LAB 13c

Introducing TypeScript

What You Will Learn

- Setting up your TypeScript environment
- Basic features of the TypeScript language

Approximate Time

The exercises in this lab should take approximately 90 minutes to complete.

Fundamentals of Web Development, 3rd Ed

Randy Connolly and Ricardo Hoar

Textbook by Pearson
http://www.funwebdev.com

Date Last Revised: January 2, 2024

Introducing TypeScript

Preparing Directories

1 The starting `lab13c` folder has been provided for you (within the zip folder downloaded from Gumroad).

This lab covers how to use the TypeScript language. This lab does so using the Node development environment, which is covered in the first exercise.

Exercise 13c.1 — Setting up TypeScript Environment

1 The TypeScript compiler must be installed first, using a tool like npm, yarn, or pnpm. In this lab, the instructions use npm. You must also decide to install TypeScript locally (i.e., within your application's folder) or globally, using one of the following commands in your `lab13c` folder:

```
npm install -g typescript
```

This will install the TypeScript compiler globally so TypeScript is available in all folders (this is probably the easiest approach).

```
npm install typescript
```

This will instead install the TypeScript compiler only in your current folder. This has the advantage that if future versions of TypeScript cause breaking changes in your application, your application will be shielded from them (since it will continue to use the version that you have installed in this folder).

2 TypeScript can be used for client-side JavaScript environments (e.g., browser) as well as server-side environments such as Node. To make TypeScript work more seamlessly with Node, you will install the ts-node execution engine for node by running the following commands:

```
npm install -g ts-node
```

You can forgo the -g flag if you wish to only install TypeScript locally.

3 Test via:

```
tsc  -v
ts-node  -v
```

Exercise 13c.2 — Hello World, Of Course

1 Create a new file named `hello.ts` in Visual Code with the following content:

```
let msg: string = 'Hello World';
console.log(msg);
```

The variable msg has a type annotation indicating that it will only contain content of type string. This will be enforced by the TypeScript compiler.

2 You will now need to run the TypeScript compiler to convert your code into regular JavaScript. Initially, we will do this with the tsc command, either in a command window or using the terminal within Visual Code (see note below):

```
tsc hello
```

On Windows, you may get an error message that tsc "`cannot be loaded because running scripts is disabled on this system`". This error can be fixed by running the following command in command window / terminal:

```
Set-ExecutionPolicy -ExecutionPolicy Bypass -Scope CurrentUser
```

3 If you examine your application folder, you will notice that tsc has created a JavaScript version of your file named `hello.js`. Examine this file in your editor.

4 Run this file in node via:

```
node hello
```

5 Edit `hello.ts` by adding the following:

```
function calcTotal(subtotal: number, tax: number): number {
   return subtotal + (subtotal*tax);
}
msg = 'Total = ';
let total: number = calcTotal(100,0.05);
console.log(msg + total);
```

6 You can combine steps 2 and 4 into one command:

```
ts-node hello.ts
```

7 Examine `hello.js`.

Notice that it is unchanged from what you saw after step 3. By default, ts-node JIT transforms the TypeScript into JavaScript at run-time and thus doesn't create an intermediate JavaScript file. This isn't ideal from a performance standpoint but is very handy when first learning TypeScript.

8 Change the following line of code in `hello.ts` as follows and then test (using ts-node):

```
let total: number = calcTotal("100",0.05);
```

This will generate a lot of error messages! The key message in all of it will be this: "`Argument of type 'string' is not assignable to parameter of type 'number'`". This is why we use TypeScript: to catch problems at compile-time instead of run-time.

9 Modify your code as follows and test.

```
let subtotal = 200;
let total: number = calcTotal(subtotal,0.05);
```

Notice that no type annotation has been provided fir subtotal and yet the code still compiles and executes. When no annotation is provided, the compiler will infer its type based on its content.

It can be helpful to create a TypeScript project, which allows you to customize how the compiler works, where the generated code will resides, etc.

Exercise 13c.3 — Creating a TypeScript Project

1 Run the following command.

```
tsc -init
```

This will generate a configuration file named tsconfig.json.

2 Examine `tsconfig.json` in your editor.

3 Use your editor's find facility to find "outDir" and change it as follows:

```
"outDir": "dist",
```

Notice this requires uncommenting this line. We are specifying the location of any generated JavaScript.

4 Add the following after the `CompilerOptions` section:

```
"compilerOptions": {
   ...
},
"include": ["src/**/*"]
```

4 Make a folder named `src` in your current working folder.

5 Move `hello.ts` to the `src` folder. Delete the previously-generated `hello.js` file.

6 Run the following command:

```
tsc
```

This will compile all the files in the folders indicated by the include option you added in Step 4.

7 Notice that you now have a `dist` folder. Examine the hello.js file in that folder. This file contains JavaScript that could be run in Node, which you can verify via:

```
node dist/hello.js
```

Exercise 13c.4 — Fun with Types

1 In the `src` folder, create a new file named `funTypes.ts`.

2 Add the following:

```
const years: number[] = [1970,1980,1990];
years.push(2000);
years.push("fred");
```

3 Compile by running the `tsc` command.

The last line will generate a compile error. TypeScript is doing its job of catching errors at compile time instead of runtime.

4 Comment out the last line and add the following.

```
const artist: [string,string,number] =
   ["Picasso","Spain", 1881];
console.log(artist[0]);
```

Since JavaScript allows an array to contain elements of any type, so too does TypeScript. Such an array is called a tuple in TypeScript.

5 Modify the code as follows and then compile.

```
const artist: readonly [string,string,number] =
   ["Picasso","Spain", 1881];
console.log(artist[0]);
artist[0] = "Randy";
```

Arrays can be flagged as read-only in TypeScript.

5 Comment out the previous line and add the following and then compile using `tsc`.

```
const a1 = {
   name: "Picasso",
   nationality: "Spain",
   birth: 1881
};
console.log(a1.name);
a1.name = 234;
```

While this code would be perfectly legal in JavaScript, it likely wouldn't be something we want to have happen. TypeScript does implicit type casting with object properties and thus catches this error at compile time.

6 Comment out the last line, add the following, and compile.

```
//a1.name = 234;
a1.death = 1973;
```

Notice that you can not add properties at runtime to existing objects in TypeScript.

7 Comment out the last line and add the following.

```
let a2: {
   name: string;
   nationality: string;
   birth: number;
} = {
   name: "Raphael",
   nationality: "Italy",
   birth: 1483
};
console.log(a2.name);
```

While this does provide a way to define an object and initialize it in type-safe manner, it generally makes more sense to use the interface or type keywords.

8 Add the following.

```
interface Artist {
   name: string,
   nationality: string,
   birth: number
}
```

An interface defines a contract for an object's structure.

9 Add the following and then compile.

```
const a3: Artist = {
   name: "Picasso",
   nationality: "Spain",
   birth: 1881
};
```

Now we can indicate what structure an object must have using the interface.

10 Add the following and then compile.

```
function output(a: Artist) {
   console.log(`${a.name} (${a.birth})`);
}
const a4 = {};
output(a3);
output(a4);
```

The last line will generate an error.

11 Comment out the last line and add the following.

```
//output(a4);
type Painting = {
   medium: string;
   base: string;
};
type Sculpture = {
   material: string;
   height: number;
};
type ArtWork = Painting | Sculpture;
```

The `type` keyword is used for creating type aliases, which is a way of creating new names for existing types. This illustrates a union type: which describes a type that is one of several possible types. This might seem crazy, but in TypeScript, it is helpful to think of types as being a ***set of values****.*

12 Add the following.

```
const p1: ArtWork = {
   medium: "Oil",
   base: "Canvas"
};
const p2: ArtWork = {
   material: "Marble",
   height: 1.4
}
console.log(p2.material);
```

Here we have two example objects of this union type.

13 Add the following.

```
type ArtDetails = {
   name: string;
   year: number;
   artist: Artist;
};
type Art = ArtDetails & ArtWork;
```

This illustrates an intersection type: which describes a type that is a combination of multiple types. Again, this will seem less strange if you remember that in TypeScript a type is a set of values.

14 Add the following.

```
const foo: Art = {
      name: "Madonna Enthroned",
      year: 1310,
      artist: {
         name: "Giotto",
         nationality: "Italy",
         birth: 1266
      },
      medium: "Tempura",
      base: "Wood"
};
console.log(foo.name + " by " + foo.artist.name);
```

15 Compile and test via:

```
tsc
node dist/fun.js
```

Exercise 13c.5 — Functions Get Functional

1 In the `src` folder, create a new file named `functions.ts`.

2 Add the following.

```
interface Person {
   name: string,
   birth: number,
   death: number
}
const per1: Person = {
   name: "Picasso",
   birth: 1881,
   death: 1973
};
const per2: Person = {
   name: "Raphael",
   birth: 1483,
   death: 1520
};
```

3 Add the following.

```
function lifeLength(a: Person): number {
   return a.death - a.birth + 1;
}
const outputperson = (a: Person): void => {
   console.log("person = " + a.name);
}
console.log(lifeLength(per1));
outputperson(per2);
```

Notice the second function has a return type of void. The compiler will flag as an error any attempt to add a return statement to this function.

4 Compile and test via:

```
tsc
node dist/functions.js
```

5 Add the following.

```
type PersonContainer = (a: Person) => string;
```

This adds a TypeScript function type definition or call signature: that is, it is a way of telling the TypeScript compiler the precise call signature that a specific function must have. This would be used if you are going to have multiple functions with the same signature.

6 Add the following.

```
const personDiv: PersonContainer = function (a) {
   return `<div>${a.name}</div>`;
}
console.log( personDiv(per1) );

const personSpan: PersonContainer = (a) => {
   return `<span>${a.name}</span>`;
}
console.log( personSpan(per2) );
```

Here we have defined two functions (the second of which uses arrow syntax) that share the same call signature, and TypeScript will ensure they have the correct call signature.

7 Edit the call signature as follows.

```
type PersonContainer = (a: Person,className?: string) => string;
```

The question mark indicates that this parameter is optional.

8 Edit as follows and test.

```
const personDiv: PersonContainer = function (a, b) {
   return `<div class="${b}">${a.name}</div>`;
}
console.log( personDiv(per1,"w-24 h-24") );
console.log( personDiv(per2) );
```

Note that the parameter names don't need to match: what needs to match is the type. The problem with this version is it operates the same regardless of whether the second parameter is provided.

9 Edit the function as follows and test.

```
const personDiv: PersonContainer = function (a, b) {
   if (typeof b !== 'undefined')
      return `<div class="${b}">${a.name}</div>`;
   else
      return `<div>${a.name}</div>`;
}
```

10 Add the following function and test.

```
function makeNested(parent: string, child: string,
                    content: string | string[]): string {
   let tag = `<${parent}>`;
   if (Array.isArray(content)) {
      content.forEach( c => {
         tag += `<${child}>${c}</${child}>`
      });
   } else {
      tag += `<${child}>${content}</${child}>`;
   }
   tag += `</${parent}>`;
   return tag;
}
console.log( makeNested("p","strong","This is the way") );
console.log( makeNested("select","option",
                ["Arsenal","Liverpool","Chelsea"]) );
```

The third parameter can be either a string or an array of strings. It uses a union type alias to specify the possible types for that parameter. The other possible way of implementing this functionality is to use overloading, which is shown in the next exercise.

11 Edit the previous function and as follows:

```
// overload signatures
function makeNested(parent: string, child: string,
                    content: string): string;
function makeNested(parent: string, child: string,
                    content: string[]): string;
function makeNested(parent: string, child: string,
       content: unknown): string {
   let tag = `<${parent}>`;
   if (Array.isArray(content)) {
      content.forEach( c => {
         tag += `<${child}>${c}</${child}>`
      });
   } else if (typeof content === 'string') {
      tag += `<${child}>${content}</${child}>`;
   } else
      throw new Error('content not string or array');
   tag += `</${parent}>`;
   return tag;
}
```

Exercise 13c.6 — Classes During Class

1 In the `src` folder, create a new file named `classes.ts`.

2 Add the following.

```
class Movie {
   title: string;
   year: number;
   runtime: number;

   constructor(t: string, y: number, r: number) {
      this.title = t;
      this.year = y;
      this.runtime = r;
   }

   getInfo(): string {
      return `${this.title} [${this.year} - ${this.getRuntimeReadable()}]`;
   }
   getRuntimeReadable(): string {
      let hours = Math.floor(this.runtime / 60);
      let minutes = this.runtime - (hours*60);
      return `${hours}hr ${minutes}min`;
   }
}
```

As you may remember from Lab10, classes were added to JavaScript in ES6, but they are simply a new syntax that combines constructor functions and prototype definitions.

3 Add the following and test.

```
let mov1 = new Movie("Juno",2007,96);
let mov2 = new Movie("Cloud Atlas",2012,172);
mov2.title = "TypeScript Atlas";
let temp = mov2.year;
console.log(mov1.getInfo());
console.log(mov2.getInfo());
```

As you can see, classes are like any function in that their properties are accessible.

4 Modify the class as follows and then try to test.

```
class Movie {
   private title: string;
   private year: number;
   private runtime: number;
```

The compiler will not allow you to access private members of a class. If no access modifier is provided, the property will be public (available outside the class).

5 Add the following method to this class.

```
getTitle(): string {
   return this.title;
}
```

6 Modify the following and test.

```
//mov2.title = "TypeScript Atlas";
let temp = mov2.getTitle();
```

7 Add the following interface definitions to the top of the file.

```
interface IJson {
   toJson(): string
}
interface IInfo {
   getInfo(): string
}
```

Like languages such as java and C#, an interface defines expected methods that a class must implement. It is a common naming convention to begin interface definitions with the capital letter I.

8 Modify the class definition as follows.

```
class Movie implements IJson, IInfo {
```

9 Add the following method to `Movie`.

```
toJson(): string {
   return JSON.stringify(this);
}
```

10 Add the following.

```
class Play implements IJson {
   title: string;
   author: string;
   year: number;

   constructor(t: string, a: string, y: number) {
      this.title = t;
      this.author = a;
      this.year = y;
   }
   toJson(): string {
      return JSON.stringify(this);
   }
}
let play1 = new Play('Hamlet','Shakespeare',1601);
```

It is common for multiple classes to implement contracts defined within different interfaces.

11 Add the following and test.

```
function apiOutput(obj: IJson) {
   console.log([obj.toJson()]);
}
apiOutput(play1);
apiOutput(mov1);
```

This function is expecting an object that implements a specific interface. Since both Play and Movie implement this interface, both can be passed to this function.

12 Create the following new class after the interface definitions near the top of the file.

```
abstract class ArtProduction implements IJson {
   title: string;
   year: number;

   constructor(t: string, y: number) {
      this.title = t;
      this.year = y;
   }
   toJson(): string {
      return JSON.stringify(this);
   }
}
```

Just like in languages such as Java and C#, abstract classes are used to define common behaviors for derived classes to extend and cannot be instantiated directly.

13 Modify the `Play` class as follows and test.

```
class Play extends ArtProduction {
   author: string;

   constructor(t: string, a: string, y: number) {
      super(t,y);
      this.author = a;
   }
}
```

14 Try to instantiate an object based on this abstract class as follows:

```
let play2 = new ArtProduction('Macbeth',1601);
```

This will generate an error.

15 Modify your `Movie` class so it also extends `ArtProduction` but still implements `IInfo`.

Exercise 13c.7 — TypeScript Generics

1 In the `src` folder, create a new file named `generics.ts`.

2 Add the following.

```
const arr1 = [5,4,23,88,30];
const arr2 = ["foo","bar","xyz","abc","def"];
const arr3 = [true, false, true];
```

3 To begin let's try to define a function that will search a passed array for a specific value, and if found, then return the array index for that value using standard JavaScript.

```
function findIndexWith(searchIn,searchFor) {
   let indx = 0;
   for (let item of searchIn) {
      if (item == searchFor) return indx;
      indx++;
   }
}
```

This won't work however in TypeScript since we need to specify parameter types.

4 Change the function signature as follows:

```
function findIndexWith(searchIn: any,searchFor: any) {
```

The any data type matches any data type. The compile errors now go away … but at what cost?

5 Test the function by adding the following.

```
console.log(findIndexWith(arr1,88));
console.log(findIndexWith(arr2,"abc"));
console.log(findIndexWith(arr3,false));
console.log(findIndexWith(55,88));
```

6 Compile and test.

While it compiles, we get a run-time error because we passed a non-array to the function in the last line. Instead of using the any data type, which opens up our code to potential type errors, we will instead use generics to create a type-safe version of this function.

7 Modify the function signature as follows.

```
function findIndexWith<T>(searchIn: T[],searchFor: T) {
```

Here we are providing additional information to the function: namely, the data type of information that the function will work on. You could think of it as akin to passing an additional parameter.

8 Try to compile.

It will now give a compile error on the attempted usage that doesn't pass an array, which is the type of error checking we want!

9 Comment out the following line, compile, and test.

```
//console.log(findIndexWith(55,88));
```

It now works.

10 Modify the function signature as follows and test.

```
function findIndexWith<Randy>(searchIn: Randy[],searchFor: Randy) {
```

Just as with parameter variables, the name can be anything. It is conventional however to use letters like T, X, or Y.

11 Add the following code and test.

```
function makeTuple<X,Y>(a: X, b: Y) {
   return [a, b];
}
console.log( makeTuple(false,"hello") );
console.log( makeTuple(13,{id: 401, status: "Ok"}) );
```

You can provide any number of possible types using generics simply by providing different type names.

12 Change the function definition as follows and test.

```
const makeTuple = <X,Y>(a: X, b: Y) => {
```

This illustrates the syntax for using generics with arrow function notation.

13 Change the function definition as follows and test.

```
const makeTuple = <X,Y>(a: X, b: Y): [X,Y] => {
```

This adds type information for the return value from the function.

14 Add the following.

```
type Codes = {
   id: number;
   status: string;
};
const arr4: Codes[] = [ {id: 401, status: "Ok"},
                        {id: 500, status: "Unavailable"}];

type Countries = {
   name: string;
   id: string;
}
const arr5: Countries[] = [
   {name: "Canada", id: "CA"},
   {name: "France", id: "FR"},
   {name: "United States", id: "US"}
];
```

This defines two typed array of objects; each of these objects contain a field named "id".

15 We want to create a function that will return the object that has a matching `id` value. While this is easy enough with the standard array `.find` function, it is not type safe. This last example illustrates how we can use constraints with our generic functions.

Add the following function and then try to compile.

```
function findById<ArrType,KeyType>(content: ArrType[],
                                   key: KeyType) {
   for (let item of content) {
      if (item.id == key) return item;
   }
}
```

You should see a compile error that says item doesn't have an id property. We will solve this using generic constraints.

16 Define the following interface before your function definition:

```
interface ContainsId {
   id: number | string;
}
```

17 Modify your function signature as follows:

```
function findById<ArrType extends ContainsId,KeyType>(
             content: ArrType[],
             key: KeyType) {
```

This provides information to the compiler that objects in the first parameter will contain an id property that is either number or string.

18 Add the following code and test.