Lab 1 - Introduction to TypeScript

TypeScript is a superset of JavaScript ES6. Its main contribution is to make ES6 strongly typed. The compiler can catch many type related problems. Strong typing also allows an editor to offer code completion, refactoring and code navigation.

In this lab we will get introduced to the type system of TypeScript.

Part 1 - Variable Types

__1. Within the **C:\LabWork\hello\src\app** folder create a file called **play.ts**. Make sure the file does not have an extension like '.txt' automatically added to it.

__2. In this file add these lines to define and print a numerical variable.

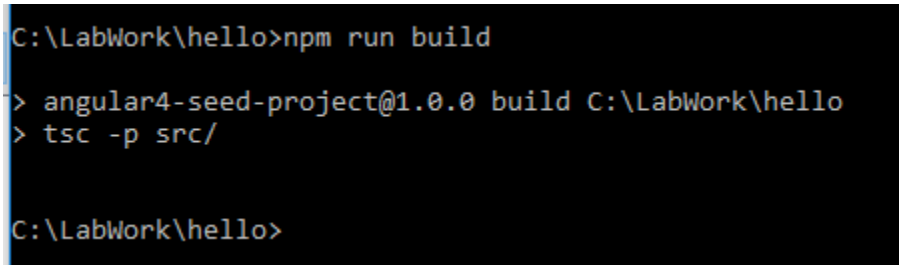
```
var age: number = 20

console.log(age)
```

This ":number" bit comes from TypeScript. The rest is basic JavaScript.

__3. Save changes.

__4. From the command prompt go to the **C:\LabWork\hello** folder. Then run this command to compile the code.

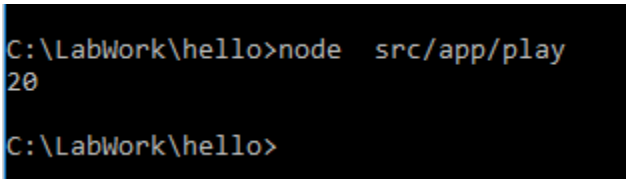
```
npm  run  build
```



__5. Make sure that there are no errors.

__6. Then enter this command to run the code.

```
node  src/app/play
```



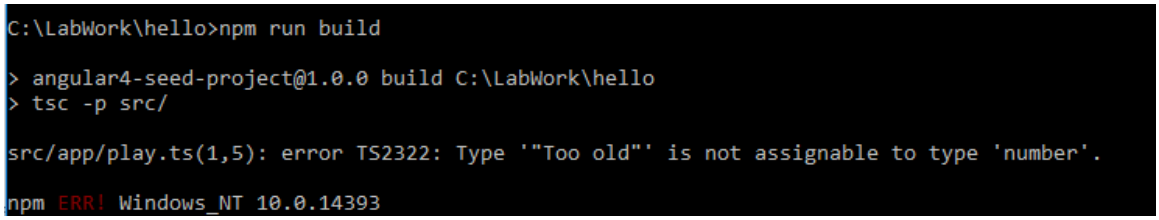
You should see the '20' output to the prompt.

__7. Now we will deliberately introduce a type error. In play.ts initialize the age variable like this.

```
var age: number = "Too old"
```

__8. Save changes.

__9. Compile the code again ('npm run build'). This time there will be an error like this.



This is a simple illustration of type safety.

Part 2 - Function Argument Types

__1. Delete all lines in **play.ts**.

__2. Add a function like this.

```
function printPerson(name:string, age:number) {
  console.log(`Name: ${name} age: ${age}`)
}

printPerson("Billy", 8)
```

Watch out for the string with back ticks. It is a template string where you can insert variables using ${variable} syntax.

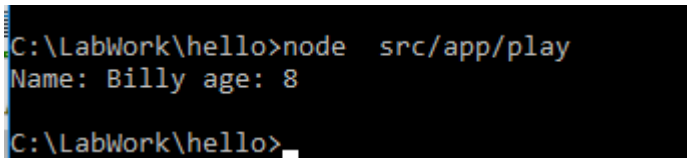
__3. Save changes.

__4. Compile the code.

```
npm run build
```

__5. Run it.

```
node src/app/play
```



__6. Try calling the function with invalid data type like this and save.

```
printPerson(8, "Billy")
```

__7. Verify that this does not compile.

Part 3 - Class and Inheritance

We will learn about class and inheritance in this section. We will model various products sold by a travel company like tours, shows and dining.

__1. Delete all lines in **play.ts**.

__2. Add a class like this.

```
class Product {
  title: string;
  price: number;
  id: number;

  constructor(id: number) {
    this.id = id
  }

  printDetails() {
    console.log(`Title: ${this.title}`)
    console.log(`ID: ${this.id}`)
    console.log(`Price: ${this.price}`)
  }
}
```

__3. Add another class that inherits from Product.

```
class Tour extends Product {
  duration: string;

  constructor(id: number, duration: string) {
    super(id);

    this.duration = duration
  }

  printDetails() {
    super.printDetails()

    console.log(`Duration: ${this.duration}`)
  }
```

```
}
```

__4. Write a function that takes a Product as argument.

```
function test(p: Product) {
  p.printDetails()
}
```

__5. Next, create an instance of Tour and call the 'test' method.

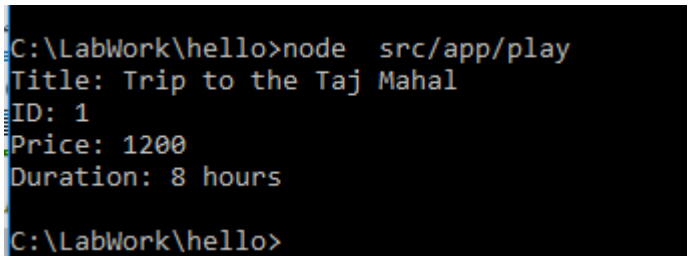
```
var t = new Tour(1, "8 hours")

t.title = "Trip to the Taj Mahal"
t.price = 1200.00

test(t)
```

The key thing to observe here is that even though test() takes as argument a Product it will actually invoke the printDetails() method of the Tour class.

__6. Save changes.

__7. Compile.

__8. Run.



__9. You can do this optional step all on your own. Add a class called Dining that extends Product and has these fields:

- **cuisine** of type string.
- **childPrice** of type number.

Create the constructor and a reasonable implementation of the 'printDetails' method.

Create an instance of it and pass it to the test() function.

This ability for a Tour object to appear as Product is called polymorphism. Inheritance is one of the ways to achieve polymorphism. The other being interface. We will get into that next.

Part 4 - Interface

An interface in TS describes the shape of an object. An interface lists a set of variables and methods. An object or class claiming to conform to that interface must declare those variables and methods.

In our ongoing example we have received a few new requirements.

- Products like Tour and Dining are bookable. We need to store a list of available dates for them. During booking a user will select a date.
- A Tour is cancelable and there is a cancelation fee associated with it. There may be other cancelable products in the future.

We realize that there may be similar requirements coming in the future. These features (bookable, cancelable etc.) can be associated with products in an independent manner. This makes it hard to model them using inheritance alone (TypeScript allows inheriting from a single class only). Interface is the right approach here.

__1. Create an interface called Bookable like this.

```
interface Bookable {
          availableDates: [Date]
}
```

__2. Create an interface called Cancelable like this.

```
interface Cancelable {
          cancelationFee: number
}
```

__3. Make the Tour class implement these two interfaces. This is shown in bold below.

```
class Tour extends Product implements Bookable, Cancelable {
...
```

__4. Save changes.

__5. Try to compile the code. You will get the following error.

```
C:\LabWork\hello>npm run build

> angular4-seed-project@1.0.0 build C:\LabWork\hello
> tsc -p src/

src/app/play.ts(17,7): error TS2420: Class 'Tour' incorrectly implements interface 'Bookable'.
  Property 'availableDates' is missing in type 'Tour'.
src/app/play.ts(17,7): error TS2420: Class 'Tour' incorrectly implements interface 'Cancelable'.
  Property 'cancelationFee' is missing in type 'Tour'.
```

That is because right now Tour class does not conform to these interfaces.

__6. Add these two fields to the Tour class.

```
availableDates: [Date]
cancelationFee: number
```

__7. Save and compile the code. Now it should compile.

__8. Write a global method that cancels a booking like this. This should be outside of any class definition.

```
function cancelBooking(c: Cancelable) {
  console.log("Canceling booking. Charges: %d", c.cancelationFee)
}
```

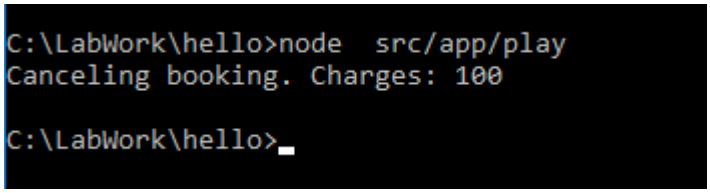
__9. Add these lines of code in bold at the end of the file to exercise the cancelBooking() function.

```
var t = new Tour(1, "8 hours")

t.title = "Trip to the Taj Mahal"
t.price = 1200.00
t.cancelationFee = 100.00

cancelBooking(t)
```

__10. Remove this code:

```
test(t)
```

__11. Save, compile and run.



A Tour class object can now appear as a Product, a bookable item and a cancelable item. The last two polymorphic behaviors are achieved using interfaces.

Part 5 - Generics

Currently our cancelBooking() method takes as input a Cancelable object. Unfortunately, we do not have access to the original price of the product or the product ID. Without knowing that we can not properly issue a refund. The correct data type should be a Product that is also Cancelable. How do we model that? Generics constraints will come to the rescue.

__1. Change the signature of cancelBooking() as shown in bold face below.

```
function cancelBooking<T extends Cancelable & Product>(c: T) {
```

Here T is a generic place holder for a type. As per our constraints, T must extend Product and implement Cancelable.

__2. Change the body of the function as shown below.

```
function cancelBooking<T extends Cancelable & Product>(c: T) {
  console.log("Canceling: %s (%d)", c.title, c.id)
  console.log("Price: %d", c.price)
  console.log("Cancelation fee: %d", c.cancelationFee)
  console.log("Total refund: %d", c.price - c.cancelationFee)
}
```

__3. Save, compile and run. You should see this output.

```
C:\LabWork\hello>node  src/app/play
Canceling: Trip to the Taj Mahal (1)
Price: 1200
Cancelation fee: 100
Total refund: 1100

C:\LabWork\hello>
```

Generics lets you work with completely unknown data types from a method or class. Optionally, as we did here, you can put certain constraints to that type. The end result is a mix of flexibility and type safety.

Part 6 - Implementing an Interface from an Object

Previously we have seen how a class implements an interface. In TS an object can also conform to an interface. This comes in handy for type safe programming.

Consider a typical JavaScript scenario where you supply several configuration options in an object.

```
configSomething({
    directory: "/foo",
    file: "bar.txt",
    maxSize: 1024
});
```

JavaScript can not enforce any kind of structure to this object. In TS we can specify that this object must conform to an interface.

__1. Let's say that in the example above **directory** and **file** are mandatory fields. And **maxSize** is optional. In **play.ts** add an interface like this to model that.

```
interface ConfigOption {
    directory: string
    file: string
    maxSize?: number
}
```

That ? after maxSize makes it an optional.

__2. Add a global method like this.

```
function configSomething(op: ConfigOption) {
    op.maxSize = op.maxSize || 1024

    console.log("Directory: %s", op.directory)
    console.log("File: %s", op.file)
    console.log("Max size: %s", op.maxSize)
}
```

__3. Call that method like this.

```
configSomething({
    directory: "/dir1",
    file: "persons.json"
})
```

__4. Remove the call to 'cancelBooking'.

__5. Save and compile.

```
npm run build
```

__6. Run the code.

```
node src/app/play
```

__7. You should see this output.

```
Directory: /dir1
File: persons.json
Max size: 1024
```

__8. Now introduce an error by changing the code as shown in bold below.

```
configSomething({
    directory: "/dir1",
    path: "persons.json"
})
```

__9. Save and compile. You should get a compilation error.

```
C:\LabWork\hello>npm run build

> angular4-seed-project@1.0.0 build C:\LabWork\hello
> tsc -p src/

src/app/play.ts(77,5): error TS2345: Argument of type '{ directory: string; path: string; }' is not assignable to parame
ter of type 'ConfigOption'.
  Object literal may only specify known properties, and 'path' does not exist in type 'ConfigOption'.

npm ERR! Windows_NT 10.0.14393
```

__10. Fix the problem, save and compile.

__11. Close all open files.

Part 7 - Review

In this lab we got a quick introduction to the most commonly used features of TypeScript. In summary:

- Variables and function parameters can have strong typing.
- You can develop classes and model inheritance between them.
- A class can inherit from only one class but implement many interfaces.
- Both inheritance and interface implementation leads to polymorphism.
- Generics are a powerful way to write classes and methods without knowing the exact nature of the types. This is mostly useful for framework and library developers who may not anticipate all possible scenarios in which their code may be used. Optionally, you can define constraints for added type safety.