Typescript quick start

A handy reference to the fundamental building blocks of Typescript.

Basic Types

Boolean

```
let completed: boolean = false;
```

String

```
let name: string = 'Sally';
```

Number

```
let aNumber: number = 20;
```

Array

```
let ages: number[] = [21, 30, 70]

// equivalent but less common in real-world apps
let ages: Array<number> = [21, 30, 70]
```

Enum

```
enum Gender { Male, Female, Unspecified };
let gender: Gender = Gender.Female;

// override the enum values (underlying numbers)
enum Gender { Male = 1, Female = 2, Unspecified = 3 };
```

Any

For interacting with javascript code or situations where the type is simply unknown and you want to avoid compile-time errors...

```
let something: any = 1;
something = "a string";
something = false;

// all work because type is "any"
```

Any for arguments

```
function handleData(age: any){
    console.log(age);
    // age could be anything!
}
```

Any for return values

```
function handleData(age: any) : any {
    if(age !== undefined){
        let message = `you are ${age} years old`;
        return message;
    }
    return false;
    // we know it's a string but we could return anything
}
```

Pro Tip: Try to avoid over-use of any as it somewhat defeats the point (and benefits) of using Typescript in the first place!

Void

Mostly used in function declarations, to indicate that the function does not return a value.

```
function log(message: string): void {
    console.log(message);
}
```

Object

Anything which isn't a primitive (isn't a number, boolean, string etc.);

```
function callApi(){
    let thing:object = backend.get('/api/users');
    thing = 42; // error: can't assign a number

    thing = {name: 'bob'}; // works, we've assigned an object
}
```

Template literals (aka Template strings)

A handy way to concatenate strings without the usual ugly syntax.

```
// without Template strings
let greeting = 'Hello ' + name + ', how are you today?';

// with Template strings
let greeting = `Hello ${name}, how are you today?`;
```

The difference is that subtle backtick used instead of an apostrophe.

Interfaces

```
interface IUser {
    firstName: string;
    lastName: string;
}
```

Interfaces make it easy to declare the "shape" of an object.

Objects defined using the interface will trigger compile errors if you a) don't specify values for required fields or b) try to assign a value to a field which isn't defined.

```
// doesn't compile because firstName, LastName missing
let me: IUser = {};

// works
let me: IUser = { firstName: 'Jon', lastName: 'Hilton' };

// doesn't compile because IUser has no an 'age' property
me.age = 21;
```

Required vs optional properties

```
interface IUser {
    firstName: string;
    lastName: string;
    age?: number;
}
```

Here `age` is optional. Everything else is required.

```
// doesn't compile because firstName, LastName missing
let me: IUser = {};

// works, age is optional
let me: IUser = { firstName: 'Jon', lastName: 'Hilton' };

// works because IUser does specify an age property
me.age = 21;

// fails compilation (mistyped name of age property)
me.aeg = 21;
```

Union Types

Sometimes, an object can be one of several types.

You can let the compiler know that multiple types are valid by declaring a union type.

```
let ambiguous: string | number;
ambiguous = 'Not any more';
ambiguous = 21;
```

Functions

For the most part functions in TS work just as functions in Javascript do with the obvious difference being that you can specify argument and return types.

Here's a named function with it's argument and return types specified.

```
function subtract(x: number, y: number): number {
    return x - y;
}
```

And here's an anonymous equivalent.

```
let subtract = function (x: number, y: number): number {
    return x - y;
}
```

Note that we haven't explicitly defined the type of `subtract` here. The Typescript compiler has inferred it from the type of our function.

This is the same as... arguments and returns a

```
let subtract: (x:number, y:number) => number;
```

This declares a variable (`subtract`) and indicates that it's type is a function which takes two `number`

`number` ...

From here we can then assign our anonymous function to `subtract`.

```
subtract = function (x: number, y: number): number {
    return x - y;
}
```

Putting it all together gives us...

```
let subtract: (x:number, y:number) => number
    = function (x: number, y: number): number {
    return x - y;
}

let result = subtract(10, 1);
alert(result);
```

Optional arguments

With default values

```
let add = function(x: number, y:number, z:number = 0): number {
    return x + y + z;
}

add(1,2); // 3
add(10,10,10); //30
```

If you specify a default value for an argument then don't pass in a value for it when you call the function, the default will be used.

Without default values

```
let greet = function(firstName: string, lastName?: string): string {
    if(lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}
```

Here no default will be used, if you don't pass the optional parameter in, the function needs to check whether it has a value and act accordingly.

Rest Parameters

If you've used C# you may have come across the `param` `d` which lets you declare that any number of arguments can be passed to a function and added to an array which holds each of the passed values.

Typescript's equivalent is Rest parameters.

```
function greetTheRoom(...names: string[]){
    console.log('hello ' + names.join(", "));
}

// 'hello Jon, Nick, Jane, Sam'
greetTheRoom('Jon', 'Nick', 'Jane', 'Sam');
```

When you call a function with a Rest parameter, you can specify as many values as you like (or none).

Arrow functions

If you've used C# you've probably used Lambdas. Javascript/Typescript has them too...

```
let sayHello = () => console.log('hello');

// is equivalent to...

var sayHello = function () {
    return console.log('hello');
};
```

And with arguments...

```
let sayHello = (name:string) => console.log(`hello ${name}`);

// is the same as...

var sayHello = function (name) {
    return console.log("hello " + name);
};
```

Classes

Javascript is very much a functional language but in recent times has introduced the concept of "classes" as a way to structure your code and objects.

Typescript supports this too...

```
class Calculator {
    private result: number;

    constructor(){
        this.result = 0;
    }

    add(x:number) {
        this.result += x;
    }

    subtract(x:number) {
        this.result -= x;
    }

    getResult():number {
        return this.result;
    }
}

// use it
let calculator = new Calculator();
calculator.add(10);
calculator.subtract(9);
console.log(calculator.getResult()); // 1
```

Modifiers

Everything in a Typescript class defaults to public.

Private modifier

You can mark class members as private and this will prevent it being accessed outside of the class.

```
class Person {
    private passportNumber: string;

    constructor(passportNumber: string){
        this.passportNumber = passportNumber;
    }
}

let person = new Person("13252323223");

// error: passportNumber is private
console.log(person.passportNumber);
```

Readonly modifier

Readonly members must be initialized when they're declared, or in the constructor

```
class Person {
    private readonly passportNumber: string;

    constructor(passportNumber: string){
        this.passportNumber = passportNumber;
    }

    changePassport(newNumber: number){
        // error: cannot assign to passportNumber because readonly
        this.passportNumber = newNumber;
    }
}
```

Parameter properties

If you find yourself declaring fields, passing values in via constructors and assigning those values to the fields, you can use a handy shorthand to save writing so much boilerplate code.

```
class Person {
    constructor(private readonly passportNumber: string){
    }
}
```

This is exactly the same as...

```
class Person {
    private readonly passportNumber: string;

    constructor(passportNumber: string){
        this.passportNumber = passportNumber;
    }
}
```

When we add accessibility modifiers (`pri` `b` `a` `o` our constructor parameters, Typescript automatically declares a class member and assigns the parameters' value to it.

Static properties

You can also declare static members on a class.

```
class Salutations {
    static Mr(name: string){
        return "Mr " + name;
    }
    static Mrs(name: string){
        return "Mrs " + name;
    }
}

// hello Mr Brian Jones
console.log ('hello ' + Salutations.Mr('Brian Jones'));

// greetings Mrs Joy Smith
console.log ('greetings ' + Salutations.Mrs('Joy Smith'));
```

Here we access the static members via the class name (`Salutations`) without having to create an instance of the class.

- Comprehensive Guide to TypeScript

-- Introduction to TypeScript

--- Brief Overview
TypeScript is a statically typed superset of JavaScript that compiles to plain JavaScript. It was developed by Microsoft and offers additional features not found in JavaScript, such as static typing, interfaces, and more robust tooling options. The main goal of TypeScript is to provide a more structured and reliable development experience while still allowing developers to write flexible and maintainable code.

--- Why Use TypeScript?
- **Enhanced Code Quality**: Static typing helps catch errors at compile time, reducing runtime errors and improving code quality.
- **Improved Tooling**: Better IDE support with features like autocompletion, type checking, and refactoring.
- **Scalability**: More suitable for large codebases with complex type relationships.
- **Modern Features**: Supports the latest JavaScript features and provides additional tools like interfaces and generics.
- **Interoperability**: Seamlessly integrates with existing JavaScript code and libraries.

-- Getting Started

--- Installation Instructions
To install the TypeScript compiler (`tsc`), you need Node.js installed on your machine. Then, you can install TypeScript globally using npm:
```bash
npm install -g typescript
```

--- Setting Up a New TypeScript Project

1. **Initialize a new project**:
   ```bash
   mkdir my-typescript-project
   cd my-typescript-project
   npm init -y
   ```
2. **Install TypeScript locally**:
   ```bash
   npm install --save-dev typescript
   ```
3. **Create a `tsconfig.json` file**:
   ```bash
   npx tsc --init
   ```
   This file contains TypeScript compiler options and project configurations.

--- Integrating TypeScript with Existing JavaScript Projects
1. **Install TypeScript**:
   ```bash
   npm install --save-dev typescript
   ```
2. **Add TypeScript files (`.ts`) alongside your JavaScript files**.
3. **Configure `tsconfig.json`** to include both TypeScript and JavaScript files:
   ```json
   {
     "include": ["src/**/*.ts", "src/**/*.js"],
     "allowJs": true
   }
   ```
4. **Compile the project**:
   ```bash
   npx tsc
   ```

-- Basic Syntax and Types

--- TypeScript Syntax vs JavaScript
TypeScript syntax is very similar to JavaScript, with the addition of type annotations and other TypeScript-specific constructs.

--- Basic Data Types
- **number**: Represents both integers and floating-point numbers.
  ```typescript
  let age: number = 25;
  ```
- **string**: Represents text data.
  ```typescript
  let name: string = "Alice";
  ```
- **boolean**: Represents true/false values.
  ```typescript
  let isActive: boolean = true;
  ```
- **null** and **undefined**: Represent absence of value.

```typescript
let empty: null = null;
let notAssigned: undefined = undefined;
```

--- Type Annotations and Type Inference
Type annotations explicitly specify variable types:
```typescript
let score: number = 100;
```
Type inference automatically deduces types based on initial values:
```typescript
let greeting = "Hello, world!";  // inferred as string
```

-- Static Typing

--- Benefits of Static Typing
- **Early Error Detection**: Catch type-related errors during development.
- **Improved Code Readability**: Makes the code more self-documenting.
- **Enhanced Refactoring**: Safer and easier code refactoring with type information.

--- Declaring Variable Types
```typescript
let isCompleted: boolean = false;
let total: number = 42;
let title: string = "TypeScript Guide";
```

--- Type Inference
TypeScript infers the type if not explicitly annotated:
```typescript
let count = 10;  // inferred as number
```

-- Interfaces

--- Definition and Usage
Interfaces define the shape of objects and can be used to ensure objects conform to a specific structure.
```typescript
interface User {
  name: string;
  age: number;
}

let user: User = {
  name: "John",
  age: 30
};
```

--- Optional and Read-Only Properties
- **Optional properties**: Mark properties that might not be present using `?`.
  ```typescript
  ```

```
interface User {
  name: string;
  age?: number;
}
```
- **Read-only properties**: Prevent modification after initialization using `readonly`.
```typescript
interface User {
  readonly id: number;
  name: string;
}
```

-- Classes

--- Object-Oriented Programming in TypeScript
TypeScript enhances JavaScript's class syntax with types and visibility modifiers.

--- Defining Classes
```typescript
class Person {
  name: string;
  age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}
```

--- Constructors and Access Modifiers
- **public** (default): Accessible from anywhere.
- **private**: Accessible only within the class.
- **protected**: Accessible within the class and subclasses.

```typescript
class Employee {
  private id: number;
  public name: string;

  constructor(id: number, name: string) {
    this.id = id;
    this.name = name;
  }
}
```

--- Inheritance and Method Overriding
```typescript
```

```
class Manager extends Employee {
  constructor(id: number, name: string, public department: string) {
    super(id, name);
  }

  greet() {
    console.log(`Hello, my name is ${this.name} and I manage the ${this.department} department.`);
  }
}
```

-- Generics

--- Introduction to Generics
Generics provide a way to create reusable components that work with any data type.

--- Creating Reusable Components
```typescript
function identity<T>(arg: T): T {
  return arg;
}

let output1 = identity<string>("Hello");
let output2 = identity<number>(42);
```

--- Generic Constraints
Constrain the types that can be used with generics:
```typescript
interface Lengthwise {
  length: number;
}

function logLength<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}
```

-- Advanced TypeScript Concepts

--- Union Types and Intersection Types
- **Union types**: Variables that can hold multiple types.
  ```typescript
  let value: string | number;
  value = "Hello";
  value = 42;
  ```
- **Intersection types**: Combine multiple types into one.
  ```typescript
  interface Drivable {
    drive(): void;
  }
  ```

```typescript
interface Flyable {
  fly(): void;
}

type Vehicle = Drivable & Flyable;
```

--- Type Aliases and Type Assertions
- **Type aliases**: Create new names for types.
  ```typescript
  type StringOrNumber = string | number;
  ```
- **Type assertions**: Override inferred types.
  ```typescript
  let someValue: any = "this is a string";
  let strLength: number = (someValue as string).length;
  ```

--- Type Guards
Type guards are functions that allow you to narrow down the type within a conditional block.
```typescript
function isString(value: any): value is string {
  return typeof value === "string";
}

function print(value: string | number) {
  if (isString(value)) {
    console.log(`String: ${value}`);
  } else {
    console.log(`Number: ${value}`);
  }
}
```

--- Conditional Types and Mapped Types
- **Conditional types**: Types that depend on a condition.
  ```typescript
  type Message<T> = T extends string ? string : number;
  ```
- **Mapped types**: Create new types by transforming properties of existing types.
  ```typescript
  type Readonly<T> = {
    readonly [P in keyof T]: T[P];
  };
  ```

This comprehensive guide provides an overview of TypeScript's key features, syntax, and usage. Whether you are starting a new project or integrating TypeScript into an existing one, these concepts will help you leverage TypeScript's powerful features to write robust and maintainable code.