TypeScript Exercises

Set up VS-code and a project for the TypeScript Exercise

If not already done, set up Visual Studio Code for TypeScript with Node as explained in this link:
https://code.visualstudio.com/docs/languages/typescript

a) Create a new empty project (folder) and add package.json to the project.

b) Add a tsconfig.json file (do `tsc --init` in the root of the project) with this content:

```
{
   "compilerOptions": {
       "target": "es6",
       "strict": true,
       "module": "commonjs",
       "outDir": "build",
       "sourceMap": true
   },
  "include": ["src/**/*"]
}
```

c) Create a folder **src** and in this folder a new typescript file (*.ts) and:

1. Verify that you can use Node modules by requiring one of nodes built-in modules, for example:
   `let http = require("http");`
2. Verify that you can use external node-modules, for example by using node-fetch:
   `npm install --save @types/node-fetch`

Interfaces 1

a) Create a TypeScript interface `IBook`, which should encapsulate information about a book, including:

- title, author: all strings
- published: Date
- pages: number

b) Create a function that takes an `IBook` instance and test it with an object instance.

c) Given the example above, explain what is meant by the term Duck Typing, when TypeScript interfaces are discussed.

d) Change the interface to make `published` and `pages` become optional - Verify the new behaviour.

e) Change the interface to make `author` readonly - Verify the new behaviour.

f) Create a class `Book` and demonstrate the "Java way" of implementing an interface.

Interfaces 2 (Function types)

a) Create an interface to describe a function: `myFunc` that should take three string parameters and return a String Array.

b) Design a function "implementing" this interface which returns an array with the three strings

c) Design another implementation that returns an array, with the three strings uppercased.

d) The function, given below, uses the ES-6 (and TypeScript) feature for destructuring Arrays into individual variables, to simulate a method that uses the interface.

```
let f2 = function logger(f1: myFunc){
    //Simulate that we get data from somewhere and uses the provided function
    let [ a, b, c] = ["A", "B", "C"];
    console.log(f1(a,b,c));
}
```

e) Test `f2` with the two implementations created in b+c.

f) Verify that `f2` cannot be used with functions that do not obey the `myFunc` interface

Promises

Forgive me for, in these me-too times, for using the Chuck Norris API again :-) but I have never encountered a situation where it was down. By the way, did you know that Chuck Norris was a programmer? - I got this while preparing for this exercise "Chuck Norris doesn't pair program" ;-)

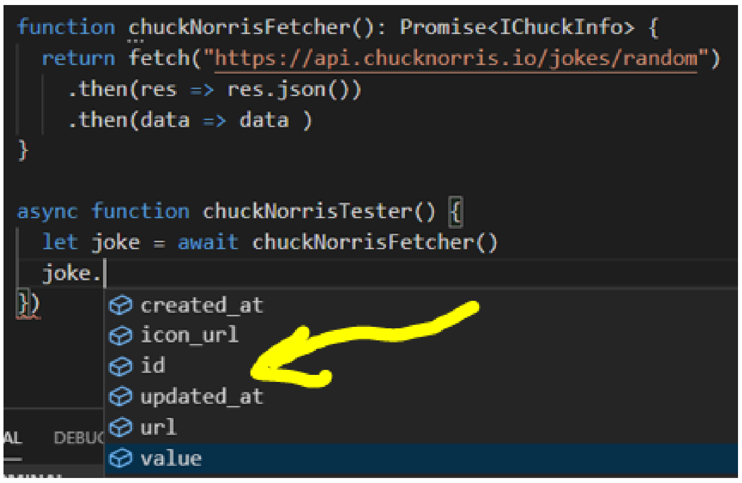
install this before you start (make sure you understand the highlighted part):

`npm install node-fetch and npm install @types/node-fetch --save-dev`

Create a file `chuckNorrisFetcher.ts` and add the following content

```
function chuckNorrisFetcher(){
  return fetch("https://api.chucknorris.io/jokes/random")
    .then(res => res.json())
    .then(data => data )
}
async function chuckNorrisTester() {
  let joke = await chuckNorrisFetcher()
  console.log(joke.value)
}
```

Without changing ANYTHING in the highlighted part, add the necessary changes to get code-completion hints as indicated in this figure.



```
function chuckNorrisFetcher(): Promise<IChuckInfo> {
  return fetch("https://api.chucknorris.io/jokes/random")
    .then(res => res.json())
    .then(data => data )
}

async function chuckNorrisTester() {
  let joke = await chuckNorrisFetcher()
  joke.
})
```

Generics

a) Implement a generic function which will take an array of any kind, and return the array reversed (just use the built-in reverse function), so the three first calls below will print the reversed array, and the last call will fail.

```
console.log(reverseArr<string>(["a","b","c"]));
console.log(reverseArr<number>([1,2,3]));
console.log(reverseArr<boolean>([true,true,false]));
console.log(reverseArr<number>(["a","b","c"]));
```

b) Implement a generic Class `DataHolder` that will allow us to create instances as sketched below:

```
let d = new DataHolder<string>("Hello");
console.log(d.getValue());
d.setValue("World");
console.log(d.getValue());

let d2 = new DataHolder<number>(123);
console.log(d2.getValue());
d2.setValue(500);
console.log(d2.getValue());
```

Verify that once created, an instance can only be used with the type it was created from.

c) Rewrite the example above to user getters and setters instead of the silly getXX and setXX methods

Classes and Inheritance

ES2015 and ES-next have added classes and inheritance to JavaScript somehow (but ONLY somehow) similar to what we know from Java.

Alternative to this exercise: If you come to like the functional style of programming, you could implement this example using Type Aliases as explained by Hejlsberg in his video (43min)

TypeScript, however, adds a lot of extras to this topic, which this exercise should demonstrate. Make sure to observe the following:

- A top-level interface IShape, to define the Shape class.
- The constructor shorthand to automatically create properties
- All of the Access Modifiers `public, private, protected` (and perhaps also `readonly`)
- `Abstract`
- `Static` (make a counter that counts the total number of instances)

A) The declaration below defines a Shape class, which as it's only properties has a `color` field + a `getArea()` and a `getPerimeter()` function which both returns undefined. This is the closest we get to an abstract method in Java.

```
abstract class Shape {
  private _color:string;
  constructor(color:string){
      this._color = color;
  }
  abstract get area():number;
  abstract get perimeter(): number;
}
```

Provide the class with a nice (using template literals) `toString()` method + a getter/setter for the colour property. Verify that you cannot (why) make an instance of this class.

B) Create a new class Circle that should extend the Shape class.

Provide the class with:

- A radius field
- A constructor that takes both colour and radius.
- Overwritten versions of the methods defined in the Base
- Getter/Setter for radius

Test the class constructor, the getters/setters and the three methods.

C) Create a new class Cylinder (agreed, definitely not a perfect inheritance example) that should extend the Circle class.

Provide the class with:

- A height field
- A constructor that takes colour, radius and height.
- Overwritten versions of relevant methods defined in the Base (getter for `perimeter` should throw "not implemented")
- A `getVolume()` method (or better, a getter called volume)
- Getter/Setter for height

Test the new class