TypeScript is a language that aims at easing development of large scale applications written in JavaScript. TypeScript adds common concepts such as classes, modules, interfaces, generics and (optional) static typing to JavaScript. It is a superset of JavaScript: all JavaScript code is valid TypeScript code so it can be added seamlessly to any project. The TypeScript compiler emits JavaScript.

This article will focus only on TypeScript extra syntax, as opposed to [JavaScript] (../javascript/).

To test TypeScript's compiler, head to the [Playground] (http://www.typescriptlang.org/Playground) where you will be able to type code, have auto completion and directly see the emitted JavaScript.

```
// There are 3 basic types in TypeScript
var isDone: boolean = false;
var lines: number = 42;
var name: string = "Anders";

// When it's impossible to know, there is the "Any" type
var notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean

// For collections, there are typed arrays and generic arrays
var list: number[] = [1, 2, 3];
// Alternatively, using the generic array type
var list: Array<number> = [1, 2, 3];

// For enumerations:
enum Color {Red, Green, Blue};
var c: Color = Color.Green;

// Lastly, "void" is used in the special case of a function returning nothing
function bigHorribleAlert(): void {
  alert("I'm a little annoying box!");
}

// Functions are first class citizens, support the lambda "fat arrow" syntax and
// use type inference

// The following are equivalent, the same signature will be infered by the
// compiler, and same JavaScript will be emitted
var f1 = function(i: number): number { return i * i; }
// Return type inferred
var f2 = function(i: number) { return i * i; }
var f3 = (i: number): number => { return i * i; }
// Return type inferred
var f4 = (i: number) => { return i * i; }
// Return type inferred, one-liner means no return keyword needed
var f5 = (i: number) =>  i * i;

// Interfaces are structural, anything that has the properties is compliant with
// the interface
interface Person {
  name: string;
  // Optional properties, marked with a "?"
  age?: number;
  // And of course functions
  move(): void;
}

// Object that implements the "Person" interface
// Can be treated as a Person since it has the name and move properties
var p: Person = { name: "Bobby", move: () => {} };
// Objects that have the optional property:
var validPerson: Person = { name: "Bobby", age: 42, move: () => {} };
// Is not a person because age is not a number
var invalidPerson: Person = { name: "Bobby", age: true };

// Interfaces can also describe a function type
interface SearchFunc {
  (source: string, subString: string): boolean;
}
// Only the parameters' types are important, names are not important.
var mySearch: SearchFunc;
mySearch = function(src: string, sub: string) {
  return src.search(sub) != -1;
}

// Classes - members are public by default
class Point {
  // Properties
  x: number;

  // Constructor - the public/private keywords in this context will generate
  // the boiler plate code for the property and the initialization in the
  // constructor.
  // In this example, "y" will be defined just like "x" is, but with less code
  // Default values are also supported

  constructor(x: number, public y: number = 0) {
    this.x = x;
  }

  // Functions
  dist() { return Math.sqrt(this.x * this.x + this.y * this.y); }

  // Static members
  static origin = new Point(0, 0);
}

var p1 = new Point(10 ,20);
var p2 = new Point(25); //y will be 0

// Inheritance
class Point3D extends Point {
  constructor(x: number, y: number, public z: number = 0) {
    super(x, y); // Explicit call to the super class constructor is mandatory
  }

  // Overwrite
  dist() {
    var d = super.dist();
    return Math.sqrt(d * d + this.z * this.z);
  }
}

// Modules, "." can be used as separator for sub modules
module Geometry {
  export class Square {
    constructor(public sideLength: number = 0) {
    }
    area() {
      return Math.pow(this.sideLength, 2);
    }
  }
}

var s1 = new Geometry.Square(5);

// Local alias for referencing a module
import G = Geometry;

var s2 = new G.Square(10);

// Generics
// Classes
class Tuple<T1, T2> {
  constructor(public item1: T1, public item2: T2) {
  }
}

// Interfaces
interface Pair<T> {
  item1: T;
  item2: T;
}

// And functions
var pairToTuple = function<T>(p: Pair<T>) {
  return new Tuple(p.item1, p.item2);
};

var tuple = pairToTuple({ item1:"hello", item2:"world"});

// Including references to a definition file:
/// <reference path="jquery.d.ts" />

// Template Strings (strings that use backticks)
// String Interpolation with Template Strings
var name = 'Tyrone';
var greeting = `Hi ${name}, how are you?`
// Multiline Strings with Template Strings
var multiline = `This is an example
of a multiline string`;
```

Further Reading

- [TypeScript Official website] (http://www.typescriptlang.org/)
- [TypeScript language specifications (pdf)] (http://go.microsoft.com/fwlink/?LinkId=267238)
- [Anders Hejlsberg - Introducing TypeScript on Channel 9] (http://channel9.msdn.com/posts/Anders-Hejlsberg-Introducing-TypeScript)
- [Source Code on GitHub] (https://github.com/Microsoft/TypeScript)
- [Definitely Typed - repository for type definitions] (http://definitelytyped.org/)