Function Basics

```
let greet = () => {
  console.log("Hey!");
}

~~greet = "12";~~
```

If you implicitly define a function you cannot change that variable to a different data type

```
let greet: Function;

greet = () => {
  console.log("Hey!");
}
```

When explicitly defined as a function, you must use the capitol "F" to do so

```
let adder: Function;

adder = (a: number, b: number) => {
  console.log(a + b);
}
```

```
const adder = (a: number, b: number) => {
  console.log(a + b);
}
```

You can specify input variable data types

These accomplish the same thing

Using const here is possible

Using const here wouldn't work because it would render the function immutable

union type (uses pipes)

This ? makes the 'c' variable optional

```
const adder = (a: number, b: number, c?: number | string) => {
  console.log(a + b + c);
}
```

This | (aka 'pipe') allows the c variable to be a number or a string

```
const adder = (a: number, b: number, c: number | string = 10) => {
  console.log(a + b + c);
}

add(5, 10);      //=> 15
add(5, 10, 20);  //=> 35
```

Instead of making the 'c' value optional, you can give it a default value and choose whether or not to use it when the function is called

```
const minus = (a: number, b: number) => {
  return a - b;
}

let result = minus(10, 3);
```

This will have a number data type because TypeScript will infer the data type produced by a return value in a function

```
const minus = (a: number, b: number): number => {
  // some really big function
  // return value data type unclear
}
```

You can explicitly define the function's return value data type in case it is not clear to other developers

```
const minus = (a: number, b: number): void => {
  console.log(a - b);
}
```

If the function doesn't have a return statement it still returns a value, but as far as TypeScript is concearned, it has a return value data type of void You must have a return statement inside a function in order to explicitly define the function's return value data type as anything other than void

type alias

The input variables for this function are verbose

```
const greeter = (user: {name: string, uid: string | number}) => {
  console.log(`${user.name} says hello!`);
}
```

```
type StringOrNum = string | number;
type userObject = {name: string, uid: StringOrNum};

const greeter = (user: userObject) => {
  console.log(`${user.name} says hello!`);
}
```

The input variables for this function are defined in type aliases that can be reused in other functions

Generics in TypeScript

Generics in TypeScript allow us to create reusable blocks of code which can be used with different types

This is a standard function it takes in an object as a parameter but the object's proiperties don't matter

```
const addUID = (obj: object) => {
  let uid = Math.floor(Math.random() * 100);
  return {...obj, uid};
}
```

The spread operator is used to add the uid variable to the end of the object like so

```
let docOne = addUID({name: "yoshi", age: 40});
```

Now let's create a new variable called docOne and make it equal to the return value of the addUID() function that is passed in the object

```
console.log(docOne);
//=> {name: "yoshi", age: 40, uid: 52}
```

This works just fine but…

```
~~console.log(docOne.name);~~
//=> This will not work and produce an error
```

…this will not work and will produce an error because when we pass in an object into the addUID() function, we're not specifying what exactly this object should be, and when the addUID() returns the new object, what properties were on the original object; those need to be captured with a generic

```
const addWorkingUID = <T extends {name: string}>(obj: T) => {
  let uid = Math.floor(Math.random() * 100);
  return {...obj, uid};
}
```

we solve this by adding a <T> for "type" and specify the data type being extended

This {name: string } is optional and more specific; it is saying that we will only extend the object if it happens to have a key called name with a value set to the data type of string

<T>

generics with interfaces

https://www.youtube.com/watch?v=IOzkOXSz9gE&list=PL4cUxeGkcC9gUgr39Q_yD6v-bSyMwKPUI&index=18
TypeScript Tutorial #18 - Generics (5:26 / 9:43 = Generics with Interfaces)

Sometimes we want to use an interface with some flexibility; here, the Resource interface accepts a generic (<T>) as a parameter so that when a variable is declared and told to comply with the interface, that variable can specify what data type it want to pass in for the generic

interface

```
interface Resource<T> {
  uid: number;
  resourceName: string;
  data: T;
}
```

variable

```
const docThree: Resource<object> = {
  uid: 1,
  resourceName: 'person',
  data: {name: 'shaun'}
}
```

This variable passes in an object for the interface data value

```
const docFour: Resource<string[]> = {
  uid: 2,
  resourceName: 'shoppingList',
  data: ['bread', 'milk', 'sugar']
}
```

This variable passes in an array of strings for the interface data value

```
console.log(docThree, docFour);
//=> {uid: 1, resourceName: "person", data: {…}}
     {uid: 2, resourceName: "shoppingList", data: Array(3)}
```

both of them work

Explicit Types(:)

TypeScript is a statically typed language and uses uses strict types. JavaScript does not use strict types and is dynamically typed.

variable declaration-variable-name: data-type
let character string

```
let character: string;
character = 'Dan';
character = 'Tom';
~~character = 23;~~

let age: number;
age = 23;
age = 34;
~~age = true;~~

let group: string[] = [];
group = ['Tom', 'Dan'];
group[2] = "Sam";
~~group[2] = 34;~~

group.push('Bob')
```

If a variable's type is explicitly defined, the value cannot change to a different data type later on

Arrays

An array can be defined as only containing strings

let array[] = [];

If the array is instanatiated as empty, you cannot push to the array if you do not specify it's value is an empty array first.

Union Types (...|...)[]=[]

```
let group: (string|number)[] = [];
group = ['Tom', 342];
group[2] = "Sam";
~~group[2] = false;~~

group.push('Bob');

let uid: string | number | boolean;
uid = "Sam";
uid = 23;
uid = true;
~~uid = [1, 2, 3];~~

let ninja1: object;

ninja1 = [1, 2, 3]

ninja1 = {name: "Raphael"}
```

Union types allow you to have 2 or more data types in an array such as:

string • number

Union types can be used for any variable and don't need "()" if they're in front of normal data type.

This works because arrays are a type of object…
But this will produce an error b/c it's previously defined as an array above

explicitly type variables before assigning variables

```
let ninja1: {
  name: string,
  weapon: string,
  age: number,
  qualified: boolean
}
```

These can only change to a string
This can only change to a number
This can only change to a boolean

```
ninja1 = {
  name: 'Raphael',
  weapon: 'sais',
  age: 14,
  qualified: false
}
```

If I alter this object in any way, I must keep the exact same structure as when the object was originally declared. I can not add or remove a key-value pair

Function Signature

Function signatures describe the general structure of a function, what arguments it takes in, and what type of data it returns

we can name these keys anything we want to

```
let greet: (x: string, y: string) => void;
```

function signature

```
greet = (name: string, greeting: string) => {
  console.log(`${name} says hi!`);
}
```

function itself

The function complies with the function signature because it has the correct return value and the same variable data types and count

If a function does not have a return statement, its return value is void, otherwise its return value could be any other data type

```
let logDetails: (myObject: {date: number, age: number}) => number;
```

function signature

```
logDetails = (secretCode: {date: number, age: number}) => {
  return secretCode.date + secretCode.age;
}
```

function itself

```
let mySecretCode = {date: 771957, age: 21};
console.log(logDetails(mySecretCode));
```

calling the function

```
let calculator: (x: number, y: number, z: string) => number;
```

function signature

```
calculator = (numOne: number, numTwo: number, action: string) => {
  if (action === 'add') {
    return numOne + numTwo;
  } else {
    return numOne - numTwo
  }
}
```

function itself

If a function body contains a conditional, then all possible return values must have the same data type as the expected return value data type specified in the function signature

```
console.log(
  calculator(2, 5, 'add')
);
```

calling the function

Dynamic (Any) Types

```
let age: any;
age = 25;
```

```
let age: any = 25;
```

These two code snippets produce the same result

```
age = "seven";
age = 7;
age = object;
age = true
```

With the data type set to any it can now be changed to any data type

Using any pretty much strips out the benefits of using TypeScript and makes it more or less the same as JavaScript

```
let ninja7: {
  name: any,
  weapon: any,
  age: any
};
```

Here the values for any of the three keys can be any data type you want

```
ninja7 = {
  name: 'Raphael',
  weapon: 'sais',
  age: 14,
}
```

You cannot change, add to or delete the name., weapon, and age keys.

```
let ninjas: any[] = [];

ninjas.push('Raphel');
ninjas.push(23);
ninjas.push(false)
```

You can push anything you want onto this array

Enums in TypeScript

Enums are a special type in TypeScript that allow you to store a set of constants or keywords and associate them with a numeric value

```
enum ResourceType { BOOK, AUTHOR, FILM, DIRECTOR };
```

enum

[0] [1] [2] [3]

interface

```
interface SecondResource<T> {
  uid: number;
  resourceType: ResourceType;
  data: T;
}
```

variable

```
const docFive: SecondResource<object> = {
  uid: 12934004,
  resourceType: ResourceType.BOOK,
  data: {title: 'name of the wind'}
}
```

```
const docSix: SecondResource<object> = {
  uid: 12934002,
  resourceType: ResourceType.AUTHOR,
  data: {title: 'name of the sea'}
}
```

```
console.log(docFive); //=> {uid: 12934004, resourceType: 0, data: {…}}
console.log(docSix); //=> {uid: 12934002, resourceType: 1, data: {…}}
```

Classes in TypeScript

In the most basic format, classes have their instance variables defined, then the instance variabels are passed into the constructor

If not specified, the class instance variables will be public

```
class MyClass {
  name: string;
  age: number;
  alive: boolean;

  constructor(x: string, y: number, z: boolean){
    this.name = x;
    this.age = y;
    this.alive = z;
  }

  format(){
    if (this.alive === true) {
      return `${this.name} is ${this.age} years old`
    } else {
      return `${this.name} is already dead`
    }
  }
}
```

we can name these keys anything we want to…

…as long as they match these values

class method returning a template string

class constructor

```
const personOne = new MyClass('Bob', 24, true)
console.log(personOne);
```

new class instance created

```
let rolodex: MyClass[] = []
rolodex.push(personOne)
```

This newly created array called rolodex will only allow instances of MyClass to be added to it

access modifiers

```
class MyClass {
  public name: string;
  private age: number;
  readonly alive: boolean;

  constructor(x: string, y: number, z: boolean){
    this.name = x;
    this.age = y;
    this.alive = z;
  }

  format(){
    if (this.alive === true) {
      return `${this.name} is ${this.age} years old`
    } else {
      return `${this.name} is already dead`
    }
  }

  changeThings(){
    this.name = "Felix"
    this.age = 12
    ~~this.alive === false~~
    return `My name is ${this.name} and I am ${this.age} and it's ${this.alive} that I'm alive`
  }
}
```

typing out public is optional

This can only be accessed inside the class

When an instance variable is readonly it can be read outside the class, but not changed or deleted

Though this will not produce an error, you cannot change a readonly class instance variable in a class method

```
const personOne = new MyClass('Bob', 24, true)
personOne.name = 'Ron'

const personTwo = new MyClass('Sam', 37, false)
console.log(personTwo.alive)

const personThree = new MyClass('Tom', 39, true)
~~console.log(personThree.age)~~

console.log(personOne.changeThings());
//=> My name is Felix and I am 12 and it's true that I'm alive
```

The only way to access age in this case is to call the class method that exposes age within the class:

```
console.log(personThree.format())
//=> Tom is 39 years old
```

because this won't work

simplify

using access modifiers to simplify class constructors

If you use access modifiers for each instance variable, then the instance variables can be defined in the class constructor:

```
export class MyClass {
  public name: string;
  private age: number;
  readonly alive: boolean;

  constructor(x: string, y: number, z: boolean){
    this.name = x;
    this.age = y;
    this.alive = z;
  }
}
```

```
export default class MyClass {
  // public name: string;
  // private age: number;
  // readonly alive: boolean;

  constructor(
    public name: string,
    private age: number,
    readonly alive: boolean.
  ){}
}
```

these are the same

don't forget this

DOM & Type Casting

TypeScript can interact with the DOM just like JavaScript, but it doesn't know if the thing you're selecting on the DOM actually exists or not, so some TypeScript environments will give you a warning that the object is possibly null; to get rid of such an error, add a "!" at the end of the query if you're sure that node exists

```
const anchor = document.querySelector('a')!;
console.log(anchor)
```

This guy here…

```
<a href="https://www.thenetninja.co.uk">The Net Ninja</a>
```

will console log this

reference: https://github.com/Richard-Burd/typescript-sandbox/blob/master/public/index.html

```
console.log(anchor.href) //=> https://www.thenetninja.co.uk
```

TypeScript knows this is an anchor with a (possible) href value, but when you select a node by class name, it could be referring to any element in the HTML page and TypeScript doesn't know if its a form or another HTML element type so we use this typecasting method to tell TypeScript what type of HTML element we are grabbing so that the TypeScript environment (such as with Visual Studio) will display all of the available methods for that HTML element type

```
const myform = document.querySelector('.new-item-form') as HTMLFormElement;
console.log(myform.children);
//=> HTMLCollection(5) [div.field, div.field, div.field, div.field, button]
```

we don't use the "!" when typecasting

https://microsoft.github.io/PowerBI-JavaScript/interfaces/_node_modules_typedoc_node_modules_typescript_lib_lib_dom_d_.htmlelement.html

Interface HTML elements

./project-path/src/app.ts

```
const form = document.querySelector('.new-item-form') as HTMLFormElement;
const type = document.querySelector('#type') as HTMLInputElement;
const tofrom = document.querySelector('#tofrom') as HTMLInputElement;
const details = document.querySelector('#details') as HTMLInputElement;
const amount = document.querySelector('#amount') as HTMLInputElement;
```

selected by class

selected by id

```
form.addEventListener('submit', (e: Event) => {
  e.preventDefault();

  console.log(
    type.value,
    tofrom.value,
    details.value,
    amount.value
  );

}
```

this is a TypeScript event object

listening for the user to submit the form

The values entered by the user into the form will get logged to the console

Tuples in TypeScript

Touples are like arrays but differ in that once the data types in the tuple have been declared in the initialization, those data types cannot change; you cannot add or delete elements in a touple either

```
let myArray = [false, 23, "string"];
myArray[0] = "I can be a string or any other data type";
```

array

Arrays are defined with "="
Touples are defined with ":"

```
let myTuple: [string, number, boolean] = ["a string", 34, false];
myTuple[0] = "I have to be a string!"

~~myTuple[0] = 340~~
//=> This will produce a TypeScript error but still compile
     to JavaScript
```

tuple

```
~~let anotherTuple: [] = ["a string", 34, false]~~
~~anotherTuple[0] = "another string"~~
//=> This will produce a TypeScript error but still compile
     to JavaScript
```

not allowed

Modules in TypeScript

The module generation is controlled by the "module" key in the tsconfig.json file and must be un-commented to use modules

./project-path/index.html

```
<html lang="en">
...
<body>
...
<script type="module" src='app.js'></script>
...
</body>
</html>
```

This needs to be set to type="module" in order to use modules

This is the startpoint from which everything else must be imported this HTML file will look for ./app.js in its same directory

./project-path/app.js

```
import { MyClass } from './classes/MyClass.js'
```

This is .js not .ts because ultimately we are referencing the JavaScript files even though we're working in TypeScript

./project-path/app.js

```
import MyClass from './classes/MyClass.js'
```

When using export default, you don't need to specify the block being exported in the { curly braces } like you do above

There are two drawbacks top using this modular system:

1.) Older browsers do not support this so they may generate an arror

2.) It does not bundle our JavaScript code into a single .js file; this means the browser has to make seperate network requests for each .js file the client wishes to access, this slows down the application Webpack can be used to compile seperate .ts files into a single .js file.

Interfaces with Objects

Interfaces allow you to enforce a certain structure on a class or an object in TypeScript, this functionality isn't available in JavaScript. In oder to comply, a class or object must have all of the properties specified in the interface, the class object cannot be missing properties (not found in the interface) or have additional ones (not found in the interface)

this interface is set to an object

doesn't comply with interface because it is empty

interface

```
interface IsPerson {
  name: string;
  age: number;
  speak(x: string): void;
  spend(y: number): number;
}
```

object

```
~~const myself: IsPerson = {}~~

const myself: IsPerson = {
  name: 'Bob',
  age: 30,

  speak(someText: string): void {
    console.log(someText);
  }

  spend(amount: number): number {
    console.log('I spent', amount);
    return amount;
  }
}
```

An object using this interface must have the following characteristic:

a spend() function that takes in a number and returns a number value

a speak() function that takes in a string and returns a void value

an age property set to a number

a name property set to a string

This is unnecessary because the return values of the functions are implied but this is considered good practice

```
const greetPerson = (person: IsPerson) => {
  console.log('Hello', person.name);
}

greetPerson(myself)
//=> Hello Bob
```

This function will only accept an IsPerson object as a parameter

sources: https://www.youtube.com/playlist?list=PL4cUxeGkcC9gUgr39Q_yD6v-bSyMwKPUI
TypeScript Tutorial #16 - Interfaces with Classes
https://github.com/Richard-Burd/typescript-sandbox/blob/master/src/app.ts

Interfaces with Classes

When a class object uses an interface, we say it implements it, whereas when a non-class object uses an interface, we use :

```
export interface HasFormatter {
  format(): string;
}
```

because the class constructor itself is a sort of interface for the class instances

This interface must return a string

./project-path/src/classes/Invoice.ts

```
import { HasFormatter } from '../interfaces/HasFormatter.js'

export class Invoice implements HasFormatter {
  constructor(
    readonly client: string,
    private details: string,
    public amount: number,
  ){}

  format(){
    return `${this.client} owes $${this.amount} for ${this.details}`
  }
}
```

The inclusion of this format() class method implements the HasFormatter interface

```
let myNewArray: HasFormatter[] = []
```

This will create a new array that will only accept objects as elements that implement the HasFormatter interface

```
const ul = document.querySelector("ul")!;
const list = new ListTemplate(ul);
```

When a variable is declared initially, it can be given an interface only and no actual value

./project-path/src/app.ts

```
form.addEventListener('submit', (e: Event) => {
  e.preventDefault();

  let doc: HasFormatter;
  if (type.value === 'invoice') {
    doc = new Invoice(tofrom.value, details.value, amount.valueAsNumber);
  } else {
    doc = new Payment(tofrom.value, details.value, amount.valueAsNumber);
  }

  console.log(doc);

  list.render(doc, type.value, 'end')

});
```

This is a TypeScript keyword

this changes s.thing like "2" to a 2

this keyword grabs the user input from the HTML form

TypeScript Installation (NPM)

Setting up and running a TypeScript environment

```
user@ubuntu ~/any/path/here $ sudo npm install -g typescript
// => This will install TypeScript onto your Ubuntu environment

user@ubuntu ~/any/path/here $ tsc index.ts someFile.js
// => Will convert the index.ts file in this directory into a someFile.js file in the same directory

user@ubuntu ~/any/path/here $ tsc index.ts someFile.js -w
=> This command is the same as the one above, except it updates the .js file live as you change the .ts file

user@ubuntu ~/any/path/here $ tsc index.ts -w
=> This will do the same as the command above assuming both the .ts and .js file have the same exact name (index)
```

The first step is to configure the tsconfig.json file in the root directory of any application using TypeScript; here is an example:

./project-path/tsconfig.json

```
{
  "compilerOptions": {
    ...
    "target": "es6",
    "module": "es2015",
    ...
    "outDir": "./public",
    "rootDir": "./src",
    ...
    "strict": true,
    ...
    "esModuleInterop": true,
    ...
    "forceConsistentCasingInFileNames": true
  },
  "include": ["src"]
}
```

Most of the tsconfig.json file can be left commented out; specify the JavaScript version to be transpiled, the directory where the JavaScript will be transpiled to, and the directory that will store the Typescript files. When fully setup, run this command and edit the TypeScript files, the commensurate JavaScript files will automatically update live!

only TypeScript files in here will be transpiled into JavaScript

```
user@ubuntu ~/typescript-sandbox (master) $ tsc -w     // => Run this from the main directory
```

This is the only command you should need after initial setup

Some of the Basics

TypeScript is a statically typed language and uses uses strict types. JavaScript does not use strict types and is dynamically typed.

```
let character = 'mario';

~~character = 43;~~

character = 'luigi';
```

once a variable is declared as a string, it cannot be set to a number & vice-versa

This is because strings & numbers are different data types, here are some of the different data types in TypeScript:

any • object • array • boolean • tuple • string • number

source: Documentation

```
const circle = (diameter: number) => {
  return diameter * Math.PI;
}
```

we can specify the variable type needed by a function

You should use any as little as possible since it negates the whole purpose of TypeScript

Arrays

```
let names = ['Tom', 'Bob', 'Dan']

names.push('Sam')
//=> ['Tom', 'Bob', 'Dan', 'Sam']

~~names.push(345)~~
~~names[2] = 423~~

~~names = "another data type";~~
```

When you declare an array with only one data type, you cannot add another data type to that array. When you declare a variable as an array, you cannot change that variable to another data type.

```
let names = ['Tom', true, 'Dan', 23]
```

Because this array contains a string, a boolean, and a number, you can add any of those 3 data types to this array.

Objects

When you declare an object like so, you must keep the value types

```
let ninja1 = {
  name: 'Michelangelo',
  weapon: 'nunchucks',
  age: 15,
  qualified: true
}
```

These can only change to a string
This can only change to a number
This can only change to a boolean

```
ninja1 = {
  name: 'Raphael',
  weapon: 'sais',
  age: 14,
  qualified: false
}
```

If I alter this object in any way, I must keep the exact same structure as when the object was originally declared. I can not add or remove a key-value pair

sources: https://www.youtube.com/playlist?list=PL4cUxeGkcC9gUgr39Q_yD6v-bSyMwKPUI
TypeScript Tutorial #17 - Rendering an HTML Template
https://github.com/Richard-Burd/typescript-sandbox/blob/master/src/classes/ListTemplate.ts

Rendering an HTML Template

TypeScript HTML templates use the render() method like React to insert elements on the DOM; this render() method takes in a transaction type (payment or invoice) a heading, and a position in the DOM it is going to be placed in:

./project-path/src/classes/ListTemplate.ts

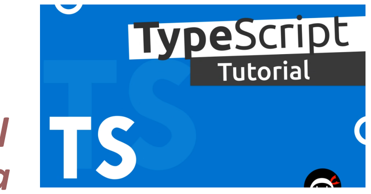
```
export class ListTemplate {
  constructor(private container: HTMLUListElement) {
      this.container = container;
  }

  render(item: HasFormatter, heading: string, pos: 'start' | 'end') {
    const li = document.createElement('li');
    const h4 = document.createElement('h4');
    h4.innerText = heading;
    li.append(h4);
    const p = document.createElement('p');
    p.innerText = item.format();
    li.append(p);

    if (pos === 'start') {
      this.container.prepend(li);
    }
    else {
      this.container.append(li);
    }
  }
}
```

Same as JavaScript

Based on the TypeScript Tutorial by NetNinja



TypeScript Illustrated

last updated @ 11:25am on 9/AUG/2021 by Richard Burd

rick.a.burd@gmail.com

github.com/Richard-Burd/typescript-sandbox
https://www.linkedin.com/in/richardburd/