# TypeScript Exercises


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

Create a new blank project, and in this, create 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`

**2) Execute and play with all examples in this tutorial:**

http://tutorialzine.com/2016/07/learn-typescript-in-30-minutes/

**3) 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 explained.

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.

**4) 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 cool 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 does not obey the `myFunc` interface

### 5) Classes and Inheritance (Skip this is you have a "time issue")

The exercise given below is the exact same exercise as you were given with the es2015 exercises.

TypeScript however, adds a great deal of extras to this topic, so do the exercise one more time, and this time make sure to include:

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

```
class Shape {
  constructor(color){
    this._color = color;
  }
  getArea() {
    return undefined;
  }
  getPerimeter() {
    return undefined;
  }
}
```

Provide the class with a nice (using template literals) `toString()` method + a getter/setter for the colour property. Test the class constructor, the getter/setter and the two methods.

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 three 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, 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 the three methods defined in the Base (`getPerimeter()` should return undefined)
- A `getVolume()` method
- Getter/Setter for height

Test the new class

D) The getX() methods (getArea(), getPerimeter() and getVolume()) are all candidates for a getter.

Rewrite the three methods to use the getter syntax; that is `console.log(circle.radius)` instead of `console.log(circle.getRadius())`

## 6) Generics

a)

Implement a generic function, which when called like this: `printType<string>("Hello")` will print `"String"` (and similar for other types);

Hint: From an object's constructor function, you can its name property (`instance.constructor.name`)

b)

Implement a generic function which when called like this:

```
printTypes<number,string,Date>(1,"a",new Date())
```

Will return this `['Number', 'String', 'Date']` (and similar for other types);

Hint: Let your generic method return an array of *any* (`Array<any>`)

c)

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"]));
```

d)

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.

e)

Using this interface:

```
interface Owner {
  owner: String;
}
```

Create a method `printOwner(..)` which will only accept arguments with an owner property, and print this value.

Hint: Use the section "Generic Constraints" in http://www.typescriptlang.org/docs/handbook/generics.html to see how to do this.