Employee implements Person {
    name: string;
    age: number;

    constructor(n: string, a: number) {
        this.name = n;
        this.age = a;
    }
```

```
    greet(phrase: string) {
        console.log(`${phrase} ${this.name}`);
    }
}
```

Here's what's happening:

- The `Employee` class implements the `Person` interface.
- It has properties (`name` and `age`) matching the structure defined in the interface.
- The `greet` method is implemented as required by the interface.

> This approach is handy when creating various types of persons (like Manager, CEO), ensuring they all adhere to the same interface contract. It maintains consistency in the structure and behavior across different classes.

Types

In TypeScript, **types** allow you to aggregate data together in a manner very similar to interfaces. They provide a way to define the structure of an object, similar to how interfaces do. Here's an example:

```
type User = {
    firstName: string;
    lastName: string;
    age: number;
};
```

Features

1. **Unions:**
   Unions allow you to define a type that can be one of several types. This is

useful when dealing with values that could have different types. For instance, imagine you want to print the ID of a user, which can be either a number or a string:

```
type StringOrNumber = string | number;

function printId(id: StringOrNumber) {
  console.log(`ID: ${id}`);
}

printId(101);     // ID: 101
printId("202");   // ID: 202
```

Unions provide flexibility in handling different types within a single type definition.

2. **Intersection:**
   Intersections allow you to create a type that has every property of multiple types or interfaces. If you have types like `Employee` and `Manager`, and you want to create a `TeamLead` type that combines properties of both:

```
type Employee = {
  name: string;
  startDate: Date;
};

type Manager = {
  name: string;
  department: string;
};

type TeamLead = Employee & Manager;
```

```
const teamLead: TeamLead = {
  name: "harkirat",
  startDate: new Date(),
  department: "Software Developer"
};
```

Intersections provide a way to create a new type that inherits properties from multiple existing types.

> In summary, while types and interfaces are similar in defining object structures, types in TypeScript offer additional features like unions and intersections, making them more versatile in certain scenarios.

Interfaces vs Types

Major Differences

1. Declaration Syntax:

- **Type:**
  - Uses the `type` keyword.
  - More flexible syntax, can represent primitive types, unions, intersections, and more.
- **Interface:**
  - Uses the `interface` keyword.
  - Typically used for defining the structure of objects.

2. Extension and Merging:

- **Type:**
  - Supports extending types.

    - Can't be merged; if you define another type with the same name, it will override the previous one.
- **Interface:**
    - Supports extending interfaces using the `extends` keyword.
    - Automatically merges with the same-name interfaces, combining their declarations.

3. Declaration vs. Implementation:

- **Type:**
    - Can represent any type, including primitives, unions, intersections, etc.
    - Suitable for describing the shape of data.
- **Interface:**
    - Mainly used for describing the shape of objects.
    - Can also be used to define contracts for classes.

Other Differences

- **Type Overriding:**
    - Types cannot be overridden or merged. Redefining a type with the same name replaces the previous one.
    - Interfaces automatically merge if declared with the same name.
- **Object Literal Strictness:**
    - Types are more lenient when dealing with object literal assignments.
    - Interfaces enforce strict object literal shapes.
- **Implementation for Classes:**
    - Interfaces can be used to define contracts for class implementations.
    - Types are more versatile for creating complex types and reusable utility types.

When to Use Which

- **Use Types:**
  - For advanced scenarios requiring union types, intersections, or mapped types.
  - When dealing with primitive types, tuples, or non-object-related types.
  - Creating utility types using advanced features like conditional types.
- **Use Interfaces:**
  - When defining the structure of objects or contracts for class implementations.
  - Extending or implementing other interfaces.
  - When consistency in object shape is a priority.

Examples

Type Example:

```
type StringOrNumber = string | number;

function printId(id: StringOrNumber) {
  console.log(`ID: ${id}`);
}

printId(101);       // ID: 101
printId("202");     // ID: 202
```

Interface Example:

```
interface Employee {
  name: string;
  startDate: Date;
}

interface Manager {
```

```
  name: string;
  department: string;
}

type TeamLead = Employee & Manager;

const teamLead: TeamLead = {
  name: "Harkirat",
  startDate: new Date(),
  department: "Software Developer",
};
```

> In summary, choose types for flexibility and advanced type features, and use interfaces for object shapes, contracts, and class implementations, ensuring a consistent and readable codebase.