Lab 4: Classes, Interfaces and Enums

Object Oriented programming (OOP) is a programming paradigm that relies on the concept of classes and objects. Typecript developers can build their applications using this object-oriented class-based approach.

In this lab, we'll learn how to use OOP concepts within a Typesript project.

Working with Classes

First, we'll create a class and define its fields and methods:

1. Open the starter project with VSCode

2. Go to `src/models.ts` and transform the `Task` interface into a class

   ```
   export  class Task {

   }
   ```

3. Add a `constructor` to the `Task` class

   ```
       constructor(id: number, title: string, description: string, date: Date,
                               assignedTo: Person, data?: any ) {

           this.id = id;
           this.title = title;
           this.description = description;
           this.date = date;
           this.assignedTo = assignedTo;
           this.data = data

       }
   ```

4. Make the `id` property `private` and rename it to `_id`

   ```
   private _id: number;
   ```

5. Add a getter method `id` for the property `_id`

```
get id() {
    return  this._id
}
```

6. Add a `public` method `update` that takes all the properties of class as parameters except the `id` (`id` is not editable)

```
public update(title: string, description: string, date: Date,
                                assignedTo: Person, data?: any ) {
    //TODO Implement
}
```

7. Implement the `update` method

8. Go to data.ts file and replace all object instances by class instances, example

```
let task: Task = new Task(
    1,
    "Fix bugs",
    "Fix all issues in the code, and pass tests",
    new Date(),
    me as Person //or <Person> me
)
```

9. Edit the `updateTask` function so it uses the method `update` of the class `Task`

```
tasks[i].update(title, description, date, assignedTo, data)
```

10. Run `tsc` and check for errors

Implementing Interfaces

Another core concept of OOP, is interfaces which are a powerful way of defining contracts within your code as well as contracts with code outside of your project:

1. Open `models.ts` file and delete the `Tasks` type, `addTask` , `updateTask` and `deleteTask` functions and the `tasks` variable

2. Open `data.ts` file and create a new interface called `Crud`

3. Add three method signatures : `add`, `update` and `delete`

```
interface Crud {
    add(id: number, title: string, description: string, date: Date, assignedTo: Person ,
    update(id: number, title: string, description: string, date: Date, assignedTo: Person
    delete(id: number): boolean;
}
```

4. In the same file create a new class `TaskManager` that `implements` `Crud` interface

```
class TaskManager implements Crud {

    add(id: number, title: string, description: string, date: Date, assignedTo: Person, d
        throw  new Error("Method not implemented.");
    }

    update(id: number, title: string, description: string, date: Date, assignedTo: Person
        throw  new Error("Method not implemented.");
    }

    delete(id: number): boolean {
        throw  new Error("Method not implemented.");
    }

}
```

5. Add a `private` property items of type `Array<Task>`, initialize to empty array

```
private _tasks: Task[] = []
```

6. Add a getter method for tasks

7. Implement `add`, `update` and `delete` methods

8. Update the dafault export of the module (file)

```
export  default TaskManager;
```

9. Create an instance of `TaskManager` in `main.ts` file and use it

```
import TaskManager from  "@data"
```

```
let manager = new TaskManager()
```

10. Add some tasks by calling `add(...)`
11. Run `tsc` to type-check
12. Run the output script

```
node dist/main.js
```

Declaring an enum

In this section, we'll group a set of values into an eumeration and use it as a type:

1. Go to `models.ts` and create a en enum `Priority`

```
export  enum Priority {

}
```

2. Add three members `Low`, `Normal`, `High`, use string value for each member

```
export  enum Priority {
    Low = "low",
    Normal = "normal",
    High = "normal"
}
```

3. Add a new property `priority` in the `Task` class, initialize it to `Normal` by default

```
priority: Priority = Priority.Normal
```

Class inheritance

One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance:

1. In the same file create a classe `ImportantTask` which **extends** the Task class

```
export  class ImportantTask extends Task {

}
```

2. Add a `constructor` add call the `super` constructor before setting the `priority` property to `High` value

```
constructor(id: number, title: string, description: string, date: Date, assignedTo: Perso
    super(id, title, description, date, assignedTo, data )
    this.priority = Priority.High
}
```

Generics

One of the main tools for creating reusable components is generics, that is, being able to create a component that can work over a variety of types rather than a single one.

`TaskManager` class now show manage both classes `Task` and `ImportantTask` , to do so we can use generics.

1. Go to `data.ts` file and add a type argument for the `Crud` interface

```
interface Crud<T> {
...
}
```

2. Update the signature of the methods

```
add(value: T): T;
update(value: T): T;
```

3. Fix the class definition of `TaskManager` so it became also generic,

```
class TaskManager<T> implements Crud<T> {
...
}
```

4. Add generic types **constraints** to the TaskManager class so it accept only the subclasses of `Task` base class

```
class TaskManager<T extends Task> implements Crud<T> {
...
}
```

5. Refactor the implementation of the `add`, `update` and `delete` methods

6. Go to `main.ts` and test with dummy data

7. Run `tsc` to type-check

8. Run the output script

```
node dist/main.js
```