TypeScript

Deep Dive

Basarat Ali Syed

Table of Contents

Introduction	0
Gettting Started	1
Why TypeScript	1.1
Future JavaScript Now	2
Classes	2.1
Classes Emit	2.1.1
Classes Super	2.1.2
Classes Extensibility	2.1.3
Arrow Functions	2.2
Rest Parameters	2.3
let	2.4
const	2.5
Destructuring	2.6
for...of	2.7
Template Strings	2.8
Spread Operator	2.9
Enums	2.10
Generators	2.11
Async Await	2.12
Project	3
Compilation Context	3.1
tsconfig.json	3.1.1
Declaration Spaces	3.2
Modules	3.3
File Module Details	3.3.1
globals.d.ts	3.3.2
Namespaces	3.4
TypeScript's Type System	4
JS Migration Guide	4.1
Ambient Declarations	4.2

- Declaration Files 4.2.1
- Variables 4.2.2
- Interfaces 4.2.3
- `lib.d.ts` 4.3
- Functions 4.4
- Type Assertion 4.5
- Freshness 4.6
- NodeJS 5
- JSX 6
- TIPs 7
- Quick Object Return 7.1
- String Based Enums 7.2
- Nominal Typing 7.3
- Stateful Functions 7.4
- TypeScript Compiler Internals 8
- Program 8.1
- AST 8.2
- TIP: Visit Children 8.2.1
- TIP: SyntaxKind enum 8.2.2
- Trivia 8.2.3
- Scanner 8.3
- Parser 8.4
- Parser Functions 8.4.1
- Binder 8.5
- Binder Functions 8.5.1
- Binder Declarations 8.5.2
- Binder Container 8.5.3
- Binder SymbolTable 8.5.4
- Binder Error Reporting 8.5.5
- Checker 8.6
- Checker Diagnostics 8.6.1
- Checker Error Reporting 8.6.2
- Emitter 8.7
- Emitter Functions 8.7.1

Emitter SourceMaps 8.7.2
Contributing 8.8
Glossary

TypeScript Deep Dive

I've been looking at the issues that turn up commonly when people start using TypeScript. This is based on the lessons from StackOverflow / DefinitelyTyped and general engagement with the TypeScript community. You can follow for updates.

If you are here to read the book get started.

You can also do one of the following:

- EPUB for iPad,iPhone,Mac
- PDF for Windows and others
- MOBI for Kindle

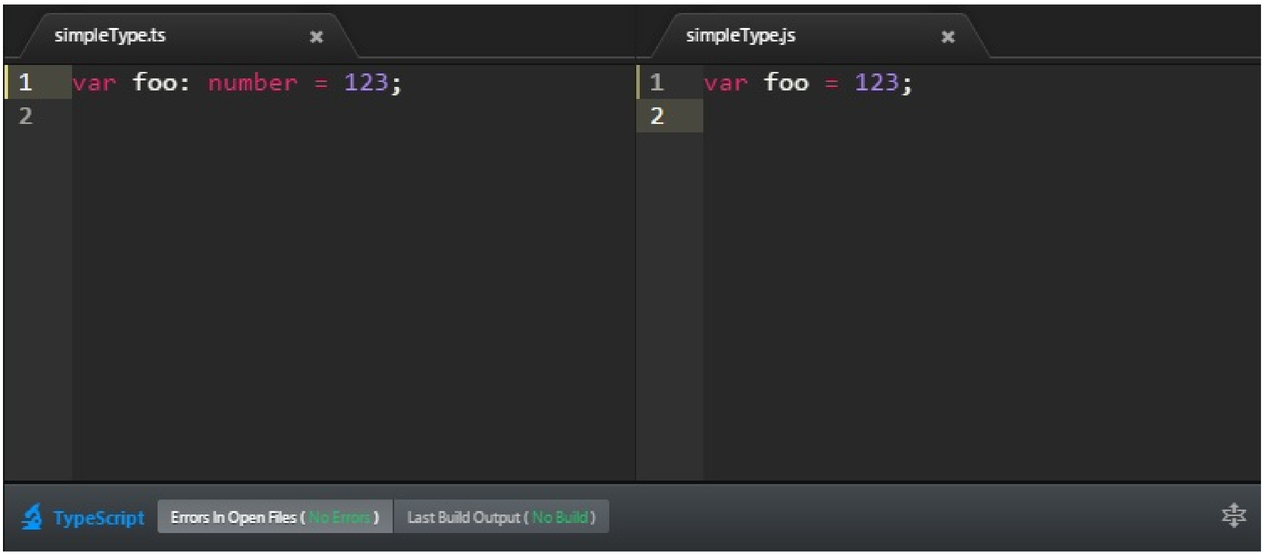
Share URL: http://basarat.gitbooks.io/typescript/

Getting Started With TypeScript

TypeScript compiles into JavaScript. JavaScript is what you are actually going to execute (either in the browser or on the server). So you are going to need the following:

- TypeScript compiler (OSS available in source and on NPM)
- A TypeScript editor (traditionally visual studio)
- Some build pipeline for a build server

Traditionally you would need to set all these up (and more) but we're trying to consolidate all this into a single Atom Package : Atom-TypeScript. The only thing you need to share and collaborate on TypeScript projects across platforms (Windows / Mac / Linux).



So:

1. Install atom.
2. `apm install atom-typescript`
3. Fire up atom. Wait (around 5 mins) for the message: `AtomTS: Dependencies installed correctly. Enjoy TypeScript ♥`

Now create a new `.ts` TypeScript file and start hacking away. AtomTS will take care of compiling it to `.js` and create a default `tsconfig.json` TypeScript project file for you.

Getting the Source Code

The source for this book is available in the books github repository https://github.com/basarat/typescript-book/tree/master/code most of the code samples can be copied in to atom-typescript and run as is. For code samples that need additional setup (e.g. npm modules), we will link you to the code sample before presenting the code. e.g.

`this/will/be/the/link/to/the/code.ts`

```
// This will be the code under discussion
```

Nightly TypeScript

Instead of using the official *stable* TypeScript compiler we will be presenting a lot of new stuff in this book that may not be released. For this purpose we recommend using nightly typescript versions.

```
npm install -g typescript@next
```

TypeScript definitions

TypeScript has a concept of a *declaration file* for external JavaScript code bases. *High quality* files exist for nearly 90% of the top JavaScript libraries out there in a project called DefinitelyTyped. You will need `tsd` to get these defintions. Don't worry, we will explain what this means later ... just install for now:

```
npm install -g tsd
```

With a dev setup out of the way lets jump into TypeScript syntax.

Why TypeScript

There are two main goals of TypeScript:

- Provide an *optional type system* for JavaScript.
- Provide planned features from future JavaScript editions to current JavaScript engines

The desire for these goals is motivated below.

The TypeScript type system

You might be wondering "**Why add types to JavaScript?**"

Types have proven ability to enhance code quality and understandability. Large teams (google,microsoft,facebook) have continually arrived at this conclusion. Specifically:

- Types increase your agility when doing refactoring. *Its better for the compiler to catch errors than to have things fail at runtime*.
- Types are one of the best forms of documentation you can have. *The function signature is a theorem and the function body is the proof*.

However types have a way of being unnecessarily ceremonious. TypeScript is very particular about keeping the barrier to entry as low as possible. Here's how:

Your JavaScript is TypeScript

TypeScript provides compile time type safety for your JavaScript code. This is no surprise given its name. The great thing is that the types are completely optional. Your JavaScript code `.js` file can be renamed to a `.ts` file and TypeScript will still give you back valid `.js` equivalent to the original JavaScript file. TypeScript is *intentionally* and strictly a superset of JavaScript with optional Type checking.

Types can be Implicit

TypeScript will try to infer as much of the type information as it can in order to give you type safety with minimal cost of productivity during code development. For example, in the following example TypeScript will know that foo is of type `number` below and will give an error on the second line as shown:

```
var foo = 123;
foo = '456'; // Error: cannot assign `string` to `number`

// Is foo a number or a string?
```

This type inference is well motivated. If you do stuff like shown in this example, then, in the rest of your code, you cannot be certain that `foo` is a `number` or a `string`. Such issues turn up often in large multi-file code bases. We will deep dive into the type inference rules later.

Types can be Explicit

As we've mentioned before, TypeScript will infer as much as it can safely, however you can use annotations to:

1. Help along the compiler, and more importantly document stuff for the next developer who has to read your code (that might be future you!).
2. Enforce that what the compiler sees, is what you thought it should see. That is your understanding of the code matches an algorithmic analysis of the code (done by the compiler).

TypeScript uses postfix type annotations popular in other *optionally* annotated languages (e.g. ActionScript and F#).

```
var foo: number = 123;
```

So if you do something wrong the compiler will error e.g.:

```
var foo: number = '123'; // Error: cannot assign a `string` to a `number`
```

We will discuss all the details of all the annotation syntax supported by TypeScript in a later chapter.

Types are structural

In some languages (specifically nominally typed ones) static typing results in unnecessary ceremony because even though *you know* that the code will work fine the language semantics force you to copy stuff around. This is why stuff like automapper for C# is *vital* for C#. In TypeScript because we really want it to be easy for JavaScript developers with a minimum cognitive overload, types are *structural*. This means that *duck typing* is a first class language construct. Consider the following example. The function `iTakePoint2D` will accept anything that contains all the things (`x` and `y`) it expects:

```
interface Point2D {
    x: number;
    y: number;
}
interface Point3D {
    x: number;
    y: number;
    z: number;
}
var point2D: Point2D = { x: 0, y: 10, }
var point3D: Point3D = { x: 0, y: 10, z: 20 }
function iTakePoint2D(point: Point2D) { /* do something */ }

iTakePoint2D(point2D); // exact match okay
iTakePoint2D(point3D); // extra information okay
iTakePoint2D({ x: 0 }); // Error: missing information `y`
```

Type errors do not prevent JavaScript emit

To make it easy for you to migrate your JavaScript code to TypeScript, even if there are compilation errors, by default TypeScript *will emit valid JavaScript* the best that it can. e.g.

```
var foo = 123;
foo = '456'; // Error: cannot assign a `string` to a `number`
```

will emit the following js:

```
var foo = 123;
foo = '456';
```

So you can incrementally upgrade your JavaScript code to TypeScript. This is very different from how many other language compilers work and yet another reason to move to TypeScript.

Types can be ambient

A major design goal of TypeScript was to make it possible for you to safely and easily use existing JavaScript libraries in TypeScript. TypeScript does this by means of *declaration*. TypeScript provides you with a sliding scale of how much or how little effort you want to put in your declarations, the more effort you put the more type safety + code intelligence you get. Note that definitions for most of the popular JavaScript libraries have already been written for you by the DefinitelyTyped community so for most purposes either:

1. The definition file already exists.
2. Or at the very least, you have a vast list of well reviewed TypeScript declaration templates already available

As a quick example of how you would author your own declaration file, consider a trivial example of jquery. By default (as expect in good JS code) TypeScript expects you to declare (i.e. use `var` somewhere) before you use a variable

```
$('.awesome').show(); // Error: cannot find name `$`
```

As a quick fix *you can tell TypeScript* that there is indeed something called `$` :

```
declare var $:any;
$('.awesome').show(); // Okay!
```

If you want you can build on this basic definition and provide more information to help protect you from errors:

```
declare var $:{
    (selector:string)=>any;
};
$('.awesome').show(); // Okay!
$(123).show(); // Error: selector needs to be a string
```

We will discuss the details of creating TypeScript definitions for existing JavaScript in detail later once you know more about TypeScript (e.g. stuff like `interface` and the `any`).

Future JavaScript => Now

TypeScript provides a number of features that are planned in ES6 for current JavaScript engines (that only support ES5 etc). The typescript team is actively adding these features and this list is only going to get bigger over time and we will cover this in its own section. But just as a specimen here is an example of a class:

```
class Point {
    constructor(public x: number, public y: number) {
    }
    add(point: Point) {
        return new Point(this.x + point.x, this.y + point.y);
    }
}

var p1 = new Point(0, 10);
var p2 = new Point(10, 20);
var p3 = p1.add(p2); // {x:10,y:30}
```

and the lovely fat arrow function:

```
var inc = (x)=>x+1;
```

Summary

In this section we have provided you with the motivation and design goals of TypeScript. With this out of the way we can dig into the nitty gritty details of TypeScript.

Future JavaScript: Now

One of the main selling points of TypeScript is that it allows you to use a bunch of features from ES6 and beyond in current (ES3 and ES5 level) JavaScript engines (like current browsers and NodeJS). Here we deep dive into why these features are useful followed by how these features are implemented in TypeScript.

Note: Not all of these features are slated for immediate addition to JavaScript but provide great utility to your code organization and maintenance. Also note that you are free to ignore any of the constructs that don't make sense for your project, although you will end up using most of them eventually ;)

Classes

The reason why its important to have classes in JavaScript as a first class item is that:

1. People like to use classes
2. Provides a consistent way for developers to use classes instead of every framework (emberjs,reactjs etc) coming up with their own version.

Finally JavaScript developers can *have* `class`. Here we have a basic class called Point:

```
class Point {
    x: number;
    y: number;
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
    add(point: Point) {
        return new Point(this.x + point.x, this.y + point.y);
    }
}

var p1 = new Point(0, 10);
var p2 = new Point(10, 20);
var p3 = p1.add(p2); // {x:10,y:30}
```

This class generates the following JavaScript on ES5 emit:

```
var Point = (function () {
    function Point(x, y) {
        this.x = x;
        this.y = y;
    }
    Point.prototype.add = function (point) {
        return new Point(this.x + point.x, this.y + point.y);
    };
    return Point;
})();
```

This is a fairly idiomatic traditional JavaScript class pattern now as a first class language construct. Note that `constructor` is optional.

Inheritance

Classes in TypeScript (like other langauges) support *single* inheritance using the `extends` keyword as shown below:

```
class Point3D extends Point {
    z: number;
    constructor(x: number, y: number, z: number) {
        super(x, y);
        this.z = z;
    }
    add(point: Point3D) {
        var point2D = super.add(point);
        return new Point3D(point2D.x, point2D.y, this.z + point.z);
    }
}
```

If you have a constructor in your class then you *must* call the parent constructor from your constructor (TypeScript will point this out to you). This ensures that the stuff that it needs to set on `this` gets set. Followed by the call to `super` you can add any additional stuff you want to do in your constructor (here we add another member `z`).

Note that you override parent member functions easily (here we override `add`) and still use the functionality of the super class in your members (using `super.` syntax).

Statics

TypeScript classes support `static` properties that are shared by all instances of the class. A natural place to put (and access) them is on the class itself and that is what TypeScript does:

```
class Something {
    static instances = 0;
    constructor() {
        Something.instances++;
    }
}

var s1 = new Something();
var s2 = new Something();
console.log(Something.instances); // 2
```

You can have static members as well as static functions.

Access Modifiers

TypeScript supports the common access modifiers that control if *a variable is accessible outside the class directly on instances* and *is the variable accessible in child classes* :

1. `public` : available on instances everywhere

2. `private` : not available for access outside the class.
3. `protected` : available on child classes but not on instances directly.

Note that at runtime (in the generated JS) these have no significance but will give you compile time errors if you use them incorrectly. An example of each is shown below:

```
class FooBase {
    public x: number;
    private y: number;
    protected z: number;
}

// EFFECT ON INSTANCES
var foo = new FooBase();
foo.x; // okay
foo.y; // ERROR : private
foo.z; // ERROR : protected

// EFFECT ON CHILD CLASSES
class FooChild extends FooBase {
    constructor(){
      super();
        this.x; // okay
        this.y; // ERROR: private
        this.z; // okay
    }
}
```

As always these modifiers work for both member properties and member functions.

Abstract

`abstract` can be thought of as an access modifier. We present it separately because opposed to the previously mentioned modifiers it can be on a `class` as well as any member of the class. Having an `abstract` modifier primarily means that such functionality *cannot be directly invoked*.

`abstract` members are commonly used as a means of providing a contract for some functionality that a child class must provide. `abstract` *class* es cannot be directly instantiated. Instead the user must create some `class` that inherit from the `abstract class` .

Define using constructor

Having a member in a class and initializing it like below:

```
class Foo{
    x: number;
    constructor(x:number){
        this.x = x;
    }
}
```

is such a common pattern that TypeScript provides a shorthand where you can prefix the member with an *access modifier* and it is automatically declared on the class and copied from the constructor. So the previous example can be re-written as (notice `public x:number`):

```
class Foo{
    constructor(public x:number){
    }
}
```

Property initializer

This is a nifty feature supported by TypeScript (from ES7 actually). You can initialize any member of the class outside the class constructor, useful to provide default (notice `members = []`)

```
class Foo{
    members = [];  // Initialize directly
    add(x){
        this.members.push(x);
    }
}
```

Whats up with the IIFE

The js generated for the class could have been:

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}
Point.prototype.add = function (point) {
    return new Point(this.x + point.x, this.y + point.y);
};
```

The reason its wrapped in an Immediately-Invoked Function Expression (IIFE) i.e.

```
(function () {

    // BODY

    return Point;
})();
```

has to do with inheritance. It allows TypeScript to capture the base class as a variable `_super` e.g.

```
var Point3D = (function (_super) {
    __extends(Point3D, _super);
    function Point3D(x, y, z) {
        _super.call(this, x, y);
        this.z = z;
    }
    Point3D.prototype.add = function (point) {
        var point2D = _super.prototype.add.call(this, point);
        return new Point3D(point2D.x, point2D.y, this.z + point.z);
    };
    return Point3D;
})(Point);
```

Notice that the IIFE allows TypeScript to easily capture the base class `Point` in a `_super` variable and that is used consistently in the class body.

`__extends`

You will notice that as soon as you inherit a class TypeScript also generates the following function:

```
var __extends = this.__extends || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    __.prototype = b.prototype;
    d.prototype = new __();
};
```

Here `d` refers to the derived class and `b` refers to the base class. This function does two things:

1. copies the static members of the base class onto the child class i.e. `for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];`
2. sets up the child class function's prototype to optionally lookup members on the parent's `proto` i.e. effectively `d.prototype.__proto__ = b.prototype`

People rarely have trouble understanding 1, but many people struggle with 2. so an explanation is in order

`d.prototype.__proto__ = b.prototype`

After having tutored many people about this I find the following explanation to be simplest. First we will explain how the code from `__extends` is equivalent to the simple `d.prototype.__proto__ = b.prototype`, and then why this line in itself is significant. To understand all this you need to know these things:

1. `__proto__`
2. `prototype`
3. effect of `new` on `this` inside the called function
4. effect of `new` on `prototype` and `__proto__`

All objects in JavaScript contain a `__proto__` member. This member is often not accessible in older browsers (sometimes documentation refers to this magical property as `[[prototype]]`). It has one objective: If a property is not found on an object during lookup (e.g. `obj.property`) then it is looked up at `obj.__proto__.property`. If it is still not found then `obj.__proto__.__proto__.property` till either: *it is found* or *the latest* `.__proto__` *itself is null*. This explains why JavaScript is called to support *prototypal inheritance* out of the box. This is shown in the following example, which you can run in the chrome console or nodejs:

```
var foo = {}

// setup on foo as well as foo.__proto__
foo.bar = 123;
foo.__proto__.bar = 456;

console.log(foo.bar); // 123
delete foo.bar; // remove from object
console.log(foo.bar); // 456
delete foo.__proto__.bar; // remove from foo.__proto__
console.log(foo.bar); // undefined
```

Cool so you understand `__proto__`. Another useful information is that all `function`s in JavaScript have a property called `prototype` and that it has a member `constructor` pointing back to the function. This is shown below:

```
function Foo() { }
console.log(Foo.prototype); // {} i.e. it exists and is not undefined
console.log(Foo.prototype.constructor === Foo); // Has a member called `constructor` poin
```

Now lets look at *effect of `new` on `this` inside the called function*. Basically `this` inside the called function is going to point to the newly created object that will be returned from the function. It's simple to see if you mutate a property on `this` inside the function:

```
function Foo() {
    this.bar = 123;
}

// call with the new operator
var newFoo = new Foo();
console.log(newFoo.bar); // 123
```

Now the only other thing you need to know is that calling `new` on a function copies the `prototype` of the function into the `__proto__` of the newly created object that is returned from the function call. Here is code you can run to completely understand it:

```
function Foo() { }

var foo = new Foo();

console.log(foo.__proto__ === Foo.prototype); // True!
```

That's it. Now look at the following straight out of `__extends`. I've take the liberty to number these lines:

```
1  function __() { this.constructor = d; }
2   __.prototype = b.prototype;
3   d.prototype = new __();
```

Reading this function in reverse the `d.prototype = new __()` on line 3 effectively means `d.prototype = {__proto__ : __.prototype}` (because of 4, effect of `new` on `prototype`), combine it with the previous line (i.e. line 2 `__.prototype = b.prototype;`) you get `d.prototype = {__proto__ : __.prototype}`.

But wait we wanted `d.prototype.__proto__` i.e. just the proto changed and maintain the old `d.prototype.constructor`. This is where the significance of the first line (i.e. `function __() { this.constructor = d; }`) comes in. Here we will effectively have `d.prototype = {__proto__ : __.prototype, d.constructor = d}` (because of 3, effect of `new` on `this` inside the called function). So since we restore `d.prototype.constructor`, the only thing we have truly mutated is the `__proto__` hence `d.prototype.__proto__ = b.prototype`.

`d.prototype.__proto__ = b.prototype` significance

The significance is that it allows you to add member functions to a child class and inherit others from the base class. This is demonstrated by the following simple example:

```
function Animal() { }
Animal.prototype.walk = function () { console.log('walk') };

function Bird() { }
Bird.prototype.__proto__ = Animal.prototype;
Bird.prototype.fly = function () { console.log('fly') };

var bird = new Bird();
bird.walk();
bird.fly();
```

Basically `bird.fly` will be looked up from `bird.__proto__.fly` (remember that `new` makes the `bird.__proto__` point to `Bird.prototype`) and `bird.walk` (an inherited member) will be looked up from `bird.__proto__.__proto__.walk` (as `bird.__proto__ == Bird.prototype` and `bird.__proto__.__proto__` == `Animal.prototype`).

`super`

Note that if you call `super` on a child class it is redirected to the `prototype` as shown below:

```
class Base {
    log() { console.log('hello world'); }
}

class Child extends Base {
    log() { super.log() };
}
```

generates:

```
var Base = (function () {
    function Base() {
    }
    Base.prototype.log = function () { console.log('hello world'); };
    return Base;
})();
var Child = (function (_super) {
    __extends(Child, _super);
    function Child() {
        _super.apply(this, arguments);
    }
    Child.prototype.log = function () { _super.prototype.log.call(this); };
    return Child;
})(Base);
```

Notice `_super.prototype.log.call(this)`.

This means that you cannot use `super` on member properties. Instead you should just use `this`.

```
class Base {
    log = () => { console.log('hello world'); }
}

class Child extends Base {
    logWorld() { this.log() };
}
```

Notice since there is only one `this` shared between the `Base` and the `Child` class you need to use *different* names (here `log` and `logWorld`).

Also Note that TypeScript will warn you if you try to misuse `super`:

```
module quz {
    class Base {
        log = () => { console.log('hello world'); }
    }

    class Child extends Base {
        // ERROR : only `public` and `protected` methods of base class are accessible via
        logWorld() { super.log() };
    }
}
```

Arrow Functions

Lovingly called the *fat arrow* (because `->` is a thin arrow and `=>` is a fat arrow) and also called a *lambda function* (because of other languages). Another commonly used feature is the fat arrow function `()=>something`. The motivation for a *fat arrow* is:

1. You don't need to keep typing `function`
2. It lexically captures the meaning of `this`
3. It lexically captures the meaning of `arguments`

For a language that claims to be functional, in JavaScript you tend to be typing `function` quite a lot. The fat arrow makes it simple for you to create a function

```
var inc = (x)=>x+1;
```

`this` has traditionally been a pain point in JavaScript. As a wise man once said "I hate JavaScript as it tends to lose the meaning of `this` all too easily". Fat arrows fix it by capturing the meaning of `this` from the surrounding context. Consider this pure JavaScript class:

```
function Person(age) {
    this.age = age
    this.growOld = function(){
        this.age++;
    }
}
var person = new Person(1);
setTimeout(person.growOld,1000);

setTimeout(function(){ console.log(person.age); },2000); // 1, should have been 2
```

If you run this code in the browser `this` within the function is going to point to `window` because `window` is going to be what executes the `growOld` function. Fix is to use an arrow function:

```
function Person(age) {
    this.age = age
    this.growOld = () => {
        this.age++;
    }
}
var person = new Person(1);
setTimeout(person.growOld,1000);

setTimeout(function(){ console.log(person.age); },2000); // 2
```

The reason why this works is the reference to `this` is captured by the arrow function from outside the function body. This is equivalent to the following JavaScript code (which is what you would write yourself if you didn't have TypeScript):

```
function Person(age) {
    this.age = age
    var _this = this;  // capture this
    this.growOld = function() {
        _this.age++;   // use the captured this
    }
}
var person = new Person(1);
setTimeout(person.growOld,1000);

setTimeout(function(){ console.log(person.age); },2000); // 2
```

Note that since you are using TypeScript you can be even sweeter in syntax and combine arrows with classes:

```
class Person {
    constructor(public age:number){}
    growOld = () => {
        this.age++;
    }
}
var person = new Person(1);
setTimeout(person.growOld,1000);

setTimeout(function(){ console.log(person.age); },2000); // 2
```

Tip: Arrow Function Need

Beyond the terse syntax, you only *need* to use the fat arrow if you are going to give the function to someone else to call. Effectively:

```
var growOld = person.growOld;
// Then later someone else calls it:
growOld();
```

If you are going to call it yourself, i.e.

```
person.growOld();
```

then `this` is going to be the correct calling context (in this example `person`).

Tip: Arrow Function Danger

In fact if you want `this` *to be the calling context* you should *not use the arrow function*. This is the case with callbacks used by libraries like jquery, underscore, mocha and others. If the documentation mentions functions on `this` then you should probably just use a `function` instead of a fat arrow. Similarly if you plan to use `arguments` don't use an arrow function.

Tip: Arrow functions with libraries that use `this`

Many libraries do this e.g `jQuery` iterables (one example http://api.jquery.com/jquery.each/) will use `this` to pass you the object that it is currently iterating over. In this case if you want to access the library passed `this` as well as the surrounding context just use a temp variable like `_self` like you would in the absence of arrow functions.

```
let _self = this;
something.each(function(){
    console.log(_self); // the lexically scoped value
    console.log(this); // the library passed value
});
```

Rest Parameters

Rest parameters (denoted by `...argumentName` for the last argument) allow you to quickly accept multiple arguments in your function and get them as an array. This is demonstrated in the below example.

```
function iTakeItAll(first, second, ...allOthers) {
    console.log(allOthers);
}
iTakeItAll('foo', 'bar'); // []
iTakeItAll('foo', 'bar', 'bas', 'qux'); // ['bas','qux']
```

Rest parameters can be used in any function be it `function` / `()=>` / `class member` .

let

Variables in JavaScript are *function scoped*. This is different from many other languages (C# / Java etc.) where the variables are *block scoped*. If you bring a *block scoped* mindset to JavaScript you would expect the following to print `123`, instead it will print `456`

```
var foo = 123;
if (true) {
    var foo = 456;
}
console.log(foo); // 456
```

This is because `{` does not create a new *variable scope*. The variable `foo` is the same inside the if *block* as it is outside the if block. This is a common source of errors in JavaScript program. This is why TypeScript (and ES6) introduces the `let` keyword to allow you to define variables with true *block scope*. That is if you use `let` instead of `var` you get a true unique element disconnected from what you might have defined outside the scope. The same example is demonstrated with `let`:

```
let foo = 123;
if (true) {
    let foo = 456;
}
console.log(foo); // 123
```

Another place where `let` would save you from errors is loops.

```
var index = 0;
var array = [1, 2, 3];
for (let index = 0; index < array.length; index++) {
    console.log(array[index]);
}
console.log(index); // 0
```

In all sincerity we find it better to use `let` whenever possible as it leads to lesser surprises for new and existing multi-lingual developers.

Functions create a new scope

Since we mentioned it, we'd like to demonstrate that functions create a new variable scope in JavaScript. Consider the following:

```
var foo = 123;
function test(){
    var foo = 456;
}
test();
console.log(foo); // 123
```

This behaves as you would expect. Without this it would be very difficult to write code in JavaScript.

Generated JS

The JS generated by TypeScript is simple renaming of the `let` variable if a similar name already exists in the surrounding scope. E.g. the following is generated as is with a simple replacement of `var` with `let` :

```
if (true) {
    let foo = 123;
}

// becomes //

if (true) {
    var foo = 123;
}
```

However if the variable name is already taken by the surrounding scope then a new variable name is generated as shown (notice `_foo`):

```
var foo = '123';
if (true) {
    let foo = 123;
}

// becomes //

var foo = '123';
if (true) {
    var _foo = 123; // Renamed
}
```

let in closures

A common programming interview question for a JavaScript developer is what is the log of this simple file:

```
var funcs = [];
// create a bunch of functions
for (var i = 0; i < 3; i++) {
    funcs.push(function() {
        console.log(i);
    })
}
// call them
for (var j = 0; j < 3; j++) {
    funcs[j]();
}
```

One would have expected it to be `0,1,2` . Surprisingly it is going to be `3` for all three functions. Reason is that all three functions are using the variable `i` from the outer scope and at the time we execute them (in the second loop) the value of `i` will be `3` (that's the termination condition for the first loop).

A fix would be to create a new variable in each loop specific to that loop iteration. As we've learnt before we can create a new variable scope by creating a new function and immediately executing it (i.e. the IIFE pattern from classes `(function() { /* body */ })();`) as shown below:

```
var funcs = [];
// create a bunch of functions
for (var i = 0; i < 3; i++) {
    (function() {
        var local = i;
        funcs.push(function() {
            console.log(local);
        })
    })();
}
// call them
for (var j = 0; j < 3; j++) {
    funcs[j]();
}
```

Here the functions close over (hence called a `closure`) the *local* variable (conveniently named `local`) and use that instead of the loop variable `i` . Note that closures come with a performance impact (they need to store the surrounding state) and therefore even though the ES6 `let` keyword in a loop would have the same behavior as the previous example, the following is an error in TypeScript if you target something less than ES6:

```
var funcs = [];
// create a bunch of functions
for (let i = 0; i < 3; i++) {
    // Error: Loop contains block-scoped variable 'i' referenced by a function in the loo
    // This is only supported in ECMAScript 6 or higher.
    funcs.push(function() {
        console.log(i);
    })
}
// call them
for (var j = 0; j < 3; j++) {
    funcs[j]();
}
```

Note: This limitation may be removed in a future version of TypeScript.

Summary

Despite a few limitations, we find `let` to be extremely useful to have for the vast majority of code. It can greatly enhance your code readability and decrease the chance of a programming error.

const

`const` is a very welcome addition offered by ES6 / TypeScript. It allows you to be immutable with variables. This is good from a documentation as well as a runtime perspective. To use const just replace `var` with `const` :

```
const foo = 123;
```

> The syntax is much better (IMHO) than other languages that force the user to type something like `let constant foo` i.e. a variable + behavior specifier.

`const` is a good practice for both readability and maintainability and avoids using *magic literals* e.g.

```
// Low readability
if (x > 10) {
}

// Better!
const maxRows = 10;
if (x > maxRows) {
}
```

const declarations must be initialized

The following is a compiler error:

```
const foo; // ERROR: const declarations must be initialized
```

Left hand side of assignment cannot be a constant

Constants are immutable after creation, so if you try to assign them to a new value it is a compiler error:

```
const foo = 123;
foo = 456; // ERROR: Left-hand side of an assignment expression cannot be a constant
```

Block Scoped

A `const` is block scoped like we saw with `let` :

```
const foo = 123;
if (true) {
    const foo = 456; // Allowed as its a new variable limited to this `if` block
}
```

Deep immutability

A `const` works with object literals as well, as far as protecting the variable *reference* is concerned:

```
const foo = { bar: 123 };
foo = { bar: 456 }; // ERROR : Left hand side of an assignment expression cannot be a con
```

However it still allows sub properties of objects to be mutated, as shown below:

```
const foo = { bar: 123 };
foo.bar = 456; // Allowed!
console.log(foo); // { bar: 456 }
```

For this reason I recommend using `const` with literals or immutable data structures.

Destructuring

TypeScript supports the following forms of Destructuring (literally named after de-stucturing i.e. breaking up the structure):

1. Object Destructuring
2. Array Destructuring

It is easy to think of destructuring as an inverse of *structuring*. The method of *structuring* in JavaScript is the object literal:

```
var foo = {
    bar: {
        bas: 123
    }
};
```

Without the awesome *structuring* support built into JavaScript creating new objects on the fly would indeed be very cumbersome. Destructuring brings the same level of convenience to getting data out of a structure.

Object Destructuring

Destructuring is useful because it allows you to do in a single line, what would otherwise require multiple lines. Consider the following case:

```
var rect = { x: 0, y: 10, width: 15, height: 20 };

// Destructuring assignment
var {x, y, width, height} = rect;
console.log(x, y, width, height); // 0,10,15,20
```

Here in the absence of destructuring you would have to pick off `x,y,width,height` one by one from `rect`.

Additionally you can get *deep* data out of a structure using destructuring. This is shown in the following example:

```
var foo = { bar: { bas: 123 } };
var {bar: {bas}} = foo; // Effectively `var bas = foo.bar.bas;`
```

Array Destructuring

A common programming question : Swap two variables without using a third one. The TypeScript solution:

```
var x = 1, y = 2;
[x, y] = [y, x];
console.log(x, y); // 2,1
```

Note that array destructuring is effectively the compiler doing the `[0], [1], ...` and so on for you. There is no guarantee that these values will exist.

Array Destructuring with rest

You can pick up any number of elements from the array and get *an array* of the remaining elements using array destructuring with rest.

```
var [x, y, ...remaining] = [1, 2, 3, 4];
console.log(x, y, remaining); // 1, 2, [3,4]
```

Array Destructuring with ignores

You can ignore any index by simply leaving its location empty i.e. `, ,` in the left hand side of the assignment. For example:

```
var [x, , ...remaining] = [1, 2, 3, 4];
console.log(x, remaining); // 1, [3,4]
```

JS Generation

The JavaScript generation for non ES6 targets simply involves creating temporary variables, just like you would have to do yourself without native language support for destructuring e.g.

```
var x = 1, y = 2;
[x, y] = [y, x];
console.log(x, y); // 2,1

// becomes //

var x = 1, y = 2;
_a = [y,x], x = _a[0], y = _a[1];
console.log(x, y);
var _a;
```

Summary

Destructuring can make your code more readable and maintainable by reducing the line count and making the intent clear. Array destructuring can allow you to use arrays as though they were tuples.

for...of

A common error experienced by beginning JavaScript developers is that `for...in` for an array does not iterate over the array items. Instead it iterates over the *keys* of the object passed in. This is demonstrated in the below example. Here you would expect `9,2,5` but you get the indexes `0,1,2` :

```
var someArray = [9, 2, 5];
for (var item in someArray) {
    console.log(item); // 0,1,2
}
```

This is one of the reasons why `for...of` exists in TypeScript (and ES6). The following iterates over the array correctly logging out the members as expected:

```
var someArray = [9, 2, 5];
for (var item of someArray) {
    console.log(item); // 9,2,5
}
```

Similarly TypeScript has no trouble going through a string character by character using `for...of` :

```
var hello = "is it me you're looking for?";
for (var char of hello) {
    console.log(char); // is it me you're looking for?
}
```

JS Generation

For pre ES6 targets TypeScript will generate the standard `for (var i = 0; i < list.length; i++)` kind of loop. For example here's what gets generated for our previous example:

```
var someArray = [9, 2, 5];
for (var item of someArray) {
    console.log(item);
}

// becomes //

for (var _i = 0; _i < someArray.length; _i++) {
    var item = someArray[_i];
    console.log(item);
}
```

You can see that using `for...of` makes *intent* clearer and also decreases the amount of code you have to write (and variable names you need to come up with).

Limitations

If you are not targeting ES6 or above, the generated code assumes the property `length` exists on the object and that the object can be indexed via numbers e.g `obj[2]`. So it is only supported on `string` and `array` for these legacy JS engines.

If TypeScript can see that you are not using an array or a string it will give you a clear error *"is not an array type or a string type"*;

```
let articleParagraphs = document.querySelectorAll("article > p");
// Error: Nodelist is not an array type or a string type
for (let paragraph of articleParagraphs) {
    paragraph.classList.add("read");
}
```

Use `for...of` only for stuff that *you know* to be an array or a string. Note that this limitation might be removed in a future version of TypeScript.

Summary

You would be surprised at how many times you will be iterating over the elements of an array. The next time you find yourself doing that, give `for...of` a go. You might just make the next person who reviews your code happy.

Template Strings

Syntactically these are strings that use backticks (i.e. `) instead of single (') or double (") quotes. The motivation of Template Strings is three fold:

- Multiline Strings
- String Interpolation
- Tagged Templates

Multiline Strings

Ever wanted to put a newline in a JavaScript string? Perhaps you wanted to embed some lyrics? You would have needed to *escape the literal newline* using our favorite escape character `\`, and then put a new line into the string manually `\n` at the next line. This is shown below:

```
var lyrics = "Never gonna give you up \
\nNever gonna let you down";
```

With TypeScript you can just use a template string:

```
var lyrics = `Never gonna give you up
Never gonna let you down`;
```

String Interpolation

Another common use case is when you want to generate some string out of some static strings + some variables. For this you would need some *templating logic* and this is where *template strings* get their name from. Here's how you would potentially generate an html string previously:

```
var lyrics = 'Never gonna give you up';
var html = '<div>' + lyrics + '</div>';
```

Now with template strings you can just do:

```
var lyrics = 'Never gonna give you up';
var html = `<div>${lyrics}</div>`;
```

Note that any placeholder inside the interpolation (`${` and `}`) is treated as a JavaScript expression and evaluated as such e.g. you can do fancy math.

```
console.log(`1 and 1 one make ${1 + 1}`);
```

Tagged Templates

You can place a function (called a `tag`) before the template string and it gets the opportunity to pre process the template string literals plus the values of all the placeholder expressions and return a result. A few notes:

- All the static literals are passed in as an array for the first argument.
- All the values of the placeholders expressions are passed in as the remaining arguments. Most commonly you would just use rest parameters to convert these into an array as well.

Here is an example where we have a tag function (named `htmlEscape`) that escapes the html from all the placeholders:

```
var say = "a bird in hand > two in the bush";
var html = htmlEscape `<div> I would just like to say : ${say}</div>`;

// a sample tag function
function htmlEscape(literals, ...placeholders) {
    let result = "";

    // interleave the literals with the placeholders
    for (let i = 0; i < placeholders.length; i++) {
        result += literals[i];
        result += placeholders[i]
            .replace(/&/g, '&amp;')
            .replace(/"/g, '&quot;')
            .replace(/'/g, '&#39;')
            .replace(/</g, '&lt;')
            .replace(/>/g, '&gt;');
    }

    // add the last literal
    result += literals[literals.length - 1];
    return result;
}
```

Generated JS

For pre ES6 compile targets the code is fairly simple. Multiline strings become escaped strings. String interpolation becomes *string concatenation*. Tagged Templates become function calls.

Summary

Multiline strings and string interpolation are just great things to have in any language. It's great that you can now use them in your JavaScript (thanks TypeScript!). Tagged templates allow you to create powerful string utilities.

Spread Operator

The main objective of the spread operator is to *spread* the objects of an array. This is best explained with examples.

Apply

A common use case is to spread an array into the function arguments. Previously you would need to use `Function.prototype.apply` :

```
function foo(x, y, z) { }
var args = [0, 1, 2];
foo.apply(null, args);
```

Now you can do this simply by prefixing the arguments with `...` as shown below:

```
function foo(x, y, z) { }
var args = [0, 1, 2];
foo(...args);
```

Here we are *spreading* the `args` array into positional `arguments` .

Destructuring

We've already seen one usage of this in *destructuring*

```
var [x, y, ...remaining] = [1, 2, 3, 4];
console.log(x, y, remaining); // 1, 2, [3,4]
```

The motivation here is to simply make it easy for you to capture the remaining elements of an array when destructuring.

Array Assignment

The spread operator allows you easily place an *expanded version* of an array into another array. This is demonstrated in the below example:

```
var list = [1, 2];
list = [...list, 3, 4];
console.log(list); // [1,2,3,4]
```

Summary

`apply` is something that you would inevitably do in JavaScript, so it's good to have a better syntax where you don't have that ugly `null` for the `this` argument. Also having a dedicated syntax for moving arrays out of (destructuring) or into (assignment) other arrays provides neat syntax for when you are doing array processing on partial arrays.

Enums

An enum is a way to organize to a collection of related values. Many other programming languages (C/C#/Java) have an `enum` data type but JavaScript does not. However TypeScript does. Here is an example definition of a TypeScript enum:

```
enum CardSuit {
    Clubs,
    Diamonds,
    Hearts,
    Spades
}

// Sample usage
var card = CardSuit.Clubs;

// Safety
card = "not a member of card suit"; // Error : string is not assignable to type `CardSuit
```

Enums and Numbers

TypeScript enums are number based. This means that numbers can be assigned to an instance of the enum, and so can anything else that is compatible with `number` .

```
enum Color {
    Red,
    Green,
    Blue
}
var col = Color.Red;
col = 0; // Effectively same as Color.Red
```

Enums and Strings

Before we look further into enums lets look at the JavaScript that it generates, here is a sample TypeScript:

```
enum Tristate {
    False,
    True,
    Unknown
}
```

generates the following JavaScript

```
var Tristate;
(function (Tristate) {
    Tristate[Tristate["False"] = 0] = "False";
    Tristate[Tristate["True"] = 1] = "True";
    Tristate[Tristate["Unknown"] = 2] = "Unknown";
})(Tristate || (Tristate = {}));
```

lets focus on the line `Tristate[Tristate["False"] = 0] = "False";`. Within it `Tristate["False"] = 0` should be self explanatory, i.e. sets `"False"` member of `Tristate` variable to be `"0"`. Note that in JavaScript the assignment operator returns the assigned value (in this case `0`). Therefore the next thing executed by the JavaScript runtime is `Tristate[0] = "False"`. This means that you can use the `Tristate` variable to convert a string version of the enum to a number or a number version of the enum to a string. This is demonstrated below:

```
enum Tristate {
    False,
    True,
    Unknown
}
console.log(Tristate[0]); // "False"
console.log(Tristate["False"]); // 0
console.log(Tristate[Tristate.False]); // "False" because `Tristate.False == 0`
```

Changing the number associated with an Enum

By default enums are `0` based and then each subsequent value increments by 1 automatically. As an example consider the following

```
enum Color {
    Red,     // 0
    Green,   // 1
    Blue     // 2
}
```

However you can change the number associated with any enum member by assigning to it specifically. This is demonstrated below where we start at 3 and start incrementing from there:

```
enum Color {
    DarkRed = 3,  // 3
    DarkGreen,    // 4
    DarkBlue      // 5
}
```

Enums are open ended

Here is the generated JavaScript for an enum shown again:

```
var Tristate;
(function (Tristate) {
    Tristate[Tristate["False"] = 0] = "False";
    Tristate[Tristate["True"] = 1] = "True";
    Tristate[Tristate["Unknown"] = 2] = "Unknown";
})(Tristate || (Tristate = {}));
```

We already explained the `Tristate[Tristate["False"] = 0] = "False";` portion. Now notice the surrounding code `(function (Tristate) { /*code here */ })(Tristate || (Tristate = {}));` specifically the `(Tristate || (Tristate = {}));` portion. This basically captures a local variable `TriState` that will either point to an already defined `Tristate` value or initialize it with a new empty `{}` object.

This means that you can split (and extend) an enum definition across multiple files. For example below we have split the definition for `Color` into two blocks

```
enum Color {
    Red,
    Green,
    Blue
}

enum Color {
    DarkRed = 3,
    DarkGreen,
    DarkBlue
}
```

Note that you *should* reinitialize the first member (here `DarkRed = 3`) in a continuation of an enum to get the generated code not clobber values from a previous definition (i.e. the `0`, `1`, ... so on values). TypeScript will warn you if you don't anyways (error message `In an enum with multiple declarations, only one declaration can omit an initializer for its first enum element.`)

Enums as flags

One excellent use of the ability to use enums as `Flags`. Consider the following example

```
enum AnimalFlags {
    None           = 0,
    HasClaws       = 1 << 0,
    CanFly         = 1 << 1,
    EatsFish       = 1 << 2,
    Endangered     = 1 << 3
}
```

Here we are using the left shift operator to move `1` around a certain level of bits to come up with bitwise disjoint numbers `0001`, `0010`, `0100` and `1000` (these are decimals `1`, `2`, `4`, `8` if you are curious). The bitwise operators `|` (or) / `&` (and) / `~` (not) are your best friend when working with flags and are demonstrated below:

```

enum AnimalFlags {
    None           = 0,
    HasClaws       = 1 << 0,
    CanFly         = 1 << 1,
}

function printAnimalAbilities(animal) {
    var animalFlags = animal.flags;
    if (animalFlags & AnimalFlags.HasClaws) {
        console.log('animal has claws');
    }
    if (animalFlags & AnimalFlags.CanFly) {
        console.log('animal can fly');
    }
    if (animalFlags == AnimalFlags.None){
        console.log('nothing');
    }
}

var animal = { flags: AnimalFlags.None };
printAnimalAbilities(animal); // nothing
animal.flags |= AnimalFlags.HasClaws;
printAnimalAbilities(animal); // animal has claws
animal.flags &= ~AnimalFlags.HasClaws;
printAnimalAbilities(animal); // nothing
animal.flags |= AnimalFlags.HasClaws | AnimalFlags.CanFly;
printAnimalAbilities(animal); // animal has claws, animal can fly
```

Here:

- we used `|=` to add flags
- a combination of `&=` and `~` to clear a flag
- `|` to combine flags

Note : you can combine flags to create convenient shortcuts within the enum definition e.g. `EndangeredFlyingClawedFishEating` below.

```
enum AnimalFlags {
    None           = 0,
    HasClaws       = 1 << 0,
    CanFly         = 1 << 1,
    EatsFish       = 1 << 2,
    Endangered     = 1 << 3,

    EndangeredFlyingClawedFishEating = HasClaws | CanFly | EatsFish | Endangered,
}
```

Const Enums

If you have an enum definition like the following:

```
enum Tristate {
    False,
    True,
    Unknown
}

var lie = Tristate.False;
```

the line `var lie = Tristate.False` is compiled to the JavaScript `var lie = Tristate.False` (yes output is same as input). This means that at execution the runtime will need to lookup `Tristate` and then `Tristate.False`. To get a performance boost here you can mark the `enum` as a `const enum`. This is demonstrated below:

```
const enum Tristate {
    False,
    True,
    Unknown
}

var lie = Tristate.False;
```

generates the JavaScript:

```
var lie = 0;
```

i.e. the compiler :

1. *inlines* any usages of the enum (`0` instead of `Tristate.False`).
2. does not generate any JavaScript for the enum definition (there is no `Tristate` variable at runtime) as its usages are inlined.

Const enum preserveConstEnums

Inlining has obvious performance benefits. The fact that there is no `Tristate` variable at runtime is simply the compiler helping you out by not generating JavaScript that is not actually used at runtime. However you might want the compiler to still generate the JavaScript version of the enum definition for stuff like *number to string* or *string to number* lookups as we saw. In this case you can use the compiler flag `--preserveConstEnums` and it will still generate the `var Tristate` definition so that you can use `Tristate["False"]` or `Tristate[0]` manually at runtime if you want. This does not impact *inlining* in any way.

Project

To create a successful project using TypeScript you need to understand the various project organization language features available. In this section we will cover "compilation context", declaration spaces and modules.

Compilation Context

The compilation context is basically just a fancy term for grouping of the files that TypeScript will parse and analyze to determine what is valid and what isn't. Along with the information about which files, the compilation context contains information about *which compiler options*. A great way to define this logical grouping (we also like to use the term *project*) is using a `tsconfig.json` file.

Basic

It is extremely easy to get started with tsconfig.json as the basic file you need is:

```
{}
```

i.e. an empty JSON file at the *root* of your project. This way TypeScript will include *all* the `.ts` files in this directory (and sub directories) as a part of the compilation context. It will also select a few sane default compiler options.

compilerOptions

You can customize the compiler options using `compilerOptions` .

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "commonjs",
        "declaration": false,
        "noImplicitAny": false,
        "removeComments": true,
        "noLib": false
    }
}
```

These (and more) compiler options will be discussed later.

TypeScript compiler

Good IDEs come with built in support for on the fly `ts` to `js` compilation. If however you want to run the TypeScript compiler manually from the command line when using `tsconfig.json` you can do it in a few ways.

- Just run `tsc` and it will look for `tsconfig.json` in the current as well as all parent folders till it finds it.
- Run `tsc -p ./path-to-project-directory` . Of course the path can be a complete or relative to the current directory.

You can even start the TypeScript compiler in *watch* mode using `tsc -w` and it will watch your TypeScript project files for changes.

Declaration Spaces

There are two declaration spaces in TypeScript: The *variable* declaration space and the *type* declaration space. These concepts are explored below.

Type Declaration Space

The type declaration space contains stuff that can be used as a type annotation. E.g the following are a few type declarations:

```
class Foo { }
interface Bar { }
type Bas = {}
```

This means that you can use `Foo` , `Bar` , `Bas` etc. as a type annotation. E.g.:

```
var foo: Foo;
var bar: Bar;
var bas: Bas;
```

Notice that even though you have `interface Bar` , *you can't use it as a variable* because it doesn't contribute to the *variable declaration space*. This is shown below:

```
interface Bar {};
var bar = Bar; // ERROR: "cannot find name 'Bar'"
```

The reason why it says `cannot find name` is because the name `Bar` *is not defined* in the *variable* declaration space. That brings us to the next topic "Variable Declaration Space".

Variable Declaration Space

The variable declaration space contains stuff that you can use as a variable. We saw that having `class Foo` contributes a type `Foo` to the *type* declaration space. Guess what?, it also contributes a *variable* `Foo` to the *variable* declaration space as shown below:

```
class Foo { }
var someVar = Foo;
var someOtherVar = 123;
```

This is great as sometimes you want to pass classes around as variables. Remember that

- We couldn't use something like an `interface` that is *only* in the *type* declaration space as a variable.

Similarly something that you declare with `var`, is *only* in the *variable* declaration space and cannot be used as a type annotation:

```
var foo = 123;
var bar: foo; // ERROR: "cannot find name 'foo'"
```

The reason why it says `cannot find name` is because the name `foo` *is not defined* in the *type* declaration space.

TIPS

Copying Stuff around in the Type Declaration Space

If you want to move a class around you might be tempted to do the following:

```
class Foo { }
var Bar = Foo;
var bar: Bar; // ERROR: "cannot find name 'Bar'"
```

This is an error because `var` only copied the `Foo` into the *variable* declaration space and you therefore cannot use `Bar` as a type annotation. The proper way is to use the `import` keyword. Note that you can only use the `import` keyword in such a way if you are using *namespaces* or *modules* (more on these later):

```
namespace importing {
    export class Foo { }
}

import Bar = importing.Foo;
var bar: Bar; // Okay
```

Capturing the type of a variable

You can actually use a variable in a type annotation using the `typeof` operator. This allows you to tell the compiler that one variable is the same type as another. Here is an example to demonstrate this:

```ts
var foo = 123;
var bar: typeof foo; // `bar` has the same type as `foo` (here `number`)
bar = 456; // Okay
bar = '789'; // ERROR: Type `string` is not `assignable` to type `number`
```

Modules

Global Module

By default when you start typing code in a new TypeScript file your code is in a *global* namespace. As a demo consider a file `foo.ts` :

```
var foo = 123;
```

If you now create a *new* file `bar.ts` in the same project, you will be *allowed* by the TypeScript type system to use the variable `foo` as if it was available globally:

```
var bar = foo; // allowed
```

Needless to say having a global namespace is dangerous as it opens your code up for naming conflicts. We recommend using file modules which are presented next.

File Module

Also called *external modules*. If you have an `import` or an `export` at the root level of a TypeScript file then it creates a *local* scope within that file. So if we were to change the previous `foo.ts` to the following (note the `export` usage):

```
export var foo = 123;
```

We will no longer have `foo` in the global namespace. This can be demonstrated by creating a new file `bar.ts` as follows:

```
var bar = foo; // ERROR: "cannot find name 'foo'"
```

If you want to use stuff from `foo.ts` in `bar.ts` *you need to explicitly import it*. This is shown in an updated `bar.ts` below:

```
import {foo} from "./foo";
var bar = foo; // allowed
```

Using an `import` in `bar.ts` not only allows you to bring in stuff from other files, but also marks the file `bar.ts` as a *module* and therefore `bar.ts` doesn't pollute the global namespace either.

What JavaScript is generated from a given TypeScript file that uses external modules is driven by the compiler flag called `module`.

External modules

There is a lot of power and usability packed into the TypeScript external module pattern. Here we discuss its power and some patterns needed to reflect real world usages.

File lookup

The following statement:

```
import foo = require('foo');
```

Tells the TypeScript compiler to look for an external module declaration of the form:

```
declare module "foo" {
    /// Some variable declarations

    export var bar:number; /*sample*/
}
```

An import with a relative path e.g.:

```
import foo = require('./foo');
```

Tells the TypeScript compiler to look for a TypeScript file at the relative location `./foo.ts` or `./foo.d.ts` with respect to the current file.

This is not the complete specification but it's a decent mental model to have and use.

Compiler Module Option

The following statement:

```
import foo = require('foo');
```

will generate *different* JavaScript based on the compiler *module* option (`--module commonjs` or `--module amd` or `--module umd` or `--module system`).

Here is how to chose which one is right for you:

- Want the package on NPM : `--module commonjs`
- Only want to use the code in the browser : `--module amd`

- Want to deploy the code on NPM *and* use it in the browser *without* any dependency on something (like *requirejs*, *webpack* or *browserify* etc). : `--module umd`
- Ready for the promised future of a *truly unified* and ECMA approved module system : `--module system`

I recommend that for new projects you just use `--module system`. But it is good to be aware of this compiler option.

Import type only

The following statement:

```
import foo = require('foo');
```

actually imports *two* things:

- The type information from the imported file.
- Takes are runtime dependency on the `foo` module.

You can pick and choose so that only *the type information* is loaded and no runtime dependency occurs. Before continuing you might want to recap the *declaration spaces* section of the book.

If you do not use the imported name in the variable declaration space then the import is completely removed from the generated JavaScript. This is best explained with examples. Once you understand this we will present you with use cases.

Example 1

```
import foo = require('foo');
```

will generate the JavaScript:

```

```

Thats right. An *empty* file as foo is not used.

Example 2

```
import foo = require('foo');
var bar: foo;
```

will generate the JavaScript:

```
var bar;
```

This is because `foo` (or any of its properties e.g. `foo.bas`) is never used as a variable.

Example 3

```
import foo = require('foo');
var bar = foo;
```

will generate the JavaScript (assuming commonjs):

```
var foo = require('foo');
var bar = foo;
```

This is because `foo` is used as a variable.

Use case: Lazy loading

Type inference needs to be done *upfront*. This means that if you want to use some type from a file `foo` in a file `bar` you will have to do:

```
import foo = require('foo');
var bar: foo.SomeType;
```

However you might want to only load the file `foo` at runtime under certain conditions. For such cases you should use the `import`ed name only in *type annotations* and **not** as a *variable*. This removes any *upfront* runtime dependency code being injected by TypeScript. Then *manually import* the actual module using code that is specific to your module loader.

As an example, consider the following `commonjs` based code where we only load a module `'foo'` on a certain function call

```
import foo = require('foo');

export function loadFoo(){
    // This is lazy loading `foo` and using the original module *only* as a type annotati
    var _foo: typeof foo = require('foo');
    // Now use `_foo` as a variable instead of `foo`.
}
```

A similar sample in `amd` (using requirejs) would be:

```
import foo = require('foo');

export function loadFoo(){
    // This is lazy loading `foo` and using the original module *only* as a type annotati
    require(['foo'], (_foo: typeof foo) => {
        // Now use `_foo` as a variable instead of `foo`.
    });
}
```

This pattern is commonly used:

- in web apps where you load certain JavaScript on particular routes
- in node applications where you only load certain modules if needed to speed up application bootup.

Use case: Breaking Circular dependencies

Similar to the lazy loading use case certain module loaders (commonjs/node and amd/requirejs) don't work well with circular dependencies. In such cases it is useful to have *lazy loading* code in one direction and loading the modules upfront in the other direction.

globals.d.ts

We discussed *global* vs. *file* modules when covering projects and recommended using file based modules and not polluting the global namespace.

Nevertheless it is convenient to have *some* files just with type declarations (for smaller projects preferably one called `globals.d.ts`) in the global namespace to make it easy to have some *types* just *magically* available for consumption in *all* your TypeScript code. For any code that is going to generate *JavaScript* we still recommend using *file modules*.

`globals.d.ts` is great for adding extensions to `lib.d.ts`.

Namespaces

Namespaces provide you with a convenient syntax around a common pattern used in JavaScript:

```
(function(something) {

    something.foo = 123;

})(something || something = {})
```

Basically `something || something = {}` allows an anonymous function `function(something) {}` to *add stuff to an existing object* (the `something ||` portion) or *start a new object then add stuff to that object* (the `|| something = {}` portion). This means that you can have two such blocks split by some execution boundary :

```
(function(something) {

    something.foo = 123;

})(something || something = {})

console.log(something); // {foo:123}

(function(something) {

    something.bar = 456;

})(something || something = {})

console.log(something); // {foo:123, bar:456}
```

This is commonly used in the JavaScript land for making sure that stuff doesn't leak into the global namespace. With file based modules you don't need to worry about this, but the pattern is still useful for *logical grouping* of a bunch of functions. Therefore TypeScript provides the `namespace` keyword to group these e.g.

```
namespace utility {
    export function log(msg) {
        console.log(msg);
    }
    export function error(msg) {
        console.error(msg);
    }
}

// usage
utility.log('Call me');
utility.error('maybe!');
```

The `namespace` keyword generates the same JavaScript that we saw earlier:

```
(function (utility) {

// Add stuff to utility

})(utility || (utility = {}));
```

One thing to note is that namespaces can be nested so you can do stuff like `namespace utility.messaging` to nest a `messaging` namespace under `utility`.

For most projects we recommend using external modules and using `namespace` for quick demos and porting old JavaScript code.

TypeScript Type System

We covered the main features of the TypeScript Type System back when we disucssed *Why TypeScript?*. The following are a few key takeaways from that discussion which don't need further explanation:

- The type system in typescript is designed to be *optional* so that *your javascript is typescript*.
- TypeScript does not block *JavaScript emit* in the presence of Type Errors, allowing you to *progressively update your JS to TS*.

Now lets start with the *syntax* of the TypeScript type system. This way you can start using these annotations in your code immediately and see the benefit. This will prepare you for a deeper dive later.

Basic Annotations

As mentioned before Types are annotated using `:TypeAnnotation` syntax. Anything that is available in the type declaration space can be used as a Type Annotation.

The following example demonstrates type annotations can be used for variables, function parameters and function return values.

```
var num: number = 123;
function identity(num: number): number {
    return num;
}
```

Primitive Types

The JavaScript primitive types are well represented in the TypeScript type system. This means `string` , `number` , `boolean` as demonstrated below:

```
var num: number;
var str: string;
var bool: boolean;

num = 123;
num = 123.456;
num = '123'; // Error

str = '123';
str = 123; // Error

bool = true;
bool = false;
bool = 'false'; // Error
```

Arrays

TypeScript provides dedicated type syntax for arrays to make it easier for you to annotate and document your code. The syntax is basically postfixing `[]` to any valid type annotation (e.g. `:boolean[]`). It allows you to safely do any array manipulation that you would normally do and protects you from errors like assigning a member of the wrong type. This is demonstrated below:

```
var boolArray: boolean[];

boolArray = [true, false];
console.log(boolArray[0]); // true
console.log(boolArray.length); // 2
boolArray[1] = true;
boolArray = [false, false];

boolArray[0] = 'false'; // Error!
boolArray = 'false'; // Error!
boolArray = [true, 'false']; // Error!
```

Interfaces

Interfaces are the core way in TypeScript to compose multiple type annotations into a single named annotation. Consider the following example :

```
interface Name {
    first: string;
    second: string;
}

var name: Name;
name = {
    first: 'John',
    second: 'Doe'
};

name = {           // Error : `second` is missing
    first: 'John'
};
name = {           // Error : `second` is the wrong type
    first: 'John',
    second: 1337
};
```

Here we've composed the annotations `first: string` + `second: string` into a new annotation `Name` that enforces the type checks on individual members. Interfaces have a lot of power in TypeScript and we will dedicate an entire section to how you can use that to your advantage.

Inline Type Annotation

Instead of creating a new `interface` you can annotate anything you want *inline* using `:{ /*Structure*/ }`. The previous example presented again with an inline type:

```
var name: {
    first: string;
    second: string;
};
name = {
    first: 'John',
    second: 'Doe'
};

name = {           // Error : `second` is missing
    first: 'John'
};
name = {           // Error : `second` is the wrong type
    first: 'John',
    second: 1337
};
```

Inline types are great for quickly providing a one off type annotation for something. It saves you the hassle of coming up with (a potentially bad) type name. However, if you find yourself putting in the same type annotation inline multiple times its a good idea to consider refactoring it into an interface (or a `type alias` covered later in this section).

Special Types

Beyond the primitive types that have covered there are few types that have special meaning in TypeScript. These are `any` , `null` , `undefined` , `void` .

any

The `any` type holds a special place in the TypeScript type system. It gives you an escape hatch from the type system to tell the compiler to bugger off. `any` is compatible with *any and all* types in the type system. This means that *anything can be assigned to it* and *it can be assigned to anything*. This is demonstrated it the below example:

```
var power: any;

// Takes any and all types
power = '123';
power = 123;

// Is compatible with all types
var num: number;
power = num;
num = power;
```

If you are porting JavaScript code to TypeScript, you are going to be close friends with `any` in the beginning. However, don't take this friendship too seriously as it means that *it is up to you to ensure the type safety*. You are basically telling the compiler to *not do any meaningful static analysis*.

`null` and `undefined`

The `null` and `undefined` JavaScript literals are effectively treated by the type system the same as something of type `any` . These literals can be assigned to any other type. This is demonstrated in the below example:

```
var num: number;
var str: string;

// These literals can be assigned to anything
num = null;
str = undefined;
```

`:void`

Use `:void` to signify that a function does not have a return type.

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

Generics

Many algorithms and data structures in computer science do not depend on the *actual type* of the object. A simple toy example is a function that takes a list of items and returns a reversed list of items:

```
function reverse<T>(items: T[]): T[] {
    var toreturn = [];
    for (let i = items.length - 1; i >= 0; i--) {
        toreturn.push(items[i]);
    }
    return toreturn;
}

var sample = [1, 2, 3];
var reversed = reverse(sample);
console.log(reversed); // 3,2,1

// Safety!
reversed[0] = '1';     // Error!
reversed = ['1', '2']; // Error!

reversed[0] = 1;       // Okay
reversed = [1, 2];     // Okay
```

Here you are basically saying that the function `reverse` takes an array (`items: T[]`) of *some* type `T` (notice the type parameter in `reverse<T>`) and returns an array of type `T` (notice `: T[]`). Because the `reverse` function returns items of the same type as it takes, TypeScript knows the `reversed` variable is also of type `number[]` and will give you Type safety. Similarly if you pass in an array of `string[]` to the reverse function the returned result is also an array of `string[]` and you get similar type safety as shown below:

```
var strArr = ['1', '2'];
var reversedStrs = reverse(strArr);

reversedStrs = [1, 2]; // Error!
```

In fact JavaScript arrays already have a `.reverse` function and TypeScript does indeed use generics to define its structure:

```
interface Array<T> {
 reverse(): T[];
 // ...
}
```

This means that you get type safety when calling `.reverse` on any array as shown below:

```
var numArr = [1, 2];
var reversedNums = numArr.reverse();

reversedNums = ['1', '2']; // Error!
```

We will discuss more about the `Array<T>` interface later when we present `lib.d.ts` in the section **Ambient Declarations**.

Union Type

Quite commonly in JavaScript you want to allow a property to be one of multiple types e.g *a `string` or a `number`*. This is where the *union type* (denoted by `|` in a type annotation e.g. `string|number`) comes in handy. A common use case is a function that can take a single object or an array of the object e.g.

```
function formatCommandline(command: string[]|string) {
    var line = '';
    if (typeof command === 'string') {
        line = command.trim();
    } else {
        line = command.join(' ').trim();
    }

    // Do stuff with line:string
}
```

Intersection Type

`extend` is a very common pattern in JavaScript where you take two objects and create a new one that has the features of both these objects. An **Intersection Type** allows you to use this pattern in a safe way as demonstrated below:

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U> {};
    for (let id in first) {
        result[id] = first[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            result[id] = second[id];
        }
    }
    return result;
}

var x = extend({ a: "hello" }, { b: 42 });

// x now has both `a` and `b`
var a = x.a;
var b = x.b;
```

Tuple Type

JavaScript doesn't have first class tuple support. People generally just use an array as a tuple. This is exactly what the TypeScript type system supports. Tuples can be annotated using `:[typeofmember1, typeofmember2]` etc. A tuple can have any number of members. Tuples are demonstrated in the below example:

```
var nameNumber: [string, number];

// Okay
nameNumber = ['Jenny', 8675309];

// Error!
nameNumber = ['Jenny', '867-5309'];
```

Combine this with the destructuring support in TypeScript, tuples feel fairly first class despite being arrays underneath.

```
var nameNumber: [string, number];
nameNumber = ['Jenny', 8675309];

var [name, num] = nameNumber;
```

Type Alias

TypeScript provides convenient syntax for providing names for type annotations that you would like to use in more than one place. The aliases are created using the `type SomeName = someValidTypeAnnotation` syntax. An example is demonstrated below:

```
type StrOrNum = string|number;

// Usage: just like any other notation
var sample: StrOrNum;
sample = 123;
sample = '123';

// Just checking
sample = true; // Error!
```

Unlike an `interface` you can give a type alias to literally any type annotation (useful for stuff like union and intersection types). Here are a few more examples to make you familiar with the syntax:

```
type Text = string | { text: string };
type Coordinates = [number, number];
type Callback = (data: string) => void;
```

> TIP: If you need to have deep hierarchies of Type annotations use an `interface`. Use a type alias for simpler object structures (like `Coordinates`) just to give them a semantic name.

Summary

Now that you can start annotating most of your JavaScript code we can jump into the nitty gritty details of all the power available in the TypeScript's Type System.

Migrating From JavaScript

In general the process consists of the following steps:

- Add a `tsconfig.json`
- Change your source code file extensions from `.js` to `.ts`. Start *suppressing* errors using `any`.
- Write new code in TypeScript and make as little use of `any` as possible.
- Go back to the old code and start adding type annotations and fix identified bugs.
- Use ambient definitions for third party JavaScript code.

Let us discuss a few of these points further.

Note that all JavaScript is *valid* TypeScript. That is to say that if you give the TypeScript compiler some JavaScript -> the JavaScript emitted by the TypeScript compiler will behave exactly the same as the original JavaScript. This means that changing the extension from `.js` to `.ts` will not adversely affect your codebase.

Suppressing Errors

TypeScript will immediately start TypeChecking your code, and your original JavaScript code *might not be as neat as you thought it was* and hence you get diagnostic errors. Many of these errors you can suppress with using `any` e.g.

```
var foo = 123;
var bar = 'hey';

bar = foo; // ERROR: cannot assign a number to a string
```

Even though the **error is valid** (and in most cases the inferred information will be better than what the original authors of different portions of the code bases imagined), your focus will probably be writing new code in TypeScript while progressively updating the old code base. Here you can suppress this error with a type assertion as shown below:

```
var foo = 123;
var bar = 'hey';

bar = <any>foo; // Okay!
```

In other places you might want to annotate something as `any` e.g.

```
function foo() {
    return 1;
}
var bar = 'hey';
bar = foo(); // ERROR: cannot assign a number to a string
```

Suppressed:

```
function foo(): any { // Added `any`
    return 1;
}
var bar = 'hey';
bar = foo(); // Okay!
```

> Note: Suppressing errors is dangerous, but it allows you to take notice of errors in your *new* TypeScript code. You might want to leave `// TODO:` comments as you go along.**

Third Party JavaScript

You can change your JavaScript to TypeScript, but you can't change the whole world to use JavaScript. This is where TypeScript's ambient definition support comes in. In the beginning we recommend you create a `vendor.d.ts` (the `.d.ts` extension specifies the fact that this is a *declaration file*) and start adding dirty stuff to it. Alternatively create a file specific for the library e.g. `jquery.d.ts` for jquery.

> Note : Well maintained and strongly typed definitions for nearly the top 90% JavaScript libraries out there exists in an OSS Repository called DefinitelyTyped. We recommend looking there before creating your own definitions as we present here. Nevertheless this quick and dirty way is vital knowledge to decrease your initial friction with TypeScript**.

Consider the case of `jquery`, you can create a *trivial* definition for it quite easily:

```
declare var $: any;
```

Sometimes you might want to add an explicit annotation on something (e.g. `JQuery`) and you need something in *type declaration space*. You can do that quite easily using the `type` keyword:

```
declare type JQuery = any;
declare var $: JQuery;
```

This provides you an easier future update path.

Again, a high quality `jquery.d.ts` exists at DefinitelyTyped. But you now know how to overcome any JavaScript -> TypeScript friction *quickly* when using third party JavaScript. We will look at ambient declarations in detail next.

Ambient Declarations

As we mentioned in why TypeScript:

> A major design goal of TypeScript was to make it possible for you to safely and easily use existing JavaScript libraries in TypeScript. TypeScript does this by means of *declaration*

Ambient declarations allow you to *safely use existing popular JavaScript libraries* and *incrementally migrate your JavaScript/CoffeeScript/Others-Compile-To-Js-Language project to TypeScript*.

Studying patterns in ambient declarations for *third party JavaScript code* is good practice for annotating *your* TypeScript code base as well. This is why we present it so early on.

Declaration file

You can tell TypeScript that you are trying to describe code that exists elsewhere (e.g. written in JavaScript/CoffeeScript/The runtime environment like the browser or nodejs) using the `declare` keyword. As a quick example:

```
foo = 123; // Error: `foo` is not defined
```

vs.

```
declare var foo:any;
foo = 123; // allowed
```

You have the option of putting these declarations in a `.ts` file or in a `.d.ts` file. We highly recommend you in your real world projects you use a separate `.d.ts` (start with one called something like `globals.d.ts` or `vendor.d.ts`).

If a file has the extension `.d.ts` then each root level definition much have the `declare` keyword prefixed to it to make it clear that the author knows that there will be *no code emitted by TypeScript to ensure that this defined item will exist at runtime*.

> - Ambient declarations is a promise that you are making with the compiler. If these do not exist at runtime and you try to you them, things will break without warning.
> - Ambient declarations are like docs. If the source changes the docs need to be kept updated. So you might have new behaviours that work at runtime but no one's updated the ambient declaration and hence you get compiler errors.

Variables

For example to tell TypeScript about the `process` variable you *can* do:

```
declare var process:any;
```

> You don't *need* to do this for `process` as there is already a community maintained `node.ts`

This allows you to use the `process` variable without TypeScript complaining:

```
process.exit()
```

We recommend using an interface wherever possible e.g:

```
interface Process {
    exit(code?:number):void;
}
declare var process: Process;
```

This allows other people to *extend* the nature of these global variables while still telling TypeScript about such modifications. E.g. consider the following case where we add an `exitWithLogging` function to process for our amusement:

```
interface Process {
    exitWithLogging(code?:number):void;
}
process.exitWithLogging = function() {
    console.log("exiting");
    process.exit.apply(process,arguments);
}
```

Lets look at interfaces in a bit more detail next.

Interfaces

Interfaces have *zero* runtime JS impact. There is a lot of power in TypeScript interfaces to declare the structure of variables.

The following two are equivalent declarations, the first uses an *inline annotation*, the second uses an *interface*:

```
// Sample A
declare var myPoint: { x: number; y: number; };

// Sample B
interface Point {
    x: number; y: number;
}
declare var myPoint: Point;
```

However the beauty of *Sample B* is that if someone authors a library that builds on the `myPoint` library to add new members they can do that with if you used an interface:

```
// Lib a.d.ts
interface Point {
    x: number; y: number;
}
declare var myPoint: Point;

// Lib b.d.ts
interface Point {
    x: number; y: number; z: number;
}

// Your code
var myPoint.z; // Allowed!
```

This is because **interfaces in TypeScript are open ended.** This is a vital tenant of TypeScript that it allows you to mimic the extensibility of JavaScript using *interfaces*.

`lib.d.ts`

A special declaration file `lib.d.ts` ships with every installation of TypeScript. This file contains the ambient declarations for various common JavaScript constructs present in JavaScript runtimes and the DOM.

- This file is automatically included in the compilation context of a TypeScript project.
- The objective of this file to make it easy for you start writing *type checked* JavaScript code.

You can exclude this file from the compilation context by specifying the `--noLib` compiler command line flag (or `"noLib" : true` in `tsconfig.json`).

Example Usage

As always lets look at examples of this file being used in action.

```
var foo = 123;
var bar = foo.toString();
```

This code type checks fine *because* the `toString` function is defined in `lib.d.ts` for all JavaScript objects.

If you use the same sample code with the `noLib` option you can a type check error:

```
var foo = 123;
var bar = foo.toString(); // ERROR: Property 'toString' does not exist on type 'number'.
```

So now that you understand the importance of `lib.d.ts` what does its contents look like? We examine that next.

`lib.d.ts` inside look

The contents of `lib.d.ts` are primarily a bunch of *variable* declarations e.g. `window` , `document` , `math` and a bunch of similar *interface* declarations e.g. `Window` , `Document` , `Math` .

The simplest way to discover what is what is to type in code *that you know works* e.g. `Math.floor` and then F12 (go to definition) using your IDE (atom-typescript has great support for this).

Lets look at a sample *variable* declaration, e.g. `window` is defined as:

```
declare var window: Window;
```

That is just a simple `declare var` followed by the variable name (here `window`) and an interface for a type annotation (here the `Window` interface). These variables generally point to some global *interface* e.g. here is a small sample of the (actually quite massive) `Window` interface:

```
interface Window extends EventTarget, WindowTimers, WindowSessionStorage, WindowLocalStor
    animationStartTime: number;
    applicationCache: ApplicationCache;
    clientInformation: Navigator;
    closed: boolean;
    crypto: Crypto;
    // so on and so forth...
}
```

You can see that here is a *lot* of type information in these interfaces. In the absence of TypeScript *you* would need to keep this in *your* head. Now you can offload that knowledge on the compiler with easy access to it using things like `intellisence` .

There is a good reason for using *interfaces* for these globals. It allows you to *add additional properties* to these globals *without* a need to change `lib.d.ts` . We will cover this concept next.

Modifying native types

Since an `interface` in TypeScript is open ended this means that you can just add members to the interfaces declared in `lib.d.ts` and TypeScript will pick up on the additions. Note that you need to make these changes in a *global module* for these interfaces to get associated with `lib.d.ts` . We even recommend creating a special file called `globals.d.ts` for this purpose.

Here are a few example cases where we add stuff to `window` , `Math` , `Date` :

Example `window`

Just add stuff to the `Window` interface e.g.

```
interface Window {
    helloWorld():void;
}
```

This will allow you to use it in a *type safe* manner:

```
// Add it at runtime
window.helloWorld = () => console.log('hello world');
// Call it
window.helloWorld();
// Misuse it and you get an error:
window.helloWorld('gracius'); // Error: Supplied parameters do not match the signature of
```

Example `Math`

The global variable `Math` is defined in `lib.d.ts` as (again, use your dev tools to navigate to definition):

```
/** An intrinsic object that provides basic mathematics functionality and constants. */
declare var Math: Math;
```

i.e. the variable `Math` is an instance of the `Math` interface. The `Math` interface is defined as:

```
interface Math {
    E: number;
    LN10: number;
    // others ...
}
```

This means that if you want to add stuff to the `Math` global variable you just need to add it to the `Math` global interface, e.g. consider the `seedrandom` project which adds a `seedrandom` function to the global `Math` object. This can be declared quite easily:

```
interface Math {
    seedrandom(seed?: string);
}
```

And then you can just use it:

```
Math.seedrandom();
// or
Math.seedrandom("Any string you want!");
```

Example `Date`

If you look the definition of the `Date` *variable* in `lib.d.ts` you will find:

```
declare var Date: DateConstructor;
```

The interface `DateConstructor` is similar to what you have seen before with `Math` and `Window` in that it contains members you can use off of the `Date` global variable e.g. `Date.now()` . In addition to these members it contains *construct* signatures which allow you to create `Date` instances (e.g. `new Date()`). A snippet of the `DateConstructor` interface is shown below:

```
interface DateConstructor {
    new (): Date;
    // ... other construct signatures

    now(): number;
    // ... other member functions
}
```

Consider the project `datejs` . DateJS adds members to both the `Date` global variable and `Date` instances. Therefore a TypeScript definition for this library would look like (BTW the community has already written this for you in this case):

```
/** DateJS Public Static Methods */
interface DateConstructor {
    /** Gets a date that is set to the current date. The time is set to the start of the
    today(): Date;
    // ... so on and so forth
}

/** DateJS Public Instance Methods */
interface Date {
    /** Adds the specified number of milliseconds to this instance. */
    addMilliseconds(milliseconds: number): Date;
    // ... so on and so forth
}
```

This allows you to do stuff like the following in a TypeSafe manner:

```
var today = Date.today();
var todayAfter1second = today.addMilliseconds(1000);
```

Example `string`

If you look inside `lib.d.ts` for string you will find stuff similar to what we saw for `Date` (`String` global variable, `StringConstructor` interface, `String` interface). One thing of note though is that the `String` interface impacts string *literals* as well as demonstrated in the below code sample:

```
interface String {
    endsWith(suffix: string): boolean;
}

String.prototype.endsWith = function(suffix: string): boolean {
    var str: string = this;
    return str && str.indexOf(suffix, str.length - suffix.length) !== -1;
}

console.log('foo bar'.endsWith('bas')); // false
console.log('foo bas'.endsWith('bas')); // true
```

Similar variable / interfaces exist for other things that have both static and instance member like `Number` , `Boolean` , `RegExp` etc. and these interfaces affect literal instances of these types as well.

Using your own custom lib.d.ts

As we mentioned earlier using the `noLib` boolean compiler flag causes TypeScript to exclude the automatic inclusion of `lib.d.ts` . There are various reasons why this is a useful feature. Here are a few of the common ones:

- You are running in a custom JavaScript environment that differs *significantly* from the standard browser based runtime environment.
- You like to have *strict* control over the *globals* available in your code. E.g. lib.d.ts defines `item` as a global variable and you don't want this to leak into your code.

Once you have excluded the default `lib.d.ts` you can include a similarly named file into your compilation context and TypeScript will pick it up for type checking.

Note: Be careful with `--noLib` . Once you are in noLib land, if you chose to share your project others, they will be *forced* into noLib land (or rather *your lib* land). Even worse if you bring *their* code into your project you might need to port it to *your lib* based code.

Compiler target effect on `lib.d.ts`

Setting the compiler target to be `es6` causes the `lib.d.ts` to include *addtional* ambient declarations for more modern stuff like `Promise`. This magical effect of the compiler target changing the *ambience* of the code is desirable for some people and for others its problematic as it conflates *code generation* with *code ambience*. For people that want to compile with both targets *and actually use the modern es6 features* using poly-fills, it is recommended that they compile with `--noLib` and include their own customized `lib.d.ts` as mentioned before.

Type Assertion

TypeScript allows you to override its inferred and analyzed view of types any way you want to. This is done by a mechanism called "type assertion". TypeScript's type assertion are purely you telling the compiler that you know about the types better than it does, and that it should not second guess you.

A common use case for type assertion is when you are porting over code from JavaScript to TypeScript. For example consider the following pattern:

```
var foo = {};
foo.bar = 123; // error : property 'bar' does not exist on `{}`
foo.bas = 'hello'; // error : property 'bas' does not exist on `{}`
```

Here the code errors because the *inferred* type of `foo` is `{}` i.e. an object with zero properties. Therefore you are not allowed to add `bar` or `bas` to it. You can fix this simply by a type assertion `as Foo` :

```
interface Foo {
    bar: number;
    bas: string;
}
var foo = {} as Foo;
foo.bar = 123;
foo.bas = 'hello';
```

`as foo` vs. `<foo>`

Originally the syntax that was added was `<foo>` . This is demonstrated below:

```
var foo: any;
var bar = <string> foo; // bar is now of type "string"
```

However there is an ambiguity in the language grammar when using `<foo>` style assertions in JSX:

```
var foo = <string>bar;
</string>
```

Therefore it is now recommended that you just use `as foo` for consistency.

Type Assertion vs. Casting

The reason why it's not called "type casting" is that *casting* generally implies some sort of runtime support. However *type assertions* are purely a compile time construct and a way for you to provide hints to the compiler on how you want your code to be analyzed.

Assertion considered harmful

In many cases assertion will allow you to easily migrate legacy code (and even copy paste other code samples into your codebase), however you should be careful with your use of assertions. Take our original code as a sample, the compiler will not protect you from forgetting to *actually add the properties you promised*:

```
interface Foo {
    bar: number;
    bas: string;
}
var foo = {} as Foo;
// ahhhh .... forget something?
```

Also another common thought is using an assertion as a means of providing *autocomplete* e.g.:

```
interface Foo {
    bar: number;
    bas: string;
}
var foo = <Foo>{
    // the compiler will provide autocomplete for properties of Foo
    // But it is easy for the developer to forget adding all the properties
    // Also this code is likely to break if Foo gets refactored (e.g. a new property adde
};
```

but the hazard here is the same, if you forget a property the compiler will not complain. It is better if you do the following:

```
interface Foo {
    bar: number;
    bas: string;
}
var foo:Foo = {
    // the compiler will provide autocomplete for properties of Foo
};
```

In some cases you might need to create a temporary variable, but at least you will not be making (possibly false) promises and instead relying on the type inference to do the checking for you.

Double assertion

The type assertion despite being a bit unsafe as we've shown, is not *completely open season*. E.g the following is a very valid use case (e.g. the user things the event passed in will be a more specific case of an event) and the type assertion works as expected

```
function handler (event: Event){
    let mouseEvent = event as MouseEvent;
}
```

However the following is most likely an error and TypeScript will complain as shown despite the user's type assertion:

```
function handler(event: Event) {
    let element = event as HTMLElement; // Error : Neither 'Event' not type 'HTMLElement'
}
```

If you *still want TypeScript you can use a double assertion*, but first asserting to `any` which is compatible with all types and therefore the compiler no longer complains:

```
function handler(event: Event) {
    let element = event as any as HTMLElement; // Okay!
}
```

How typescript determines if a single assertion is not enough

Basically it allows the assertion from type `S` to `T` succeed if either `S` is a subtype of `T` or `T` is a subtype of `S`. This is to provide extra safety when doing type assertions ... completely wild assertions can be very unsafe and you need to use `any` to be that unsafe.

Freshness

(Note: you need typescript nightly for this at the moment).

TypeScript provides a concept of **Freshness** (also called *strict object literal checking*) to make it easier to type check object literals that would otherwise be structurally type compatible.

Structural typing is *extremely convenient*. Consider the following piece of code. This allows you to *very conviniently* upgrade your JavaScript to TypeScript while still preserving a level of type safety.

```
function logName(something: { name: string }) {
    console.log(something.name);
}

var person = { name: 'matt', job: 'being awesome' };
var animal = { name: 'cow', diet: 'vegan, but has milk of own species' };
var random = { note: `I don't have a name property` };

logName(person); // okay
logName(animal); // okay
logName(random); // Error : property `name` is missing
```

However *structural* typing has a weakness in that it allows you to misleadingly think that something accepts more data than it actually does. This is demonstrated in the following code which TypeScript will error on as shown:

```
function logName(something: { name: string }) {
    console.log(something.name);
}

logName({ name: 'matt' }); // okay
logName({ name: 'matt', job: 'being awesome' }); // Error: object literals must only spec
```

Note that this error *only happens on object literals*. Without this error one might look at the call `logName({ name: 'matt', job: 'being awesome' })` and think that *logName* would do something useful with `job` where as in reality it will completely ignore it.

Another big use case is with interfaces that have optional members, without such object literal checking, a typo would type check just fine. This is demonstrated below:

```
function logIfHasName(something: { name?: string }) {
    if (something.name) {
        console.log(something.name);
    }
}
var person = { name: 'matt', job: 'being awesome' };
var animal = { name: 'cow', diet: 'vegan, but has milk of own species' };
var random = { note: `I don't have a name property` };

logIfHasName(person); // okay
logIfHasName(animal); // okay
logIfHasName(random); // okay
logIfHasName({neme: 'I just misspelled name to neme'}); // Error: object literals must on
```

The reason why only object literals are type checked this way is because having a object literal created on the spot and passed in with additional properties *that aren't actually used* is almost always a typo or a misunderstanding of the API.

Allowing extra properties

A type can include an index signature to explicitly indicate that excess properties are permitted.

```
var x: { foo: number, [x: string]: any };
x = { foo: 1, baz: 2 };  // Ok, `baz` matched by index signature
```

TypeScript with NodeJS

TypeScript has had *first class* support for NodeJS since inception. Here's how to get setup with a NodeJS project in TypeScript:

1. Compile with `--module` set to `"commonjs"` (as we mentioned in modules)
2. Compile with `--declaration` set to `true`. This gets TypeScript to generate a `.d.ts` file which is what TypeScript looks for in your `node_modules` for using TypeScript NPM modules from other TypeScript modules.
3. Add `node.d.ts` (`tsd install node`) to your compilation context.

That's it! Now you can use all the built in node modules (e.g. `import fs = require('fs')`) with all the safety and developer ergonomics of TypeScript!

Creating TypeScript node modules

You can even use other node modules written in TypeScript. As a module author, two things you should do:

- have `"declaration": true` to get the `.d.ts` file that other modules will look for.
- you might want to have a `typings` field (e.g. `src/index`) in your `package.json` similar to the `main` field to point to the default TypeScript definition export. For an example look at `package.json` for csx.

Example package: `npm install csx` for csx, usage: `import csx = require('csx')` .

JSX Support

TypeScript supports JSX transpilation and code analysis. If you are unfamiliar with JSX here is an excerpt from the official website:

> JSX is a XML-like syntax extension to ECMAScript without any defined semantics. It's NOT intended to be implemented by engines or browsers. It's NOT a proposal to incorporate JSX into the ECMAScript spec itself. It's intended to be used by various preprocessors (transpilers) to transform these tokens into standard ECMAScript.

The motivation behind JSX is to allow users to write HTML like views *in JavaScript* so that you can:

- Have the view Type Checked by the same code that is going to check your JavaScript
- Have the view be aware of the context it is going to operate under (i.e. strethen the *controller-view* connection in traditional MVC)

This decreases the chances of errors and increases the maintainability of your user interfaces. The main consumer of JSX at this point is ReactJS from facebook. This is the usage of JSX that we will discuss here.

Setup

- Use files with the extension `.tsx` (instead of `.ts`).
- Use `"jsx" : "react"` in your `tsconfig.json` 's `compilerOptions` .
- Install the definitions for JSX and React into your project : (`tsd install react --save --resolve`).
- Import react into your `.tsx` files (`import * as React from "react"`).

HTML Tags vs. Components

React can either render HTML tags (strings) or React components (classes). The JavaScript emit for these elements is different (`React.createElement('div')` vs. `React.createElement(MyComponent)`). The way this is determined is by the *case* of the *first* letter. `foo` is treated as an HTML tag and `Foo` is treated as a component.

Type Checking

HTML Tags

An HTML Tag `foo` is to be of the type `JSX.IntrinsicElements.foo`. These types are already defined for all the major tags in a file `react-jsx.d.ts` which we had you install as a part of the setup. Here is a sample of the the contents of the file:

```
declare module JSX {
    interface IntrinsicElements {
        a: React.HTMLAttributes;
        abbr: React.HTMLAttributes;
        div: React.HTMLAttributes;
        span: React.HTMLAttributes;

        /// so on ...
    }
}
```

Components

Components are type checked based on the `props` property of the component. This is modeled after how JSX is transformed i.e. the attributes become the `props` of the component.

To create React components we recommend using ES6 classes. The `react.d.ts` file defines the `React.Component<Props,State>` class which you should extend in your own class providing your own `Props` and `State` interfaces. This is demonstrated below:

```
interface Props {
  foo: string;
}
class MyComponent extends React.Component<Props, {}> {
    render() {
        return <span>{this.props.foo}</span>
    }
}

<MyComponent foo="bar" />
```

Non React JSX

TypeScript provides you with the ability to use something other than React with JSX in a type safe manner. The following lists the customizability points, but note that this is for advanced UI framework authors:

- You can disable `react` style emit by using `"jsx" : "preserve"` option. This means that JSX is emitted *as is* and then you can use your own custom transpiler to transpile the JSX portions.
- Using the `JSX` global module:
  - You can control what HTML tags are available and how they are type checked by customizing the `JSX.IntrinsicElements` interface members.
  - When using components:
    - You can control which `class` must be inherited by components by customizing the default `interface ElementClass extends React.Component<any, any> { }` declaration.
    - You can control which property is used to type check the attributes (the default is `props`) by customizing the `declare module JSX { interface ElementAttributesProperty { props: {}; } }` declaration.

TIPs

In this section we present a number of tips that we have collected over the course of using TypeScript in the real world.

Return an object literal

Sometimes you need a function that just returns a simple object literal. However, something like

```
var foo = ()=>{
    bar: 123
};
```

is a parsing compiler error. You can fix it but surrounding the object literal in `()`:

```
var foo = ()=>({
    bar: 123
});
```

String enums

Sometimes you need a collection of strings collected under a common key. TypeScript does have enum support but it is `number` based. You can create something similar that is string based quite easily using a variable definition, e.g.:

```
let Tristate = {
    False : '',
    True: '',
    Unknown: ''
};

// make values same as keys
Object.keys(Tristate).map((key) => Tristate[key] = key);
```

Because of TypeScript's inference engine only the provided members are accessible (e.g. `Tristate.False` , `Tristate.True` , `Tristate.Unknown` in our case) and the next line makes the values the same as the keys (so that you don't have typo's and refactoring the key automatically changes the value).

You can use such an enum as follows:

```
// Assigning
let state = Tristate.True;

// Checking if it matches
if (state === Tristate.True) {

}
```

This is just a pattern to :

- reduce your reliance on magic strings, and provides easy documentation about all supported values for a particular variable.
- make strings less brittle, if you choose to refactor a member all instances will get refactored (or error).

> One thing missing from this is a good type annotation. Sadly at the moment it needs to be `: string`, however in practice it hasn't been a big issue.

Nominal Typing

The TypeScript type system is structural and this is one of the main motivating benefits. However, there are real-world use cases for a system where you want two variables to be differentiated because they have a different *type name* even if they have the same structure. A very common use case is *identity* structures (which are generally just strings with semantics associated with their *name* in languages like C#/Java).

The workaround involves the following:

- adding an unused property on a type to break structural compatability.
- using a type assertion when needing to new up or cast down.

This is demonstrated below:

```
// FOO
interface FooId extends String {
    _fooIdBrand: string; // To prevent type errors
}

// BAR
interface BarId extends String {
    _barIdBrand: string; // To prevent type errors
}

/**
 * Usage Demo
 */
var fooId: FooId;
var barId: BarId;

// Safety!
fooId = barId; // error
barId = fooId; // error
fooId = <FooId>barId; // error
barId = <BarId>fooId; // error

// Newing up
fooId = 'foo' as any;
barId = 'bar' as any;

// If you need the base string
var str: string;
str = fooId as any;
str = barId as any;
```

Using `_` prefix and a `Brand` suffix is a convention I strongly recommend (and the one followed by the TypeScript team).

Stateful Functions

A common feature in other programming languages is usage of the `static` keyword to increase the *lifetime* (not *scope*) of a function variable to live beyond function invocations. Here is a `c` sample that achieves this:

```
void called(){
    static count = 0;
    count++;
    printf("Called : %d", count);
}

int main (){
    called(); // Called : 1
    called(); // Called : 2
    return 0;
}
```

Since JavaScript (or TypeScript) doesn't have function statics you can active the same thing using various abstractions that wrap over a local variable e.g. using a `class` :

```
class Called {
    count = 0;
    called = () => {
        this.count++;
        console.log(`Called : ${this.count}`);
    }
}

let {called} = new Called();

called(); // Called : 1
called(); // Called : 2
```

> C++ developers also try and achieve this using a pattern they call `functor` (a class that overrides the operator `()`).

Compiler

The typescript compiler source is located under the `src/compiler` folder.

It is split into the follow key parts:

- Scanner (`scanner.ts`)
- Parser (`parser.ts`)
- Binder (`binder.ts`)
- Checker (`checker.ts`)
- Emitter (`emitter.ts`)

Each of these get their own unique files in the source. These parts will be explained later on in this chapter.

NTypeScript

We have a project called NTypeScript which makes it easier to play around with the compiler API e.g. by exposing internal APIs. You use it the same way you would use `typescript` but just have an `n` prefix for all things (binary : `ntsc` , require: `ntypescript`). This is also the compiler used by atom-typescript and the one we will use to present these examples.

Syntax vs. Semantics

Just because something is *syntactically* correct doesn't mean it is *semantically* correct. Consider the following piece of TypeScript code which although *syntactically* valid is *semantically* wrong

```
var foo: number = "not a number";
```

`Semantic` means "meaning" in English. This concept is useful to have in your head.

Processing Overview

The following is a quick review of how these key parts of the TypeScript compiler compose:

```
SourceCode ~~ scanner ~~> Token Stream
```

```
Token Stream ~~ parser ~~> AST
```

```
AST ~~ binder ~~> Symbols
```

`Symbol` is the primary building block of the TypeScript *semantic* system. As shown the symbols are created as a result of binding. Symbols connect declaration nodes in the AST to other declarations contributing to the same entity.

Symbols + AST are what is used by the checker to *semantically* validate the source code

```
AST + Symbols ~~ checker ~~> Type Validation
```

Finally When a JS output is requested:

```
AST + Checker ~~ emitter ~~> JS
```

There are a few additional files in the TypeScript compiler that provide utilities to many of these key portions which we cover next.

File: Utilities

`core.ts` : core utilities used by the TypeScript compiler. A few important ones:

- `let objectAllocator: ObjectAllocator` : is a variable defined as a singleton global. It provides the definitions for `getNodeConstructor` (Nodes are covered when we look at `parser` / `AST`), `getSymbolConstructor` (Symbols are covered in `binder`), `getTypeConstructor` (Types are covered in `checker`), `getSignatureConstructor` (Signatures are the index, call and construct signatures).

File: Key Data Structures

`types.ts` contains key data structures and interfaces uses throughout the compiler. Here is a sampling of a few key ones:

- `SyntaxKind` The AST node type is identified by the `SyntaxKind` enum.
- `TypeChecker` This is the interface provided by the TypeChecker.
- `CompilerHost` This is used by the `Program` to interact with the `System` .
- `Node` An AST node.

File: System

`system.ts` . All interaction of the TypeScript compiler with the operating system goes through a `System` interface. Both the interface and its implementations (`WScript` and `Node`) are defined in `system.ts` . You can think of it as the *Operating Environment* (OE).

Now that you have an overview of the major files, we can look at the concept of `Program`

Program

Defined in `program.ts`. The compilation context (a concept we covered previously) is represented within the TypeScript compiler as a `Program`. It consists of `SourceFile` s and compiler options.

Usage of `CompilerHost`

Its interaction mechanism with the OE:

`Program` *-uses->* `CompilerHost` *-uses->* `System`

The reason for having a `CompilerHost` as a point of indirection is that it allows it's interface to be more finely tuned for `Program` needs and not bother with OE needs (e.g. the `Program` doesn't care about `fileExists` a function provided by `System`).

There are other users of `System` as well (e.g. tests).

SourceFile

The program provides an API to get the Source Files `getSourceFiles(): SourceFile[];`. Each is represented as a root-level node for an AST (called `SourceFile`).

Node

The basic building block of the Abstract Syntax Tree (AST). In general node represent non-terminals in the language grammar; some terminals are kept in the tree such as identifiers and literals.

Two key things make up an AST node documentation. Its `SyntaxKind` which identifies it within the AST and its `interface`, the API the node provides when instantiated for the AST.

Here are a few key `interface Node` members:

- `TextRange` members that identify the node's `start` and `end` in the source file.
- `parent?: Node` the parent of the node in the AST.

There are other additional members for node flags and modifiers etc. that you can lookup by searching `interface Node` in the source code but the ones we mentioned are vital for node traversal.

SourceFile

- `SyntaxKind.SourceFile`
- `interface SourceFile`.

Each `SourceFile` is a top-level AST node that is contained in the `Program`.

AST Tip: Visit Children

There is a utility function `ts.forEachChild` that allows you to visit all the child nodes of any Node in the AST.

Here is simplified snippet of the source code to demonstrate how it functions:

```
export function forEachChild<T>(node: Node, cbNode: (node: Node) => T, cbNodeArray?: (nod
        if (!node) {
            return;
        }
        switch (node.kind) {
            case SyntaxKind.BinaryExpression:
                return visitNode(cbNode, (<BinaryExpression>node).left) ||
                    visitNode(cbNode, (<BinaryExpression>node).operatorToken) ||
                    visitNode(cbNode, (<BinaryExpression>node).right);
            case SyntaxKind.IfStatement:
                return visitNode(cbNode, (<IfStatement>node).expression) ||
                    visitNode(cbNode, (<IfStatement>node).thenStatement) ||
                    visitNode(cbNode, (<IfStatement>node).elseStatement);

            // .... lots more
```

Basically it checks `node.kind` and based on that assumes an interface offered by the `node` and calls the `cbNode` on the children. Note however that this function doesn't call `visitNode` for *all* children (e.g. SyntaxKind.SemicolonToken). If you want *all* the children of a node in the AST just call `.getChildren` member function of the `Node`.

E.g. here is a function that prints the verbose `AST` of a node:

```
function printAllChildren(node: ts.Node, depth = 0) {
    console.log(new Array(depth+1).join('----'), ts.syntaxKindToName(node.kind), node.pos
    depth++;
    node.getChildren().forEach(c=> printAllChildren(c, depth));
}
```

We will see a sample usage of this function when we discuss the parser further.

AST Tip: SyntaxKind

`SyntaxKind` is defined as a `const enum`, here is a sample:

```
export const enum SyntaxKind {
    Unknown,
    EndOfFileToken,
    SingleLineCommentTrivia,
    // ... LOTS more
```

It's a `const enum` (a concept we covered previously) so that it gets *inlined* (e.g. `ts.SyntaxKind.EndOfFileToken` becomes `1`) and we don't get a dereferencing cost when working with AST. However the compiler is compiled with `--preserveConstEnums` compiler flag so that the enum *is still available at runtime*. So in JavaScript you can use `ts.SyntaxKind.EndOfFileToken` if you want. Additionally you can convert these enum members to display strings using the following function:

```
export function syntaxKindToName(kind: ts.SyntaxKind) {
    return (<any>ts).SyntaxKind[kind];
}
```

Trivia

Trivia (called that because its `trivial`) represent the parts of the source text that are largely insignificant for normal understanding of the code, such as whitespace, comments, and even conflict markers. Trivia is *not stored* in the AST (to keep it lightweight). However it can be fetched *on demand* using a few `ts.` APIs. Before we show them you need to understand

Trivia Ownership

In General:

- A token owns any trivia after it on the *same* line *upto* the next token.
- Any comment *after that line* is associated with the following token.

For leading and ending comments in a file:

- The first token in the source file gets all the initial trivia
- The last sequence of trivia in the file is tacked onto the end-of-file token, which otherwise has zero width.

The first token in the source file gets all the initial trivia, and the last sequence of trivia in the file is tacked onto the end-of-file token, which otherwise has zero width.

Trivia APIs

For most basic uses, comments are the "interesting" trivia. The comments that belong to a Node which can be fetched through the following functions:

Function	Description
`ts.getLeadingCommentRanges`	Given the source text and position within that text, returns ranges of comments between the first line break following the given position and the token itself (probably most useful with `ts.Node.getFullStart`).
`ts.getTrailingCommentRanges`	Given the source text and position within that text, returns ranges of comments until the first line break following the given position (probably most useful with `ts.Node.getEnd`).

As an example, imagine this portion of a source file:

```
debugger;/*hello*/
    //bye
  /*hi*/    function
```

`getLeadingCommentRanges` for the `function` will only return the last 2 comments `//bye` and `/*hi*/`.

Appropriately, calling `getTrailingCommentRanges` on the end of the debugger statement will extract the `/*hello*/` comment.

Token Start/Full Start

Nodes have what is called a "token start" and a "full start".

- Token Start: the more natural version, which is the position in file where the text of a token begins
- Full Start: the point at which the scanner began scanning since the last significant token

AST nodes have an API for `getStart` and `getFullStart`. In the following example:

```
debugger;/*hello*/
    //bye
  /*hi*/    function
```

for `function` the token start is at `function` whereas *full* start is at `/*hello*/`. Note that full start even includes the trivia that would otherwise be owned by the previous node.

Scanner

The sourcecode for the TypeScript scanner is located entirely in `scanner.ts`. Scanner is *controlled* internally by the `Parser` to convert the source code to an AST. Here is what the desired outcome is.

```
SourceCode ~~ scanner ~~> Token Stream ~~ parser ~~> AST
```

Usage by Parser

There is a *singleton* `scanner` created in `parser.ts` to avoid the cost of creating scanners over and over again. This scanner is then *primed* by the parser on demand using the `initializeState` function.

Here is a *simplied* version of the actual code in the parser that you can run demonstrating this concept:

`code/compiler/scanner/runScanner.ts`

```
import * as ts from "ntypescript";

// TypeScript has a singelton scanner
const scanner = ts.createScanner(ts.ScriptTarget.Latest, /*skipTrivia*/ true);

// That is initialized using a function `initializeState` similar to
function initializeState(text: string) {
    scanner.setText(text);
    scanner.setOnError((message: ts.DiagnosticMessage, length: number) => {
        console.error(message);
    });
    scanner.setScriptTarget(ts.ScriptTarget.ES5);
    scanner.setLanguageVariant(ts.LanguageVariant.Standard);
}

// Sample usage
initializeState(`
var foo = 123;
`.trim());

// Start the scanning
var token = scanner.scan();
while (token != ts.SyntaxKind.EndOfFileToken) {
    console.log(ts.syntaxKindToName(token));
    token = scanner.scan();
}
```

This will print out the following :

```
VarKeyword
Identifier
FirstAssignment
FirstLiteralToken
SemicolonToken
```

Scanner State

After you call `scan` the scanner updates its local state (position in the scan, current token details etc). The scanner provides a bunch of utility functions to get the current scanner state. In the below sample we create a scanner and then use it to identify the tokens as well as their positions in the code.

`code/compiler/scanner/runScannerWithPosition.ts`

```
// Sample usage
initializeState(`
var foo = 123;
`.trim());

// Start the scanning
var token = scanner.scan();
while (token != ts.SyntaxKind.EndOfFileToken) {
    let currentToken = ts.syntaxKindToName(token);
    let tokenStart = scanner.getStartPos();
    token = scanner.scan();
    let tokenEnd = scanner.getStartPos();
    console.log(currentToken, tokenStart, tokenEnd);
}
```

This will print out the following:

```
VarKeyword 0 3
Identifier 3 7
FirstAssignment 7 9
FirstLiteralToken 9 13
SemicolonToken 13 14
```

Standalone scanner

Even though the typescript parser has a singleton scanner you can create a standalone scanner using `createScanner` and use its `setText` / `setTextPos` to scan at different points in a file for your amusement.

Parser

The sourcecode for the TypeScript parser is located entirely in `parser.ts`. Scanner is *controlled* internally by the `Parser` to convert the source code to an AST. Here is a review of what the desired outcome is.

```
SourceCode ~~ scanner ~~> Token Stream ~~ parser ~~> AST
```

The parser is implemented as a singleton (similar reasons to `scanner`, don't want to recreate it if we can reinit it). It is actually implemented as `namespace Parser` which contains *state* variables for the Parser as well as a singleton `scanner`. As mentioned before it contains a `const scanner`. The parser functions manage this scanner.

Usage by program

Parser is driven indirectly by Program (indirectly as its actually by `CompilerHost` which we mentioned previously). Basically this is the simplified call stack:

```
Program ->
    CompilerHost.getSourceFile ->
        (global function parser.ts).createSourceFile ->
            Parser.parseSourceFile
```

The `parseSourceFile` not only primes the state for the Parser but also primes the state for the `scanner` by calling `initializeState`. It then goes on to parse the source file using `parseSourceFileWorker`.

Sample Usage

Before we dig too deep into the parser internals, here is a sample code that uses the TypeScript's parser to get the AST of a source file (using `ts.createSourceFile`), and then print it.

`code/compiler/parser/runParser.ts`

```
import * as ts from "ntypescript";

function printAllChildren(node: ts.Node, depth = 0) {
    console.log(new Array(depth + 1).join('----'), ts.syntaxKindToName(node.kind), node.p
    depth++;
    node.getChildren().forEach(c=> printAllChildren(c, depth));
}

var sourceCode = `
var foo = 123;
`.trim();

var sourceFile = ts.createSourceFile('foo.ts', sourceCode, ts.ScriptTarget.ES5, true);
printAllChildren(sourceFile);
```

This will print out the following:

```
SourceFile 0 14
---- SyntaxList 0 14
-------- VariableStatement 0 14
------------ VariableDeclarationList 0 13
---------------- VarKeyword 0 3
---------------- SyntaxList 3 13
-------------------- VariableDeclaration 3 13
------------------------ Identifier 3 7
------------------------ FirstAssignment 7 9
------------------------ FirstLiteralToken 9 13
------------ SemicolonToken 13 14
---- EndOfFileToken 14 14
```

This looks like a (very right sided) tree if you tilt your head to the left.

Parser Functions

As mentioned `parseSourceFile` sets up the initial state and passes the work onto `parseSourceFileWorker` function.

`parseSourceFileWorker`

Starts by creating a `SourceFile` AST node. Then it goes into parsing source code starting from the `parseStatements` function. Once that returns, it then completes the `SourceFile` node with additional information such as its `nodeCount` , `identifierCount` and such.

`parseStatements`

One of the most significant `parseFoo` style functions (a concept we cover next). It switches by the the current `token` returned from the scanner. E.g. if the current token is a `SemicolonToken` it will call out to `parseEmptyStatement` to create an AST node for an empty statement.

Node creation

The parser has a bunch of `parserFoo` functions with bodies that create `Foo` nodes. These are generally called (from other parser functions) at a time where a `Foo` node is expected. A typical sample of this process is the `parseEmptyStatement()` function which is used to parse out empty statements like `;;;;;;` . Here is the function in its entirety

```
function parseEmptyStatement(): Statement {
    let node = <Statement>createNode(SyntaxKind.EmptyStatement);
    parseExpected(SyntaxKind.SemicolonToken);
    return finishNode(node);
}
```

It shows three critical functions `createNode` , `parseExpected` and `finishNode` .

`createNode`

The parser's `createNode` function `function createNode(kind: SyntaxKind, pos?: number): Node` is responsible for creating a Node, setting up its `SyntaxKind` as passed in, and set the initial position if passed in (or use the position from the current scanner state).

`parseExpected`

The parser's `parseExpected` function `function parseExpected(kind: SyntaxKind, diagnosticMessage?: DiagnosticMessage): boolean` will check that the current token in the parser state matches the desired `SyntaxKind`. If not it will either report the `diagnosticMessage` sent in or create a generic one of the form `foo expected`. It internally uses the `parseErrorAtPosition` function (which uses the scanning positions) to give good error reporting.

`finishNode`

The parser's `finishNode` function `function finishNode<T extends Node>(node: T, end?: number): T` sets up the `end` position for the node and additional useful stuff like the `parserContextFlags` it was parsed under as well as if there were any errors before parsing this node (if there were then we cannot reuse this AST node in incremental parsing).

Binder

Most JavaScript transpilers out there are simpler than TypeScript in that they provide little in the way of code analysis. The typical JavaScript transpilers only have the following flow:

```
SourceCode ~~Scanner~~> Tokens ~~Parser~~> AST ~~Emitter~~> JavaScript
```

While the above architecture is true as a simplified understand of TypeScript js generation, a key feature of TypeScript is its *Semantic* system. In order to assist type checking (performed by `checker`), the `binder` (in `binder.ts`) is used to connect the various parts of the source code into a coherent type system that can then be used by the `checker`. The main responsibility of the binder is to create the *Symbols*.

Symbol

Symbols connect declaration nodes in the AST to other declarations contributing to the same entity. Symbols are the basic building block of the Semantic system. The symbol constructor is defined in `core.ts` (and `binder` actually uses the `objectAllocator.getSymbolConstructor` to get its hands on it). Here is the contructor:

```
function Symbol(flags: SymbolFlags, name: string) {
    this.flags = flags;
    this.name = name;
    this.declarations = undefined;
}
```

`SymbolFlags` is a flag enum and is really used to identify additional classifications of the symbol (e.g the scope of a variable flags `FunctionScopedVariable` or `BlockScopedVariable` or others)

Usage by Checker

The `binder` is actually used internally by the type `checker` which in turn is used by the `program`. The simplified call stack looks like:

```
program.getTypeChecker ->
    ts.createTypeChecker (in checker)->
        initializeTypeChecker (in checker) ->
            for each SourceFile `ts.bindSourceFile` (in binder)
            // followed by
            for each SourceFile `ts.mergeSymbolTable` (in checker)
```

The unit of work for the binder is a SourceFile. The `binder.ts` is driven by `checker.ts`.

Binder function

Two critical binder functions are `bindSourceFile` and `mergeSymbolTable`. We will take a look at these next.

`bindSourceFile`

Basically checks if the `file.locals` is defined, if not it hands over to (a local function) `bind`.

Note: `locals` is defined on `Node` and is of type `SymbolTable`. Note that `SourceFile` is also a `Node` (in fact a root node in the AST).

TIP: local functions are used heavily within the TypeScript compiler. A local function very likely uses variables from the parent function (captured by closure). In the case of `bind` (a local function within `bindSourceFile`) it (or function it calls) will setup the `symbolCount` and `classifiableNames` among others, that are then stored on the returned `SourceFile`.

`bind`

Bind takes any `Node` (not just `SourceFile`). First thing it does is assign the `node.parent` (if `parent` variable has been setup ... which again is something the binder does during its processing within the `bindChildren` function), then hands off to `bindWorker` which does the *heavy* lifting. Finally it calls `bindChildren` (a function that simply stores the binder state e.g. current `parent` within its function local vars, then calls `bind` on each child, and then restores the binder state). Now lets look at `bindWorker` which is the more interesting function.

`bindWorker`

This function switches on `node.kind` (of type `SyntaxKind`) and delegates work to the appropriate `bindFoo` function (also defined within `binder.ts`). For example if the `node` is a `SourceFile` it calls (eventually and only if its an external file module) `bindAnonymousDeclaration`

`bindFoo` functions

There are few pattern common to `bindFoo` functions as well as some utility functions that these use. One function that is almost always used is the `createSymbol` function. It is presented in its entirety below:

```
function createSymbol(flags: SymbolFlags, name: string): Symbol {
    symbolCount++;
    return new Symbol(flags, name);
}
```

As you can see it is simply keeping the `symbolCount` (a local to `bindSourceFile`) up to date and creating the symbol with the specified parameters.

Symbols and Declarations

Linking between a `node` and a `symbol` is performed by a few functions. One function that is used to bind the `SourceFile` node to the source file Symbol (in case of an external module) is the `addDeclarationToSymbol` function

Note : the `Symbol` for an external module source file is setup as `flags : SymbolFlags.ValueModule` and `name: '"' + removeFileExtension(file.fileName) + '"'`).

```
function addDeclarationToSymbol(symbol: Symbol, node: Declaration, symbolFlags: SymbolFla
    symbol.flags |= symbolFlags;

    node.symbol = symbol;

    if (!symbol.declarations) {
        symbol.declarations = [];
    }
    symbol.declarations.push(node);

    if (symbolFlags & SymbolFlags.HasExports && !symbol.exports) {
        symbol.exports = {};
    }

    if (symbolFlags & SymbolFlags.HasMembers && !symbol.members) {
        symbol.members = {};
    }

    if (symbolFlags & SymbolFlags.Value && !symbol.valueDeclaration) {
        symbol.valueDeclaration = node;
    }
}
```

The important linking portions:

- creates a link to the Symbol from the AST node (`node.symbol`).
- add the node as *one of* the declarations of the Symbol (`symbol.declarations`).

Declaration

Declaration is just a `node` with an optional name. In `types.ts`

```
interface Declaration extends Node {
    _declarationBrand: any;
    name?: DeclarationName;
}
```

Container

An AST node can be a container. This determines the kinds of `SymbolTables` the Node and associated Symbol will have. Container is an abstract concept (i.e. has no associated data structure). The concept is driven by a few things, one being the `ContainerFlags` enum. The function `getContainerFlags` (in `binder.ts`) drives this flag and is presented below:

```
function getContainerFlags(node: Node): ContainerFlags {
    switch (node.kind) {
        case SyntaxKind.ClassExpression:
        case SyntaxKind.ClassDeclaration:
        case SyntaxKind.InterfaceDeclaration:
        case SyntaxKind.EnumDeclaration:
        case SyntaxKind.TypeLiteral:
        case SyntaxKind.ObjectLiteralExpression:
            return ContainerFlags.IsContainer;

        case SyntaxKind.CallSignature:
        case SyntaxKind.ConstructSignature:
        case SyntaxKind.IndexSignature:
        case SyntaxKind.MethodDeclaration:
        case SyntaxKind.MethodSignature:
        case SyntaxKind.FunctionDeclaration:
        case SyntaxKind.Constructor:
        case SyntaxKind.GetAccessor:
        case SyntaxKind.SetAccessor:
        case SyntaxKind.FunctionType:
        case SyntaxKind.ConstructorType:
        case SyntaxKind.FunctionExpression:
        case SyntaxKind.ArrowFunction:
        case SyntaxKind.ModuleDeclaration:
        case SyntaxKind.SourceFile:
        case SyntaxKind.TypeAliasDeclaration:
            return ContainerFlags.IsContainerWithLocals;

        case SyntaxKind.CatchClause:
        case SyntaxKind.ForStatement:
        case SyntaxKind.ForInStatement:
        case SyntaxKind.ForOfStatement:
        case SyntaxKind.CaseBlock:
            return ContainerFlags.IsBlockScopedContainer;

        case SyntaxKind.Block:
            // do not treat blocks directly inside a function as a block-scoped-container
            // Locals that reside in this block should go to the function locals. Othewis
            // would not appear to be a redeclaration of a block scoped local in the foll
            // example:
            //
            //      function foo() {
```

```
            //          var x;
            //          let x;
            //      }
            //
            // If we placed 'var x' into the function locals and 'let x' into the locals
            // the block, then there would be no collision.
            //
            // By not creating a new block-scoped-container here, we ensure that both 'va
            // and 'let x' go into the Function-container's locals, and we do get a colli
            // conflict.
            return isFunctionLike(node.parent) ? ContainerFlags.None : ContainerFlags.IsB
    }

    return ContainerFlags.None;
}
```

It is *only* invoked from the binder's `bindChildren` function which sets up a node as a `container` and/or a `blockScopedContainer` depending upon the evaluation of the `getContainerFlags` function. The function `bindChildren` is presented below:

```
// All container nodes are kept on a linked list in declaration order. This list is used
// the getLocalNameOfContainer function in the type checker to validate that the local na
// used for a container is unique.
function bindChildren(node: Node) {
    // Before we recurse into a node's chilren, we first save the existing parent, contai
    // and block-container.  Then after we pop out of processing the children, we restore
    // these saved values.
    let saveParent = parent;
    let saveContainer = container;
    let savedBlockScopeContainer = blockScopeContainer;

    // This node will now be set as the parent of all of its children as we recurse into
    parent = node;

    // Depending on what kind of node this is, we may have to adjust the current containe
    // and block-container.   If the current node is a container, then it is automaticall
    // considered the current block-container as well.  Also, for containers that we know
    // may contain locals, we proactively initialize the .locals field. We do this becaus
    // it's highly likely that the .locals will be needed to place some child in (for exa
    // a parameter, or variable declaration).
    //
    // However, we do not proactively create the .locals for block-containers because it'
    // totally normal and common for block-containers to never actually have a block-scop
    // variable in them.  We don't want to end up allocating an object for every 'block'
    // run into when most of them won't be necessary.
    //
    // Finally, if this is a block-container, then we clear out any existing .locals obje
    // it may contain within it.  This happens in incremental scenarios.  Because we can
    // reusing a node from a previous compilation, that node may have had 'locals' create
    // for it.  We must clear this so we don't accidently move any stale data forward fro
```

```
        // a previous compilation.
        let containerFlags = getContainerFlags(node);
        if (containerFlags & ContainerFlags.IsContainer) {
            container = blockScopeContainer = node;

            if (containerFlags & ContainerFlags.HasLocals) {
                container.locals = {};
            }

            addToContainerChain(container);
        }

        else if (containerFlags & ContainerFlags.IsBlockScopedContainer) {
            blockScopeContainer = node;
            blockScopeContainer.locals = undefined;
        }

        forEachChild(node, bind);

        container = saveContainer;
        parent = saveParent;
        blockScopeContainer = savedBlockScopeContainer;
    }
```

As you might recall from section on binder functions : `bindChildren` is called from the `bind` function. So we have the recursive bindig setup : `bind` calls `bindChildren` calls `bind` for each child.

SymbolTable

Its implemented as a simple HashMap. Here is the interface (`types.ts`):

```
interface SymbolTable {
    [index: string]: Symbol;
}
```

SymbolTables as initialized by binding. There are a few SymbolTables used by the compiler.

On `Node` :

```
locals?: SymbolTable;                   // Locals associated with node
```

On `Symbol` :

```
members?: SymbolTable;                  // Class, interface or literal instance members
exports?: SymbolTable;                  // Module exports
```

Note: We saw `locals` getting initialized (to `{}`) by `bindChildren` based on `ContainerFlags` .

SymbolTable population

SymbolTable are populated with `Symbols` primarily by a call to `declareSymbol` . This function is presented below in entirety:

```
/**
 * Declares a Symbol for the node and adds it to symbols. Reports errors for conflicting
 * @param symbolTable - The symbol table which node will be added to.
 * @param parent - node's parent declaration.
 * @param node - The declaration to be added to the symbol table
 * @param includes - The SymbolFlags that node has in addition to its declaration type (e
 * @param excludes - The flags which node cannot be declared alongside in a symbol table.
 */
function declareSymbol(symbolTable: SymbolTable, parent: Symbol, node: Declaration, inclu
    Debug.assert(!hasDynamicName(node));

    // The exported symbol for an export default function/class node is always named "def
    let name = node.flags & NodeFlags.Default && parent ? "default" : getDeclarationName(

    let symbol: Symbol;
    if (name !== undefined) {
```

```
        // Check and see if the symbol table already has a symbol with this name.  If not
        // create a new symbol with this name and add it to the table.  Note that we don'
        // give the new symbol any flags *yet*.  This ensures that it will not conflict
        // with the 'excludes' flags we pass in.
        //
        // If we do get an existing symbol, see if it conflicts with the new symbol we're
        // creating.  For example, a 'var' symbol and a 'class' symbol will conflict with
        // the same symbol table.  If we have a conflict, report the issue on each
        // declaration we have for this symbol, and then create a new symbol for this
        // declaration.
        //
        // If we created a new symbol, either because we didn't have a symbol with this n
        // in the symbol table, or we conflicted with an existing symbol, then just add t
        // node as the sole declaration of the new symbol.
        //
        // Otherwise, we'll be merging into a compatible existing symbol (for example whe
        // you have multiple 'vars' with the same name in the same container).  In this c
        // just add this node into the declarations list of the symbol.
        symbol = hasProperty(symbolTable, name)
            ? symbolTable[name]
            : (symbolTable[name] = createSymbol(SymbolFlags.None, name));

        if (name && (includes & SymbolFlags.Classifiable)) {
            classifiableNames[name] = name;
        }

        if (symbol.flags & excludes) {
            if (node.name) {
                node.name.parent = node;
            }

            // Report errors every position with duplicate declaration
            // Report errors on previous encountered declarations
            let message = symbol.flags & SymbolFlags.BlockScopedVariable
                ? Diagnostics.Cannot_redeclare_block_scoped_variable_0
                : Diagnostics.Duplicate_identifier_0;
            forEach(symbol.declarations, declaration => {
                file.bindDiagnostics.push(createDiagnosticForNode(declaration.name || dec
            });
            file.bindDiagnostics.push(createDiagnosticForNode(node.name || node, message,

            symbol = createSymbol(SymbolFlags.None, name);
        }
    }
    else {
        symbol = createSymbol(SymbolFlags.None, "__missing");
    }

    addDeclarationToSymbol(symbol, node, includes);
    symbol.parent = parent;

    return symbol;
}
```

Which SymbolTable gets populated is driven by the first argument to this function. e.g. when adding a declaration to a *container* of kind `SyntaxKind.ClassDeclaration` or `SytanxKind.ClassExpression` the function `declareClassMember` will get called which has the following code:

```
function declareClassMember(node: Declaration, symbolFlags: SymbolFlags, symbolExcludes:
    return node.flags & NodeFlags.Static
        ? declareSymbol(container.symbol.exports, container.symbol, node, symbolFlags, sy
        : declareSymbol(container.symbol.members, container.symbol, node, symbolFlags, sy
}
```

Binder Error Reporting

Binding errors are added to the sourceFile's list of `bindDiagnostics` .

An example error detected during binding is the use of `eval` or `arguments` as a variable name in `use strict` scenario. The relevant code is presented in its entirety below (`checkStrictModeEvalOrArguments` is called from multiple places, call stacks originating from `bindWorker` which calls different functions for different node `SyntaxKind`):

```
function checkStrictModeEvalOrArguments(contextNode: Node, name: Node) {
    if (name && name.kind === SyntaxKind.Identifier) {
        let identifier = <Identifier>name;
        if (isEvalOrArgumentsIdentifier(identifier)) {
            // We check first if the name is inside class declaration or class expression
            // otherwise report generic error message.
            let span = getErrorSpanForNode(file, name);
            file.bindDiagnostics.push(createFileDiagnostic(file, span.start, span.length,
                getStrictModeEvalOrArgumentsMessage(contextNode), identifier.text));
        }
    }
}

function isEvalOrArgumentsIdentifier(node: Node): boolean {
    return node.kind === SyntaxKind.Identifier &&
        ((<Identifier>node).text === "eval" || (<Identifier>node).text === "arguments");
}

function getStrictModeEvalOrArgumentsMessage(node: Node) {
    // Provide specialized messages to help the user understand why we think they're in
    // strict mode.
    if (getContainingClass(node)) {
        return Diagnostics.Invalid_use_of_0_Class_definitions_are_automatically_in_strict
    }

    if (file.externalModuleIndicator) {
        return Diagnostics.Invalid_use_of_0_Modules_are_automatically_in_strict_mode;
    }

    return Diagnostics.Invalid_use_of_0_in_strict_mode;
}
```

Checker

Like we mentioned before *checker* is the thing that makes TypeScript uniquely more powerful than *just another JavaScript transpiler*. The checker is located in `checker.ts` and at this moment it is 15k+ lines of code (largest part of the compiler).

Usage by Program

The `checker` is initialized by `program`. The following is a sampling of the call stack (we showed the same one when looking at `binder`):

```
program.getTypeChecker ->
    ts.createTypeChecker (in checker)->
        initializeTypeChecker (in checker) ->
            for each SourceFile `ts.bindSourceFile` (in binder)
            // followed by
            for each SourceFile `ts.mergeSymbolTable` (in checker)
```

Association with Emitter

True type checking happens once a call is made to `getDiagnostics`. This function is called e.g. once a request is made to `Program.emit`, in which case the checker returns an `EmitResolver` (progarm calls the checkers `getEmitResolver` function) which is just a set of functions local to `createTypeChecker`. We will mention this again when we look at the emitter.

Here is the call stack right down to `checkSourceFile` (a function local to `createTypeChecker`).

```
program.emit ->
    emitWorker (program local) ->
        createTypeChecker.getEmitResolver ->
            // First call the following functions local to createTypeChecker
            call getDiagnostics ->
                getDiagnosticsWorker ->
                    checkSourceFile

            // then
            return resolver
            (already initialized in createTypeChecker using a call to local createResolve
```

Global Namespace Merging

Within `initializeTypeChecker` the following code exists :

```
// Initialize global symbol table
forEach(host.getSourceFiles(), file => {
    if (!isExternalModule(file)) {
        mergeSymbolTable(globals, file.locals);
    }
});
```

Which basically merges all the `global` symbols into the `let globals: SymbolTable = {};` (in `createTypeChecker`) SymbolTable. `mergeSymbolTable` primarily calls `mergeSymbol` .

Checker error reporting

The checker uses the local `error` function to report errors. Here is the function:

```
function error(location: Node, message: DiagnosticMessage, arg0?: any, arg1?: any, arg2?:
    let diagnostic = location
        ? createDiagnosticForNode(location, message, arg0, arg1, arg2)
        : createCompilerDiagnostic(message, arg0, arg1, arg2);
    diagnostics.add(diagnostic);
}
```

Emitter

There are two `emitters` provided with the TypeScript compiler:

- `emitter.ts` : this is the emitter you are most likely to be interested in. Its the TS -> JavaScript emitter.
- `declarationEmitter.ts` : this is the emitter used to create a *declaration file* (a `.d.ts`) for a *TypeScript source file* (a `.ts` file).

We will look at `emitter.ts` in this section.

Usage by `program`

Program provides an `emit` function. This function primarily delegates to `emitFiles` function in `emitter.ts` . Here is the call stack:

```
Program.emit ->
    `emitWorker` (local in program.ts createProgram) ->
        `emitFiles` (function in emitter.ts)
```

One thing that the `emitWorker` provides to the emitter (via an argument to `emitFiles`) is an `EmitResolver` . `EmitResolver` is provided by the program's TypeChecker, basically it a subset of *local* functions from `createChecker` .

`emitFiles`

Defined in `emitter.ts` here is the function signature:

```
// targetSourceFile is when users only want one file in entire project to be emitted. Thi
export function emitFiles(resolver: EmitResolver, host: EmitHost, targetSourceFile?: Sour
```

`EmitHost` is a just a simplified (as in narrowed down) version of `CompilerHost` (and is at runtime actually a CompilerHost for many use cases).

The most interesting call stack from `emitFiles` is the following:

```
emitFiles ->
    emitFile(jsFilePath, targetSourceFile) ->
        emitJavaScript(jsFilePath, targetSourceFile);
```

`emitJavaScript`

There is a lot of good comments in this function so we present it below :

```
function emitJavaScript(jsFilePath: string, root?: SourceFile) {
    let writer = createTextWriter(newLine);
    let write = writer.write;
    let writeTextOfNode = writer.writeTextOfNode;
    let writeLine = writer.writeLine;
    let increaseIndent = writer.increaseIndent;
    let decreaseIndent = writer.decreaseIndent;

    let currentSourceFile: SourceFile;
    // name of an exporter function if file is a System external module
    // System.register([...], function (<exporter>) {...})
    // exporting in System modules looks like:
    // export var x; ... x = 1
    // =>
    // var x;... exporter("x", x = 1)
    let exportFunctionForFile: string;

    let generatedNameSet: Map<string> = {};
    let nodeToGeneratedName: string[] = [];
    let computedPropertyNamesToGeneratedNames: string[];

    let extendsEmitted = false;
    let decorateEmitted = false;
    let paramEmitted = false;
    let awaiterEmitted = false;
    let tempFlags = 0;
```

```
    let tempVariables: Identifier[];
    let tempParameters: Identifier[];
    let externalImports: (ImportDeclaration | ImportEqualsDeclaration | ExportDeclaration
    let exportSpecifiers: Map<ExportSpecifier[]>;
    let exportEquals: ExportAssignment;
    let hasExportStars: boolean;

    /** Write emitted output to disk */
    let writeEmittedFiles = writeJavaScriptFile;

    let detachedCommentsInfo: { nodePos: number; detachedCommentEndPos: number }[];

    let writeComment = writeCommentRange;

    /** Emit a node */
    let emit = emitNodeWithoutSourceMap;

    /** Called just before starting emit of a node */
    let emitStart = function (node: Node) { };

    /** Called once the emit of the node is done */
    let emitEnd = function (node: Node) { };

    /** Emit the text for the given token that comes after startPos
      * This by default writes the text provided with the given tokenKind
      * but if optional emitFn callback is provided the text is emitted using the callbac
      * @param tokenKind the kind of the token to search and emit
      * @param startPos the position in the source to start searching for the token
      * @param emitFn if given will be invoked to emit the text instead of actual token e
    let emitToken = emitTokenText;

    /** Called to before starting the lexical scopes as in function/class in the emitted
      * @param scopeDeclaration node that starts the lexical scope
      * @param scopeName Optional name of this scope instead of deducing one from the dec
    let scopeEmitStart = function(scopeDeclaration: Node, scopeName?: string) { };

    /** Called after coming out of the scope */
    let scopeEmitEnd = function() { };

    /** Sourcemap data that will get encoded */
    let sourceMapData: SourceMapData;

    if (compilerOptions.sourceMap || compilerOptions.inlineSourceMap) {
        initializeEmitterWithSourceMaps();
    }

    if (root) {
        // Do not call emit directly. It does not set the currentSourceFile.
        emitSourceFile(root);
    }
    else {
        forEach(host.getSourceFiles(), sourceFile => {
            if (!isExternalModuleOrDeclarationFile(sourceFile)) {
```

```
                emitSourceFile(sourceFile);
            }
        });
    }

    writeLine();
    writeEmittedFiles(writer.getText(), /*writeByteOrderMark*/ compilerOptions.emitBOM);
    return;

    /// BUNCH OF LOCAL FUNCTIONS
}
```

Basically it sets up a bunch of locals (these function form the *bulk* of `emitter.ts`) and then hands off to a local function `emitSourceFile` which kicks off the emit. The `emitSourceFile` function just sets up the `currentSourceFile` and in turn hands off to a local `emit` function.

```
function emitSourceFile(sourceFile: SourceFile): void {
    currentSourceFile = sourceFile;
    exportFunctionForFile = undefined;
    emit(sourceFile);
}
```

The `emit` function handles *comment* emit + *actual JavaScript* emit. The *actual JavaScript* emit is the job of `emitJavaScriptWorker` function.

`emitJavaScriptWorker`

The complete function:

```
function emitJavaScriptWorker(node: Node) {
    // Check if the node can be emitted regardless of the ScriptTarget
    switch (node.kind) {
        case SyntaxKind.Identifier:
            return emitIdentifier(<Identifier>node);
        case SyntaxKind.Parameter:
            return emitParameter(<ParameterDeclaration>node);
        case SyntaxKind.MethodDeclaration:
        case SyntaxKind.MethodSignature:
            return emitMethod(<MethodDeclaration>node);
        case SyntaxKind.GetAccessor:
        case SyntaxKind.SetAccessor:
            return emitAccessor(<AccessorDeclaration>node);
        case SyntaxKind.ThisKeyword:
            return emitThis(node);
        case SyntaxKind.SuperKeyword:
            return emitSuper(node);
        case SyntaxKind.NullKeyword:
```

```
            return write("null");
        case SyntaxKind.TrueKeyword:
            return write("true");
        case SyntaxKind.FalseKeyword:
            return write("false");
        case SyntaxKind.NumericLiteral:
        case SyntaxKind.StringLiteral:
        case SyntaxKind.RegularExpressionLiteral:
        case SyntaxKind.NoSubstitutionTemplateLiteral:
        case SyntaxKind.TemplateHead:
        case SyntaxKind.TemplateMiddle:
        case SyntaxKind.TemplateTail:
            return emitLiteral(<LiteralExpression>node);
        case SyntaxKind.TemplateExpression:
            return emitTemplateExpression(<TemplateExpression>node);
        case SyntaxKind.TemplateSpan:
            return emitTemplateSpan(<TemplateSpan>node);
        case SyntaxKind.JsxElement:
        case SyntaxKind.JsxSelfClosingElement:
            return emitJsxElement(<JsxElement|JsxSelfClosingElement>node);
        case SyntaxKind.JsxText:
            return emitJsxText(<JsxText>node);
        case SyntaxKind.JsxExpression:
            return emitJsxExpression(<JsxExpression>node);
        case SyntaxKind.QualifiedName:
            return emitQualifiedName(<QualifiedName>node);
        case SyntaxKind.ObjectBindingPattern:
            return emitObjectBindingPattern(<BindingPattern>node);
        case SyntaxKind.ArrayBindingPattern:
            return emitArrayBindingPattern(<BindingPattern>node);
        case SyntaxKind.BindingElement:
            return emitBindingElement(<BindingElement>node);
        case SyntaxKind.ArrayLiteralExpression:
            return emitArrayLiteral(<ArrayLiteralExpression>node);
        case SyntaxKind.ObjectLiteralExpression:
            return emitObjectLiteral(<ObjectLiteralExpression>node);
        case SyntaxKind.PropertyAssignment:
            return emitPropertyAssignment(<PropertyDeclaration>node);
        case SyntaxKind.ShorthandPropertyAssignment:
            return emitShorthandPropertyAssignment(<ShorthandPropertyAssignment>node);
        case SyntaxKind.ComputedPropertyName:
            return emitComputedPropertyName(<ComputedPropertyName>node);
        case SyntaxKind.PropertyAccessExpression:
            return emitPropertyAccess(<PropertyAccessExpression>node);
        case SyntaxKind.ElementAccessExpression:
            return emitIndexedAccess(<ElementAccessExpression>node);
        case SyntaxKind.CallExpression:
            return emitCallExpression(<CallExpression>node);
        case SyntaxKind.NewExpression:
            return emitNewExpression(<NewExpression>node);
        case SyntaxKind.TaggedTemplateExpression:
            return emitTaggedTemplateExpression(<TaggedTemplateExpression>node);
        case SyntaxKind.TypeAssertionExpression:
```

```
            return emit((<TypeAssertion>node).expression);
        case SyntaxKind.AsExpression:
            return emit((<AsExpression>node).expression);
        case SyntaxKind.ParenthesizedExpression:
            return emitParenExpression(<ParenthesizedExpression>node);
        case SyntaxKind.FunctionDeclaration:
        case SyntaxKind.FunctionExpression:
        case SyntaxKind.ArrowFunction:
            return emitFunctionDeclaration(<FunctionLikeDeclaration>node);
        case SyntaxKind.DeleteExpression:
            return emitDeleteExpression(<DeleteExpression>node);
        case SyntaxKind.TypeOfExpression:
            return emitTypeOfExpression(<TypeOfExpression>node);
        case SyntaxKind.VoidExpression:
            return emitVoidExpression(<VoidExpression>node);
        case SyntaxKind.AwaitExpression:
            return emitAwaitExpression(<AwaitExpression>node);
        case SyntaxKind.PrefixUnaryExpression:
            return emitPrefixUnaryExpression(<PrefixUnaryExpression>node);
        case SyntaxKind.PostfixUnaryExpression:
            return emitPostfixUnaryExpression(<PostfixUnaryExpression>node);
        case SyntaxKind.BinaryExpression:
            return emitBinaryExpression(<BinaryExpression>node);
        case SyntaxKind.ConditionalExpression:
            return emitConditionalExpression(<ConditionalExpression>node);
        case SyntaxKind.SpreadElementExpression:
            return emitSpreadElementExpression(<SpreadElementExpression>node);
        case SyntaxKind.YieldExpression:
            return emitYieldExpression(<YieldExpression>node);
        case SyntaxKind.OmittedExpression:
            return;
        case SyntaxKind.Block:
        case SyntaxKind.ModuleBlock:
            return emitBlock(<Block>node);
        case SyntaxKind.VariableStatement:
            return emitVariableStatement(<VariableStatement>node);
        case SyntaxKind.EmptyStatement:
            return write(";");
        case SyntaxKind.ExpressionStatement:
            return emitExpressionStatement(<ExpressionStatement>node);
        case SyntaxKind.IfStatement:
            return emitIfStatement(<IfStatement>node);
        case SyntaxKind.DoStatement:
            return emitDoStatement(<DoStatement>node);
        case SyntaxKind.WhileStatement:
            return emitWhileStatement(<WhileStatement>node);
        case SyntaxKind.ForStatement:
            return emitForStatement(<ForStatement>node);
        case SyntaxKind.ForOfStatement:
        case SyntaxKind.ForInStatement:
            return emitForInOrForOfStatement(<ForInStatement>node);
        case SyntaxKind.ContinueStatement:
        case SyntaxKind.BreakStatement:
```

```
            return emitBreakOrContinueStatement(<BreakOrContinueStatement>node);
        case SyntaxKind.ReturnStatement:
            return emitReturnStatement(<ReturnStatement>node);
        case SyntaxKind.WithStatement:
            return emitWithStatement(<WithStatement>node);
        case SyntaxKind.SwitchStatement:
            return emitSwitchStatement(<SwitchStatement>node);
        case SyntaxKind.CaseClause:
        case SyntaxKind.DefaultClause:
            return emitCaseOrDefaultClause(<CaseOrDefaultClause>node);
        case SyntaxKind.LabeledStatement:
            return emitLabelledStatement(<LabeledStatement>node);
        case SyntaxKind.ThrowStatement:
            return emitThrowStatement(<ThrowStatement>node);
        case SyntaxKind.TryStatement:
            return emitTryStatement(<TryStatement>node);
        case SyntaxKind.CatchClause:
            return emitCatchClause(<CatchClause>node);
        case SyntaxKind.DebuggerStatement:
            return emitDebuggerStatement(node);
        case SyntaxKind.VariableDeclaration:
            return emitVariableDeclaration(<VariableDeclaration>node);
        case SyntaxKind.ClassExpression:
            return emitClassExpression(<ClassExpression>node);
        case SyntaxKind.ClassDeclaration:
            return emitClassDeclaration(<ClassDeclaration>node);
        case SyntaxKind.InterfaceDeclaration:
            return emitInterfaceDeclaration(<InterfaceDeclaration>node);
        case SyntaxKind.EnumDeclaration:
            return emitEnumDeclaration(<EnumDeclaration>node);
        case SyntaxKind.EnumMember:
            return emitEnumMember(<EnumMember>node);
        case SyntaxKind.ModuleDeclaration:
            return emitModuleDeclaration(<ModuleDeclaration>node);
        case SyntaxKind.ImportDeclaration:
            return emitImportDeclaration(<ImportDeclaration>node);
        case SyntaxKind.ImportEqualsDeclaration:
            return emitImportEqualsDeclaration(<ImportEqualsDeclaration>node);
        case SyntaxKind.ExportDeclaration:
            return emitExportDeclaration(<ExportDeclaration>node);
        case SyntaxKind.ExportAssignment:
            return emitExportAssignment(<ExportAssignment>node);
        case SyntaxKind.SourceFile:
            return emitSourceFileNode(<SourceFile>node);
    }
}
```

Recursion is done by simply calling other `emitFoo` function from these functions as needed e.g. from `emitFunctionDeclaration` :

```
function emitFunctionDeclaration(node: FunctionLikeDeclaration) {
    if (nodeIsMissing(node.body)) {
        return emitOnlyPinnedOrTripleSlashComments(node);
    }

    if (node.kind !== SyntaxKind.MethodDeclaration && node.kind !== SyntaxKind.MethodSign
        // Methods will emit the comments as part of emitting method declaration
        emitLeadingComments(node);
    }

    // For targeting below es6, emit functions-like declaration including arrow function
    // When targeting ES6, emit arrow function natively in ES6 by omitting function keywo
    if (!shouldEmitAsArrowFunction(node)) {
        if (isES6ExportedDeclaration(node)) {
            write("export ");
            if (node.flags & NodeFlags.Default) {
                write("default ");
            }
        }

        write("function");
        if (languageVersion >= ScriptTarget.ES6 && node.asteriskToken) {
            write("*");
        }
        write(" ");
    }

    if (shouldEmitFunctionName(node)) {
        emitDeclarationName(node);
    }

    emitSignatureAndBody(node);
    if (languageVersion < ScriptTarget.ES6 && node.kind === SyntaxKind.FunctionDeclaratio
        emitExportMemberAssignments((<FunctionDeclaration>node).name);
    }
    if (node.kind !== SyntaxKind.MethodDeclaration && node.kind !== SyntaxKind.MethodSign
        emitTrailingComments(node);
    }
}
```

Emitter SourceMaps

We said that the bulk of the `emitter.ts` is the local function `emitJavaScript` (we showed the initialization routine of this function before). It basically sets up a bunch of locals and hits off to `emitSourceFile`. The following is a revisiting of the function, this time focusing on SourceMap stuff:

```
function emitJavaScript(jsFilePath: string, root?: SourceFile) {

    // STUFF ........... removed

    let writeComment = writeCommentRange;

    /** Write emitted output to disk */
    let writeEmittedFiles = writeJavaScriptFile;

    /** Emit a node */
    let emit = emitNodeWithoutSourceMap;

    /** Called just before starting emit of a node */
    let emitStart = function (node: Node) { };

    /** Called once the emit of the node is done */
    let emitEnd = function (node: Node) { };

    /** Emit the text for the given token that comes after startPos
      * This by default writes the text provided with the given tokenKind
      * but if optional emitFn callback is provided the text is emitted using the callbac
      * @param tokenKind the kind of the token to search and emit
      * @param startPos the position in the source to start searching for the token
      * @param emitFn if given will be invoked to emit the text instead of actual token e
    let emitToken = emitTokenText;

    /** Called to before starting the lexical scopes as in function/class in the emitted
      * @param scopeDeclaration node that starts the lexical scope
      * @param scopeName Optional name of this scope instead of deducing one from the dec
    let scopeEmitStart = function(scopeDeclaration: Node, scopeName?: string) { };

    /** Called after coming out of the scope */
    let scopeEmitEnd = function() { };

    /** Sourcemap data that will get encoded */
    let sourceMapData: SourceMapData;

    if (compilerOptions.sourceMap || compilerOptions.inlineSourceMap) {
        initializeEmitterWithSourceMaps();
    }

    if (root) {
```

```
        // Do not call emit directly. It does not set the currentSourceFile.
        emitSourceFile(root);
    }
    else {
        forEach(host.getSourceFiles(), sourceFile => {
            if (!isExternalModuleOrDeclarationFile(sourceFile)) {
                emitSourceFile(sourceFile);
            }
        });
    }

    writeLine();
    writeEmittedFiles(writer.getText(), /*writeByteOrderMark*/ compilerOptions.emitBOM);
    return;

    /// BUNCH OF LOCAL FUNCTIONS
```

The imporant function call here : `initializeEmitterWithSourceMaps` which is a function local to `emitJavaScript` that overrides some locals that were already defined here. At the bottom of `initializeEmitterWithSourceMaps` you will notice the overriding:

```
    // end of `initializeEmitterWithSourceMaps`

    writeEmittedFiles = writeJavaScriptAndSourceMapFile;
    emit = emitNodeWithSourceMap;
    emitStart = recordEmitNodeStartSpan;
    emitEnd = recordEmitNodeEndSpan;
    emitToken = writeTextWithSpanRecord;
    scopeEmitStart = recordScopeNameOfNode;
    scopeEmitEnd = recordScopeNameEnd;
    writeComment = writeCommentRangeWithMap;
```

This means that the bulk of emitter code can not care about SourceMap and just use these local functions the same way with or without SourceMaps.

Contributing

TypeScript is OSS and on GitHub and the team welcomes community input.

Setup

Super easy:

```
git clone https://github.com/Microsoft/TypeScript.git
cd TypeScript
npm install -g jake
npm install
```

Setup Fork

You would obviously need to setup Microsoft/TypeScript as an `upstream` remote and your own *fork* (use the GitHub *fork* button) as `origin` :

```
git remote rm origin
git remote rm upstream
git remote add upstream https://github.com/Microsoft/TypeScript.git
git remote add origin https://github.com/basarat/TypeScript.git
```

Additionally I like to work off branches like `bas/` to have it show up cleaner in the branch listings.

Running Tests

There are lots of `test` and `build` options in their JakeFile. You can run *all* tests with `jake runtests`

Baselines

Baselines are used to manage if there are any changes in the *expected* output of the TypeScript compiler. Baselines are located in `tests/baselines` .

- Reference (*expected*) baselines: `tests/baselines/reference`
- Generated (*in this test run*) baselines : `tests/baselines/local` (this folder is in **.gitignore**)

> If there are any differences between these folders tests will fail. You can diff the two folders with tools like BeyondCompare or KDiff3.

If you think these changes in generated files are valid then accept baselines using `jake baseline-accept`. The changes to `reference` baselines will now show as a git diff you can commit.

> Note that if you don't run *all* tests then use `jake baseline-accept[soft]` which will only copy over the new files and not delete the whole `reference` directory.

Test Categories

There are different categories for different scenarios and even different test infrastructures. Here are a few of these explained.

Compiler Tests

These ensure that compiling a file :

- generates errors as expected
- generated JS as expected
- types are identified as expected
- symbols are identified as expected

These expectations are validated using the baselines infrastructure.

Creating a Compiler Test

Test can be created by adding a new file `yourtest.ts` to `tests/cases/compiler`. As soon as you do so and run the tests you should get baseline failure. Accept these baselines (to get them to show up in git), and tweak them to be what you *expect* them to be ... now get the tests to pass.

Run all of these in isolation using `jake runtests tests=compiler`, or just your new file using `jake runtests tests=compiler/yourtest`

I will even often do `jake runtests tests=compiler/yourtest || jake baseline-accept[soft]` and get the diff in `git`.

Glossary

Duck Typing

If it walks like a duck and quacks like a duck, it is a duck. For TypeScript if it has all the members structurally then it is okay for other things (irrespective of name) that accept that structure.

1.1. Why TypeScript

Incremental Parsing

Re-Parsing as the user edits the code.

8.4.1. Parser Functions

OE

Operating Environment. I'd like to use the term Operating System, but that is not necessarily what I mean here. Think Browser,Node.js,WScriptHost etc.

8. TypeScript Compiler Internals 8.1. Program