

TypeScript in Plain Language Volume I

A Comprehensive Starting Up Guide

For Complete Beginners

Tony de Araujo

Technical Instructor

New Jersey, U.S.A.

October 2016

TypeScript 2.0 compatible

Copyright © Tony de Araujo

All Rights Reserved

Download Amazon's *free* Reading App for your system:

[Kindle Reading Apps](#).

Author's profile at [Amazon](#)

My gratitude goes to Amazon, family, friends, teachers, authors and all readers around the world.

"Excellence is not a skill. It is an attitude." -- Soren Kierkegaard

Why this book

Dear reader,

As all my other publications, this is a book of exercises.

The only way to learn how to program in a certain language is by doing it. Even if you have taken a TypeScript course before, you will forget the rudiments of the language unless you do periodic maintenance by practicing basic language skills.

As all the other books I have written, the book's purpose is to empower the reader. It will provide a solid beginning for you to master TypeScript syntax and language semantics. It will also supply you with a programmed reviewing set of exercises for a weekend routine drill.

TypeScript in Plain Language Volume I will not make you a TypeScript expert (and it shouldn't), but it will point you in the right direction by giving you the necessary foundation to take the rest of the journey on your own.

By design, this volume uses the TypeScript playground because it is the best way to learn and test basic syntax. **Volume II** will show you how to install your own local development server and how to start with **Angular 2** using TypeScript. Volume II will be available in November 2016.

Tony de Araujo

At the end of the book, empty your cup and return to the beginning.

Practice, Practice, Practice

Other books from the author

For other publications of the author, please visit Amazon:

<https://www.amazon.com/Tony-de-Araujo/e/B00D7V08WY/>

Table of Contents

[TypeScript in Plain Language Volume I A Comprehensive Starting Up Guide](#)

[Why this book](#)

[Other books from the author](#)

[Table of Contents](#)

[Tools needed for this project](#)

[How to read this book](#)

[PART I....](#)

[1- What is TypeScript](#)

[Why should I write my JavaScript using TypeScript?](#)

[2- VARIABLES and data TYPES](#)

[Introduction](#)

[Declaring a variable and giving it a type classification](#)

[Using the keyword any as a variable data type:](#)

[LAB WORK: variable declaration](#)

[Notes about types and variables](#)

[How to write comments in your code](#)

[Using ALERT, PROMPT, CONFIRM and CONSOLE.LOG](#)

[The enum data type](#)

[LAB WORK: practicing enumerations](#)

[3- Math, logical and comparison OPERATORS](#)

[Math operators](#)

[Comparison operators](#)

[Logical operators](#)

[The IF, ELSE IF, ELSE conditional statements](#)

[Using the ternary operator as a conditional shortcut](#)

[The Switch statement](#)

[LAB WORK: practicing conditional statements](#)

[4- Short introduction to OBJECTS, ARRAYS and LOOPS](#)

[What is an object in TypeScript?](#)

[The For In loop](#)

[CONSOLE.LOG: How to visualize the result on the screen](#)

[What is an array?](#)

[The for loop](#)

[Initializing arrays in TypeScript](#)

[Using de-structuring to individualize data from arrays or objects](#)

[The Spread Operator](#)

[5- Introduction to FUNCTIONS](#)

[About functions](#)

[Making sure the function input data type is validated](#)

[The return value](#)

[Setting default parameter values](#)

[Optional Parameters](#)

[A REST parameter accepts 'all the other arguments'](#)

[ARROW functions – an introduction](#)

[About the *THIS* in arrow functions](#)

[a\) Arrow functions have no "this"](#)

[b\) An arrow function inside of another function](#)

[c\) An arrow function inside of a single object](#)

[d\) Using arrow functions in objects instantiated from classes](#)

[e\) Arrow functions assure the correct context in classes](#)

[f\) Summary](#)

Function OVERLOADING

LAB WORK: Three small projects using functions

PART II

6- Variable prefixes: VAR, LET, CONST, DECLARE

Function Scope

Planning for block scope

Use let for loop counters

The CONST prefix

Using the const prefix in function expressions

The DECLARE keyword

7- Interfaces

What is an interface?

How to create an interface

Assigning an interface type to an object

Declaring an object assigned to an interface but with no property values

How to create optional properties

Interfacing objects with unlimited properties

Using an interface to describe a function

Adding a function (or method) to an object Interface

Using an interface to describe an Array type

a) Numeric indexed array interfaces

b) String-indexed array interfaces

Extending Interfaces

8- Classes

What is a class?

Creating a class

LAB WORK: creating classes and objects

Implementing classes from interfaces

Extending classes by using other classes

- a) A simple extended class
- b) Adding one more property to the super class
- c) What happens when we add a method to the *child* class?
- d) What happens when we add a property to the *derived* class?
- e) How to add a constructor to the child class

LAB WORK: Extending classes

Private and public properties

- a) Public properties
- b) Readonly – a TypeScript 2.0 feature
- c) Private properties
- d) Getting: enabling reading data from private properties
- e) Setting: enabling adding or editing data in private properties
- f) Instantiating an object and accessing its private properties

Static properties in a class

LAB WORK: Adding static properties to classes

9- Namespaces

What are namespaces and modules?

Creating an alias for namespace functions

Using external Modules

10- Casting and Conversion

Parsing strings to numbers

Using the toString conversion method

Type Compatibility

11- Last but not least

Where to go from here

In conclusion

Tools needed for this project

In order to do these exercises you will need to be online. The author demonstrates the exercises by using *Google Chrome* but please feel free to use any browser you prefer.

This book uses the online *TypeScript playground* for most exercises. The idea is to learn TypeScript syntax by practicing the concepts incrementally introduced.

At the end of the book, the reader/student will be better prepared to approach advanced topics as found in other books and online articles.

Do not worry about the simplicity of the playground. It is actually very useful when it comes to learning code, and its testing capabilities are amazingly productive. For what we cover in this book (and we cover a lot), the playground is the best tool of all. It also assures we are using the latest version of the language.

The author assumes the reader knows basic ***JavaScript***. However, most concepts of JavaScript that apply to the topics treated in the book will be reviewed as we approach its TypeScript counterpart.

The electronic version of this book was written for *Kindle*, especially the *Kindle App*.

The author recommends using the *free Kindle app* by downloading it to your system. This will facilitate linking directly to the exercise samples on the internet.

You can download the free Kindle app from the following Amazon URL:

<http://www.amazon.com/Amazon-Digital-Services-Inc-Download/dp/B00UB76290/>

How to read this book

This book will be more rewarding if you read it straight through in a linear fashion because many of the concepts build on one another.

If your desire is to review a certain subject, there are eleven major classifications, which are numbered and self-contained. Everything else is within those groups. Please refer to the table of contents in your Kindle for a complete list of topics.

Major Topics:

- 1- What is TypeScript
- 2- VARIABLES and data TYPES
- 3- Math, logical and comparison OPERATORS
- 4- Short introduction to OBJECTS, ARRAYS and LOOPS
- 5- Introduction to FUNCTIONS
- 6- Variable prefixes: VAR, LET, CONST, DECLARE
- 7- Interfaces
- 8- Classes
- 9- Modules
- 10- Casting and Conversion
- 11- Where to go from here

Getting the most out of this book

To get the greatest benefit from this book, I highly recommend that you do all the exercises shown in the book. You should study the exercises and try modifying them for testing purposes. There is a reason why an exercise is included and you may not discover it until you take a hard look at the problem. This is a hands-on activity.

PART I....

Awareness empowers

1- What is TypeScript

Simply said, TypeScript is JavaScript with some added bonuses. Conceptually, I look at TypeScript as a “*virtual editor*” that allows me to write JavaScript in an easier and safer way.

As the code in a project gets larger or complex, writing it in TypeScript improves productivity.

- This is due to the added features of the TypeScript language.
- With fewer instructions, we are able to create long lines of JavaScript code.
- It is also because TypeScript is more object-oriented whereas JavaScript is functional.
- Finally, it is much easier to visualize a solution in TypeScript than it is to conceptualize it in common JavaScript.

The good thing is that native TypeScript understands JavaScript and converts our classes and modules into functional code.

On the other hand, TypeScript code is easier to understand. You will know exactly what an expression is doing rather than taking a guess about it, as sometimes happens when inspecting JavaScript.

If you already know JavaScript, you can use it in your TypeScript files because JavaScript is TypeScript without the added bonus features. Once you paste your existing JavaScript code into a new TypeScript file, you can then optionally add the new features that come with TypeScript, which will convert back into a more complex functional JavaScript project when you compile the code.

In order to take advantage of this entire concept you need to start thinking in terms of TypeScript instead of common JavaScript. If this thought bothers you, don't worry about it because TypeScript is a subset of JavaScript and your visualization of it will come naturally, as you write more code.

Eventually, it may come a time when your mind shifts and you start thinking of TypeScript as a language on its own, one that shares features with JavaScript.

Why should I write my JavaScript using TypeScript?

In case you are not yet convinced of approaching JavaScript by writing in TypeScript, please read on:

- It is a lot easier to write JavaScript code in *TypeScript mode* than doing it directly as JavaScript. This is especially true when it comes to larger or more complex programs.
- TypeScript is also safer to write due to its type checking features, which minimize potential problems as the code expands, or when writing code as a team.
- The learning curve for mastering the language is a lot shorter when using TypeScript. Advanced JavaScript is much more complex.
- Just like JavaScript, TypeScript is an *open source* computer language that transcompiles (language-to-language translation) to JavaScript. It was specifically created by Microsoft to work under JavaScript syntax rules. This means that almost any JavaScript code is also TypeScript.
- There are many advantages on writing in TypeScript as opposed to doing it directly in JavaScript such as code safety, which prevents unnoticeable errors, added features like *classes*, *interfaces*, *modules*, *static* data types, creation of enumerable *lists* and much, much more.

On the other hand, you may have heard of JavaScript version 6 (also known as [ES6](#) or [JavaScript 2015](#)) and you might ask yourself why you shouldn't learn that language version, instead.

The answer depends on what you are trying to accomplish. Yes, we all should eventually learn ES6. However, ES6 is not yet compatible with all browsers on the field and probably will not be compatible for years.

If we want more and better features right now, TypeScript is a good alternative to common JavaScript because it brings ES6 features and converts them into functional common JavaScript, the version currently compatible with all browsers.

On the other hand, many of the features waiting to be implemented in JavaScript 2015 are available in the current TypeScript version, which compiles them into common JavaScript by using functional syntax.

This advanced JavaScript syntax (created by TypeScript), is not well known by many JavaScript developers. Even if they know it, doing it manually is time consuming and prone to human error.

In other words, you tell TypeScript what you want to accomplish by using its succinct syntax, and TypeScript does the magic conversion by creating functional JavaScript.

Another reason why TypeScript is becoming very popular is Google's **Angular 2**:

- The new AngularJS version is officially tuned to use TypeScript instead of JavaScript and this is because of its *type checking* capabilities. That in itself is reason enough to learn the language features added to TypeScript so that we can use them while creating *Angular* projects.

Once you grasp the basics of TypeScript, you may not write pure JavaScript ever again because the advantage of TypeScript becomes obvious. At least that has been my experience with it since TypeScript provides more tools to work with than the ones provided by common JavaScript.

On the other hand, if you come from a **C#** or **Java** background, using TypeScript will make more sense to you than common JavaScript.

In summary:

Just keep in mind that TypeScript is not another language. TypeScript is JavaScript with some added bonuses. Once you learn TypeScript you will notice that ES6 makes more sense to you as well. This is because you are actually learning ES6 when you study TypeScript.

2- VARIABLES and data TYPES

Introduction

Please read along and understand the concepts. These lectures are purposely short. In the end of each section, there is an opportunity to review the concepts by doing some exercises. Later, you might want to code along as I explain concepts but in the earlier chapters, it is more efficient to wait until we get to the lab session that follows the explanation.

What is a variable?

In order to reserve memory space for the data we want to store in a computer program sequence, we need to tell the TypeScript compiler what kind of data we want to save in a certain named location. This type *declaration* is a contract between the programmer and the TypeScript compiler.

Since TypeScript is compiled to JavaScript, the size of the reserved memory is not important because JavaScript variables can change type at any time; what the type declaration does, is to tell the TypeScript interpreter about our intentions for that variable. Then, the interpreter will inspect the code as we write it, and warns us if we try to change the contract by inadvertently changing the type.

Think of TypeScript as your code assistant that double-checks your work.

There are several pre-sized data types in the TypeScript library but we can also create our own customized types as well.

The simplest and most common types used in variables are the following:

- Type **string** – this represents a string of characters such as a sentence, a word, a single character or even a set of numeric characters (when wrapped in single or double quotes):
"Hello" <-- this value is a string of characters. This data is of type *string*.
"33" <-- this value is a two-character representation because of the quotation marks around it. Numbers wrapped in quotes are always of type *string*.
'grape' <-- In TypeScript (as in JavaScript) both single or double quotes represent *string* values.

- Type **number** – it represents any number as long as it is not wrapped in quotes:
33 <-- this is a number with no quotes around it. Therefore, it is of type *number*.
- Type **boolean** – A variable of type boolean has one of two values: **true** or **false**. Since TypeScript is case sensitive, the **t** or **f** must be in lower case in order for the value to be a boolean. **Note:** It is a fact that, in JavaScript/TypeScript, when a variable contains a value other than *zero*, *null* or *undefined*, the value is considered *true*; otherwise, *false*. However, those are not Boolean values, those are evaluations of whether the variable contains or does not contain a value. The only Boolean values from the *boolean* type are *true* and *false*.

Declaring a variable and giving it a type classification

Before we store data in memory, we need to create a *handle* to grab the memory location.

Otherwise, we cannot physically access that location in order to add or edit our data.

This 'handle' serves as an *alias* for the memory location address, and in general terms, we call it a **variable**.

To declare (or introduce) a variable, we start with the keyword **var**, then the variable name, and then the optional data itself.

In TypeScript we can also replace the keyword **var** for the keyword **let** which grounds the variable to the code block where it is being introduced (more about *let* on a later section).

Please note:

var is only used once, when we first initialize a variable.

The difference between *var* and other declaring prefixes will be expanded on *chapter 6* when we get to *VAR, LET, CONST, DECLARE*.

Method 1 – Declaration and assignment

Examples of variable declarations:

```
var myName = "Tony";
```

```
var myNumber = 99;
```

```
var isAparent = true;
```

Please note:

true is not a string of characters; *true* is a reserved keyword, which represents a boolean value [2] (more about boolean values later).

Because *true* is a reserved word, *true* is not wrapped in quotes.

In all of the above examples, based on the type of data assigned to the variable, the TypeScript interpreter will know what kind of variable it is (the *type* is implicitly assigned).

Method 2 – Declaration without assignment

We can also declare variables without assignment of data.

However, contrary to JavaScript, when we don't assign initial data to a variable we must specify which **type** of data we want to store in that location.

The reason for specifying a data type is to inform the TypeScript interpreter of what our intention are for that particular variable.

Once we tell the interpreter what we intend to do with the variable, the interpreter will assure that we don't make a mistake later on in the code. It enforces the original intention by flagging the code if we insert the wrong data type into the variable.

The following are examples of the same variable declarations as before. This time, however, I am not assigning data to them:

```
var myName: string;  
var myNumber: number;  
var isAparent: boolean;
```

Notice how I had to specify the *type of data* after the variable name. I separated the variable name and the type declaration with a *colon :* (the colon indicates inheritance. The variable inherits the *traits* available from the *type* that follows. Also, white space does not matter):

```
var myName : string
```

is the same as `var myName:string`

Question:

Why didn't I specify the *data type* on my first examples (method 1)?

Since my first examples included the data at the time of variable declaration, the TypeScript interpreter implicitly knows what type of data we are going to store in that memory location. This allows us to skip the type declaration step.

However, as a curiosity, this is how we could write the whole declaration and assignment if we wanted to:

Method 3

```
var myName: string = "Tony";  
var myNumber: number = 33;  
var isAparent: boolean = true;
```

Notice how *string*, *number* and *boolean* are in lower case. This is important because those are the reserved names for these types. Upper case will not work.

Using the keyword *any* as a variable data type:

Sometimes we want a variable to stay dynamically open to accept any type of data inputted during runtime. In this case we use the keyword *any* as a type.

In TypeScript hierarchy, the *any* type is actually above the types *number*, *string* and *boolean*. This means that when we declare a type *any*, TypeScript will allow us to subsequently use data of any one of the other types.

Notice that I've written “*use*”. You cannot explicitly change the type to another type because this variable was initially declared as an *any*, and *any* is a type. What we can do, is to insert data of *any other* type in an *any* type variable.

Example:

```
var xyz: any;
```

Then...

```
xyz = 6; //ok
```

```
xyz = "Tony"; // ok
```

```
xyz : string = "test"; // ERROR because xyz already has an  
assigned type
```

As seen, all these variables will accept any type of data without getting a complaint from the TypeScript interpreter. On the last example, when I specifically try to change the type of *xyz* to *string*, I get an error to warn me that I may be making a mistake.

We can also **declare and assign** a variable of type *any* just as we did with other types:

```
var myEntry: any = "blablabla";
```

Please note:

After the code is translated to JavaScript, it does not matter what type of data the variable really accepts since JavaScript does not care. This can sometimes be a good thing and most of the times a bad thing because it may lead to errors **while we program** the code.

The reason to use TypeScript at design time is to prevent unexpected results later on. Once TypeScript transcompiles the code into JavaScript, the type checking capability disappears but by now, the project is finished without data type errors.

LAB WORK: variable declaration

For this lab test, we are going to use the playground at the TypeScriptlang.org website:

Here's the link to the playground: typescriptlang.org/play [3]

Erase the data shown on the left column of the playground editor. The data on the right side will erase automatically since it is a read-only column.

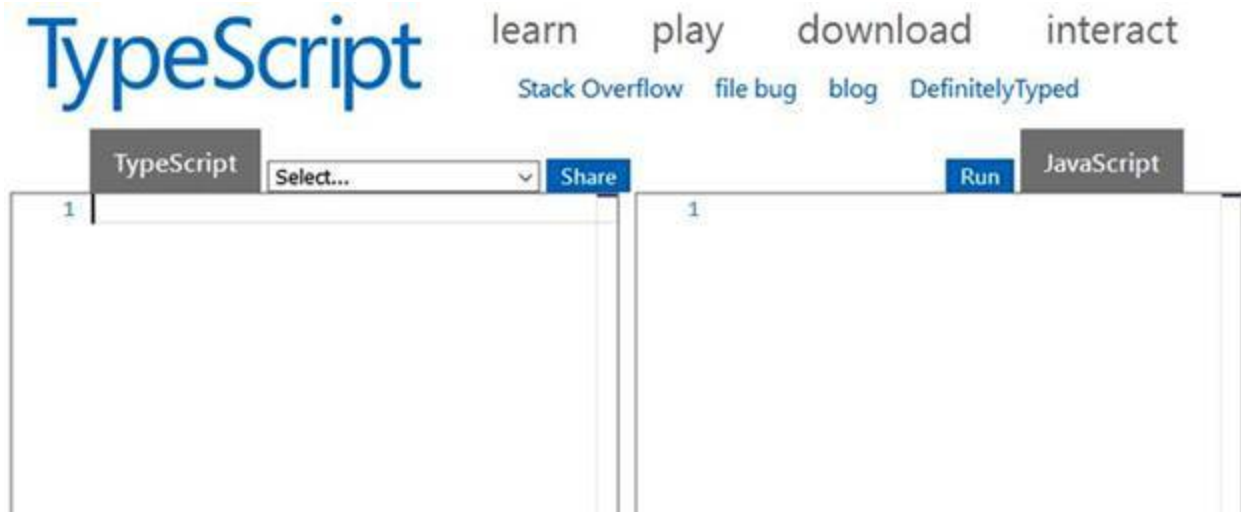


Fig 1 This is how your playground will look like after erasing the original code

1. On the left column, declare a variable of type *string*.

Do not add data to it just yet:

```
var myName:string;
```

Look at what happens on the right column. The playground automatically converts your TypeScript code into JavaScript. This is called [transpiling](#) [4] or transcompiling (source to source). Notice how the *type* of data has been stripped from the JavaScript result on the right column. This is because in JavaScript the data *type* does not matter since all variables accept any data type we throw at them.

However, by declaring the intended type of data as we did on the TypeScript column, we are preventing possible human errors while programming.

This prevention happens because the editor is assisting us as we program. These advantages will be more obvious later on. Integrity of data *type* becomes extremely important, as programs get larger and therefore more prone to human error.

2. Let's now try to assign the variable *myName* to a number and see what happens.

On the next line write:

```
myName = 33;
```

On the playground layout, notice the red square mark at the right of the left column (see image below). Even though the JavaScript transpiling worked out ok, TypeScript flagged your variable assignment because you are assigning a number to a variable you have previously declared as a *string* variable. If you were writing this code in a real TypeScript editor, you would not be able to process your TypeScript code.



Fig 2 a numeric data type should not be stored on a string type variable

3. Replace the number 33 with the following string value (include the quotes):

```
myName = "Tony";
```

4. Now let's declare a variable of type *number* and assign the number 33 to it at the same time:

```
var myNumber = 33;
```

Notice how we do not have to declare the *type of data* when we declare the variable since we are including the data in the declaration. TypeScript will implicitly default to *type number* based on the data we initially include.

- 5- Declare a variable *x* of type *any*.

Then assign it to a *string* like for example *"Hello"*.

Later reassign variable *x* to the number 999. It should accept this change without complaining:

```
var x:any;  
x = "Hello";  
x = 999;
```

- 6- Try a few more 'variable declarations and assignments'.

For some of your variables, exclude the data at the time of declaration and assign it

afterword on the next line.

Here are a few more examples:

```
var inStock: boolean = true;
var myColor: string;
myColor = "Orange";
var myArray: number[];
myArray = [1,2,3,4];
var mySecArray: string[];
mySecArray = ["green","red"];
```

I have included *array types* on my example in case you were curious about them. We will cover arrays later. Just practice with data types that you already know.

Please note:

This is a course in TypeScript, not a JavaScript course. However, most people interested in learning TypeScript are doing so because of JavaScript. They want to convert or transcompile the TypeScript code into JavaScript in order to use it on their projects. The *playground* is a quick way to get some JavaScript code going. You can actually just copy the JavaScript code resulted from our TypeScript original and then paste it on a JavaScript console for testing purposes or on your own Angular/HTML project. You might ask why I shouldn't just do it directly with JavaScript and avoid the extra step. Well, it is a valid point (for now) but TypeScript is actually easier to write than JavaScript is. As the code gets more complicated, JavaScript becomes entangled and the chances of creating errors in our code are exponentially greater.

You will appreciate your skills in TypeScript as you write programs that are more complex. At this point, JavaScript is easier to write but this will be reversed later in the book. Actually, you will need to know a great deal of JavaScript to write some of the simplest programs we are going to do in TypeScript during this course.

Also, always look at the JavaScript version on the right side to see what TypeScript does to it. You will be amazed about the complexity of JavaScript for some of the easy tasks we are going to program.

Notes about types and variables

If you are coming from a language like for example C#, you will notice how TypeScript does not subdivide primitive types into different categories to allow for better memory management. This is because TypeScript must limit itself to the capabilities and nature of JavaScript, which adjusts its types dynamically, and it has a very limited classification of types.

About type declarations

TypeScript implicitly assigns all numbers to type ***number***, which is a double-precision 64-bit floating-point value.

All characters within quotes are assigned the type ***string***, which is the same as the JavaScript primitive type and it represents a sequence of characters stored as Unicode UTF-16 code.

A ***boolean*** type is either a logic value of *true* or a logic value *false*.

TypeScript includes three other types based on the JavaScript language but we do not assign those types to our variable declarations: *undefined*, *null* and *void*.

Type ***void*** is used in functions when a function does not have to return a value. Functions will be covered later.

About variable names

For *name-styling* please refer to the GitHub TypeScript Wiki for suggestions. This wiki is constantly being updated:

github.com/Microsoft/TypeScript/wiki/Coding-guidelines [5]

As for variable names, use the following guidelines:

TypeScript is a case-sensitive language. This means that a variable name such as *myName* is different from the variable name *MYName*. Variable names can be of any length. The rules for creating legal variable names are as follows:

- The first character must be an ASCII letter (either uppercase or lowercase), or an underscore (`_`) character (underscores are not recommended for normal variable names). Note that a number cannot be used as the first character.
- Subsequent characters must be letters, numbers, or underscores (`_`).
- The variable name must not be a TypeScript reserved word.

See the following link for a list of reserved words:

github.com/Microsoft/TypeScript/issues/2536 [6]

When in doubt, use the tool on the following link to verify if your word is a forbidden word. This tool validates your word against TypeScript/ECMAScript6 reserved terms:

mothereff.in/js-variables [7].

How to write comments in your code

The most common way to comment code is by doing it line by line. For that, we use two forward slashes at the beginning of the line:

```
// this is a comment
// this is another comment, etc.
```

An alternate way for writing multiple lines of comments is as follows:

```
/* this is a very long comment and I really don't want to write forward
slashes on every line. However, the forward slashes actually make a comment much
more obvious when the comment spans for many lines. */
```

Caution: Any code to the right of a `//` will be considered a comment. We can comment to the right of a code line, but we can never add code to the right of a comment:

Good comment:

```
var x = 123; // x is a numeric variable
```

Bad comment:

```
// x is a numeric variable var x = 123;
```

Commenting code in the playground

Besides commenting notes about our code, we can also comment one or several lines of code to make it inactive when we want to test the program.

In the playground, (for Window systems) you can inactivate a code block by highlighting the code and then press **CTRL+ /**.

You can also reactivate the code back by repeating the process, which removes the forward slashes from the comment block.

Using ALERT, PROMPT, CONFIRM and CONSOLE.LOG

TypeScript is a superset of current and future versions of JavaScript, and it includes additional features that may or may not be available in upcoming JavaScript versions. As a refresher, a "superset" is a set that includes another. JavaScript is included in TypeScript.

Because of this inclusion, any JavaScript program is valid TypeScript code. One can literally paste JavaScript into a TypeScript editor and TypeScript will convert it back to JavaScript when the program is finalized.

TypeScript does a better job in its JavaScript code creation than what most JavaScript programmers can manually do (Just this alone is a reason to learn TypeScript).

TypeScript is a [strongly typed](#) [8] language that converts to a weakly typed language.

Since TypeScript has been designed to translate into JavaScript, its strongly typed features are molded to make this process seamless. This is the advantage of TypeScript over other languages attempting to transcompile to JavaScript. TypeScript was designed from the ground up for this purpose.

In order to output our test results when studying or programming, we will be using the following display methods on the exercises:

console.log() <-- Displays a message on the *console*. I will show you how to access the Console when we get to loops.

alert() <-- Displays a popup box with a message

confirm() <-- Displays a popup box with a message, and expects a user confirmation.

prompt() <-- Displays a popup box with a request for more information from the user. The user's input can then be assigned to a variable.

You will have a chance to play with these methods during our lab sessions.

NOTE: If you are new to this syntax and do not know the meaning of **()** , this will be explained on the next section when I talk about functions.

The enum data type

Up to this point we have discussed simple "data types" such as *number*, *string* and *boolean*. The next data type is a more complex one in the sense that it may contain several data values, rather than just one value in it.

enum stands for *enumeration*.

Enumeration is used whenever we want to create a collection of fixed named elements, such as the days of the week, or the months of the year.

These names are also known as *constants*.

Side note: A "constant" is a *quality* of a given data and it means that the original data cannot be changed. If I declare a variable as a constant (more about this later), I will have to assign its data at the time of declaration, and this data cannot be changed. This is because TypeScript (as in other languages) will treat this value as a fixed value, not as a variable that can be manipulated.

In an *enum*, each name in the collection of names is assigned to a number. This happens automatically (unless we customize the numbers). Then later, we can recall each name by calling its corresponding index number.

By default, an enumeration starts from zero (as the first value) but we can override the starting position by assigning the first name to a different number, in which case all other consecutive numbers will sequentially apply. (For example, if you assign your first value to start at index 3, the next value will start at 4, and so on).

Besides assigning a beginning index number to an enumeration, we can also override all the indexes by assigning specific numbers to each one of the named values.

Example of a regular **enum**:

```
enum WeekDays{  
    Monday,  
    Tuesday,  
    Wednesday  
}
```

Notice how the W is capitalized. This is not mandatory (the interpreter does not care) but it is conventionally accepted to capitalize enums, interfaces and classes in order to visually separate them from regular variables, functions, arrays and objects.

Now, when we log or alert **WeekDays [1]** to screen, we get the value of Tuesday displayed, which is the second item counting from zero:

```
alert(WeekDays[1]);
```

Notice how we address the first element of *WeekDays*: We use **bracket syntax** (the square brackets) to pass in the index number we want to address.

Side note:

In programming, we usually use either **bracket syntax** or **dot syntax** to address preexisted values.

Dot syntax is used when the value resulted from the expression presented, is known and fixed. (Think of it as *hardwired*).

In addition, dot syntax does not accept numbers or string values (those wrapped in quotes).

Bracket syntax is used when the values are not known until the time of code execution. (Think of it as *softwired*).

Going back to *enums*, let's say that we want our first item to start at index 6. In this case, we must specifically declare our intentions because the first location normally defaults to zero.

Example of a modified starting index:

```
enum SummerMonths{  
    June = 6,  
    July,  
    August  
}
```

Now, when we call *index 8* from *SummerMonths*, we get *August* as a result because the addresses of all other values were sequentially incremented from 6:

```
alert("The hottest month is " + SummerMonths[8]);
```

The displayed result will be *The hottest month is August*.

Please remember, we can also assign a specific number to each item of an enumeration.

Enumerations come handy as lists of constants we can recall throughout the program.

We use *enumerations* whenever the same constant data value is needed throughout the program.

It makes it easy to edit the code if we ever want to change that value because we will only change it once, instead of possibly having to do it hundreds or thousands of times.

LAB WORK: practicing enumerations

- a) Please navigate to the TypeScript Playground:
typescriptlang.org/play [3]
- b) Then, erase the default code shown on the display. If you have written any code recently please do a browser hard refresh
On PC use **CTRL + f5**
(To find a shortcut for systems other than Windows please refer to the following webpage: wiki.scratch.mit.edu/wiki/Hard_Refresh) [9]

Project 1

- 1- Let's create an enumeration list of all the months of the year.
Notice what happens on the right column as soon as you type the keyword *enum*.
- 2- My first constant value will start at index 1 instead of zero.
- 3- I am calling my list ***MonthList***.
- 4- At the end (outside of the *enum* code block) I will display the following message using ***alert()***:
"My favorite month of the year is October"
Where October is accessed by using bracket syntax.
- 5- After you have finish coding, click on the **run** button to see your message on the browser.

Do this exercise on your own first before looking at my own sample.

If you are not sure on how to do it, go back a read the topic, or take a quick peek at my sample, then try doing it on your own. Avoid copying line by line. If you copy line by line, erase it at the end and try doing it again on your own.

Here's a link to my own sample code: *[10]*

jsplain.com/javascript/index.php/Thread/151-TypeScript-enum-months-of-the-year.

Questions:

- Did you notice the complexity of JavaScript?
- Isn't TypeScript much easier to write?

Project 2

Please clear the left column of your playground:

[TypeScript playground](#). [3]

On this next project, I am also reviewing variable declarations so that we keep those in mind.

- 1- Declare a **PriceList** enumeration with the following prices: "\$23.50", "\$11.00", "\$36.99".

By including the quotations marks, I am telling TypeScript to use these numbers as a string of characters since I do not intend to do calculations with them.

Also, index the first listed price as price 1.

- 2- Outside of the *enum*, declare a variable named **book1** with the following value *"TypeScript in Plain Language"*.
- 3- Finally create an alert that displays the book title and the price listed as "\$11.00". The result output should look like this:

"TypeScript in Plain Language costs \$11.00"

The alert code expression could look like the following:

```
book1 + " costs " + PriceList[2]
```

- 4- After you finish coding, run the program.

See my raw file on the following link: [11]

jsplain.com/javascript/index.php/Thread/152-TypeScript-enum-priceList.

Compare your code with its JavaScript equivalent. Which one would you rather write?

3- Math, logical and comparison

OPERATORS

Math operators

+ The *plus operator* works as follows:

When both operands are numbers, it adds the left operand to the right operand.

If one or both operands are of type *string*, it concatenates the two words or characters together.

Examples:

```
alert(2+3) ; // returns 5
```

```
alert("hello" + " world") ; // returns hello world
```

```
var a = 7;
```

```
var b = 9;
```

```
var c = a + b;
```

```
alert(c) ; //returns 16.
```

- The *minus operator* subtracts the right numeric operand from the left numeric operand.

Examples:

```
alert(4-3); // returns 1
```

```
var d = 10;
```

```
var e = 6;
```

```
var f = d - e;
```

```
alert(f); // returns 4
```

+= The *plus-equals* operator adds the left operand to the right operand and automatically assigns the total result to the left**

Examples:

```
var g = 10;  
var h = 6;  
g += h;  
alert(g) ; // returns 16
```

****Notes:**

This note applies to the following operators (some will be discussed ahead):

+=

-=

/=

***=**

Because all these operators are assignment operators, the left operand must be a variable, it cannot be a direct number

-= The *minus-equals* operator subtracts the right operand from the left operand and automatically assigns to the left.

Examples:

```
var j = 10;
```

```
var k = 6;
```

```
j -= k;
```

```
alert(j); // returns 4
```

/ The *forward slash* is the division operator. It divides the left operand by the right operand.

Examples:

```
var k = 12;
```

```
var m = 6;
```

```
alert(k / m); // returns 2
```

/= The *forward slash equals operator* divides the left operand by the right operand and automatically assigns the quotient to the left operand.

Examples:

```
var n = 12;
```

```
n /= 3
```

```
alert(n); // returns 4
```


% The *modulus operator* (fetches the remainder of the left operand divided by the right operand).

Examples:

```
var p = 24;
```

```
var q = 5;
```

```
var r = p % q;
```

```
alert(r); // returns 4
```

* The *asterisk operator* multiplies the left operand by the right operand.

Examples:

```
var s = 20;
```

```
var t = 3;
```

```
var u = s * t;
```

```
alert(u); // returns 60
```

******* = The *star equals operator* multiplies the left operand by the right operand and automatically assigns the product to the left operand.

Examples:

```
var v = 20;
```

```
v *= 6;
```

```
alert(v); // returns 120
```

Comparison operators

`=== !== < > <= >= true false`

In math we use the equals sign `=` to determine equality, but in programming the `=` operator has a different meaning.

In TypeScript the equals sign is an assignment operator: it assigns the right operand to the left operand:

```
var x = 12;
```

In the above example, 12 is given to x (stored in location x). This is an assignment, not a question to inspect equality.

To inspect equality, we use the *triple =* operator:

```
x === 12;
```

The above expression is not an assignment, it is a question and the question is:

“are the values in x and in 12 the same?” or, in this example, “Is the value contained in variable x, 12”?

Every time the TypeScript interpreter sees a triple equals (`===`), it always evaluates and replies with a boolean *true* or a boolean *false*.

What is a Boolean?

A Boolean is a result of an evaluation that has one of two possible value outcomes:

true or **false**.

In TypeScript, *boolean* (in lower case) is a data type just like *string*, *number*, etc.

The keywords *true* and *false* (in lower case) are permanently reserved words in TypeScript. We can literally write *true* and *false* without quotes because TypeScript knows what they mean. In fact, if we ever write "true" or "false" in quotes (or in uppercase), TypeScript will assume it is a string of characters and not a *boolean* value type.

Therefore, we could create a question on the console for TypeScript to answer (this question is a command to get a boolean result). Example:

```
x === 12;
```

I am assuming we have already declared variable *x* and it has some kind of a value in it. If the value is 12, TypeScript will return a boolean *true* because the statement is true. Else, it will return a boolean *false*.

Below please find a few more comparison operators and then some expressions to be evaluated by TypeScript.

Please remember, we are comparing the left operand, to the right operand.

Less than: **<**

Greater than: **>**

Less than or equals to: **<=**

Greater than or equals to **>=**

Not the same as: **!==**

Examples:

`12 < 10`; The boolean answer is *false*

`12 > 10`; The boolean answer is *true*

`12 === 12`; The boolean answer is *true*

`12 === "12"`; The boolean answer is *false**

`12 !== 13`; The boolean answer is *true*

`12 !== 12`; The boolean answer is *false*

`12 <= 12`; The boolean answer is *true*

*Note: You may have seen or used double equals == with JavaScript. In a different programming language that should be ok. However, in JavaScript the == operator is not recommended, and since any == or === written in TypeScript will be transcompiled to JavaScript in the same manner, it is my humble opinion to stick to triple === in TypeScript as well.

Logical operators

Logical AND: `&&`

Logical OR: `||`

Logical NOT: `!`

There are three basic logical operators in TypeScript.

1- The AND logical operator: `&&`

In an `&&` expression both operands need to be a boolean *true* in order to get a *true* output result.

Examples:

`(10 > 9) && (10 < 11) ;` the TypeScript answer is *true*
because *true* and *true* is *true*.

`(10 > 14) && (10 < 11) ;` the TypeScript answer is *false*
because *false* and *true* is *false*.

`5 === 7 && 5 === 5 ;` the TypeScript answer is *false*

`5 === "5" && 5 === 5 ;` JavaScript will answer as *false*. However, the TypeScript code checker will flag this operation stating that you cannot compare two different types of data.

`5 === 5 && 7 > 5 ;` the TypeScript answer is *true*

2- The OR logical operator: ||

The two vertical bars denote an OR logical operator. In US Windows keyboards you can access the `|` by pressing SHIFT and the last key on the right at the row starting with *qwerty*.

With logical `||`, one of the operands needs to be *true* in order to get a boolean *true* as the output.

Normally, TypeScript will not check the right operand in an OR statement when the left operand is already *true*. In programming “*lingo*” this is known as “*short-circuit*” and it is important to know this concept because sometimes it makes a difference in the way you have to write your code sequence.

As an example where short-circuiting makes a difference, if you are looking for a condition that includes either **A** or **B** or **A&&C**, it is best to write **A&&C** first, otherwise the program will exit when it encounters **A**. It will never see **A&&C** which might be the most correct outcome.

Examples:

`(10 > 9) || (10 < 9)` ; the TypeScript answer is *true* because the first expression is *true*.

`(10 < 9) || (10 > 9)` ; the TypeScript answer is *true* because the second expression is *true*.

`(10 < 9) || (10 < 8)` ; the TypeScript answer is *false* because none of the expressions are *true*.

3- The NOT logical operator: !

The *not* ! operator is used to invert a statement (deny a statement).

In TypeScript (as in JavaScript), any data value is considered *true* when asked if a value exists, except for the following values:

The boolean *false*, the number *zero*, the value known as *undefined*, and the value known as *null*.

The following declarations are for testing purposes:

```
var x = 3;  
var y = 0;  
var z = "Tony";  
var a;
```

Looking above, is variable *x* *true* or *false*? It is *true* because it has a value. On the other hand, variable *y* is *false* because its value is *zero*.

Let me introduce a TypeScript method that we can use to find out whether a value in a variable is *true* or is *false*:

```
Boolean();
```

Remember, we are asking TypeScript to tell us if the statement is *true* or *false*. We are not inspecting the variable to get the value itself. It's all about the statement we place as an argument to the `Boolean()` method. The statement is either *true* or it is *false*.

For example, in the expression `Boolean(x)` ; we are asking the following question “*Is this true?*”, “*I say x exists as true, is my statement true?*”, and then TypeScript replies with *true* or *false*.

Examples:

`Boolean(x)` ; the statement is *true* (since variable x from the previous examples is 3)

`Boolean(!x)` ; the statement is *false* (since x has a value, *!x* is a *false* statement)

`Boolean(y)` ; the statement is *false* (y is 0 and zero is *false*)

`Boolean(!y)` ; the statement is *true* (y is 0 and therefore *!y* is a *true* statement)

`Boolean(z)` ; the statement is *true* (z is "Tony" and Tony is *true*)

`Boolean(!z)` ; the statement is *false* (since z has a value, the *!z* statement is *false*)

`Boolean(a)` ; the statement is *false* (*a* is undefined and we can't say it is *true*)

`Boolean(!a)` ; the a statement is *true* (*a* is undefined and therefore *!a* is a *true* statement)

There is no lab session for this topic

I am not including extra lab work for this topic because most likely you are already familiar with it from writing JavaScript. However, if you want to practice a bit, go to the [TypeScript playground](#) [3] and create your own variables (or use my examples), then use `alert()` to include a Boolean expression inside of it and see the results on the screen when you run the program. We will have a chance to review some of these concepts as we use them in future lab exercises. This topic is included here as a reference in case you need it when doing other exercises.

You can also copy the following raw file and test it on the playground: [12]

icontemp.com/typ/boolean1.txt

I have published an online exercise on the following link to test your knowledge of Boolean states: [13]

jsplain.com/javascript/index.php/Thread/109-Boolean-True-or-Boolean-false

The IF, ELSE IF, ELSE conditional statements

In programming, it is necessary at times to pick one path or another, based on a certain outcome.

If the value is less than 12, display "*Good morning!*", **else if** the value is less than 18, display "*Good afternoon!*", **else**, display "*Good evening!*"

In the above sentence we have an **if** condition clause, an **else if** condition clause, and a final **else** statement. The **else** does not have a condition for displaying a message because it is the default outcome when none of the other conditions apply.

The **else if** is an intermediate condition and it is only checked when the initial **if** does not apply.

We can have many **else if** conditions, but we can only have one if and one else.

Both **else if** and **else** are children of **if**.

Therefore, "**if**" is the only conditional clause that can exist by itself.

Note: The reason why we should not use more than one *if*, is because each *if* will be treated as an independent code sequence, and the program will check each one of them, even when it encounters one that applies. This slows down the process and it may even give you unintended results. For that reason, any subsequent conditions (intermediate conditions) must be implemented in *else if* statements.

The code goes like this:

```
var time = 12;

if(time < 12){
    alert("Good morning!");
}
else if(time >= 12){
    alert("Good afternoon!");
}
else{
    alert("Good evening!");
}
```

(We will practice conditional statements on the next lab session)

Using the ternary operator as a conditional shortcut

The `?:` [ternary operator](#) [14] can be used to alternate between one condition or another.

Example:

```
condition ? first result : alternate result;
```

In the original example (from the previous topic), there were three possible outcomes, which, when using the ternary operator, can be written in the following manner:

```
condition ? result1 : second condition ? result2 : alternate  
result;
```

Applying the ternary operator to our example, we could write it as follows

```
1 var time = 12;  
2  
3 time < 12 ? alert("Good morning!") : time >= 12 ?  
4     alert("Good afternoon!") : alert("Good evening!");  
5  
6 // it displays: "Good Afternoon!"
```

Fig 3 (script was split between lines 3 and 4 to keep it shorter in the book)

Raw file: [15] icontemp.com/typ/ternaryOP.txt

Which method should you use?

It is a matter of preference since they both work the same way. This is just a shortcut for the *if*, *else if*, *else* conditional statement.

There is another optional design called a *Switch conditional statement* and we will look into it next.

The Switch statement

Sometimes we need to go beyond two or three conditional options and the “*if, else if*” mechanism becomes tedious to write and confusing to inspect later on. In this case, it may be more practical to write a *switch conditional statement*.

The switch statement is written in the following manner:

```
switch (expression)
{
    case 'condition to match expression':
        alert(message) ;
        break;
    case 'condition to match expression':
        alert(message) ;
        break;
    default:
        alert(message) ;
}
```

At runtime, all cases remain switched off, except for the one that matches the switch expression (I mean the case having a boolean *true* outcome when compared to the *switch expression*).

Please look at the following link for a code example and an explanation of a switch conditional expression: [16]

jsplain.com/javascript/index.php/Thread/153-TypeScript-switch-conditional-statement

LAB WORK: practicing conditional statements

Please use the [TypeScript Playground](#) [3] for the following exercises.

Exercise 1:

Create an *if, else* conditional statement that checks whether 7 is greater than 3, and outputs one of the following messages:

*“the first number is **greater** than the second number”*

or the following output otherwise (*else*):

*“the first number is **less** than the second number”*.

See my result here: [17] icontemp.com/typ/if1.txt

Exercise 2:

Change the expression by reversing the operands so that you can get the second output instead of the first one.

Exercise 3:

Based on exercise 1, replace your “*if, else*” statement with a *ternary expression*. Check the explanation on ternary expressions to see how it is done.

See my result here: [18] icontemp.com/typ/if2.txt

Exercise 4:

- 1- Create an enumeration list named ***weekDays*** to list *Sunday*, *Monday* and *Tuesday* where Sunday is item number 1 (remember, it defaults to zero).
- 2- Then, initialize a string type variable named ***nthDay*** and store the cardinal number "third" in it. This will be our matching test data.
- 3- Now, create a switch conditional mechanism that checks the data in ***nthDay***.
For each case (first, second, third) it should alert a message like for example, the following:
"The first day of the week is Sunday" or *"The third day of the week is Tuesday"*, etc.
However, do not write the message explicitly in English for each output. Instead, use the variable *nthDay* and the enum *weekDays* as the two expressions to be evaluated in the message itself. Example:
`"The " + nthDay + " day of the week is " + weekDay[1]`
- 4- As the default output, write an alert like for example
"That day is not on the list"

Below, please find my own version of this exercise: [19]

jsplain.com/javascript/index.php/Thread/157-TypeScript-enum-days-of-the-week

I hope this has been fun. These tools will come handy as we progress in our lab exercises.

Let's now discuss objects, arrays and loops.

4- Short introduction to OBJECTS, ARRAYS and LOOPS

This is a very short introduction to objects, arrays and loops. I will expand these concepts on the second part of this book when we get to *interfaces* and *classes*. This introduction is necessary so that I can use this material on my examples as the book progresses. If you have written JavaScript in the past, you will not see anything new here, but it may be worth skimming through it anyway.

There is no lab session in this chapter. However, it may be beneficial to fire up your playground and try the given examples.

<http://www.typescriptlang.org/play/> [3]

What is an object in TypeScript?

The word object is deceiving because almost everything we create in TypeScript is actually an object. However, in the JavaScript “lingo”, an object is a **customized data type** used to store a collection of items we normally refer to as *properties* or *fields*. These items are stored in **key/value** pairs and we address the value by calling its key.

Here is an example of an object assigned to variable *myWardrobeCabinet*:

```
var myWardrobeCabinet = {  
  topDrawer: "Towels",  
  middleDrawer: "T-shirts",  
  bottomDrawer: "Socks"  
};
```

If this code looks new to you this is what you need to know:

- There are several *key/value* paired items assigned to a variable named *myWardrobeCabinet*. This *variable* becomes the name of the *object* and this object becomes a data type named *myWardrobeCabinet*.

- The items are written inside of a pair of curly braces `{ }` and they are separated by commas except for the very last one.
- The *key* is the word located on the left side (like for example, *topDrawer*), and the *value* is what's on the right side after the colon `:` which is used to separate the *key* from the *value*. The colon represents *inheritance*.

Then we can address each item with *Dot Syntax* as shown on the example below.

To express in code the following sentence -- “*My top drawer contains towels*”, we write it in the following manner:

```
"My top drawer contains " + myWardrobeCabinet.topDrawer;
```

The *myWardrobeCabinet.topDrawer* gets evaluated at runtime to the value "Towels" .

We can insert the above statement in an *alert()* method like on the following example:

```
alert("My top drawer contains " + myWardrobeCabinet.topDrawer);
```

If you need to expand your knowledge in JavaScript objects, please refer to my eBook [JavaScript Objects Functions and Arrays Explained](#).^[20]

It is a short reading with lots of practicing exercises.

Let's now introduce loops and then we will do some lab work.

The For In loop

The purpose of a loop is to repeat an instruction a number of times until a certain condition is met. The **for in** loop works great with objects such as the one just described on the previous section.

This loop scans the object and uses the **in** operator to grab the item key name for each loop cycle. Knowing this, we can then program the body of the loop mechanism to do whatever we want to do with the object. The basic layout goes like this:

```
for(var item in objectName)
{
  // do something with the object;
}
```

The prefix *var* can be replaced with **let**, and as you will see later, in TypeScript *let* is a best option for loop counters.

The term *item* is a temporary variable acting as a placeholder for the *key* name fetched during any given loop cycle. The word *item* is arbitrary since you can call it anything.

When the loop starts, it picks and holds one of the object's properties for the length of the cycle.

The loop cycles for as many properties as the object contains.

This next script shows a way to display the items from the object I have declared earlier:

```
for (var item in myWardrobeCabinet )
{
    console.log(myWardrobeCabinet[item]) ;
}
```

Notice that I'm using **console.log()** instead of *alert()*. This is because I do not want to get a popup alert box for each item of the object. I will explain in a second how to visualize the result on the screen when using *console.log* in the playground.

Notice how I addressed each *value* location. In this case, I could not use *Dot syntax* because in Dot syntax, the values are hardwired, and in a loop, the values are dynamic. When it comes to dynamic instructions (the ones we do not know the value until the program runs), we need to use the square brackets.

myWardrobeCabinet[item] means to evaluate the value in the item location at the time of the loop cycle. Remember, the *item* variable is the placeholder for the key name.

A note about **var** in loops

I have not cover **var** vs **let** as variable initialization keywords. I am waiting to cover functions first. However, when it comes to variables used in loops, such as **item** in my previous example, it is best to use *let* as an initialization prefix instead of using *var*. The keyword *let* is new and not yet supported across browsers, but since we are using TypeScript, we can safely implement it. The keyword **let** tells the TypeScript compiler that this value is unique to this loop. In other words, do not carry the counter value forward because it may interfere with the initialization of another loop somewhere further down the pipe. Initializing variables with **let** gives them limited scope (limited to the `{ }` brackets). I will show some examples after we cover functions.

CONSOLE.LOG: How to visualize the result on the screen

The `console.log()` method was specifically designed for testing code and it only works if you are using a test [Console](#) [21].

How do we access the Console from the playground?

When you *run* your program on the TypeScript playground, you will get a blank screen if your output comes from a `console.log` and this is because the `console.log` method cannot be viewed on a regular HTML page.

You will need to access the Console that comes with your browser.

Remember that `console.log` is only used when we are testing code. I am using Google Chrome and I access the Console by clicking on the following key combination:

For *Windows* or *Linux*: **CTRL+SHIFT+ j**

or if on a *MAC*: **Command + Option + j**

- 1- Run the program (from the TypeScript playground).
- 2- When your browser's page opens up (blank or not), access the Console by using the shortcut given above.
- 3- If there is a `console.log` result, it should now be visible to you. In our case, based on the *for in* loop I have written on the previous topic, you will see a list of all the items in object `myWardrobeCabinet`.

What is an array?

Just like in a *key/value* paired object, an array is a collection of items. The difference is that in an array, the data values are programmed sequentially (an array of values), instead of by key name. There is no key name pointing to the values. Each value is assigned to an index number, starting from zero.

Example of an array:

```
var myColors = ["blue", "orange", "red"];
```

The *square brackets* represent an array (as opposed to the *curly braces* for objects).

We can add a fourth item to the end of the array by using the method **push()**

```
myColors.push("violet");
```

However, if I try to add a new item of type *number* to the above array, I will get an error from the TypeScript interpreter.

Consider the following example:

```
myColors.push(33);
```

This is because TypeScript implicitly assigned myColors to a *string type* array, based on the items I first introduced when I created the array. (We can of course place the number 33 in quotes making it a *string* of characters and that will work, but it becomes a *string* and we can't do any calculations with it).

In JavaScript, arrays may contain mixed data types. This is not allowed in TypeScript **unless we explicitly declare an array of type *any***:

```
var myColors:any = ["blue", "orange", "red"];
```

Now we can add any type of item to it.

I will come back to arrays later in the book. If you want to explore arrays and array methods in detail, please refer to [this book](#). [20]

How do we display a specific value from the array?

To display data from an array we use bracket syntax and pass in the index number:

```
myColors[2];
```

Since the array locations start at the count of zero, item 2 is the third item, which in our example evaluates to “red”.

If you want to test this, you can include it in an *alert()* method to see it on the screen:

```
alert(myColors[2]); // prints red
```

```
alert(myColors[0]); // prints blue
```

In addition, to change an item or add a new item (if we know the index of the next available position), we can do it in the following manner:

```
myColors[0] = "brown"; // changes the first item from red to brown
```

```
myColors[3] = "yellow"; // adds one more item to the array because 3 is the next available position. This is the same as
```

```
myColors.push("yellow");
```

The for loop

In order to display all the items from an array, we can use a **for** loop.

We have seen how with *key/value* pairs, we used a **for in** loop to scan the object and display the items.

With *arrays*, we use the regular **for** loop:

```
for(let i = 0; i < myArray.length; i++)  
{  
  // do something here;  
}
```

- The *let i*, could have been *var i*, but in TypeScript we should finally use **let** which gives it a local execution context and avoids possible conflicts ahead. The variable *i* is being used as a cycle counter.
- **.length** is a property of the array. It keeps track of its length. In the *myColors* example, the original length is 3. We use *.length* instead of 3 in order to make it portable. This means that if the array size ever changes (which is permissible in this language) the length used in the loop will adjust itself.
- **i++** is a shortcut for **i = i + 1**. It tells the interpreter to increment the value of *i* by one unit at a time.
- The loop starts at zero, and it processes the code block for as many times as the length of the array. The **i++** increment happens at the end of each cycle and eventually the value of *i* reaches the value of *length* and the loop stops.

Here is how we would use a *for loop* to display all the items of the array *myColors*:

```
for(let i = 0; i < myColors.length; i++)  
{  
    console.log(myColors[i]);  
}
```

Again, the `myColors[i]` represents each location of the array for each cycled position starting from zero, and when evaluated, it give us the item name (the data pointed to by the index position).

Since we are using a *console.log*, we need to access the Console. See the topic “*How to visualize the console.log result on the screen*” posted a few pages earlier. As a refresher, you call the Console after you run the code on the TypeScript playground.

When the blank page displays,

On Windows systems, press **CTRL+SHIFT+j**

On a MAC, press **Command + Option + j**

Initializing arrays in TypeScript

So far, we have seen how to initialize an array and assign values to it at the same time. However, we can also initialize an array without having to add any values at first.

Normally in basic JavaScript, we initialize an array in the following manner:

```
var myArray = [];
```

That works fine in TypeScript as well. **In TypeScript, that kind of syntax declares an array of type *any*.**

A type *any* array is one that accepts mixed data types.

In TypeScript, we also have the option of initializing arrays in the following manners:

```
var myArray: number[] = new Array();
```

```
var myArray2: string[] = new Array();
```

```
var myArray3: boolean[] = new Array();
```

and

```
var myArray4: any[] = new Array();
```

After creating the array, we can *push* values into it.

As an example, let's push() a few items into *myArray4*:

```
myArray4.push(33, "Tony", true);
```

If you `alert(myArray4)`;

you will see that *myArray4* contains three mixed item values: *33, Tony, true*

In Summary:

When we create an array and assign values at the same time, TypeScript *implicitly* assigns a *data type* to it based on the values we include in the declaration.

We can also explicitly create arrays and add the items later in which case we need to declare the *type of data* the array should contain.

To *iterate* over an array it is common to use a standard *for loop* as opposed to a ***for in*** loop which is used with *key/value* paired objects.

Arrays are indexed numerically and sequentially, but we can also jump index positions by placing items in specific locations. In this case, the empty locations will be filled with data of value ***undefined***.

In TypeScript we can also use a technique called ***de-structuring*** to split the items from an array into separate entities. This will be covered on the next topic.

Using de-structuring to individualize data from arrays or objects

TypeScript introduces a way to grab the items of an array or an *object literal* and assign them to individual variables.

Take, for example, the following array:

```
var myArray = ["red", "orange", "blue"];
```

We could manually bind each item to a separate variable as shown next:

```
var x = myArray[0]; // it assigns "red" to x
var y = myArray[1]; // it assigns "orange" to y, etc....
```

TypeScript has a **de-structuring** feature we can use as a shortcut for that purpose:

```
var [x,y,z] = myArray;
```

That's it! Now *x* is bound to red, *y* is bound to orange and *z* is bound to blue. This operation sequentially binds the data in each index to the variables on the left.

If you look at the final JavaScript transpiled version, you will notice how the TypeScript compiler has written the code in the same fashion as what I did on my first example above.

De-structuring can also be used for object literals.

```
var myObject = {firstName: "Mary", lastName: "Smith"};
var {firstName,lastName} = myObject;
alert(firstName);
```

The variables on line 2 were assigned to the object properties **based on matching names**. If you reverse the names, you still get the correct information in each one of them. If you use other names, such as for example, xyz, you will get an error. Also, notice the curly braces. The braces need to match the data structure you are attempting to de-structure.

Note:

- When it comes to de-structuring arrays, the sequence of presenting your variables is important because the values will be assigned from left to right.
- In objects, the variable declaration sequence does not matter. What matters is the name given to each object property, which should match the property name of the data source we are binding to the variable.

Next, let's look at another important technique: **spreading** items from an array.

The Spread Operator

The spread operator is represented by the three-dot ellipsis (...).

The best way to describe the spread operator is by giving you an example.

Let's assume the following two arrays:

```
let myColors = ["red", "white", "blue"];  
let myOtherColors = ["Green", "Black", "Brown"];
```

As seen before, if we want to add a new item to array `myColors` we can use `push()`:

```
myColors.push("orange");
```

Notice how the argument for `push` must be a string value (or a number, if applicable).

How would I write my code if I wanted to push the whole ***myOtherColors*** array into array ***myColors***?

In JavaScript ES5 it would take several steps to do the operation, but in Typescript all we have to do is to use the spread operator to break the source array into its individual components.

Here's how it is done (notice the three-dot ellipsis as a prefix to the source array):

```
myColors.push(...myOtherColors);
```

Although we did not include a string value as an argument for `push()`, it will work because the TypeScript interpreter will evaluate the array and submit each individual string item to `push`.

That saves us a lot of work. Spread operators can be used whenever we need to submit the contents of a complex data structure as individual data components.

5- Introduction to FUNCTIONS

About functions

There is no lab session in this chapter. However, it may be beneficial to fire up you playground and try the given examples.

A function is used to store code that performs a specific task. This task might consist of many lines of code stored in the function's body. We can later invoke the function every time we need to run the code. You can immediately see the benefit of functions since it only takes one line to invoke the function.

In other words, functions are "self-contained" *modules* of code that accomplish a specific task. (Speaking of *modules*, later we will also learn how to create modules that may contain several related functions).

Sometimes functions accept external data to be internally processed, and then they "return" an output result. Other times a function has a specific job to do and no external data is necessary.

Once a function is written into code, it can be used repeatedly (we instantiate the function) by merely invoking the function. Functions can also be invoked from inside of other functions.

Here is an example of a simple function. This function does not take any data input. Its purpose is to accomplish a task whenever we invoke the function:

```
function greeter () {  
    alert("Hello world!");  
}
```

If you write this function on the TypeScript playground and run the program, nothing will happen. This is because a function is a *potential action*, but it only becomes an action when we call/invoke it (bring it to action). *Calling* a function or *invoking* a function means the same thing.

To call a function we just write the name of the function and add a pair of parentheses as a suffix (one word):

```
greeter();
```

Now, when you run the program, you will see the "Hello world!" message on the screen.

If your intention is to feed data into the function so that it gets processed by the function, you need to add one or more input parameters inside of the parentheses of our function signature (the top line). There are other ways to do this, as we will see later:

```
function greeter (input1) {  
  alert("Hello " + input1);  
}
```

Notice how I took advantage of *input1* and used it on my output message.

Now we call the function again and *pass in* my name (or yours) as a *string* value:

```
greeter("Tony");
```

The output result is "Hello Tony".

Notes:

Make sure "Tony" is wrapped in quotes because it is a string value. If you don't wrap the input *argument* in quotes, the interpreter will assume that it is a variable, and it will look for such variable in your code.

As discussed previously, numbers will be passed into the function without being wrapped in quotes. This works because numbers cannot be the name of variables.

Another thing to keep in mind as far as terminology is concerned, is that an ***argument*** is what we call the data being passed into a ***parameter***. In other words, functions have *input parameters* and parameters accept *arguments*.

Making sure the function input data type is validated

What happens when I call the function and pass in the number 3?

```
greeter(3);
```

It displays "*Hello 3*" which does not make any sense.

TypeScript helps us create rules for the type of data we intend to use.

We can actually write a list of rules for our code and store it in a data type called *interface*.

Interfaces will be covered later in the book. Right now, let's just stick to functions since we are building our TypeScript awareness from the ground up.

In order to ensure that our function's input is always of type *string* (if that is our intention), we can program the function signature this way:

```
function greeter (input1:string) { . . .
```

Now, if we ever call the function and pass in a *number*, TypeScript complains because it is the wrong type of argument.

The return value

Each function needs to *return* a value. In our previous example we did not *return* anything, we only displayed a message on the screen monitor. Functions have a tremendous advantage over code written in the global environment (see my book [JavaScript Objects Functions and Arrays Explained](#) [20] for a greater detail on that topic).

The ***return*** mechanism is unique to functions since we cannot use it anywhere else but in a function.

The purpose of the ***return statement*** (correct terminology) is twofold:

- 1- To export a value from within the function in its raw format
- 2- To terminate the function execution (it acts as a *break*).

When the function *call* ends its process (and it always ends by the execution of a *return* which acts as a break), the execution context is cleared from working memory. Since having a *return* is very important to JavaScript in order to wipe the function execution from working memory, every JavaScript function invocation returns automatically if there is no explicit *return* stated.

The automatic *return* generated by the JavaScript interpreter is always the value of *undefined*, which is of type ***undefined***. That is what happened in our little *greeter* example, an automatic return was generated without being noticed by us.

Now, TypeScript was created to prevent ambiguity when it comes to writing JavaScript code.

The TypeScript *type checking* helps us become more conscious and better programmers. In TypeScript, we can specify what kind of *return* data type we intend to use in the function, including the usage of **void**, which means that no *return* is intended from the programmer's part and therefore JavaScript should do its *return thing* automatically.

Using an explicit return type declaration helps you and other programmers to inspect, or to create, your 30000 lines of code project.

Remember, we only want to explicitly use *return*, if we need to export the *raw value* from the function. When our purpose is just to display the output, then we tell TypeScript that our return type is *void* because we do not need the raw data:

```
function greeter ():void{  
    alert("Hello world!");  
}
```

Another example:

```
function greeter (input1):void{  
    alert("Hello " + input1);  
}
```

Using the return value

Whenever we display data on the screen (by using *alert*, *console.log*, *confirm*, etc.), the data type is always converted to *string* before it outputs, even if it is a number (**screen values are always a string type**).

On the other hand, when we need the *raw data* for further processing, then we have to *return* it, which is a way of exporting the data out of the function.

Note: We can also assign the data to an external variable in order to export it but right now, we are talking about the *return mechanism*.

Let's see how we can use the ***return*** value by modifying our example.

If, on my example, I replace the "***void return*** type declaration", with a ***string return*** type, I immediately get a warning from the TypeScript interpreter, stating that it is now expecting an explicit return statement of type string inside of the function body, and there is no such implementation written, as you can see below:

```
function greeter (input1):string{  
    alert("Hello " + input1);  
}
```

In order to solve the above problem, I need to include a return statement inside of the function.

Then, I will optionally catch the raw returned value externally, as the function returns.

The catching is done by assigning the returned value to an external variable when I call the function (see the second line from the bottom on this example):

```
function greeter (input1):string{  
    return "Hello " + input1;  
}  
  
var result= greeter("Marie");  
alert(result);
```

Of course, the external variable is optional and TypeScript will not care whether we catch the value or not. TypeScript only wants to make sure that my function signature is consistent with what I program in the body of the function.

By the way, when we declare the external variable (*result*), we may also add a *type classification* to it. Adding types is a good practice because it triggers the TypeScript editor's error correction when we attempt to call the function 300 lines later. We could declare variable *result* like this:

```
var result:string= greeter("Marie");
```

This way, the TypeScript interpreter (and any human programmer inspecting the code later) will know that the function is supposed to return a string value.

Setting default parameter values

Let's look at another example. This one will have multiple input parameters:

```
function addMe (input1:number, input2:number) {  
    alert(input1 + input2);  
}  
addMe (5, 6);
```

Declaring functions with type assignments have several benefits.

a) Type safety

In the above script, we see a function that takes two arguments via its parameters and then displays the result of their sum. These two parameters take arguments of type *number*. If we attempt to pass in any other type of arguments when we call the function, the TypeScript interpreter red flags it.

b) Making sure all parameters are active when the function is called

When I call the function *addMe* and pass in numbers 5 and 6, it results in a display of 11, and that is fine.

However, what if I call the function and pass in just the number 5?

The TypeScript interpreter will flag my function invocation because one of the parameters is missing an *argument*. (We feed arguments to parameters).

In order to prevent errors of omission when we invoke functions, **TypeScript introduces the ability to set default values to the input parameters of a function**. This can be helpful in case we miss one of the input arguments when we call the function, which normally results in error for TypeScript, or *undefined* for JavaScript. (Besides error prevention, this feature also expands the capability of function design by making it easier to write certain odd implementations).

To provide a default value to parameters, we do it in the following manner:

```
function addMe (input1:number=0, input2:number=0) {  
    alert(input1 + input2);  
}
```

Now, when I call the function with just one argument (or no arguments at all) it still processes without errors:

```
addMe (5) ;
```

If you are trying this example along with me, notice how much more work we would need to do to accomplish this in common JavaScript:

```
function addMe(input1, input2) {  
    if (input1 === void 0) { input1 = 0; }  
    if (input2 === void 0) { input2 = 0; }  
    alert(input1 + input2);  
}
```

Please note:

When mixing default parameters with regular parameters, make sure your default parameters are programmed to the right of the regular parameters because, when we invoke the function, the arguments are applied sequentially to the parameters.

Optional Parameters

Besides introducing default parameters, TypeScript introduces *optional* parameters.

An optional parameter is one that is included in the function signature, but it is not required.

Whenever we apply an argument to an optional parameter, it triggers a different part of the script.

Further explanation:

Normally we get an error, when we write a parameter in JavaScript and then *call* the function but do not pass in an argument to the parameter.

To prevent this problem, TypeScript allows for the creation of *optional parameters*.

An optional parameter is followed by a question mark “?”.

Like in *default* parameters, just make sure your optional parameters are the last parameters to the right, otherwise the option feature becomes useless when you call the function because arguments are always matched to its parameters in a left to right sequence.

Here is an example (look for the ? question mark on input3):

```
1 function addMe(input1:number, input2:number, input3?:number):void {  
2     if (input3) {  
3         alert(input1 + input2 + input3);  
4     } else {  
5         alert(input1 + input2);  
6     }  
7 }  
8
```

Fig 4

Raw file: [22] icontemp.com/typ/func3i.txt

(The *void* is just to say that I do not require a *return* statement in this example).

Now, in this example, when we can call the function with just two arguments, it triggers the *else* portion of the conditional statement:

```
addMe (3, 5) ; // it displays 8
```

When we invoke the function with three arguments, it triggers the *if()* conditional output:

```
addMe (3, 5, 6) ; // it displays 14
```

NOTE: what does the **if(input3)** mean? This is a Boolean question and the answer is either *true* or *false*. In other words, “does *input3* have a value other than *false*, *zero*, *null* or *undefined*?” if so, the answer is *true* and the code block {} is executed. Otherwise, the *else* code block is executed.

A REST parameter accepts ‘all the other arguments’

What is a rest parameter?

The rest parameter syntax allows a function to represent an indefinite number of arguments as an array of arguments. To remember the name “rest” just think it this way, *here is your first parameter, here is your second parameter, and here are the rest of the others*.

A function with a rest parameter is also called a [variadic function](#) [23] in other languages.

The *rest* parameter needs to be the last option in the parameter signature layout (or the only one).

To indicate that a parameter is a *rest* parameter we prefix it with **three dots**.

Here is an example where *c* is a rest parameter:

```
function test(a,b,...c) {  
}
```

Parameters *a* and *b* are regular parameters. Then, parameter *c* is an array of an unknown number of arguments. (Remember, *arguments* are the data being passed to *parameters*. Parameters are the variables accepting arguments).

Now we can invoke the function with at least three arguments, or perhaps more:

```
test(2,5,7,9,1);
```

Arguments 2 and 5 are assigned to parameter *a* and *b*. The rest of the numbers are assigned to an array mapped to parameter *c*.

Additionally, we can have just one parameter and make it a rest parameter because it is also the last one:

```
function myFunc(...x) {  
}
```

A rest parameter is an array

Since the *rest parameter set* is a real array, we can actually take advantage of all the methods available to array objects (see my [eBook \[20\]](#) on array methods).

Here is an example of a finished function with parameter *c* as a rest parameter:

```
1 function test(a, b, ...c) {  
2  
3     console.log("a= " + a);  
4     console.log("b= ", b);  
5  
6     for (var i = 0; i < c.length; i++) {  
7         console.log(c[i]);  
8     }  
9 }  
10  
11 test(3,5,8,9);
```

Fig 5

Raw file: [24] icontemp.com/typ/restParam1.txt

Side note: lines 3 and 4 are just two different ways of logging similar expressions. On line 4, remove the space between **b=** and **"**.

Now, when we call the function like this: **test(3,5,8,9);**

It displays **a=3 b=5 8 9**

Another example

This next function uses only one parameter, which happens to be a *rest* parameter.

This function places all the arguments passed into the parameter into an array. Then, it calculates the sum of all its arguments. You can pass as many arguments into the function as you want, and the loop will make sure all arguments are processed.

```
1 function addTotal(...x) {  
2     var total = 0;  
3     for (var i = 0; i < x.length; i++) {  
4         total += x[i];  
5     }  
6     return total;  
7 }  
8
```

Fig 6

Raw file: [25] icontemp.com/typ/restParam2.txt

I can now assign a *function call* to a variable and then pass in a bunch of numeric arguments. Then, in my example, when I alert the result it displays 32:

```
var result = addTotal(3,5,2,4,7,8,1,2);  
alert(result);
```

Notes:

The variable *total* was declared inside of the function. This variable is not seen from the outside. When the function returns, the value of the variable is reset. This is another advantage of functions because we can privatize variables, which remain in separate contexts as we call the same function. After the function execution, these variables are cleared from memory. I will cover private variable in more detail later on.

The **for** loop is using the `+=` operator to add the value on the right to the variable on the left.

At the end, the function returns the final value of variable *total* and resets *total* back to zero. If you happen to catch the returned value from the outside, you can reuse it. If you don't catch the return value, it will be lost in forever.

Questions with further explanation:

a) Which *data type* is assigned to a rest parameter by TypeScript?

A rest parameter must be of an *array type*. This is done automatically. If we try to change the data type of a rest parameter like in the following example:

```
function addTotal(...x:number) {
```

We get a **red flag**.

c) In our example, how do we assure that, when we invoke the *addTotal* function, all arguments for the rest parameter are numeric?

On our previous script, function *addTotal* does not specify what kind of arguments it accepts and this could lead into an implementation error.

For example, we could pass in "blue", "red", "green" and the function would gladly process it like "*blueredgreen*".

Since the purpose of the function is to add numbers, we should make sure that the rest parameter only accepts numbers. The way to do it is by adding the data type to the parameter, not as a number, but **as a numeric array**:

```
function addTotal(...x:number[]) {
```

From this point on, only numbers will be accepted when we call function *addTotal*.

ARROW functions – an introduction

Note: If you are new to programming, you may find this information confusing. Please read on because the new terms used here will be explained in more detail as we move along.

An *arrow function expression* is a shortcut for a function expression (which is an anonymous function assigned to a variable. See example below).

An *arrow function expression* in TypeScript is not bound to the local “this”, which is a pointer to the parent object. In fact, an arrow function in TypeScript does not have a “this” of its own at all (as we will see later) and we need to be careful when implementing a “this” in arrow functions since TypeScript will point it to an execution context that may be different than what we intend it to be.

A regular function expression goes like this:

```
var squareMe = function (x:number) {  
    return x * x;  
}
```

We call it "anonymous" because it has no name of its own. It is often assigned to a variable, which acts as the function's name. This is just another way of declaring functions. Both methods, this one and the previously discussed method, are common. Programmers use one method or the other. The reason to use anonymous functions is by preference or by necessity. The necessity part comes from the fact that anonymous functions have great flexibility - they can be embedded into existing code.

The **arrow** function expression is something new to JavaScript. It is a shortcut to represent an *anonymous* function. Aside from being a shortcut, an arrow function offers some advantages (and some pitfalls as well).

To write the above function as an *arrow* function we would do it like this:

```
var squareMe = (x:number) => x * x;
```

If you are trying these examples in the *playground*, watch how the interpreter converts the arrow function back to a regular *function expression* when transpiling to JavaScript. This is because arrow functions are not yet accepted in most browsers. However, read on because there are some useful implementations when using arrow functions in TypeScript.

What has changed on the above example?

The *function* keyword is missing, the explicit *return* statement is missing and so are the curly braces.

If you add an explicit *return* statement or any other executable code, you will need to add the curly braces back.

Example:

```
var squareMe = (x:number) => {return x * x;}
```

You may also see arrow function shortcuts looking like the one in the following example (no parentheses for the parameter):

```
var someVar = x => x + 3;
```

This is common when there is only one parameter (*x* is the parameter in our example). When we enter more parameters, we need to put the parentheses back.

On the other hand, if there are no parameters in the function, you may find the following shortcut:

```
var someOtherVar = () => "testing 123...";
```

When invoking function *someOtherVar()*, it returns *testing 123...*

Arrow functions are also known as [lambda expressions](#) [26].

About the *THIS* in arrow functions

a) Arrow functions have no "this"

TypeScript does not allow a direct link to the "*this*" keyword in an *arrow function*.

What is "this"?

In general, the "this" keyword is a placeholder for the object that owns the expression. However, there are exceptions based on the context at the time of calling.

Contrary to a regular function expression, in an arrow function there is no "this" built in.

Consider the following script to see how there is no "this" in a TypeScript arrow function:

```
var myFunc = ()=>{  
    alert(this);  
}
```

The following is the JavaScript version of the above TypeScript arrow function:

```
var _this = this;  
var myFunc = function () {  
    alert(_this);  
};
```

Notice how TypeScript changes the term "*this*" to "*_this*" and then it creates a global variable named "*_this*" which points to the global "*this*".

When we invoke function *myFunc()* we get object **Window** as the displayed result and this is correct since object Window owns this function. However, the fact that TypeScript changed our original "this" to "*_this*" and pointed it to the global "this" can be an asset or a source of trouble, depending what you are trying to accomplish.

We need to know when an arrow function is helpful, as well as when it may get us in trouble.

Consider the following object containing a regular arrow function inside it:
(Please test it on your playground)

```
1 var x = "I'm the global x";
2
3 var outerObject = {
4
5   x: "I'm the object x",
6
7   someFunc: ()=>{
8     var x= "I'm the local x";
9     alert(x);
10  }
11
12 };
13
```

Fig 7

- 1- Get the raw file (it also includes the code line stated on step 2):

[27] icontemp.com/typ/innerX1.txt

- 2- Call the function by running the program:

```
outerObject.someFunc();
```

Which x do you think will show on the screen?

(Try it on the [TypeScript playground](#) [3]).

The `alert()` method will display the inner x: *"I'm the local x"*.

That is because the program searches for variable x from inside out, and it uses the first variable x that finds. This is correct.

What if, we actually wanted the arrow function to access the property *x* of object *outerObject*, instead of its own *x*?

The normal way to access the property *x* from object *outerObject* is by writing it as

```
alert(this.x);
```

Then (normally), the "this" is replaced by the object name at runtime and the value "*I'm the object x*" is displayed. However, TypeScript arrow functions will fail when we implement them this way because they do not recognize the keyword "this" and do something else instead.

Before we modify our original TypeScript code from the above example to include a "this" in its arrow function, please look at the current *JavaScript* version in your playground. It should look like the one in the following image:

```
1 var x = "I'm the global x";
2 var outerObject = {
3   x: "I'm the object x",
4   someFunc: function () {
5     var x = "I'm the local x";
6     alert(x);
7   }
8 };
9
```

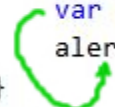


Fig 8 This is the **JavaScript version** of our first example. TypeScript replaced the arrow function for a standard function expression because browsers do not yet support arrow functions.

TypeScript replaced the arrow function with a regular JavaScript function expression. The output result for the *alert()* method is the value of local variable *x*.

As asked before, what if we wanted to access the object's x value, instead of accessing the function's x value?

Normally in JavaScript, if we intend to access the object's x property instead of the inner x variable, we need to use *this.x* instead of x. Here is what I mean:

```
1 var x = "I'm the global x";
2
3 var outerObject = {
4
5   x: "I'm the object x",
6
7   someFunc: ()=>{
8     var x= "I'm the local x";
9     alert(this.x);
10  }
11
12 };
```

Fig 9 added a "this" to x

However, the above script will not work. Notice what TypeScript does to its JavaScript version as soon as we add *this.x* instead of just x (see figure below):

```
1 var _this = this;
2 var x = "I'm the global x";
3 var outerObject = {
4   x: "I'm the object x",
5   someFunc: function () {
6     var x = "I'm the local x";
7     alert(_this.x);
8   }
9 };
10
```




Fig 10 TypeScript adds a _this to JavaScript and points it to a global "this"

Because an arrow function in TypeScript does not recognize the "this" keyword, TypeScript automatically adds a fake *_this* to JavaScript (line 7) so that our manually written "this" does not conflict with the object's "this", and then TypeScript assigns it to a global "this" (see line 1). It would have worked fine if our intention was to grab the global x, but the x we want is the one inside of the object and therefore the result is incorrect.

TypeScript assumes that the programmer wants to access the global environment.

What can we do to access the object's x property from our example?

One thing we can do is to avoid using the arrow function in cases where we are going to use the keyword "this" to access the current object. If our intention is to grab the object's local property via one of its methods, a regular anonymous function is a better choice than an arrow function.

Another thing we could do is to hardwire the object's name to variable x in the *alert()* method:

```
alert(outerObject.x) ;
```

This second alternative will work but it is not portable.

In summary

Use a regular function expression if you want to access a local "this" in a single object literal.

However, when it comes to classes instead of single objects, TypeScript acts differently.

Therefore, what we may perceive as a flaw in TypeScript is actually made by design to solve certain advanced problems. We will get there soon.

Here's an example of using a regular function expression when we want to access the parent object:

Raw file: [28] icontemp.com/typ/arrowFuncThis.txt

Please note:

Arrow functions with a modified "this" have certain benefits that need to be explored. They work this way for a reason and we will be able to see a few examples ahead, but first, let's make sure we master the concepts we have just covered by exploring the subject a bit more.

b) An arrow function inside of another function

Let's look at another example to solidify the "this" vs "this" concept.

The following function includes an inner function that adds words or *sets of words* to build a complete string. This inner function is an *arrow function*. The variable seen on **line 1** is just for testing purposes and we will use it later.

Please examine the code before continuing. I will explain it below the image but it is best if you take a hard look at it before reading my explanation.

```
1 var word: string = "groundhog";
2
3 function stringBuilder() {
4     var word: string = "";
5     return newWord => {
6         return word += " " + newWord;
7     }
8 }
9
10 var addWord = stringBuilder();
11
12 alert(addWord("The quick"));
13 alert(addWord("brown fox"));
14 alert(addWord("jumps over the"));
15 alert(addWord("lazy dog"));
```

Fig 11

Raw file: [29] icontemp.com/typ/groundHog1.txt

Lines 3 through 8 comprise of a function named *stringBuilder*.

On line 4 there is an empty string variable. This variable will **accumulate** words to build a complete string at the end. This variable has the same name as the global variable on line 1, which is not a good practice. I have done this on purpose in order to test accessibility to it later in the exercise.

The reason for including the variable **word** (line 4) inside of function *stringBuilder* is to keep it private so that no one can edit it globally. Another reason could be to avoid possible conflicts with global variables. A third reason could be to improve memory management.

The function *stringBuilder* returns another function on line 5, which I have named ***newWord***. The reason for using an inner function is so that, every time we call the function to add new data to variable *word*, the variable *word* holds this data and it does not reset its value. In other words, if we only used one function, variable *word* will reset each time we called the function. The inner function allows for permanent storage as described next.

This is what happens:

For your convenience I have included the picture online:

[30] <http://www.icontemp.com/typ/pic11.jpg>

In order to access the inner function (*newWord*), function *stringBuilder* is assigned to variable ***addWord*** on line 10. This assignment is actually a *function invocation* and therefore, the value returned to variable *addWord* is the inner function in its explicit form.

Now variable *addWord* represents a function, the inner function.

This means that the private variable *word* inside of the outer function will remain in memory for the life of the program, because the inner function assigned to *addWord* has closure on the private variable *word*. To have closure on a variable is synonymous with keeping that variable in its scope until it is no longer needed (even after the outer function ends its process).

On lines 12, 13, 14 and 15, the function is invoked and several words are passed in. The inner function adds these words to variable *word*, which concatenates them into a whole string paragraph.

The top variable assigned to *groundHog* has nothing to do with this script but it will be important on our next example.

- Test the script on the *TypeScript playground*. It should work fine since we do not have any issues yet. Take a few minutes to study the code and see why and how it works. Notice how each time the `alert()` fires up, it adds a new word to the output.

Next, let's do an experiment: what would happen if instead of an outer function, we had an object, and the inner function was a member of the object?

Let's look at that possibility next.

c) An arrow function inside of a single object

Below you will see an *object literal*, which contains an internal method to build our string.

```
1 var word: string = "groundhog";
2
3 var stringBuilder = {
4     word: "",
5     newWord: (myWord) => {
6         return this.word += " " + myWord;
7     }
8 };
9
10
11
12 alert(stringBuilder.newWord("The quick"));
13 alert(stringBuilder.newWord("brown fox"));
14 alert(stringBuilder.newWord("jumps over the"));
15 alert(stringBuilder.newWord("lazy dog"));
```

Fig 12

Raw file: [31] icontemp.com/typ/groundHog2.txt

The *stringBuilder* is now an object instead of a function. It has an internal property called *word* on line 4, and on line 5, it has a method to build our string. This method happens to be an arrow function, and because we want to access the object's property *word*, we are using the "*this*" keyword as a placeholder for the object's name (and this is wrong as you will see next).

What happens when we run the script?

When we run this script, the JavaScript interpreter uses the external variable *word* and the first term that comes up in the output string is *groundHog*.

This is because TypeScript inserts an extra variable called *_this* on the JavaScript version in lieu of the regular "this" we coded on TypeScript. Then, TypeScript assigns it to the global "this" (outside of the object) and that creates an error because we really want to address the property *word*, not the global variable *word*.

What can we do?

Do not use an arrow function in single objects if you are going to use “this” to address a local property. However, if you instantiate the object from a *class*, it will work.

We have not yet covered classes so, if the next sample script looks confusing, please move on and return later after we cover classes.

This next topic is included here for future reference (when you review the book).

d) Using arrow functions in objects instantiated from classes

The following script is similar to the last example except for how the object was created. In this case, the object is instantiated from a class. It is not a single object created from scratch and therefore the `"_this"` is not placed globally; it is placed within the class itself.

Classes are covered on a later chapter. For now we are just concerned with implementation of `"this"` in arrow functions.

```
1 var word: string = "groundhog";
2 class StringBuilder {
3     word: string = "";
4     newWord = (myWord) => {
5         return this.word += " " + myWord;
6     }
7 }
8 var stringBuilder = new StringBuilder();
9
10
11
12 alert(stringBuilder.newWord("The quick"));
13 alert(stringBuilder.newWord("brown fox"));
14 alert(stringBuilder.newWord("jumps over the"));
15 alert(stringBuilder.newWord("lazy dog"));|
```

Fig 13 This class contains an arrow function. The `"this"` remains within the class.

Link to raw file: [32] icontemp.com/typ/arrowFuncClass1.txt

The method `newWord` (lines 4 through 6) successfully addresses the correct property `word` (line 3) when called from the instantiated object `stringBuilder` (which are lines 8, and then 12,13,14,15). This is because TypeScript redirects the arrow function's `"this"` to the class' `"this"`, as opposed to when an object is created without using a class.

Test it yourself and look at the JavaScript version to see where the interpreter declared the `_this` variable.

Below, please find an image of the JavaScript version created by TypeScript:

```
1 var word = "groundhog";
2 var StringBuilder = (function () {
3     function StringBuilder() {
4         var _this = this;
5         this.word = "";
6         this.newWord = function (myWord) {
7             return _this.word += " " + myWord;
8         };
9     }
10    return StringBuilder;
11 })();
12 var stringBuilder = new StringBuilder();
13 alert(stringBuilder.newWord("The quick"));
14 alert(stringBuilder.newWord("brown fox"));
15 alert(stringBuilder.newWord("jumps over the"));
16 alert(stringBuilder.newWord("lazy dog"));
17
```

Fig 14 JavaScript file

In this case, the arrow function works just like a regular function expression.

Side note:

Just out of curiosity, how would we go about addressing the outer variable *word* (line 1) from within the *newWord* method, when implementing a class?

To address the global variable *word*, all we have to do is to remove the "this" from the returned *word* value (line 5 on the TypeScript *fig13*).

Another way, if you want to be more specific, is to declare another variable at the environment you want to target (after line 1, *fig13*) and assign it to its local "this", then point the method's *word* to it (currently on line 5).

Example:

```
var that = this; // this is declared in the target environment
```

```
return that.word... // return statement in the class method
```

Notice that I used "*that*" instead of "*_this*". I would avoid manually using "*_this*" (underscore this) since TypeScript uses it internally whenever it needs it. You can of course, use another name for the variable. Besides "that", "self" is also a popular name for this kind of application. Whatever name you choose, keep it consistent on your programs.

On the next topic, we will see how TypeScript successfully uses *arrow functions* to solve problems normally found on common JavaScript code.

e) Arrow functions assure the correct context in classes

We will cover classes in more detail later in the book. For now, please look at the following example code. **You may also skip this topic and return to it after we cover classes.**

```
1 var test = 999;
2 class myClass {
3     test = 55;
4     myFunction() {
5         setTimeout(function() {
6             alert(this.test);
7         }, 50);
8     }
9
10 }
11
12 var myObject = new myClass();
13 myObject.myFunction();
```



Fig 15

Link to raw file: [33] icontemp.com/typ/arrowFuncClass2.txt

On line 1, there is a global variable with a value of 999. This variable serves as a test against the *class property* of the same name (*test*), which is found on line 3.

The inner function on line 5 will select either the global variable *test*, or the class property *test*, via its “this” pointer.

Lines 4 through 8 comprise of a method (*myFunction*) which invokes a *window* method (*setTimeout*), which in turn triggers an *inner function* that alerts the value of *test*.

The question is: which value is displayed on the screen, 999 or 55?

The value 999 is displayed on the screen instead of 55.

That is because the method *setTimeout* is an inner function inside of a method. **Inner functions inside of methods are not methods**, they are independent functions and they belong to the global context. To make things worse, *setTimeout* is a method from the *window* object and therefore it is global by nature, but even a simple inner function would have its “this” pointing at the global object.


If `setTimeout` confuses you, here's an example of the same script using just a simple inner function:

[34] icontemp.com/typ/arrowFuncClass3.txt

So what can we do to address the inner property in the class and get the value of 55?

To address the “this” from the class itself we can **use an *arrow function* as the inner function** because, as we already know, TypeScript will use its ***this*** magic to keep the fake “this” within the class boundaries.

Please look at the following image to see what I mean:



```
1 var test = 999;
2 class myClass {
3     test = 55;
4     myFunction() {
5         setTimeout(()=> {
6             alert(this.test);
7         }, 50);
8     }
9 }
10 }
11
12 var myObject = new myClass();
13 myObject.myFunction();
```

Fig 16

Link to raw file: [35] icontemp.com/typ/arrowFuncClass4.txt

This script has already been explained below the image before last. In this current image, we now have an arrow function on line 5 instead of a regular anonymous function. The arrow function will direct the “this” to access 55 instead of 999. This is because TypeScript adds a new variable named ***this*** to the class and this variable will point to the “this” of the class.

Below, please find the JavaScript result after being transcompiled by the TypeScript compiler.

```
1 var test = 999;
2 var myClass = (function () {
3     function myClass() {
4         this.test = 55;
5     }
6     myClass.prototype.myFunction = function () {
7     var _this = this;
8         setTimeout(function () {
9             alert(_this.test);
10        }, 50);
11    };
12    return myClass;
13 })();
14 var myObject = new myClass();
15 myObject.myFunction();
16
```

Fig 17 This is JavaScript

Notice how on line 9, TypeScript replaced our “this” for “_this”, and then on line 7, it created a new variable which points the current local “this” to the inner function “_this”.

f) Summary

Arrow functions in TypeScript do not have a “*this*” keyword.

Every time we include a “*this*” in an arrow function, the TypeScript interpreter replaces it with “*_this*”, and then it creates a new variable in the global context or, if a class is being used to instantiate the object, in the class context.

Avoid using “*this*” in arrow functions unless you use it to solve problems such as the ones described on topic (e). Otherwise, use regular anonymous functions to avoid surprises.

An inner function inside of a method is not a method - it is a global function.

In classes, if one of the purposes of using an inner function is to access members of the class, use an arrow function to redirect the “*this*” from the default global access, to the class environment access (TypeScript adds a fake “*this*” to the arrow function and it resolves it based on the criteria we have seen in this chapter).

The purpose of the fake “this” in an arrow function is to set the “this” where the function is created, rather than setting it in the context where the function is invoked. This is the reason why it points to the global environment when it is created in the global environment (even inside of a global child). It can also point to a class when it is created within the class.

There is a lot more to arrow functions than what has been covered so far and I believe TypeScript is still developing more solutions. Use the *TypeScript Playground* to experiment with other possibilities. Look at the results and analyze the JavaScript version to see what TypeScript is doing in the background.

Learn code like if you were a mad scientist, test, test, and test some more.

Function OVERLOADING

Function overloading is the ability to use the same function name for multiple function implementations.

The implementation selected when invoking the function is based on the type of *arguments* we feed to the function *parameters* when we invoke the function. We have seen this before when I discussed optional parameters.

Since *JavaScript* functions accept any type of data as arguments, overloading is not as important as it is with other [strongly typed](#) [8] languages. However, TypeScript allows for overloading and that can help the compiler to warn the programmer for possible errors when he or she creates the project. This is all possible thanks to *type annotations*.

In other words, function overloading in TypeScript limits a function with type *any* parameters to the specific types available in the declared signatures, instead of authorizing all data types as parameters. **Function overloading narrows the scope of the type *any*** to the available (authorized by the programmer) signatures.

This will make sense when we look at the examples given on the next few pages.

Keep in mind that since TypeScript allows for *optional* parameters in functions, you may find the optional parameters feature to be a better solution for the outcome you want to achieve, instead of using the overloading technique I'm about to explain. Having options is a good thing!

The following example is 'totally bogus' but it serves to illustrate how we can create function overloading in TypeScript without getting into more complex code.

In the code below, the interpreter looks at the data type and decides whether it should *multiply* or *concatenate* the result. To accomplish that, we want the function to accept either numbers or strings:

```
1 function doubleMe(x: number);
2 function doubleMe(x: string);
3 function doubleMe(x: any) {
4     if (x && typeof x === "number") {
5         alert(x * 2);
6     }
7     else if (x && typeof x === "string") {
8         alert(x + " " + x);
9     }
10 }
11 doubleMe(5);
12 doubleMe("Tony");
```

Fig 18

See my raw file here: [36] icontemp.com/typ/funcOverload.txt

Lines 1 and 2 are function signatures. The reason to declare these signatures is to tell TypeScript that we intend to use the function for either numeric arguments or string arguments (the calling types are limited to only one or the other).

Then, on line 3, I declared a function that accepts a type *any* parameter and, depending on which type of argument I pass when I invoke the function, JavaScript will execute the code differently. The execution is based on the “if, else if” statements.

I could however just write the third function (the one of type *any*) and obtain the same results in JavaScript. However, here is where you may find it useful: because I have declared two different function signatures above the one that implements the code, they conceptually limit the type of data that can be passed when I call the function.

For example, if I declare an *array* and pass in the array into the function, normally this would work because the function accepts *any* type. However, in TypeScript, this particular *any* type function is limited to *number* or *string* types due to the top two signature declarations.

Example:

```
var myArray = [1,2,3,4]; // (to be used on the 3rd example
doubleMe(5); <-- OK
doubleMe("Tony"); <-- OK
doubleMe(myArray); <-- ERROR
```

The TypeScript interpreter will warn me that I am about to make a potential mistake when I attempt to pass an array type as an argument to the function call.

Summary:

Because JavaScript does not support overloading, we implement overloading in TypeScript only to establish type safety rules for ourselves or for a programming team working on the project.

LAB WORK: Three small projects using functions

Project 1

Create a program in TypeScript that converts temperatures from *Fahrenheit* to *Celsius*. You can find a link to my own code at the end of the instructions.

- 1- Wrap this program in a function named **f2c** so that you can invoke the program at any time.
- 2- This function will take a **numeric** argument via a parameter named **f**.
- 3- The function will not have to return anything (use **void**) since we are going to *alert* the result.
- 4- Write an **alert()** method within the function and display the following result:
“212 degrees Fahrenheit = 100 degrees Celsius”
Where, in place of 212, we have variable **f** and in place of 100 we have the following formula:
 $(f - 32) * 5 / 9$
- 5- Invoke the function and pass in 212 as the input argument. After running the program, the displayed result should be "212 degrees Fahrenheit = 100 degrees Celsius".
- 6- Finally, try to call the function by passing a *string* value such as for example "abc". You will see that TypeScript underlines the value in red and when you hover with your mouse over it, you will read the following error:
Argument of type 'string' is not assignable to parameter of type 'number'.

See my code sample here: [37]

jsplain.com/javascript/index.php/Thread/154-TypeScript-Fahrenheit-to-Celsius

On the next project, we are going to reverse the function mechanism so that it converts from Celsius to Fahrenheit.

Project 2

Create a program in TypeScript that converts temperatures from *Celsius* to *Fahrenheit*.

- 1- Wrap the program in a function named **c2f** so that you can call the program at any time.
- 2- This function will take a **numeric** argument via a parameter named **c**.
- 3- The function will not have to return anything (use **void**) since we are going to *alert* the result.

- 4- Write an **alert()** method within the function and display the following result:

“100 degrees Celsius = 212 degrees Fahrenheit”

Where, in place of 100, we include variable **c**, and in place of 212 we have the following formula:

(c * 9 / 5 + 32)

- 5- Call the function and pass in 100 as the input argument. After you run the program, the displayed result should be *"100 degrees Celsius = 212 degrees Fahrenheit"*.
- 6- Finally, try to call the function by passing a *string* value such as "abc". You will see that TypeScript underlines the value in red and when you hover with your mouse over it, you will read the following error:

Argument of type 'string' is not assignable to parameter of type 'number'.

See my code sample here: [38]

jsplain.com/javascript/index.php/Thread/155-TypeScript-Celsius-to-Fahrenheit

Project 3

- Create a program in TypeScript that converts *Celsius* degrees to *Fahrenheit* **or** *Fahrenheit* to *Celsius*.
 - Use a second parameter that defaults to the character "f" as an argument. The idea is that when we enter the temperature argument to be converted, it automatically calculates a conversion from *Fahrenheit* to *Celsius*. However, if we enter a second argument ("c" in this case), it calculates a conversion from Celsius to Fahrenheit. On the other hand, if we enter another character instead of "f" or "c", it will alert that the conversion is not supported.
- 1- Wrap the program in a function named **conversion()** so that you can call the program at any time.
 - 2- This function will take two parameters. Name the first parameter **x** and make it of *type number*. Name the second parameter **y**, make it of *type string* and assign a default value of **"f"** to it.
 - 3- The function will not have to return anything (use **void**) since we are going to alert the result.
 - 4- Create an **if, else if, else** conditional statement.
 - 5- The **if** condition clause should check to see if **y === "f"**. In this case, alert the result of a conversion from Fahrenheit to Celsius (remember that we are using **x** as the variable name this time).
Example code: `alert((x-32) * 5 / 9 + " Celsius");`
 - 6- The **else if** condition clause should check to see if **y === "c"**. In this case, alert the result of a conversion from Celsius to Fahrenheit (use x as the numeric variable).
Example: `alert((x * 9 / 5 + 32) + " Fahrenheit");`
 - 7- The **else** clause will display a generalized message such as:
`alert("Sorry, this conversion is not supported");`
 - 8- Call the function by passing just a number, then call it again by passing a number and the character "c", finally, call it by passing a bogus character. See samples below:
`conversion(212) ;`

```
conversion(100, "c");  
conversion(100, "g");
```

The following link takes you to my own code sample here: [39]

jsplain.com/javascript/index.php/Thread/156-Typescript-Fahrenheit-to-or-from-Celsius

Project 4

Do you want to practice a little more? Try adding other conversions to your program.

You can create additional *else if clauses* in the middle, or replace the conditional statements with a **switch** statement for a cleaner outcome.

Below please find some suggestions for creating other conversions on their own:

a) Perimeter

Suggested function design: `calcPerimeter(length,width) {`

Formula: `length + length + width + width`

b) kilometers into miles

Suggested function design: `kiloMiles(km) {`

Formula: `km * 0.6214`

c) Feet to meters

Suggested function design: `feetM(ft) {`

Formula: `ft * 0.3048.`

Spend a few hours creating new stuff.

If you would like to save your scripts for future use, copy/paste them to a plain text editor such as Windows NOTEPAD and save them with the extension of *.txt*, which represents the plain text format. This is what I did for my samples.

To save them as TypeScript code, use the *.ts* or, if you are saving the JavaScript version, use *.js*. For now, we really want the *.txt* file extension for our sample.

PART II

“For the things we have to learn before we can do them, we learn by doing them”

--- Aristotle, The Nicomachean Ethics

6- Variable prefixes: VAR, LET, CONST, DECLARE

Function Scope

In JavaScript, functions are the only code block (code inside of brackets `{ }`) that allow for private variables (those that cannot be seen from the outside).

Example:

```
var color = "blue";  
function myColor() {  
    var color = "red";  
    alert(color);  
}  
myColor(); // red  
alert(color); // blue
```

In the above example, when function *myColor* is invoked, it displays *red* as a result which is the value in the inner variable *color*. However, when we alert *color* by itself, it displays *blue* because it is the value in the global variable *color*. This shows that when we initialize a variable inside of a function (by using *var*), the variable remains private and it does not conflict with variables outside of the function with similar names.

This privacy does not apply to other code blocks in JavaScript syntax. In JavaScript, functions are the only code block that allows for private variables. In the upcoming JavaScript 6 version, this will change but this feature is not yet supported across browsers. In the future, privatization in *code blocks* will be done with the keyword **let** instead of **var**.

TypeScript allows for the usage of *let* by refactoring the code so it becomes compatible to all browsers. This will be demonstrated in the next few topics.

Planning for block scope

Although not yet recognized by browsers, TypeScript allows the usage of *let* when we create our code project. This implementation sometimes works and sometimes it does not make a difference when it comes to the final JavaScript transpiling result; it all depends on the context.

However, even if *block scope* is not supported in every script context, it is good practice to use *let* whenever we create temporary variables.

The same principle applies when we declare variables inside of code block groups because it implicitly tells TypeScript of our intentions, and TypeScript reward us with *code checking* as we continue to program the project.

In the next two examples, I will demonstrate what happens when implementing *block scope* in JavaScript by using *let* as a variable declaration prefix, instead of *var*:

Example 1

```
var x = 123;

if(true){
  var x = 7;
}
```

In the above example, the second *var x* reassigns the first *var x* from 123 to 7, and based on the fact that we used *var* the second time, this may have been done by mistake. In other words, perhaps we did not know that x already existed and we just erased its original value.

Example 2

```
var y = 123;  
  
if(true){  
  let y = 7;  
}
```

On example 2, the *let* keyword gives it a *feeling** of local scope (within the `{}`). The keyword *let* is only introduced in JavaScript version 6, which at the time of this writing, is not yet supported across all browsers, but we can safely use it in TypeScript because it will be appropriately translated to the current JavaScript supported version.

What TypeScript does in this case, it automatically changes the name of the second variable from *y* to *y_1*, so that it does not conflict with the outer variable.

*This is not real privacy, but it avoids name conflicts.

Just to understand the convenience of using *let* instead of *var*, here is another example of what TypeScript does:

```
1 var x = 123;
2
3 if(true){
4     let x = 7;
5 }
6
7 let x = 23;
```

Fig 19

We already know what happens to variable *x* on line 4: it becomes *x_1* in JavaScript in order to prevent conflict with the global variable *x*.

However, what is going to happen on line 7?

When we write *let x = 23* on line 7, TypeScript warns us that this variable already exists in this context (line 1). Had we used *var* instead of *let*, TypeScript would not have warned us about a possible conflict. It would just overwrite it.

In summary:

Although browsers do not yet support *let*, it is good practice to use it in TypeScript so that the interpreter knows of our intentions and protect us from making syntax errors along the way.

If a variable is not going to be reused outside of a code block, then we definitely should use *let* instead of *var*. This becomes important in loops. Eventually, *var* will probably fade away.

Use let for loop counters

Let's look at the following example that uses *let* in a *for loop*:

```
1 var i = 7;  
2  
3 for(let i = 0; i < 5; i++){  
4     console.log(i);  
5 }  
6  
7 alert(i);  
8 // 7
```

Fig 20

Above we have two variables named *i*.

Because variable *i* in the *for loop* was declared with the *let* keyword, it did not affect the value of the outer variable *i* which remains with a value of 7 (instead of 5 by the time the loop counter ends its cycles).

TypeScript was able to accomplish this separation of concerns by changing the name of the loop counter from *i* to **i_1** in the JavaScript output code:

```
1 var i = 7;  
2 for (var i_1 = 0; i_1 < 5; i_1++) {  
3     console.log(i_1);  
4 }  
5 alert(i);  
6 // 7  
7
```

JavaScript

Fig 21

In other words, use *let* for temporary variables because TypeScript will provide you with better support as you continue to expand the code in your project.

The CONST prefix

Constants are variables whose initial values should not be changed. In this sense, the term "variable" is not correct because constants are fixed values.

Common JavaScript does not enforce constant values. The new ES6 JavaScript version will introduce constants but this is not yet supported.

TypeScript introduces the concept of constants in the sense that it warns the programmer if he/she attempts to change the value of a variable originally designated as a constant.

To declare a variable as a constant we use the keyword **const** instead of *var* or *let*.

Example:

```
const x = 10;
```

Now, if I attempt to change the value of x I get a red dot next to x:

```
x. = 12;
```

The TypeScript interpreter is warning me to pay attention to what I am about to do. Variable x becomes highlighted in red on the TypeScript playground.

Using the const prefix in function expressions

As you probably know from JavaScript, there are two ways for declaring a function. These two ways are *function declarations* and *function expression*.

A function declaration uses the word function as its prefix.

Example:

```
function addMe(num1: number, num2: number):number{  
    return num1 + num2;  
}
```

In the above script, I am declaring a function named addME.

Now, on the example below, I am declaring a variable named addME and assigning to it an expression, which happens to be the script of a function. This is why we call this technique a function expression:

```
var addMe = function (num1: number, num2: number):number{  
    return num1 + num2;  
};
```

Which one you use depends on your preference and the application at hand. Sometimes one makes more sense than the other does.

Since the function on the second example is assigned to a variable, one must be careful not to reassign the variable to some other value later on. As a precaution you might want to declare a function expression with the keyword **const** so that TypeScript warns you if you mistakenly reassign the variable name to something else later in the program.

Variable hoisting and dead zones

When we choose which keyword to use for our declaration, we must consider **variable hoisting** and **temporal dead zones**. A dead zone is the area above the physical declaration of the variable. When it comes to **var**, **let** and **const**, one can only put the variable to use below its declaration (not before it). When we use the keyword **function**, as in function expressions (first example), we can put the function to use above its declaration. In addition, both TypeScript and ES6 use **block scope** for *let* and *const*, which means that variables declared with one of these two prefixes are limited to the code-block where they are declared, and even there, these variables are only available after the declaration since above it, is a dead zone.

Further reading:

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/
let#Temporal_dead_zone_and_errors_with_let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#Temporal_dead_zone_and_errors_with_let)

The DECLARE keyword

Sometimes you may come across a variable on TypeScript code that starts with the keyword *declare* and you might wonder about its meaning.

The *declare* keyword is used to introduce variables that come from a different source other than your TypeScript code, like for example, from a *library*. It is an *ambient declaration* (an ambient declaration is used to provide type information for code that exists somewhere else).

You will find or use ambient declarations in advanced code, which is beyond the scope of this introductory project. The purpose of this introduction is to make you aware of it and give you a reference to start exploring the subject.

To know more about the *declare* keyword please refer to the following GitHub resource:

[40] github.com/Microsoft/TypeScript/blob/master/doc/spec.md#12 .

7- Interfaces

What is an interface?

In our previous short introduction to objects, we created a simple object literal called *myWardrobeCabinet*:

```
var myWardrobeCabinet = {  
  topDrawer: "Towels",  
  middleDrawer: "T-shirts",  
  bottomDrawer: "Socks"  
};
```

The above construct is an **object** data type.

(By *construct*, I mean a paragraph in code syntax; and independent piece of code).

If we plan to write more objects in a similar manner (liker using similar property types and the same number of properties), we could take the mold of this object and create a rule set to assure that we (and other programmers on the team) follow the same criteria in object creation for the project. This set of rules is called an *interface*.

Here is an example of an interface with rules based on the previous object:

```
interface ICabinet {  
    topDrawer: string;  
    middleDrawer:string;  
    bottomDrawer:string;  
}
```

The above example serves as a preliminary overview in case you want to have an idea about it, but I will explain how to design an interface on the next topic.

An *interface* is a construct that has specific rules for the creation of other constructs (such as objects, classes, functions, arrays, etc).

The advantage of building an *interface* (or custom data type), is that we can safely recreate other individual objects based on the same rules.

Do not think of “**inheritance**”. This is different from inheritance and you will see the difference later in the book when we cover classes.

The reason for an *interface* is to establish how we, as programmers, are going to create a certain kind of code for the project at hand. It works great for teamwork but even if you program alone, it helps you along the way because the compiler will warn you when you mistakenly attempt to break the rules. You set those rules early on for yourself in order to stay consistent.

- Interfaces allow us to create custom data types in TypeScript.
- An interface acts as a contract, which needs to be followed if implemented (if assigned to another construct).
- Interfaces ensure TypeScript type-checking features. (We tell TypeScript to inspect the code as we write it).
- Interfaces do not add ‘bulk’ to memory because the overall *type information* is erased from a TypeScript program compilation. We can freely add customized data types by using interfaces without worrying about the runtime overhead.

In summary:

After this summary, we will create an interface to see how it is done.

An interface sets the rules for *object* creation (or the creation of any other type such as *arrays* and *functions*).

The interface becomes a template with rules for a certain type of object creation (this is not a class).

The only job of an interface in TypeScript is to describe a type. Later we will cover classes, and we have already seen functions... these two mechanisms deal with implementation while an interface helps us keep our programs error-free by providing information about the type of the data we work with.

On the next topic, we are going to discuss the syntax of an interface. This is an important tool for advanced TypeScript/JavaScript programming.

How to create an interface

- 1- To create an interface (which is a list of rules) we start by declaring the interface and giving it a name:

```
interface ICabinet { }
```

Notice how the *C* is capitalized. This is not mandatory but it is a common convention to capitalize *interfaces* and *classes*, and leave variables and object names in lower case. It kind of calls attention to anyone inspecting the code to see what the name represents.

Also, the prefixed *I* is just a way to tell humans that this is an interface. Again, it is not mandatory, just a good thing to do. If you name it *Cabinet* instead of *ICabinet*, it will work the same way.

Note: When we create a program, we create it with two target audiences in mind: the machine and the human eye.

The following link takes you to a guideline of Interface naming conventions as written for C# (both C# and TypeScript were created under the leadership of *Anders Hejlsberg* from Microsoft):
[41] [Interface naming guidelines](#).

- 2- The next thing to do is to establish the rules of the interface we are creating, like for example, the names of properties and the type of data these properties will accept:

```
interface ICabinet {  
    topDrawer: string;  
    middleDrawer:string;  
    bottomDrawer:string;  
}
```

If you write this interface in the [TypeScript Playground](#) [3] you will notice that nothing shows up on the right side as JavaScript. This is because it is pure TypeScript. Interfaces assist TypeScript to inspect the code we, humans, write for the rest of the project, but after the project is finished, they are not transferred when transcompiled to JavaScript. They are however, good and safe implementations that we should save as original code for future use and possible updating.

Assigning an interface type to an object

Now that we have an Interface, we can convert our original object (or any future objects of this type) to comply with the *rules* set by the interface.

Remember, the interface is just like another type; it is of type *ICabinet*.

Here's how we assign an object named **myWardrobeCabinet** to this interface:

```
var myWardrobeCabinet:ICabinet = {  
    topDrawer: "Towels",  
    middleDrawer: "T-shirts",  
    bottomDrawer: "Socks"  
};
```

The only difference between this object and the object I have shown earlier on other exercises is that we now have assigned a type to this object. *myWardrobeCabinet* is now of type *ICabinet*.

What does it mean being of type ICabinet? It means that this type is a customized type, a specific combination of other types. An object of type *ICabinet* is an object with three specific properties of type string with specific names as described in a ruleset called *ICabinet*.

If you paste this object on the playground (after the interface) you will see that the object shows up on the JavaScript column, but the interface does not show at all (Interfaces are only used in TypeScript since JavaScript does not support them).

Things to know:

1. **Interfaces regulate the data type of the properties used by their subscribers.**

When I try to change "*Socks*" to a number like for example 33 (without quotes), JavaScript accepts it but you will notice that *myWardrobeCabinet* gets underlined in red on the TypeScript side. This is because *bottomDrawer* is set as a *string* value by the interface rules and that raises a flag, which assists the programmer to stop and correct the issue.

2. **Interfaces also regulate the number of properties a subscriber should have.**

What would happen if we removed or commented the *bottomDrawer* line from the object?

Again, a *flag* is raised. This is because another rule of the interface was broken, which is the number of items that this *data type* must contain. There should be three items.

3. **Interfaces regulate the name of the properties used by its members.**

When we try to change the name of any property, we get a warning from TypeScript. This prevents syntax mistakes.

Declaring an object assigned to an interface but with no property values

We can create a blank object and assign it to an interface. Then later we can add the data to its properties.

Example (based on the previous object):

```
var myWardrobeCabinet = <ICabinet> { };
```

In general, the angle brackets specify a *protocol* of some kind. In this case we are assigning *myWardrobeCabinet* to an object `{}` that assumes a protocol named *ICabinet*.

Why didn't we get an error when creating the object, since the properties are not listed in accordance to the interface rules?

Actually, those properties were implicitly assigned to the object, but they are all set to the value of *undefined*.

You can check that by alerting one of the properties and you will see that the TypeScript interpreter will assist you writing the property name as soon as you write the *dot*.

Example:

```
alert (myWardrobeCabinet.topDrawer) ;
```

When you run the `alert()` command it displays *undefined* because *topDrawer* does not have a value yet but the property already exists. In JavaScript any empty property or variable is assigned to the value *undefined* which serves as a placeholder for future data.

Now that we have an object, we can add values to its properties:

```
myWardrobeCabinet.topDrawer = "Towels";  
myWardrobeCabinet.middleDrawer = "T-shirts";  
myWardrobeCabinet.bottomDrawer = "Socks";
```

How to create optional properties

If we want to, we can establish that some of the properties on the interface may be optional. This makes the interface much more flexible and versatile.

Optional properties in interfaces work the same way as optional parameters in functions:

We use the question mark (?).

Let's change the last item from being mandatory to being optional

```
interface ICabinet {  
    topDrawer: string;  
    middleDrawer:string;  
    bottomDrawer?:string;  
}
```

Now, when we remove or comment the last item from object *myWardrobeCabinet*, TypeScript does not complain because the last property is an optional property.

Notes:

- Unlike functions where optional parameters need to be written last, in interfaces, optional properties can be written anywhere. Any property can be listed as optional.
- When we omit an optional property in an object, it does not mean that the property is missing. It only means that the property is not being used and it is set as *undefined*, but it still exists. We just don't get a red flag warning when we ignore it.

Interfacing objects with unlimited properties

In our previous example, we had a fixed number of properties or optional properties. However, many times what we need is an object with an open number of properties that can be added later.

To create an interface where the number of properties is unlimited, we need to design the interface as if we were going to create an array, but with an index of type *string*, instead of a *numeric* index.

I will cover this implementation later, but if you are reviewing the book and not reading it in sequence, please go forward a few pages and read the topic “**Using an interface to describe an array type**”.

Using an interface to describe a function

Interfaces are cool! We can use interfaces to set rules for function signatures before we start programming functions.

A function signature is the top part of the function where we declare the parameters and the type of *return statement* we want to have. The function name is not part of the signature.

a) Example of a function interface:

```
interface IPrintItem{  
  (x:string):void  
}
```

In the above declaration, *x* is a parameter of type *string* (parameter sets are wrapped with parentheses), and *void* is the type of *return*. This means that any method or function that subscribes to an *IPrintItem* interface should have one parameter of type *string* and the *return* statement is **not mandatory** (See further explanation on the next page)*.

Note: when we create a function based on this interface, the *x* parameter can be named whatever you want; it does not have to match the name of the parameter written in the interface (this contrasts with property names from our previous topic. Those need to match the names on the interface).

What needs to be matched with an interface representing a function is the *data type* rule-set: the parameter is of type *string* and the return is *void**.

- b) Once we have created the interface, we can *initialize* some variables of type *IPrintItem*. (These variables will then be assigned to function expressions on the next step).

Example on how to assign variables to an interface:

```
var myFavCar: IPrintItem;  
var myFavFlower: IPrintItem;
```

- c) Now we can assign anonymous functions to the variables. These functions need to be based on the rules stated in the interface.

In our example, the interface calls for a *string* parameter. If you try to change this rule by declaring a *number* parameter, adding extra parameters, or adding a return value of a different *type**, TypeScript will complain.

***Actually, if the return type on the interface is either *void* or *any*, you can use any type of return declaration on your function signatures, as well as no return statement declaration in the body of the function. However, if the return type on the interface is of type *string* or type *number*, the rule is strictly enforced and these return types cannot be changed later when we create the functions. I think it is a matter of specificity since both *void* and *any* are more forgiving.**

Assigning function expressions to variables that subscribe an interface

The following are two examples of anonymous functions assigned to our previously declared variables:

```
myFavCar = function (itemName:string){  
    alert("My favorite car model: " + itemName);  
}  
  
myFavFlower = function (itemName:string){  
    alert(itemName + " is my favorite flower.");  
}
```

- d) Now we can call both functions and pass in whatever argument we want as long as the argument is of type *string*.

```
myFavCar("Corvette");
```

It displays *My favorite car model: Corvette*

```
myFavFlower("Sunflower");
```

It displays *Sunflower is my favorite flower.*

Here is my link to the complete sample file: [42]

jsplain.com/javascript/index.php/Thread/159-TypeScript-Creating-a-function-interface

Adding a function (or method) to an object Interface

We have just seen how to program an interface for multiple function creation. The interface establishes the rules ahead of time so that we are consistent with the types used in future functions that subscribe to this interface.

Now we can go back to the original object *myWardrobeCabinet* and its interface *ICabinet*, and see how to implement a set of rules for methods (functions) we may want to include in objects that subscribe to this interface.

Note: a *method* is what we call a function that belongs to an object and works on behalf of the object. However, functions inside of other functions (methods) are not methods. Those should be treated as global functions.

I have chosen to publish this script online along with an explanation instead of bringing it to the book in an effort to simplify the topic a bit.

Please refer to the following link: [43]

jsplain.com/javascript/index.php/Thread/160-TypeScript-Sample-of-an-interface-for-an-object

Using an interface to describe an Array type

We can also create interfaces to set the *type rules* for arrays.

There are two kinds of array interfaces:

numerically indexed (used for regular arrays)

and *string-indexed* (used to represent objects with an unlimited number of properties)

a) Numeric indexed array interfaces

The example below shows an *array interface* with a *numeric index* and *string data*:

```
interface IMyArray {  
  [index:number]:string;  
}
```

Then we can create arrays based on the above construct:

```
var someArray : IMyArray =["blue", "Green", "red"];
```

b) String-indexed array interfaces

You might ask at this time why should we have an array without a numeric index. Shouldn't we use an object implementation for that?

Well, there is no such thing as a string-indexed array.

What we are going to create is actually an object with the curly braces `{}`.

If you remember from when we created Interfaces for objects, we had to declare every property as well as tell TypeScript which properties were optional (if any). By using a string-indexed array interface, we can declare one single property rule that applies to all properties of the object (almost like an array of property types). **This allows for an unlimited number of properties in this object.**

Here is what the interface syntax looks like:

```
interface IGrades {  
  [index:string]:number;  
}
```

In the above example, the data type is *numeric* but it could have been of any other type as well, since this is data stored in object properties.

Please note:

On the interface syntax, the term **index** can be called anything (it is not a keyword). It is a variable to be used internally by the TypeScript compiler. I have used *index* to be more descriptive but you can use whatever name you want like for example *x* or *i*.

After creating the interface *IGrades*, we can create objects modeled after the interface:

```
var myGrades:IGrades = {};
```

Then finally, we add items to the object *myGrades*:

```
myGrades["Math"]=100;  
myGrades["History"]=90;  
  
console.log(myGrades) ;
```

To copy or review the sample file discussed above please refer to the following link: [44]

jsplain.com/javascript/index.php/Thread/162-TypeScript-Sample-of-an-interface-for-an-array

In order to cement these ideas, please create these interfaces yourself and test them on the playground.

Please note:

As a reminder, when it comes to numeric indexed arrays, we use the *for loop* to iterate over the properties. When it comes to string-indexed objects, the *for in loop* is used instead.

Extending Interfaces

Extending interfaces means to add a second or more interfaces to another interface in order to take advantage of existing *rule sets* and avoiding repetition. (For future reference, this differs from classes, which can only extend to one more class).

a) In the following example, we have an interface designed for an object and its colors:

```
interface IColors{  
    colorName:string;  
    colorIntensity:number;  
}
```

b) Then we have another interface where we describe an item and its *id* number:

```
interface IMerchandize{  
    name:string;  
    id:number;  
}
```

c) Now we create a third interface that describes two properties of its own, but it extends to include the other two interfaces as well:

```
interface IPricing extends IColors, IMerchandize{  
    inStock:Boolean;  
    price:number;  
}
```

d) Finally, we create an object that follows the rules from interface *IPricing*:

```
var sunflower=<IPricing>{};
```

Notes: In general, the angle brackets < > indicate a protocol. In this case, we are saying that the object assigned to *sunflower* follows the Interface *IPricing* protocol.

At this time, all properties from the three previously created interfaces have been assigned to object *sunflower* but the data in each of them is still *undefined* because we have not yet added any data to the object.

- e) Now we enter all the properties for the object *sunflower* in accordance with the rules from all the interfaces we had declared:

```
sunflower.colorName = "Yellow";
sunflower.colorIntensity = 5;
sunflower.name = "Sunflower";
sunflower.id = 2345;
sunflower.inStock = true;
sunflower.price = 11.00;
```

Please note: We can also create the object all at once and assign it to *IPricing* interface, which extends to all the other two interfaces:

```
var sunflower:IPricing = {
    colorName:"yellow",
    colorIntensity:5,
    name:"Sunflower",
    id: 2345,
    inStock:true,
    price:11.00
}
```

Please refer to the image below to see the whole script.

```
1 //TypeScript
2 interface IColors {
3     colorName: string;
4     colorIntensity: number;
5 }
6
7 interface IMerchandize {
8     name: string;
9     id: number;
10 }
11
12 interface IPricing extends IColors, IMerchandize {
13     inStock: Boolean;
14     price: number;
15 }
16
17 var sunFlower: IPricing = {
18     colorName: "yellow",
19     colorIntensity: 5,
20     name: "Sunflower",
21     id: 2345,
22     inStock: true,
23     price: 11.00
24 }
```

Fig 22

Link to raw file: ^[45] icontemp.com/typ/ifExtends.txt

8- Classes

What is a class?

A class is the template structure that defines a live object. Wait, isn't that what you said about interfaces?

It may sound confusing but classes and interfaces have different purposes:

- An interface in TypeScript is an assistant to the programmer; it keeps the making of a program error-free by planning ahead of time what kind of code we are going to create as far as data types are concerned. This may apply to variables, functions, arrays, individual objects or even classes. It is a contract between the programmer and the TypeScript interpreter about the code that is going to be written -- a promise to include certain properties and abide by a designated type structure. This becomes more important in teamwork projects or in medium to large programs. It avoids getting lost in a jungle of code by preventing inconsistency.
- A class is also a template structure, but it deals with implementation of objects. Each class consists of TypeScript statements that define how to perform a task or set of tasks that you want to repeat frequently (the same idea as functions). The class can contain *private methods* (which are only consumed internally to perform the classes' functions), and *public methods*, which you can use to "interface" (connect) the outside world with the class. A good class hides its inner workings and includes only the public methods that are required to provide a simple connection to its functionality. When you bundle complex blocks of programming into a class, any script that uses that class does not need to worry about how exactly a particular operation is performed. All that is required is knowledge of the class' public methods, which allow access to the class.
- Both **interfaces** and **classes** define rules: interfaces define data types, and classes define object implementations, which also include data types. **Objects** are code blocks that store related properties and functions. **Functions** are code blocks that store related code that can potentially create actions on behalf of an object such as modifying its properties or exporting its values to the outer world. Even an external function in the global

environment acts on behalf of the global object which, in a browser, it is the object Window.

- ***Interfaces*** are used for the convenience of the programmer and for the safety of code writing.
- ***Classes*** are used to instantiate new objects. ***Objects*** can be created on their own, or by mimicking the properties of a *class*, or by following the type rules from an *interface*, or by using both a *class* and an *interface* as models.

If it still sounds confusing, you will have a much better idea about it after creating a couple of classes, instantiate some objects and even subscribe to an interface or two just to see how everything works together. **Let's do that!**

Creating a class

- a) To create a class we first declare a class container like in the following example for a class named *Person*:

```
class Person {  
  
}
```

Notice the capital P. As mentioned before, classes, interfaces and enums are conventionally capitalized.

- b) Next, we list the properties and respective types available in the class.

Notice how each property is terminated with a semicolon. This is different from manually creating an object where properties are separated by a comma.

```
class Person {  
    firstName: string;  
    lastName: string;  
    age: number;  
}
```

The class is just a *blueprint* of what an object should look like. So far, it looks like an *interface* but this similarity will stop when we do the next step.

- c) Now we must create an internal method that populates the properties with data when we want to instantiate (create) objects.

This is a special method with a special name: *constructor*

```
class Person {  
    firstName: string;  
    lastName: string;  
    age: number;  
    constructor(firstname:string, lastname:string) {  
        this.firstName = firstname;  
        this.lastName = lastname;  
    }  
}
```

Notice how I wrote the parameters of the function *constructor* differently than the names of the properties. This was done on purpose to illustrate that they are independent variables. It is actually common to write the exact same name for both (property and parameter) but I wanted to show you how the data is transferred from right to left when property *this.firstName* gets the data assigned from parameter *firstname*.

Speaking of the keyword “**this**”, if I am going to create an object of type *Person* named *mary*, the “this” serves as a placeholder for *mary*. Therefore, at runtime *this.firstName* is replaced with *mary.firstName*. By using “this” we make the class code portable to any new object we may instantiate.

d) The only thing left on this class is a way of outputting the data to the screen.

Let's add a method to do just that (see lines 13 – 15 on the image below).

```
1 class Person {  
2  
3   firstName: string;  
4   lastName: string;  
5   age: number;  
6  
7   constructor(firstname:string, lastname:string, age:number){  
8     this.firstName = firstname;  
9     this.lastName = lastname;  
10    this.age = age;  
11  }  
12  
13  listPerson(){  
14    alert("Person: " + this.firstName + " " + this.lastName + " age: " + this.age);  
15  }  
16 }  
17
```

Fig 23 See complete raw files below:

Raw File:[46] jsplain.com/javascript/index.php/Thread/161-TypeScript-a-Person-class

Lines 13 through 15 represent a method to display the information of a Person member object, and line 16 closes the class code block.

e) Now we can instantiate (create) an object based on this class:

```
var mary = new Person("Mary", "Smith", 39);
```

Here we use the keyword *new* which creates a space in memory for object *mary*, then we instantiate an object based on class *Person* which automatically calls the constructor function and passes in the data arguments.

f) Once the object is instantiated, we can invoke the method *listPerson* for the object *mary*:

```
mary.listPerson();
```

The output should be: *Person: Mary Smith age: 39.*

To see my complete *raw file* please use the link under the above image.

LAB WORK: creating classes and objects

This exercise serves two purposes:

- To assess your own understanding of the topic just covered
- To practice creating classes and instantiating objects in the simplest form possible.

Exercise 1: A class with no constructor

- 1- Using the [TypeScript playground](#) [3], create a class named *Automobile*.

The class should implement the following *string* properties:

make, model, color

The class should also contain the following *numeric* properties:

mileage, rating

(As you type, notice how the interpreter creates its JavaScript counterpart).

- 2- Instantiate an object called *mySubaru*.
(no data needed at this time)

Please refer to my raw file sample if you need help on the syntax:

[47] icontemp.com/typ/class1.txt

What have we done so far?

Well, we have just created a class without a specific *constructor* method (a constructor automatically adds the data to each property when we instantiate an object).

Next we instantiated an object but there is no data in it. However, the properties of the object have already been established when we instantiated the object, except that this data is of the value *"undefined"*. This value serves as a placeholder for future data.

So, what is the purpose for creating an explicit constructor? The purpose is to provide a way for us to insert the data as we instantiate the object.

- a) What happens if I attempt to display one of the object's properties, like for example *make*?

```
alert(mySubaru.make) ;
```

It displays the value of *undefined* because we have no data to show. However, the property is already there and the keyword *undefined* is the actual current value of the property.

- b) Although we don't have a constructor method in this class, we can later add the data to the properties by doing it manually.

On the example below, notice how the properties are listed as available as soon as you type the **dot** after **mySubaru**.

Example:

```
mySubaru.make = "Subaru";
```

```
mySubaru.model = "Outback";
```

- c) We can now test the object by displaying some of its content.
Make sure to comment or delete the *alert()* from step **a**

```
alert(mySubaru.make + " " + mySubaru.model);
```

I get an alert with the following data: *Subaru Outback*

Exercise 2: Adding a constructor to the class

- 1- Erase the object *mySubaru* from your *playground* and its subsequent testing code lines (**keep the class**).

Then go back to the *class* and add a **constructor** method which automatically inserts the data in each property when we instantiate an object.

(For practicing purposes, be sure to add the *data type* for each parameter in the constructor. The importance of these data type declarations will become obvious at the end of this lab work when I cover class interfaces).

- 2- Now *instantiate* an object called *mySubaru* and pass in the following data as arguments:

```
make: "Subaru",  
model: "Outback",  
color: "blue",  
mileage: 67000,  
rating: 9
```

See my sample raw file on the link below:

[48] icontemp.com/typ/class2.txt

- 3- Finally, we can call any of the object properties to see what is in there:

```
alert(mySubaru.make + " " + mySubaru.model);
```

It displays *Subaru Outback*.

Exercise 3: Adding a customized method to the class

Let's now try to add a customized method to the class. (I am placing my method below the closing bracket of my constructor but it can also be above the constructor as well).

- 4- In the class, add a method named ***printRating*** that alerts the following messages:
This car rates 9 out of 10.
Where 9 is the inputted object rating.

(Disclaimer: I do not know the true rating of an Outback. This is just an exercise).

- 5- Call the *printRating* method for *mySubaru*.

See my sample file here:

[49] icontemp.com/typ/class3.txt

Exercise 4: Creating a *for in* loop

- 1- Create an external *for in loop* to print all the keys and corresponding data values.

Use a *console.log* in your loop instead of an *alert()*, and dynamically display the items as in the following style:

make: Subaru

model: Outback

Remember, in order to see the results of a *console.log* on the screen (in Google Chrome), we must access the Console: *CTRL+SHIFT+j* or *COMMAND + OPTION + j* if on a Mac, **after we completely run the program.**

Hint: If you remember, in loops we cannot use *dot syntax* because the data we are fetching is dynamic. Use bracket syntax to fetch the data. Example: **mySubaru[item]** when *item* is the temporary variable for the loop.

Here's my sample raw file:

[50] icontemp.com/typ/class4.txt

Exercise 5: Filtering out the “printRating”

Did you notice how all the properties, including the *printRating* method, were displayed when using the *for in* loop?

If your intention is to only display the data and not all the properties, you can filter out the method *printRating* from it.

There are several ways to do this but here are two possible solutions you can incorporate in your loop. (The first solution is not practical for multiple methods, but it helps understanding the second solution better):

First way:

```
if(item !== "printRating") {  
    console.log(item + ": " + mySubaru[item]);  
}
```

In other words, display everything except the item labeled “*printRating*”.

Here is my complete raw file:

[51] icontemp.com/typ/class5.txt

(See a better implementation on the next page)

Second way:

This second implementation is the most common since it works for all methods in the object and it makes more sense when we have several methods in the class. It uses the ***instanceof*** operator, which takes two operands: In our example, it takes the *key* name on the left, and the *data type* on the right.

If the left is an instance of the right, then the *continue* command makes the loop go back to the next cycle without executing the code below it:

```
for (let item in mySubaru) {  
    if (mySubaru[item] instanceof Function) {  
        continue;  
    } else {  
        console.log(item + ": " + mySubaru[item]);  
    }  
}
```

Notice how **Function** is capitalized. The names of the JavaScript **fundamental objects** are capitalized because that's how they were originally called, and since JavaScript is case sensitive, we need to write them exactly as they are. To see other fundamental objects please refer to the following link: [52]

developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

Here is my complete raw file for this project:

[53] icontemp.com/typ/class15.txt

Note: In loops we can use **continue** or **break**. The *continue* command returns the running process to the beginning of the loop. The *break* command makes the program completely jump out of the loop.

You can read more about the **instanceof** operator at the following url:

[54] <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof>

Exercise 6: Including the *for in loop* inside of the class (as a method)

This next exercise is important in the sense that illustrates what changes in the *for in loop* when we integrate it as a method in a class. The reason for the changes is that it needs to be portable (applicable to any object of the same class).

- 1- Add a new method to the class Automobile. Name this method **listIt**.
- 2- Next, transfer the *for in loop* from the external environment into this new method's code block.
Now we cannot use *mySubaru* as the object calling the loop. We need to substitute the object's name by the keyword **this**.
- 3- Also, use the *instanceof* operator to filter out methods from being displayed because we now have more than one method in the class and this is easier to do. In other words, the first shown way of filtering cannot be used in a class.
- 4- In the end, call the method based on object *mySubaru*

See my raw file here: [55]

icontemp.com/typ/class6.txt

DO NOT DISCARD THIS EXERCISE.

We are going to use it on the next project.

Note:

Please look at the **JavaScript** implementation of this past exercise. Notice how in JavaScript all these methods are being added by the TypeScript compiler to the *prototype object* of Automobile.

Isn't it easier to write the code in TypeScript?

It seems so, it is easier to write code in TypeScript but we still need to know JavaScript as well because we may need to analyze JavaScript code in order to fix problems or update projects. However, the learning curve to understand JavaScript for debugging purposes is less than the learning curve required to engineer a project in JavaScript from scratch.

From this perspective, it is worth mastering TypeScript for programming purposes, and learn as much as we can about JavaScript in order to debug, analyze and understand its benefits and limitations.

Implementing classes from interfaces

In TypeScript we can also assign *interfaces* to *classes*. This allow us to establish a certain **criteria for class design**.

For example, on our previous *Automobile* class, we assigned the data type *number* to property *mileage*. Let's assume that we want that property to be consistently of type ***string*** instead of type *number*. In other words, instead of 67000 we want the data as "67,000". Since we are not going to do calculations with this property, a *string* data type works out perfectly.

Suppose that, based on this decision, we want to assure consistency from our team of programmers working on this project. We can create an interface with specific rules and have the class subscribe this interface.

Here's how we create the interface and then assign it to the class.

- 1- The interface is normally written on the top (prior to the class creation), but if you write it on the bottom it still works.

```
interface IAuto{  
}
```

- 2- Now, on the class itself we add
implements IAuto as shown below:

```
class Automobile implements IAuto{  
}
```

Notice how *implements* is plural.

At this point, nothing changes on the class because there are no rules in the interface. If you test your code it will work.

- 3- Now we add the property *mileage* to the interface and assign it to a *string* type:

```
interface IAuto{  
    mileage:string;  
}
```

Notice how suddenly the class name, *Automobile*, becomes underlined in red and if we hover the mouse over it, an error message is displayed explaining how the property *mileage* is wrongly implemented.

- 4- Once we fix the data type for *mileage* in the class, another flag is raised, this time in *this.mileage = myMileage*;
This is because the **parameter** *myMileage* in the *constructor* is still assigned to a *number* type but it should now be of *string* type.
- 5- Finally we get another red flag when we instantiate object *mySubaru* because we are entering 67000 as a number. It now needs to be wrapped in quotes.
- 6- At this point, everything should be working without red flags.

In summary, interfaces help the programmer to find errors as he or she writes the code.

Here is my completed raw file sample: [56]

jsplain.com/javascript/index.php/Thread/164-TypeScript-assigning-an-interface-to-a-class

Notes:

- a) Do you remember back when we first covered interfaces how all the properties had to be listed on the interface itself whether they were mandatory or explicitly optional? Have you noticed on my current class example how we only implemented one property on the interface but TypeScript did not complain about all the extra properties in the class itself?

Interfaces are implemented differently when it comes to classes. It starts with the usage of the keyword *implements* rather than assigning the interface as a type with the *colon*.

If the interface is empty, the class still functions without any errors. If however we add a property to the interface, then that particular property must exist in the class, and conform to the interface rules.

The interface can also list **optional properties** as described earlier on the topic about interfaces (see table of contents).

- b) Another thing to know about class interfaces is that, methods can only be stated on the interface if they are *instance methods* (those that will only exist after we create an object).

From our class example, we have two instance methods: **printRating** and **listIt**. The other method, the **constructor**, is a *static method*. Static methods are those that exist in memory from the start, and can be invoked at any time. We invoke the constructor method at the time the object is instantiated with the keyword *new*, not after we instantiate the object. Static methods are permanently resident in the program. Static methods will not be recognized by the interface and therefore we cannot use them in our interface implementation.

In other words, if we ever decide to include all the members of the class in an interface, we **skip the constructor**.

Here's how I would implement the rest of the class properties on our *IAuto* interface example: [57]

icontemp.com/typ/class7.txt

Extending classes by using other classes

A word of caution: Do not use inheritance for the sake of inheritance. We need to know how to use it, but too much inheritance adds to complexity. The idea of DRY (do not repeat yourself) is not always true. It makes maintenance easier since we do not have to modify multiple takes of the same code, but in many occasions, it is easier to repeat certain properties even if we risk spending more time later maintaining the code. Before using inheritance, think of the reasons why you are going to do it that way. Coding does not have to emulate real life patterns.

a) A simple extended class

Extending classes by using functionality from other classes is a very common practice in programming, and TypeScript brings this capability to JavaScript by simplifying its implementation.

In my simple example below, we have a class named *Citizen* and another class named *Resident*. For simplicity's sake, *Citizen* only has functionality to get the *name* of a person *object*, and *Resident* has no functionality of its own, it only implements the functionality available in *Citizen*. This may not seem to be practical but it illustrates how we connect one class to the other.

1- First we create the base or parent class Citizen:

Please try along with me

```
class Citizen{  
    name:string;  
    constructor(CName:string) {  
        this.name = CName;  
    }  
}
```

Note: I could have named the constructor's parameter as *name* to coincide with its property counterpart, but by calling it *CName* will better illustrate the explanation on step 14 because we can tell which keyword is being referred to when we call this property from the child's class constructor.

Now that we have a parent class we are going to create class *Resident* and extend it to include the properties from class *Citizen*:

```
class Resident extends Citizen{ }
```

Notice the keyword *extends*. That makes *Citizen* the base, or super, or parent class, from which *Resident* is created.

2- At this point, we are ready to instantiate an object from class *Resident*.

I'll call this object *tony* and populate its property name, which [property] comes from the super class *Citizen*:

```
var tony = new Resident("Tony de Araujo");
```

3- Finally, we can verify the data in this new object as follows:

```
alert("The new resident's name is " + tony.name);
```

Summary:

- To add functionality to a class from an existing class we use the keyword *extends*.
- After we extend a *derived* class, we have access to the properties of the *super* class.

If you have written the code along with me, it may be a good idea to erase everything and try creating this exercise once again. Then continue on to the next exercise.

b) Adding one more property to the super class

- 4- Continuing with the previous class project, let's add a *phone* property to the parent class Citizen:

```
class Citizen{
    name:string;
    phone:string;
    constructor(CName:string, CPhone:string) {
        this.name = CName;
        this.phone = CPhone;
    }
}
```

Notice how now the object *tony* is underlined in red (if you still have it on your playground from the previous exercise). This is because the object *tony* declaration no longer matches the constructor from the super class.

5- We need to add the phone number to object *tony* when we instantiate the object:

```
var tony = new Resident("Tony de Araujo", "3442345");
```

I'm making my *phone* property a *string* type property, hence the quotes.

6- Comment or erase the previous *alert* (from step 4) and add a new *alert* to display *name* and *phone* number:

```
alert("Name: " + tony.name + " Phone: " + tony.phone);
```

My raw file sample can be found on the following link:

[58] <http://icontemp.com/typ/class8b.txt>

DO NOT DISCARD THE EXERCISE

Summary:

It does not matter how many properties the super class has. However, if these properties are included in the super class *constructor*, they need to be called in when we create an object modeled after the super class or after a related child class (as you will see later).

c) What happens when we add a method to the *child* class?

The idea of extending a class to use a *super* class is to get extra functionality but the derived or child class can have functionality of its own as well.

7- Let us create a method in the derived class (*Resident*).

Name the method *listIt* and transfer the very last *alert* from the previous exercise into it.

Remember that we need to replace the object name *tony* by the keyword *this* in order to make it portable:

```
class Resident extends Citizen{
    listIt(){
        alert("Name: " + this.name + " Phone: " + this.phone);
    }
}
```

8- Now at the end of the program, call the method *listIt* on object *tony*:

```
tony.listIt();
```

It should work as before and you should have gotten the following alert:

Name: tony Phone: 3442345

Here is my raw file sample:

[59] icontemp.com/typ/class9.txt

d) What happens when we add a property to the *derived* class?

We have seen how an extended or child class allows for the usage of local methods.

Let's now see if we can also add our own properties to the child class.

9- Add the following string *property* to the child class *Resident*:

```
address:string;
```

10- Also, add the *address* property to the method *listIt* in order to be displayed when we call it.

Example:

```
alert("name: " + this.name + " Phone: " + this.phone + "  
Address: " + this.address);
```

11- Now, manually add the *address data* to object *tony* and then call the *listIt* method on object *tony*:

```
tony.address = "33 Main street";  
tony.listIt();
```

See my complete raw file sample here:

[60] <http://icontemp.com/typ/class10b.txt>

Summary:

We are able to add properties to our child class and then we can populate the data manually.

We haven't seen how to actually populate this data at the time of object instantiation. I will explain that process on the next topic.

The idea is to grasp one concept at a time so that we can actually build awareness. They are all valid and self-contained concepts.

At the end of these exercises, repeat them once again like if they were musical scales.

e) How to add a constructor to the child class

So far, we have been using the constructor from the super (or parent) class *Citizen* because there were no properties to populate with data on the child/derived class. Now that we added an *address* property to the derived class *Resident*, I would like to populate the *address* property field when I instantiate an object. In this case, **we need to coordinate which constructor does what**, the parent constructor and the child constructor. This is easier than it seems.

We never modify a parent or super class. Its constructor remains unchanged. In a way the super class is like a closed box; it is what it is and we can only customize the child class. This makes sense because the parent class does not know the child class. The child class is tapping into the parent class but not the other way around.

What we need to do is to create a constructor for the child class based on the sequence of properties we want to populate.

12- In other words, in this new constructor, we copy the parameters from the parent's class constructor to the child's class constructor, then we add the child newer parameters to it:

Build the following constructor in the child's class:

```
constructor (CName:string, CPhone:string, myAddress:string) {  
    }
```

myAddress is the name I have chosen to represent this parameter. You can name it differently.

Then, in the body of the child's class constructor, assign ***myAddress to this.address*** as we normally do with all other properties. However, **we need to connect the foreign parameters (*CName* and *CPhone*) to their respective properties at the parent class and this should be done first**, at the very top of the constructor's body. We do this by using a method known as ***super ()***.

13- Here's how connect the two constructors:

```
constructor (CName:string,CPhone:string,myAddress:string) {  
    super (CName,CPhone) ;  
    this.address = myAddress;  
}
```

You can see my own code and a brief explanation on the following link: *[61]*

jsplain.com/javascript/index.php/Thread/165-TypeScript-An-Extended-class-Sample

I have also included a rudimental illustration of this concept on the next page.

```

3 class Citizen{
4     name:string;
5     phone:string;
6     constructor(CName:string, CPhone){
7         this.name = CName;
8         this.phone = CPhone;
9     }
10 }
11
12 class Resident extends Citizen{
13     address:string;
14     constructor(CName:string,CPhone:string,myAddress:string){
15         this.address = myAddress;
16         super(CName,CPhone);
17     }
18     listIt(){
19         alert("name: " + this.name + " Phone: " + this.phone + " Address: " + this.address);
20     }
21 }
22
23 var tony = new Resident("Tony de Araujo", "3442345", "33 Main Street");
24
25 //tony.address = "33 Main street";
26
27 tony.listIt();|

```

Fig 24 How the child class' constructor fetches parameters from the parent class' constructor.

CORRECTION: Please reverse lines 16 and 15. `super()` should always be the first line on the constructor's body. It didn't matter when I first published this book but that has been changed.

Summary:

Whenever we need to use a constructor on the child class, we first invoke the **super()** method to connect the real parent's parameters to the parameters being used in the child's class constructor, and then we link the local properties to the local parameters as illustrated on lines 15 and 16 (in reversed).

We never modify the parent class because the parent class is independent from those classes that use it to extend their functionality.

Once we add a constructor to the child class, this constructor takes over the instantiation of an object, and the parent constructor needs to be invoked which is done via the `super()` method.

What does the *super()* class method do?

It says: "initialize my parent class before you initialize me" and that is done by calling its default constructor.

LAB WORK: Extending classes

I hope you have tested the previous class and class inheriting exercises along with me. Here is another example for you to assess your understanding of the topic. Do not worry if you are still a bit confused with certain details, learning a language starts with a basic understanding of the topic but it is the practicing, the drilling and the challenges we encounter that open the doors for a higher awareness of concepts.

On this next lab project, imagine that you are writing a program for a car dealership and you want to create a main class to gather all the common data details, and then you want another class for describing used cars, and a third class for describing new cars.

We want three classes in total: **General class**, **Used Car class** and **New Car class**.

The challenge is to include all the common details on the main class, and some details pertaining to the used car or new car classification in their respective classes.

For my own example, I am going to simplify the details but please feel free to improve the program by making it as complex as you wish because in the end, you need to master these concepts on your own terms.

Here is the basic information about these three classes:

- Main class name: **AutoMainClass**
- Used car class name: **UsedCarClass**
- (The new car class will be created later.
It will inherit some properties from the other existing classes)
- Properties included in my **AutoMainClass**:
make:string, model:string,
- Properties included in my UsedCarClass:
year:number, mileage:number, condition:string

Exercise 1: The AutoMainClass

- a) Create a *class* named *AutoMainClass*.

Include the following properties as stated earlier:

make:string, model:string

- b) Then, create a ***constructor*** to propagate the data when we instantiate objects based on this class.

I will call my own parameters ***AMake*** and ***AModel*** so that they look different from the property names.

Here's my own version of this class:

[62] icontemp.com/typ/class11.txt

Exercise 2: The UsedCarClass

- c) Below the existing class, create another class named *UsedCarClass*.
Include the following properties:

year:number, mileage:number, condition:string

- d) Then, ***extend*** the class to include the functionality of class *AutoMainClass*.
e) Finally, add a ***constructor*** in the used car class that fetches the parameters from *AutoMainClass* as well as the properties of *UsedCarClass*.

Make sure the ***super()*** method is called before any other parameter.

Note: This time, I am using the same name of the original properties of class *UsedCarClass* to name my parameters in the constructor.

Here's my own version of this class:

[63] icontemp.com/typ/class12.txt

Exercise 3: Instantiating a used car object

- f) Instantiate an object from *UsedCarClass* and pass in all the data requested by the parameters of this and the parent classes. I am using my data as follows:

```
Object name: "ford123",  
Make: "Ford",  
Model: "Mustang",  
Year: 1983,  
Mileage: 77000,  
Condition: "fair"
```

- g) Then, **call** one of the properties just to test your script.
I'm using *alert()* and calling the property *mileage* of object *ford123* which gets me 77000 on the screen.

Here is my own complete version of the program: [64]

jsplain.com/javascript/index.php/Thread/166-TypeScript-Sample-of-class-inheritance

Exercise 4: The NewCarClass

Question: Can we extend a class to include more than one class?

When it comes to extending classes, we can only extend a class **to include one more class**. This is in contrast with interfaces where we can subscribe to more than one interface. Therefore, if we want to create a *NewCarClass* from properties of both *AutoMainClass* and *UsedCarClass*, it is not going to work because we can only extend a class to include another class.

However, we could extend the *NewCarClass* to include the *UsedCarClass* and this will automatically bring in the properties from the *AutoMainClass* **by inheritance** because the *UsedCarClass* extends to *AutoMainClass*.

Steps to perform

- h) Your challenge is to create a new class named *NewCarClass* and extend it to include the *UsedCarClass*.
- i) Also, since a new car is normally in good condition, make the *condition* parameter to default to “new” (in quotes since it is a string value).

Another way would be to make it optional (remember the **?** mark), but for now, let's make it defaulting to “new”.

- j) Please feel free to add other properties pertaining to new cars. I will not add any new properties myself since I want my example to be as simple as possible.
- k) Finally, instantiate a new object from class *NewCarClass* and display a few of the properties to test your code. I am going to use the following data for my own test (notice that there is no car *condition* stated since it defaults to *new*):

```
var mySubaru = new NewCarClass("Subaru", "Outback", 2016,
0);
alert(mySubaru.mileage);
alert(mySubaru.condition);
```

Here is my raw file sample: [65]

jsplain.com/javascript/index.php/Thread/167-TypeScript-Sample-of-multiple-class-inheritance

Private and public properties

Many times one programmer creates a class, and then the class is reused by other programmers in their code, and for this and other reasons, it is common to protect certain properties of a class by limiting direct access to them. In this case, the user of the class can only access these private properties through the implementation of public methods, which act as interfaces (connections) to private properties.

Note: This privacy only pertains to TypeScript, and it is used by the interpreter to help you write better and safer code. The JavaScript implementation after you transpile from TypeScript will not reflect this explicit privacy.

In this chapter, we will explore the following topics:

- a) **Public** properties
- b) *TypeScript 2.0: The readonly* property
- c) **Private** properties
- d) **Getting**: enabling reading data from private properties
- e) **Setting**: enabling adding or editing data in private properties
- f) Instantiating an object and **accessing** its private properties

a) Public properties

By default, in TypeScript all properties and methods are **public**. This means that they can be accessed by their instantiated objects without restrictions.

Example:

```
class Test{  
  color: string;  
}
```

The property *color* is public and it can be directly accessed by any instantiated object.

A more explicit way to write the above class is by using the keyword *public* before the property declaration:

```
class Test{  
  public color: string;  
}
```

The keyword *public* is optional, but it states the intent of the original programmer.

b) Readonly – a TypeScript 2.0 feature

Before we discuss private properties let me just introduce a new TypeScript 2.0 addition that can be used to prevent public properties from being modified.

By adding a **readonly** modifier, public properties become immutable. This simplifies the writing of code and prevents unforeseen errors if we don't pay attention.

Taken the previous code as an example, this is how we make the public property **color** immutable:

```
class Test{  
public readonly color: string = "orange";  
}
```

From this point on, the value of public property color is always "orange".

c) Private properties

If for some reason we want to make sure that a certain property does not change its *value* while we work on the project, then we declare the property access in the class as ***private***:

```
class Test{  
    color: string;  
    private _intensity: number;  
}
```

Now, after instantiating an object, we can add or modify the data of property *color*.

However, if we try to change the property *_intensity*, TypeScript will show a red line under our code to warn us that we are breaking the rules:

```
var x = new Test();  
x.color = "blue";  
x._intensity = 5; <-- This will be underlined in red to warn you that you shouldn't do this.
```

Note: It is common (but not mandatory) to begin the name of private properties with an underscore. This serves as an instant visual recognition of privacy for programmers inspecting the code later.

Here's another example. If you are coding along with me, please use this class for the exercise steps that follow.

```
class Test{  
    public color:string;  
    private _intensity: number;  
}
```

Question:

How do we *get* the value from a private property, or *set* a new value for a private property?

Let's look at getting and setting next.

d) Getting: enabling reading data from private properties

In order to *get* or *set* private properties we need to create two methods in the class: one method to read data, and/or another method to modify the data.

These methods use the predefined keywords *get* and *set* and their purpose is to either get data from the property or set a new value to the property.

Note: *get* and *set* are only recognized by **JavaScript 5** or above. To compile it outside of the *TypeScript playground* you may have to pass **-target ES5** to the compiler. Just search the term to find out how to do it for whatever compiler you want to use.

1- Let's start with the *get*. Write the following line as the third item on your class:

```
get intensity() { }
```

At this point, TypeScript does not know which property you are going to get data from because there is no reference to the property yet (the name *intensity* is just coincidental and somewhat conventional. I will explain more about it further down the page).

As soon as you write the above code in the class, you will get an error stating that a *get* method must always *return* a value. This is because we did not include the *return* statement, which is mandatory on a *get* method, as you will see.

- 2- Since the property `_intensity` is of type *number*, we also need to specify the return type for this method as *number*.

Here's the complete *get* method script for property `_intensity`:

```
get intensity():number{  
    return this._intensity;  
}
```

Please look at the complexity of the JavaScript version on the right.

The choice of *function name* for a *get* method does not matter. I have written *intensity* (without the underscore) so that we (humans) know what it does based on the other information in the class.

However, you cannot write `_intensity` (with underscore) because that name is already being used by the property itself, but *myIntensity* or even *xyz* would work as well.

Do not forget the '*this*' which serves as a placeholder for the object name calling the method. Also, sometimes and for security reasons, we may want to hide the real name of the property from the outside world. In that case, we could use a bogus name for the property inside of the class, like for example `_intensity123`.

Just know that the name of the *get* method does not have to coincide with the real property name.

Now we have a method to *read* the private property `_intensity` from the outside. We will see how to read it, on step (e).

e) Setting: enabling adding or editing data in private properties

3- To add data to/or modify data in a property, we use a *set* method:

```
set intensity(x:number) {  
    this._intensity = x;  
}
```

Notice how the same function name, *intensity*, is used for both *get* and *set*. TypeScript implicitly knows which method to use based on your request. Also, the parameter *x* could be called anything because it is an independent variable. I have chosen *x* to remove any doubts from your mind about semantics as you analyze the code.

Notice that there is no *return* statement declaration on a set method.

Question:

Can we use a different name for the *set* method other than the name we've already used for the *get* method?

The answer is yes we can. However, since TypeScript allows the same name to be used in both methods, it is probably best to use *intensity* for both *get* and *set* because it simplifies our thought process.

We can now modify the data on property *_intensity* from the outside. We will see how to do that next.

Note: If the intention is to prevent a certain property from being modified at all, we can skip the *set* method and just keep the *get* method so that we are able to read the value in the property. Without the *set* method, there is no editing access to it.

f) Instantiating an object and accessing its private properties

4- At this point, let's instantiate an object and add some properties to it:

```
var xyz = new Test();  
xyz.color = "blue";  
xyz.intensity = 8;  
alert(xyz.intensity);
```

On **line 1**, object *xyz* has been instantiated.

On **line 2**, the value *blue* was added to property *color*.

On **line 3**, the value of *8* was added to property *_intensity*.

Notice how I used *xyz.intensity* like if *intensity* was the real name of the property *_intensity*. This is how we hide the real name of the property; we use a setter method to do the internal work for us. It is almost like going to a Bank and asking the teller to deposit some money on our own account for us. We do not have access to the vault, we need to use a middle person to access the vault and deposit our money there. The same principal applies to [66] [getter and setter methods](#).

On **line 4**, I am alerting the value of *xyz*'s property *_intensity* and we should see the number 8 on the screen when we run the program.

As you can see, the methods *get* and *set*, as well as the property *_intensity*, are invisible to the outer world. The alias name for our example is *intensity*, which emulates the property name.

To look at my complete raw file please refer to the following link: [67]

jsplain.com/javascript/index.php/Thread/169-TypeScript-Getter-and-Setter-Sample

Static properties in a class

On this section, you will find the following topics:

- a) What is a static property?**
- b) How to access a static property from within an object**
- c) Can static properties be modified?**
- d) Aside from static properties we can also create static methods**

a) What is a static property?

A static property is a property associated with the class itself. What I mean by this is that, there is **no association with any instantiated object**.

The value in this property is always available through **direct access** (calling the class without having to instantiate an object).

Static properties may look like *constants* but they are not constants because they can be modified either directly, or via a *set* method. To prevent a static property from being modified, we would have to make it a *private static* property.

Here is an easy example of a static property:

```
class SomeClass{  
    static author = "Tony de Araujo";  
    currentLocation = "New York";  
}
```

In this class, the property *author* is *static*. This means that we can call it directly by referencing to the class and without having to instantiate an object:

```
alert(SomeClass.author);
```

The above *alert* displays the message on the screen as *Tony de Araujo*.

However, if you try to directly display the other property, *currentLocation*,

```
alert(SomeClass.currentLocation);
```

you will get a red mark and an error stating that property *currentLocation* does not exist.

This is because a non-static property only exists in instantiated objects (it is a dynamic property):

```
var someObject = new SomeClass();  
alert(someObject.currentLocation);
```

By instantiating an object, I can now call non-static properties from the class without a problem and *New York* is displayed on the screen when I run the program.

On the other hand, if I try to call property *author* from the object *someObject*, I will get an error as well.

This is because property *author* can only be seen when we call it directly from the class.

```
alert(someObject.author); <--- ERROR
```

Summary:

- Static properties are always available since we can call them directly by referring to the class.
- Dynamic properties are only available by referring to an instantiated object.

Question:

What if I want to use the static property of a class, but from within an instantiated object?

I will address that possibility next.

b) How to access a static property from within an object

If we want to access a class static property from an instantiated object, we need to create a class method to allow such accessibility.

That's right -- the authorization for an object to access a static property must come from the class itself. This is because static properties are on a different context than the instantiated object:

```
class SomeClass{
    static author = "Tony de Araujo";
    currentLocation = "New York";
    printAuthor():string{
        return SomeClass.author;
    }
}
```

The instantiated object accesses the *printAuthor* method, which in turn has access to the static property, *author*:

```
var someObject = new SomeClass();
alert(someObject.printAuthor());
```

c) Can static properties be modified?

- 1- Static class properties can be modified directly, like in the following example:

```
SomeClass.author = "Ada Lovelace";
```

The above statement will change the author's name from *Tony de Araujo* to [68] [Ada Lovelace](#) (known as the first computer programmer).

- 2- In contrast, however, static class properties **cannot be modified** from an **instantiated object**, unless we create a **set** method to allow such modification.
- 3- In order to prevent anyone from modifying a class static property, we need to make it private. Then we allow it to be utilized by supplying a **get** method (no **set** method included).

Example:

```
class SomeClass{
    private static theAuthor = "Tony de Araujo";
    currentLocation = "New York";
    static get author():string{
        return SomeClass.theAuthor;
    }
}
alert(SomeClass.author);
```

- 4- Now, since the static property is private, the only way to access the *static* property directly from the class is to call the static *get* method.
- 5- Since there is no *set* method, the static property *theAuthor* is not editable.

6- However, as stated earlier on section (b), we cannot access *static* properties **from instantiated objects** unless we create a method to help us access the static property.

Since we have made this property *private*, the method to access it from an object needs to be a ***get*** method (versus a regular method), and since the current *get* method is also ***static***, we need to create a second *get* method (a *non-static get* method) to allow accessibility from objects.

Please look at the following *raw file* sample to get the idea. Notice how both methods have the same name. Although this is not mandatory, using the same name makes sense and it will not cause conflict because instantiated objects will only see the regular *get* method:

[69] icontemp.com/typ/class13.txt

d) Aside from static properties we can also create static methods

Just like static properties, static methods are methods we can directly access from the class (without having to instantiate an object).

We have already seen a *static* method on our examples, the static *get* method we created earlier.

The idea behind static methods is the same as static properties and I am not going to expand it any further. However, please stop for a moment and think about a possible static method and how would you implement it. The only way to master this stuff is to understand, practice, meditate on the subject and experiment with different possibilities.

Learn to code as if you were a mad scientist.

On the next lab session, we are going to cement these concepts in case you may have found it confusing.

Let practice a bit more!

LAB WORK: Adding static properties to classes

For this project, please copy the sample script shown on the link below. This is a class script we have made earlier. Let's add a static property to it. [70]

jsplain.com/javascript/index.php/Thread/166-TypeScript-Sample-of-class-inheritance

Action:

- 1- In the class *AutoMainClass*, add a **static** property named ***nameOfDealership***.
Assign the string value "*AutoMall*" to the above static property.
- 2- Since we want all instantiated objects to have access to the data in this variable, add a regular **method** to this class with the name ***storeName***.
This will allow instantiated objects to read the data from *nameOfDealership*.
(Use a *return statement* to export the value from the function).
- 3- Finally, test the script by alerting the name of the store as applied to object *ford123*.

Here is my complete script (I have added "**Dealer name:** " to my alert in order to make the message more obvious: [71])

jsplain.com/javascript/index.php/Thread/170-TypeScript-Adding-a-static-property-to-a-class

One more step

The dealership owner decided that she wants the store name to be permanent on the program so that no one can change it.

Based on what we have covered thus far, how would you go about making the static *nameOfDealership* property immutable from the outside but still available as read-only via any instantiated object and directly from the class?

It may be a good idea to review static properties again to refresh your mind.

You can find my raw file solution on the following link:

[72] icontemp.com/typ/class14.txt

9- Namespaces

What are namespaces and modules?

As you know, the only way to encapsulate code in *common JavaScript* (ES5) is by using a function.

In a **function**, the code inside of the *code block* can be privatized so that it does not interfere with external code that may be using similar variable names.

In the next JavaScript version (ES6), we will be able to isolate code by placing it inside of any *code block*, not just in functions. This, however, is not yet available since most browsers do not support it. TypeScript circumvents this problem by renaming variables that have similar names.

Going back to namespaces and modules, as of TypeScript 1.5, internal modules are simply called **namespaces**. Now, when we refer to a **module**, we now are referring to an external folder that we link to our main code so that we can use its functionality.

A namespace is an independent block of code that we can reuse by referring to it.

Let's look at a couple of examples.

The following image represents two functions. They both do a calculation. We can group these and other similar functions in a namespace dedicated to *math* so that we can reuse the functionality for different applications.

```
1 let add = function (...num: number[]): number{
2     let total = 0;
3     for (let i = 0; i < num.length; i++){
4         total += num[i];
5     }
6     return total;
7 }
8
9 let multiply = function (...num: number[]): number{
10    let total = num[0];
11    for (let i = 1; i < num.length; i++){
12        total *= num[i];
13    }
14    return total;
15 }
16
17 alert(add(1, 2, 3, 4, 5)); //15
18
19 alert(multiply(3, 4, 2)); //24
```

Fig 25 Raw file:[73] <http://icontemp.com/typ/namespace1.txt>

To create a namespace named Calculator we write the following code:

```
namespace Calculator{}
```

Please note that, conventionally, the first character of Calculator is in upper case.

Then we insert the two functions (lines 1 through 15) inside of the curly braces.

Now, in order to use these functions you need to invoke them with the fully qualified address, which is (for the *add* function):

```
Calculator.add(1,2,3,4,5) ;//error
```

However, TypeScript does not allow you to access this function because you need to make the function **exportable** from within the namespace.

Insert the keyword **export** to the left of the **let** keyword for every function or property you want to export from the namespace. See the next image for details.

```
1 namespace Calculator {
2
3   export let add = function (...num: number[]): number {
4     let total = 0;
5     for (let i = 0; i < num.length; i++) {
6       total += num[i];
7     }
8     return total;
9   }
10
11   export let multiply = function (...num: number[]): number {
12     let total = num[0];
13     for (let i = 1; i < num.length; i++) {
14       total *= num[i];
15     }
16     return total;
17   }
18 }
19
20 alert(Calculator.add(1, 2, 3, 4, 5)); //15
21 |
22 alert(Calculator.multiply(3, 4, 2)); //24
```

Fig 26 Raw file:[74] <http://icontemp.com/typ/namespace2.txt>

Now we can use it from the outside. This is, of course, still a single file but the Calculator code is now isolated from the rest of the script.

Creating an alias for namespace functions

If you intend to use the namespace `Calculator` often in your program, you can create an alias for the functions you are most likely to repeat calling and use the shortcut instead.

Here's an example of a shortcut creation for the *add* function.

The alias is created outside of the namespace, in the environment you are going to use the function:

```
let add = Calculator.add;
```

From now on, all you have to do is to write **add(2,3,4) ;** instead of the long qualified version.

Using external Modules

This book was purposely written to use the *TypeScript Playground* in order to focus on the language syntax. Therefore, we cannot really test an external Module.

Modules are covered on Volume II. In that volume, we will download TypeScript to our system and learn how to setup an internal server for testing purposes. Then we can work with modules.

Volume II will also cover an introduction to **Angular 2** using *Angular-cli*. All the knowledge acquired in the volume will start to pay off when we get to volume II.

Please check my page on Amazon to see if it is available. The publication target date is November 2016. [75]

<https://www.amazon.com/Tony-de-Araujo/e/B00D7V08WY/>

10- Casting and Conversion

Parsing strings to numbers

As you probably know from JavaScript, a *prompt()* converts any user input to a *string* value.

If we write the following code:

```
var y:string = prompt("Please enter a number");
```

TypeScript will accept it without a problem. However, we may get the wrong result.

Take for example the following script:

```
var y:string = prompt("Please enter a number");  
alert(y + 2);
```

If, at the prompt we enter the number 45, the output result will be 452, which is incorrect. This is because JavaScript treats the input as a *string* value and concatenates the 45 to the 2 as a *string* value instead of adding them up.

We could change variable *y* to a number type variable but TypeScript will red flag it because it knows that *prompt()* will only accept string values.

Therefore, the solution is to *parse* the input to an *integer*, which is a numeric value:

```
var y:number = parseInt(prompt("Enter a number"));  
alert(y + 2);
```

Now when we enter the value 45 we get 47 as a result. The value was converted to a *number* and the calculation is correct.

We could also have done it this way:

```
var y = prompt("Enter a number");  
var x:number = parseInt(y);  
alert(y + 2);
```

Aside from **parseInt()**, we can also **parseFloat()** which returns a floating number instead of an integer when the input is a [float](#).

Using the toString conversion method

Sometimes we want a value to be reduced to a *string* type.

Take the following array as an example:

```
var myColors = ["blue", "red", "white"];
```

When we console.log it,

```
console.log(myColors);
```

the result is displayed in an array format (by showing index numbers and respective values).

Actually, any output to the screen is a string value regardless of what type it is, but what if we just want the values by themselves (without being indexed)?

We could convert the *array* to *string* before it outputs:

```
console.log(myColors.toString());
```

The method *toString* can also be used to assign a value to a variable:

```
var x = myColors.toString();  
console.log(x);
```

Here's another example of converting a *boolean* type to a *string* type:

```
var myBool = true;  
var x:string = myBool.toString();
```

Now the value of x is a string "true", not a boolean **true**.

Type Compatibility

When one variable is of a certain type and a second variable is of a different type, we can assign the data of the second variable to the first variable, but only if the type-members of the first variable are contained within the type of the second variable.

Example:

```
interface A { apples:number }
```

```
var x : A;
```

```
var y = { bananas: 1, apples: 5, oranges: 3}
```

```
x = y;
```

//it's ok to assign y to x because all members of type A are included in y with no exception. Therefore, x is compatible to y.

Please paste the above four lines on the playground so that we can test them. Then read the explanation on the next page.

We are going to compare the compatibility of variable x and variable y:

- a) The interface representing type *A* lists a property named "apples" which is declared as a *number* type value.
- b) Variable *x* subscribes the characteristics of *A*
- c) Variable *y* is an object with no subscriptions and it contains properties named as "oranges", "apples" and "bananas". All these properties are of type *number*.
- d) Since variable *x* represents type *A* which has a numeric property named "apples" and "apples" is also included in variable *y*, we can safely assign *y* to *x* because variable *x* is fully compatible to *y* and it is going to be extended to include more types rather than less.

The opposite assignment (*y* = *x*), would not work.

If you remove "apples" from interface *A*, the *y* to *x* assignment still works because there are no incompatibilities.

However, if you change "apples" to "figs" on interface *A* (or add "figs" as a second property), now *x* and *y* are no longer compatible because "figs" does not belong to *y*. The TypeScript interpreter will red flag the assignment.

Only members of the *targeted type* are checked for compatibility. In this case, only *x* is checked to see if type *A* is compatible to type *y*. That is the reason why we can put "bananas" and "oranges" in *x*, but we cannot put *x* into *y*.

For further reading, please refer to the following typescriptlang.org link: [76]

<http://www.typescriptlang.org/docs/handbook/type-compatibility.html>

I will also expand this topic here: [77]

jsplain.com/javascript/index.php/Thread/171-TypeScript-Type-Compatibility

11- Last but not least

Where to go from here

Congratulations, and thanks for coming along with me to the very end of this volume.

This project is by design, incomplete. We have come a long way and now it is up to you since your interests and needs may be different from mine.

- 1- What I would do at this point is to read the book once again and do the exercises as a refresher. This is because you now have a broader view of what's involved and you might have questions which will force you to do research and experiment with the exercises.
- 2- You can also go straight to Volume II and start creating your own local projects. Then learn Angular 2. Volume II will introduce you to the TypeScript development server as well as the Angular 2 development server via Angular-cli.
- 3- An important source of reference is the official **TypeScript Language Specification** webpage on Github. This may not be the easiest way to see how things work, but at this point I believe you will start to take advantage of this information: [78]
<https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md>
- 4- A great source for questions and answers is Stackoverflow:
[79] stackoverflow.com/questions/tagged/typescript
- 5- [80] DefinitelyTyped.org is the repository for high quality TypeScript type definitions.
- 6- [81] Wikipedia.org contains an independent page about TypeScript with great resources.
- 7- I will also continue writing about TypeScript on my own resources platform:
[82] <http://jsplain.com/javascript/index.php/Board/16-TypeScript-notes-open-forum/>

In conclusion

Thank you so much for investing your time doing these exercises. I hope they serve you well!

If you can spare another 5 minutes, please leave a good word on Amazon to help me promote this book. The future of this book is in your hands, my friend.

Here is the link to the book page on Amazon.com: [75]

<https://www.amazon.com/Tony-de-Araujo/e/B00D7V08WY/>

I will continue to advance the topic with more exercises and useful tips on JSPLAIN.com, as well as in future related publications.

Good luck ahead!

Tony de Araujo

NJ, USA