TypeScript Kursverwaltungssystem Übung

Ziel

Das Ziel dieser Übung ist es, ein einfaches Kursverwaltungssystem für eine Universität mit TypeScript zu implementieren. Aufbauend auf den Konzepten der letzten Übung, vertiefen wir nun fortgeschrittene TypeScript-Features. Der Fokus liegt auf der korrekten Modellierung von Datenstrukturen durch **Type Aliases**, **Literal Types** und **Discriminated Unions**. Des Weiteren üben wir den Einsatz von **Klassen** (inkl. Vererbung, Access Modifiers, `readonly` und Parameter Properties) und wie diese mit **Interfaces** interagieren. Schließlich führen wir **Generics** ein, um wiederverwendbare und typsichere Funktionen und Klassen zu erstellen.

Aufgaben

1. Grundlegende Typen und Interfaces

Definieren Sie zunächst die Basis-Typen, die wir im System benötigen.

- Erstellen Sie einen Type Alias `UserID` für eine `string` oder `number`.
- Erstellen Sie einen Literal Type `StudentStatus` für die Werte `"Aktiv"`, `"Beurlaubt"` oder `"Exmatrikuliert"`.
- Definieren Sie ein `Person`-Interface mit `firstName`, `lastName` und einem optionalen Feld `middleName?`. Wählen Sie passende Typen für die Felder.

```
// Typ-Aliase und Literal-Typen
type UserID = /* ... */ ;
type StudentStatus = /* ... */ ;

// Interface mit optionalem Feld
interface Person {
    /* ... */
}
```

2. Klassen, Vererbung und Modifier

Implementieren Sie eine `BaseUser`-Klasse, die als Grundlage für alle Benutzer*innen im System dient.

- Verwenden Sie **Parameter Properties** im Konstruktor, um die Eigenschaften zu initialisieren.
- Die `id` soll `public` und `readonly` sein (Typ `UserID`).
- Die `email` soll `private` sein (Typ `string`).
- Die Klasse soll das `Person`-Interface implementieren (`implements Person`). Der einfachste Weg, die erforderlichen Interface-Eigenschaften (`firstName`, `lastName`, `middleName`) zu implementieren, ist ebenfalls die Verwendung von `public` Parameter Properties direkt im Konstruktor.
- Fügen Sie einen `public` **Getter** `get fullName(): string` hinzu, der den vollen Namen (inkl. `middleName`, falls vorhanden) zurückgibt.

```
class BaseUser implements Person {
    /* ... */
}
```

Erstellen Sie eine `Student`-Klasse, die von `BaseUser` **erbt** (`extends`).

- `Student` soll zusätzlich eine `matrikelNr` (Typ `string`) und den `status` (Typ `StudentStatus`) besitzen.
- Implementieren Sie den Konstruktor, rufen Sie super() korrekt auf und initialisieren Sie die neuen Eigenschaften (gerne auch hier **Parameter Properties** verwenden).

```
class Student extends BaseUser {
/* ... */
}
```

3. Discriminated Unions und Narrowing

Nicht alle Kurse sind gleich. Wir wollen zwischen Vorlesungen und Seminaren unterscheiden. Wählen Sie sinnvolle Datentypen, wo keine explizit angegeben sind.

- Erstellen Sie ein Interface `Lecture` mit `kind: "lecture"`, `title: string` und `credits: number`.
- Erstellen Sie ein Interface `Seminar` mit `kind: "seminar"`, `title` und `topic`.
- Erstellen Sie einen **Discriminated Union** Type Alias `Course` (entweder `Lecture` oder `Seminar`).
- Implementieren Sie eine Funktion `printCourseDetails`, die einen Course entgegennimmt und je nach Typ unterschiedliche Informationen ausgibt.
    - Hinweis: Hier müssen Sie **Type Narrowing** anwenden.

```
interface Lecture {
    /* ... */
}

interface Seminar {
    /* ... */
}

type Course = /* ... */;

function printCourseDetails(course: Course): void {
    console.log(`Titel: ${course.title}`);
    // Type Narrowing mit einer if-else, basierend auf der Eigenschaft 'kind'
    /* ... */
}
```

4. Generische Funktion mit Constraints

Wir benötigen eine Hilfsfunktion, um ein beliebiges Objekt anhand seiner `id` aus einem Array zu finden.

- Implementieren Sie eine **generische Funktion** `findItemById`.
- Die Funktion soll einen Typparameter T haben.

- Sie soll einen **Generic Constraint** verwenden, um sicherzustellen, dass `T` ein Objekt mit einer `id`-Eigenschaft vom Typ `UserID` ist
  - Hinweis: Definieren Sie ein passendes Interface HasId (siehe unten)
  - Verwenden Sie dann `T extends HasId`
- Die Funktion erhält ein Array vom Typ `T[]` und eine `id` (Typ `UserID`) und gibt `T | undefined` zurück.

```
interface HasId {
    id: UserID
}
```

Der offensichtliche Vorteil dieser Funktion ist, dass sie durch **Generics** (d.h., den Typparameter) mit verschiedenen Typen (Studierende, Kursen, Professor*innen) verwendet werden kann, solange diese eine id-Eigenschaft besitzen.

Der folgende Testlauf zeigt die Verwendung Ihrer Funktion mit einem Array von `Student`-Objekten.

```
// Test:
const testStudents = [
    new Student("s1", "a@b.c", "Max", "Mustermann", "12345", "Aktiv"),
    new Student("s2", "d@e.f", "Erika", "Musterfrau", "67890", "Aktiv")
];
const foundStudent = findItemById(testStudents, "s2");
console.log(foundStudent?.fullName); // Sollte "Erika Musterfrau" ausgeben
```

5. Generische Klasse

Um verschiedene Datentypen (Studierende, Kurse, Professor*innen) verwalten zu können, erstellen wir ein generisches Repository.

- Erstellen Sie eine generische Klasse `DataRepository<T>`.
- Die Klasse soll einen Typparameter `T` haben.
- Sie soll ein `private` Array `items: T[]` verwalten.
- Implementieren Sie die Methoden `addItem(item: T): void` und `getAll(): T[]`.

Der folgende Testlauf zeigt die Verwendung Ihrer generischen Klasse mit `Student`- und `Course`-Objekten.

```
// Test:
const studentRepo = new DataRepository<Student>();
studentRepo.addItem(new Student("s3", "g@h.i", "Peter", "Pan", "11111", "Beurlaubt"));

const courseRepo = new DataRepository<Course>();
courseRepo.addItem({ kind: "lecture", title: "Web Frontend", credits: 5 });

console.log(studentRepo.getAll());
console.log(courseRepo.getAll());
```

6. Integration und Anwendung (Der Praxistest)

In dieser letzten Aufgabe soll alles zusammengeführt werden. Sie simulieren den "Start" Ihrer Anwendung, indem Sie alle bisher erstellten Bausteine verwenden und deren Zusammenspiel testen. Beachten Sie dabei folgende Anforderungen:

- **Repositorys instanziieren**
  - Erstellen Sie eine Instanz von `DataRepository` für den Typ `Student`.
  - Erstellen Sie eine Instanz von `DataRepository` für den Typ `Course`.
- **Daten erstellen und hinzufügen**
  - Erstellen Sie drei Instanzen der `Student`-Klasse. Verwenden Sie unterschiedliche Daten (z.B. eine Person mit `middleName`, eine ohne; unterschiedliche `StudentStatus`-Werte; `UserID` mal als `string`, mal als `number`).
  - Fügen Sie alle drei Studierenden zum entsprechenden `DataRepository` hinzu.
  - Erstellen Sie zwei `Course`-Objekte: eine `Lecture` und ein `Seminar`.
  - Fügen Sie beide Kurse zum entsprechenden `DataRepository` hinzu.
- **Klassen-Methoden und Getter testen**
  - Rufen Sie die `getAll()`-Methode Ihres Studierenden-Repositories auf, um ein Array aller Studierenden zu erhalten.
  - Iterieren Sie (z.B. mit `forEach`) über dieses Array und geben Sie für jede*n Studierende*n den `fullName` (über den Getter) auf der Konsole aus.
- **Narrowing-Funktion testen**
  - Rufen Sie die `getAll()`-Methode Ihres Kurs-Repositories auf.
  - Iterieren Sie über das Kurs-Array und rufen Sie für jeden Kurs die Funktion `printCourseDetails` auf. Überprüfen Sie, ob je nach `kind` die korrekten Details (Credits oder Topic) ausgegeben werden.
- **Generische Suchfunktion testen**
  - Verwenden Sie die `findItemById`-Funktion, um nach einem Ihrer Studierenden zu suchen. Übergeben Sie das Studierenden-Array (das Sie mittels `getAll()` vom Studierenden-Repository erhalten haben) und eine **existierende** `id`. Geben Sie den `fullName` der*des gefundenen Studierenden aus.
  - Rufen Sie `findItemById` erneut auf, aber mit einer **nicht-existierenden** `id` (z.B. "s99" oder 999). Geben Sie das Ergebnis (das `undefined` sein sollte) auf der Konsole aus, um zu prüfen, dass auch der "Nicht-gefunden"-Fall funktioniert.

Abgabe

Reichen Sie Ihre Hausübungen als ZIP-Datei mit allen erforderlichen Dateien (in diesem Fall vermutlich ausschließlich TypeScript) ein. Geben Sie bei Bedarf zusätzlichen Dokumentationen oder Kommentare an, um Ihren **Code zu erläutern**.

Bewertung

Die Übung wird anhand der Vollständigkeit der Implementierung, korrekten Verwendung von TypeScript-Funktionen (insb. Typ-System, Klassen, Generics), logischen Struktur des Codes und Lesbarkeit bewertet.