

Table of Contents

Introduction	1.1
Tutorials	1.2
Angular	1.2.1
ASP.NET Core	1.2.2
Gulp	1.2.3
Migrating from JavaScript	1.2.4
React & Webpack	1.2.5
React	1.2.6
TypeScript in 5 minutes	1.2.7
Handbook	1.3
Basic Types	1.3.1
Variable Declarations	1.3.2
Interfaces	1.3.3
Classes	1.3.4
Functions	1.3.5
Generics	1.3.6
Enums	1.3.7
Type Inference	1.3.8
Type Compatibility	1.3.9
Advanced Types	1.3.10
Symbols	1.3.11
Iterators and Generators	1.3.12
Modules	1.3.13
Namespaces	1.3.14
Namespaces and Modules	1.3.15
Module Resolution	1.3.16
Declaration Merging	1.3.17
JSX	1.3.18
Decorators	1.3.19
Mixins	1.3.20
Utility Types	1.3.21
Triple-Slash Directives	1.3.22
Type Checking and Javascript Files	1.3.23
Declaration Files	1.4
Library Structures	1.4.1
By Example	1.4.2
Do's and Don'ts	1.4.3

Deep Dive	1.4.4
Templates	1.4.5
Publishing	1.4.6
Consumption	1.4.7
Project Configuration	1.5
tsconfig.json	1.5.1
Compiler Options	1.5.2
Project Reference	1.5.3
Compiler Options in MSBuild	1.5.4
Integrating with Build Tools	1.5.5
Nightly Builds	1.5.6
Release Notes	1.6
TypeScript 3.1	1.6.1
TypeScript 3.0	1.6.2
TypeScript 2.9	1.6.3
TypeScript 2.8	1.6.4
TypeScript 2.7	1.6.5
TypeScript 2.6	1.6.6
TypeScript 2.5	1.6.7
TypeScript 2.4	1.6.8
TypeScript 2.3	1.6.9
TypeScript 2.2	1.6.10
TypeScript 2.1	1.6.11
TypeScript 2.0	1.6.12
TypeScript 1.8	1.6.13
TypeScript 1.7	1.6.14
TypeScript 1.6	1.6.15
TypeScript 1.5	1.6.16
TypeScript 1.4	1.6.17
TypeScript 1.3	1.6.18
TypeScript 1.1	1.6.19

TypeScript-Handbook (3.1)

The PDF is generated by gitbook. The document is based on [Typescript Handbook](#)

For a more formal description of the language, see the [latest TypeScript Language Specification](#).

Angular is a modern framework built entirely in TypeScript, and as a result, using TypeScript with Angular provides a seamless experience.

The Angular documentation not only supports TypeScript as a first-class citizen, but uses it as its primary language. With this in mind, [Angular's site](#) will always be the most up-to-date reference for using Angular with TypeScript.

Check out the [quick start guide here](#) to start learning Angular now!

Setup

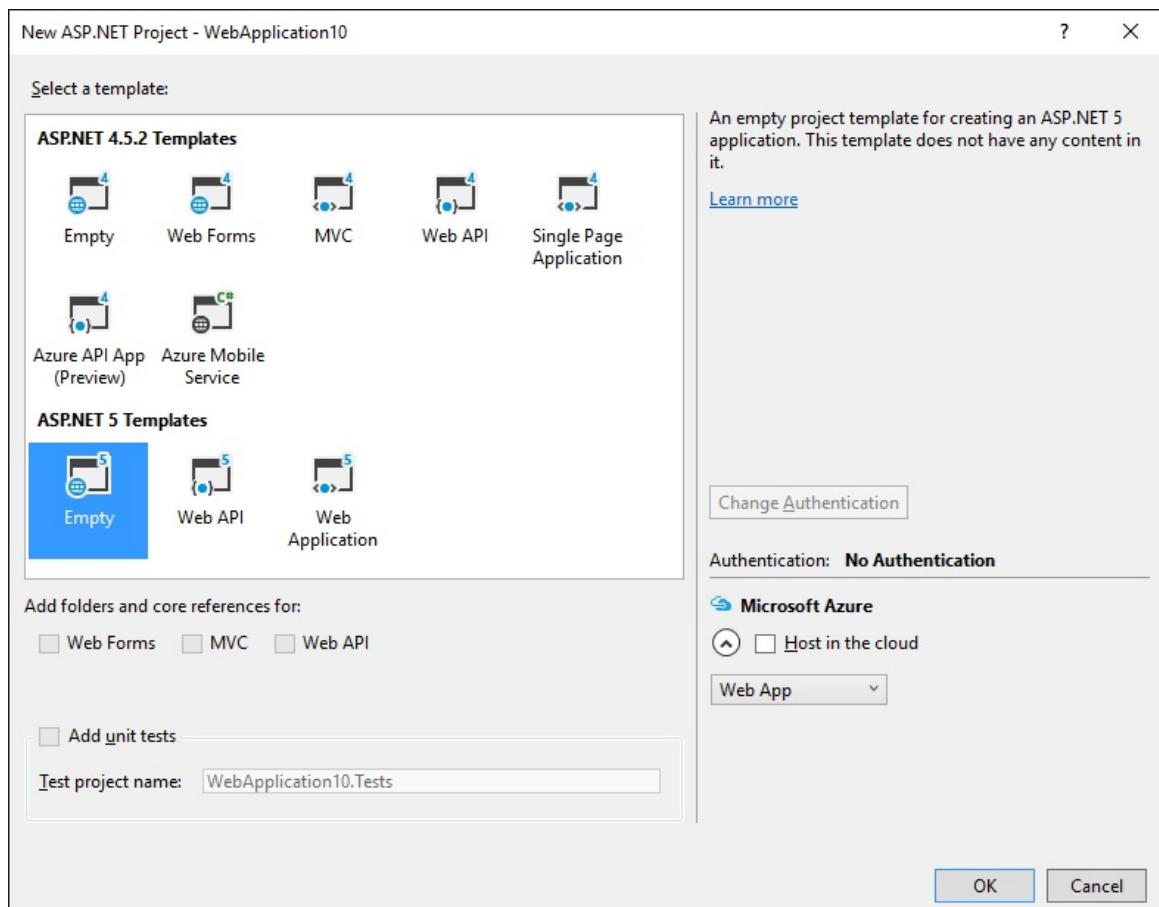
Install ASP.NET Core and TypeScript

First, [install ASP.NET Core](#) if you need it. This quick-start guide requires Visual Studio 2015 or 2017.

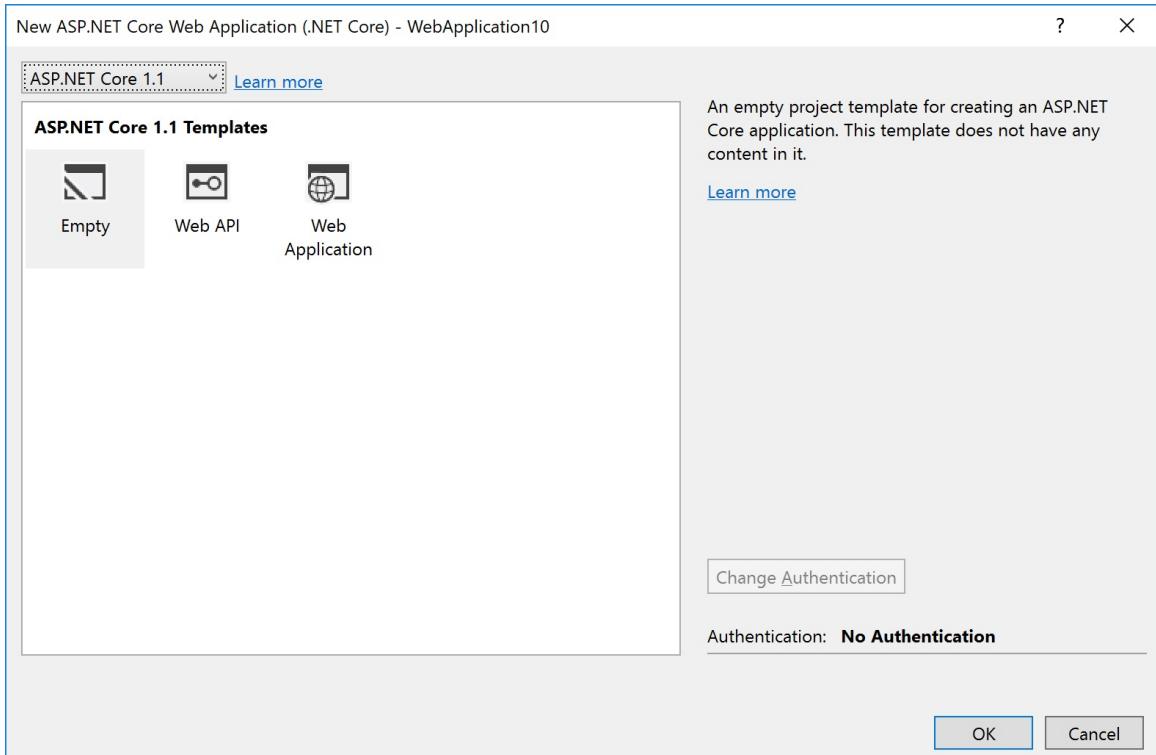
Next, if your version of Visual Studio does not already have the latest TypeScript, you can [install it](#).

Create a new project

1. Choose **File**
2. Choose **New Project** (Ctrl + Shift + N)
3. Choose **Visual C#**
4. For VS2015, choose **ASP.NET Web Application > ASP.NET 5 Empty**, and let's uncheck "Host in the cloud" since we're going to run this locally.



For VS2017, choose **ASP.NET Core Web Application (.NET Core) > ASP.NET Core 1.1 Empty** instead.



Run the application and make sure that it works.

Set up the server

VS2015

In `project.json` add another entry in `"dependencies"`:

```
"Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
```

The resulting dependencies should look like this:

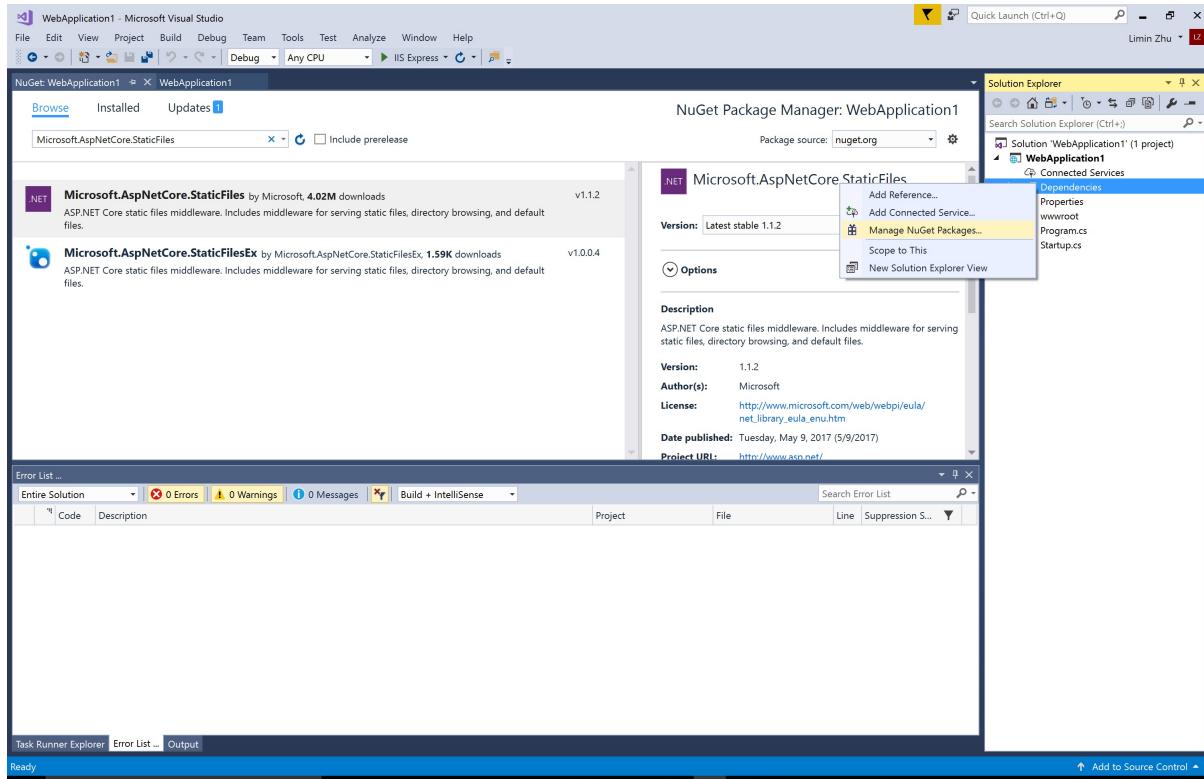
```
"dependencies": {
  "Microsoft.AspNet.IISPlatformHandler": "1.0.0-rc1-final",
  "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
  "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
},
```

Replace the body of `Configure` in `Startup.cs` with

```
public void Configure(IApplicationBuilder app)
{
  app.UseIISPlatformHandler();
  app.UseDefaultFiles();
  app.UseStaticFiles();
}
```

VS2017

Open **Dependencies > Manage NuGet Packages > Browse**. Search and install `Microsoft.AspNetCore.StaticFiles 1.1.2`:



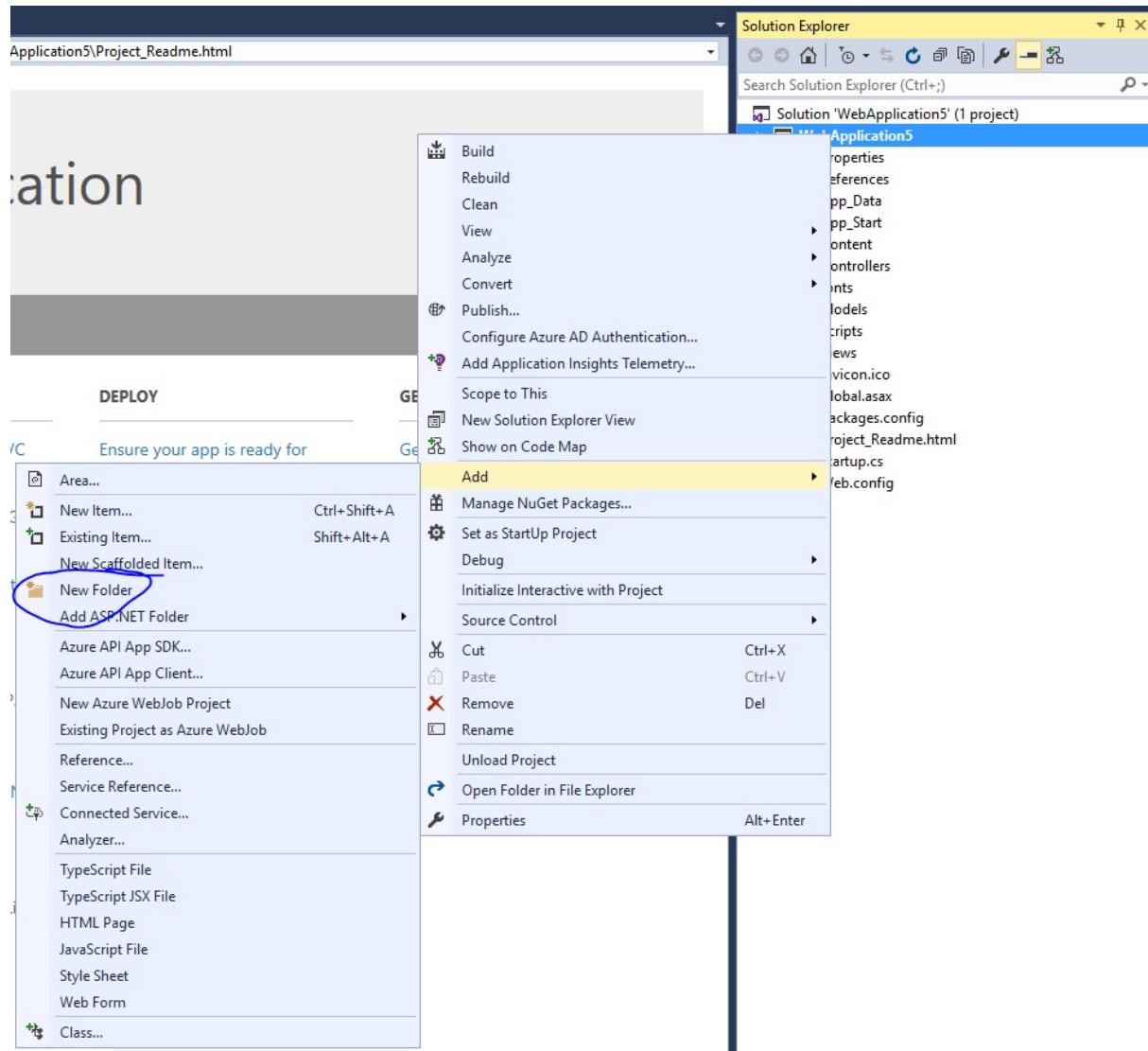
Replace the body of `Configure` in `Startup.cs` with

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```

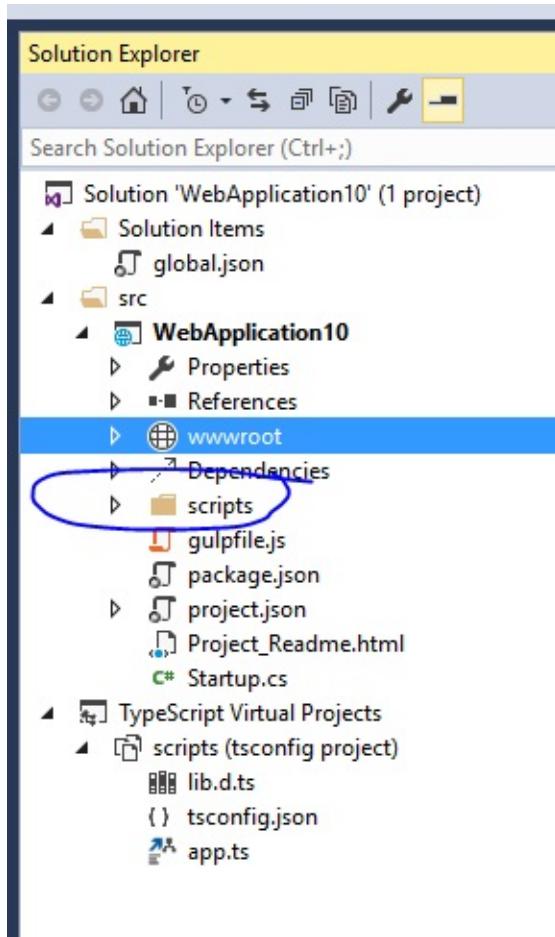
You may need to restart VS for the red squiggly lines below `UseDefaultFiles` and `UseStaticFiles` to disappear.

Add TypeScript

The next step is to add a folder for TypeScript.

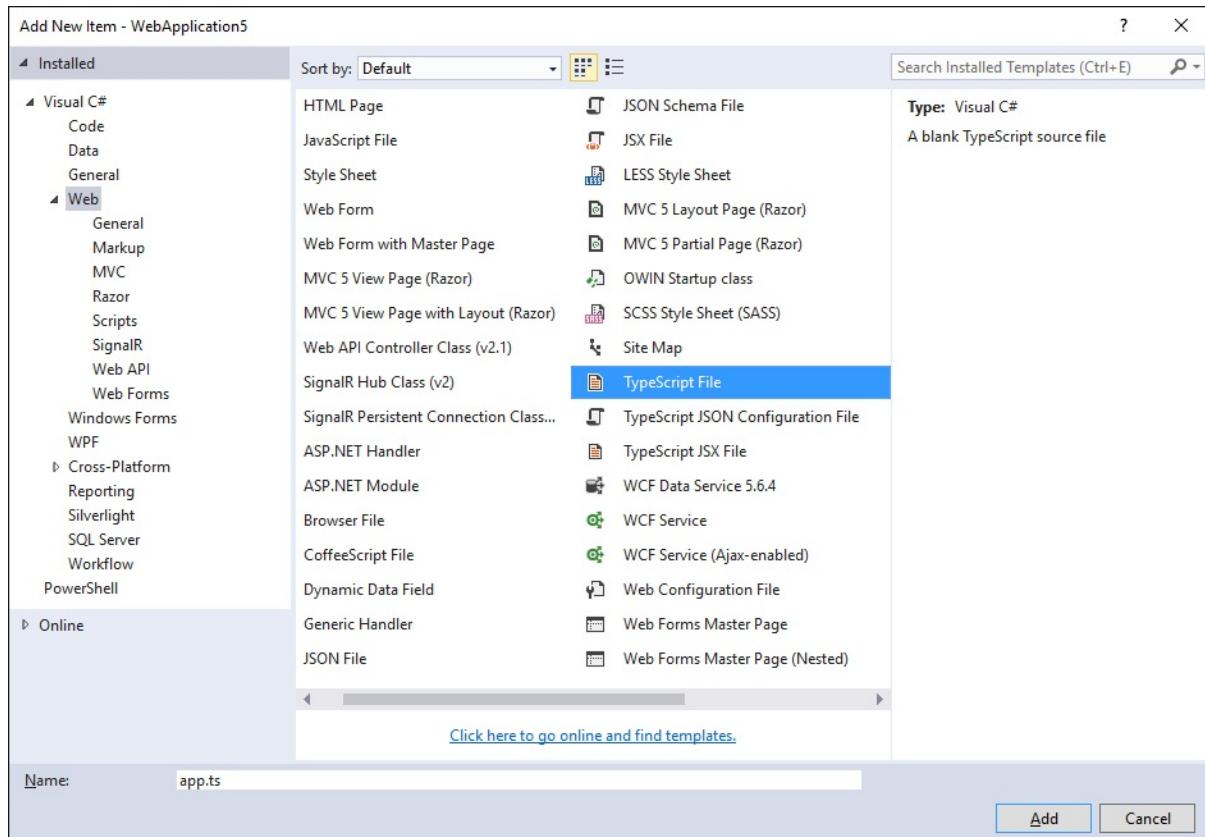


We'll just call it `scripts`.



Add TypeScript code

Right click on `scripts` and click **New Item**. Then choose **TypeScript File** (it may be in the .NET Core section) and name the file `app.ts`.



Add example code

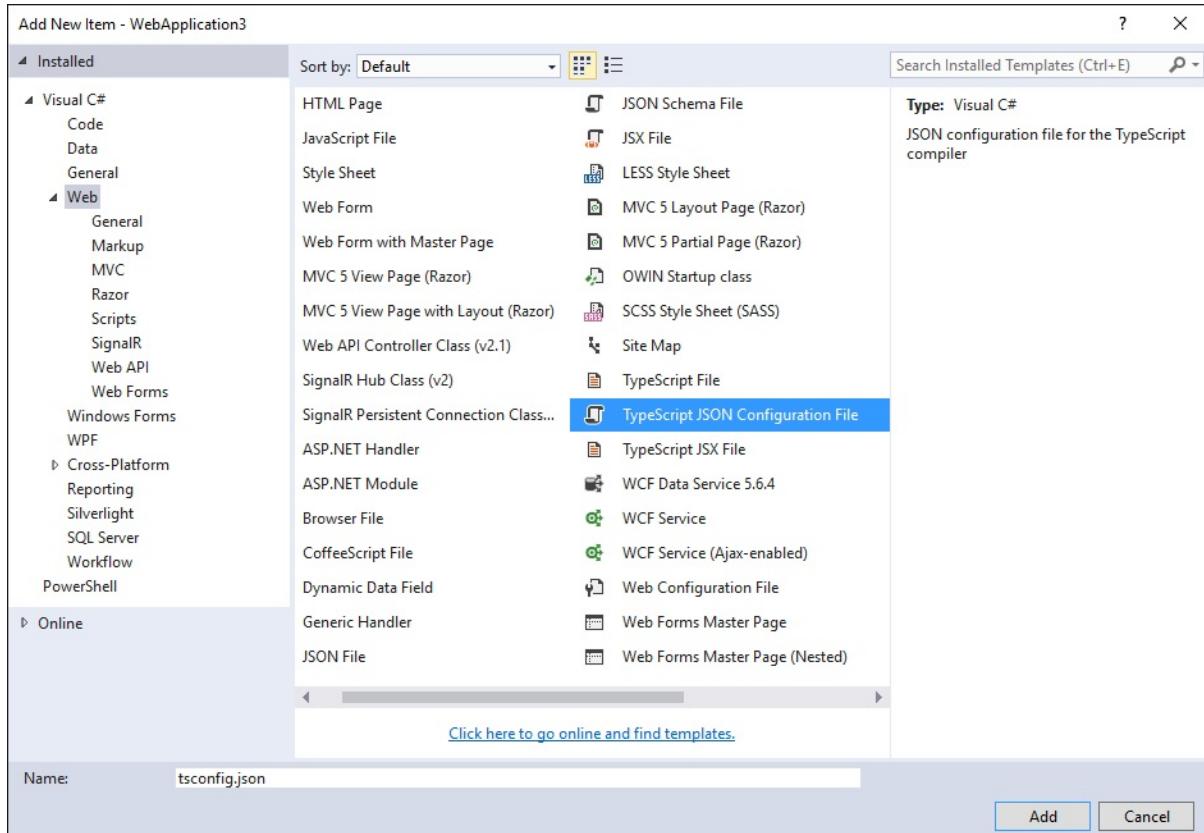
Type the following code into app.ts.

```
function sayHello() {
    const compiler = (document.getElementById("compiler") as HTMLInputElement).value;
    const framework = (document.getElementById("framework") as HTMLInputElement).value;
    return `Hello from ${compiler} and ${framework}!`;
}
```

Set up the build

Configure the TypeScript compiler

First we need to tell TypeScript how to build. Right click on the scripts folder and click **New Item**. Then choose **TypeScript Configuration File** and use the default name `tsconfig.json`.



Replace the default `tsconfig.json` with the following:

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "noEmitOnError": true,
    "sourceMap": true,
    "target": "es5"
  },
  "files": [
    "./app.ts"
  ],
  "compileOnSave": true
}
```

This is similar to the default, with the following differences:

1. It sets `"noImplicitAny": true`.
2. It explicitly lists `"files"` instead of relying on `"excludes"`.
3. It sets `"compileOnSave": true`.

`"noImplicitAny"` is good idea whenever you're writing new code — you can make sure that you don't write any untyped code by mistake. `"compileOnSave"` makes it easy to update your code in a running web app.

Set up NPM

Now we need to set up NPM so we can download JavaScript packages. Right click on the project and click **New Item**. Then choose **NPM Configuration File** and use the default name `package.json`. Inside `"devDependencies"` add `"gulp"` and `"del"`:

```
"devDependencies": {
  "gulp": "3.9.0",
```

```

    "del": "2.2.0"
}

```

Visual Studio should start installing gulp and del as soon as you save the file. If not, right-click package.json and then **Restore Packages**.

Set up gulp

Finally, add a new JavaScript file named `gulpfile.js`. Put the following code inside:

```

/// <binding AfterBuild='default' Clean='clean' />
/*
This file is the main entry point for defining Gulp tasks and using Gulp plugins.
Click here to learn more. http://go.microsoft.com/fwlink/?LinkId=518007
*/

var gulp = require('gulp');
var del = require('del');

var paths = {
  scripts: ['scripts/**/*.{js,ts,tsd,html}', 'scripts/**/*.map'],
};

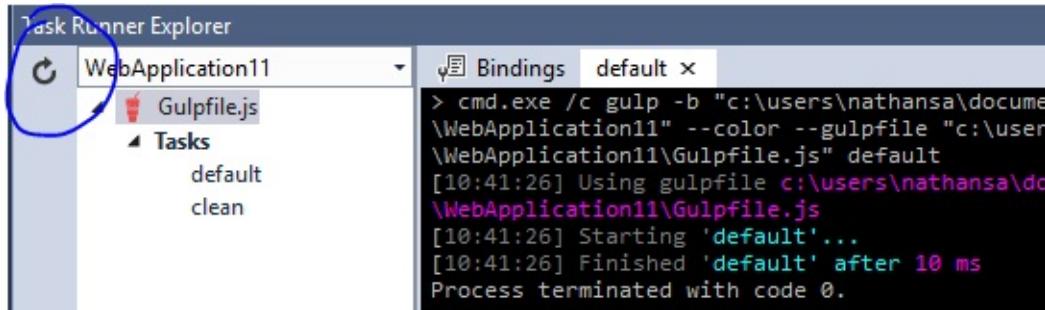
gulp.task('clean', function () {
  return del(['wwwroot/scripts/**/*']);
});

gulp.task('default', function () {
  gulp.src(paths.scripts).pipe(gulp.dest('wwwroot/scripts'));
});

```

The first line tells Visual Studio to run the task 'default' after the build finishes. It will also run the 'clean' task when you ask Visual Studio to clean the build.

Now right-click on `gulpfile.js` and click **Task Runner Explorer**. If 'default' and 'clean' tasks don't show up, refresh the explorer:



Write an HTML page

Add a New Item named `index.html` inside `wwwroot`. Use the following code for `index.html`:

```

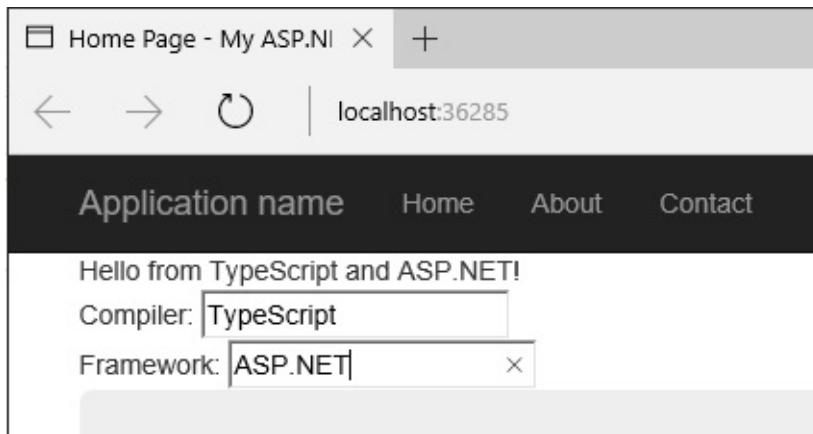
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <script src="scripts/app.js"></script>
  <title></title>
</head>
<body>

```

```
<div id="message"></div>
<div>
    Compiler: <input id="compiler" value="TypeScript" onkeyup="document.getElementById('message').innerText = sayHello()" /><br />
    Framework: <input id="framework" value="ASP.NET" onkeyup="document.getElementById('message').innerText = sayHello()" />
</div>
</body>
</html>
```

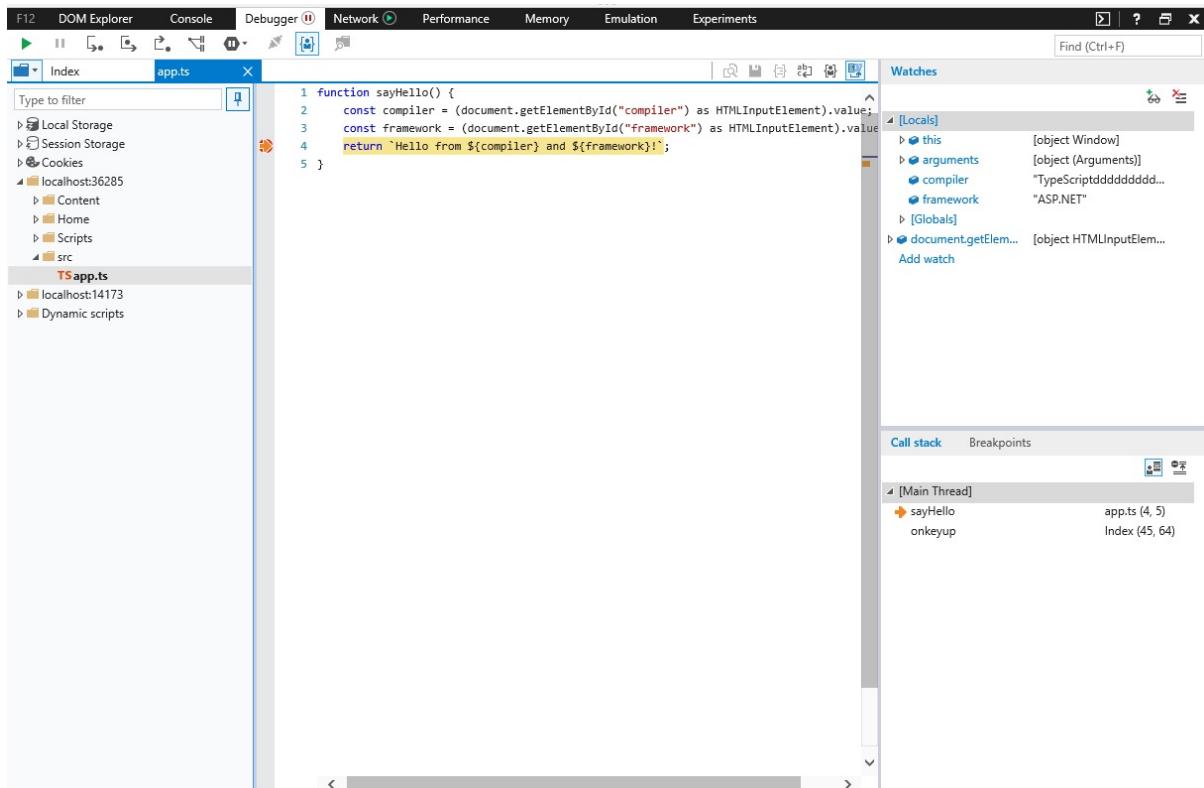
Test

1. Run the project.
2. You should see a message when you type in the input boxes:



Debug

1. In Edge, press F12 and click the **Debugger** tab.
2. Look in the first localhost folder, then scripts/app.ts
3. Put a breakpoint on the line with `return`.
4. Type in the boxes and confirm that the breakpoint hits in TypeScript code and that inspection works correctly.



That's all you need to know to include basic TypeScript in your ASP.NET project. Next we'll include Angular and write a simple Angular app.

Add Angular 2

Add NPM dependencies

Add Angular 2 and SystemJS to `dependencies` in `package.json`.

For VS2015, the new `dependencies` list:

```

"dependencies": {
  "angular2": "2.0.0-beta.11",
  "systemjs": "0.19.24",
  "gulp": "3.9.0",
  "del": "2.2.0"
},

```

For VS2017, due to the deprecation of peer dependencies in NPM3, we need to list Angular 2's peer dependencies directly as dependencies as well:

```

"dependencies": {
  "angular2": "2.0.0-beta.11",
  "reflect-metadata": "0.1.2",
  "rxjs": "5.0.0-beta.2",
  "zone.js": "^0.6.4",
  "systemjs": "0.19.24",
  "gulp": "3.9.0",
  "del": "2.2.0"
},

```

Update tsconfig.json

Now that Angular 2 and its dependencies are installed, we need to enable TypeScript's experimental support for decorators. We also need to add declarations for ES2015, since Angular uses core-js for things like `Promise`. In the future decorators will be the default and these settings will not be needed.

Add `"experimentalDecorators": true`, `"emitDecoratorMetadata": true` to the `"compilerOptions"` section. Next, add `"lib": ["es2015", "es5", "dom"]` to `"compilerOptions"` as well to bring in declarations from ES2015. Finally, we'll need to add a new entry in `"files"` for another file, `./model.ts`, which we'll create. Our `tsconfig` should now look like this:

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "noEmitOnError": true,
    "sourceMap": true,
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "target": "es5",
    "lib": [
      "es2015", "es5", "dom"
    ]
  },
  "files": [
    "./app.ts",
    "./model.ts",
    "./main.ts"
  ],
  "compileOnSave": true
}
```

Add Angular to the gulp build

Finally, we need to make sure that the Angular files are copied as part of the build. We need to add:

1. The paths to the library files.
2. Add a `lib` task to pipe the files to `wwwroot`.
3. Add a dependency on `lib` to the `default` task.

The updated `gulpfile.js` should look like this:

```
/// <binding AfterBuild='default' Clean='clean' />
/*
This file is the main entry point for defining Gulp tasks and using Gulp plugins.
Click here to learn more. http://go.microsoft.com/fwlink/?LinkId=518007
*/

var gulp = require('gulp');
var del = require('del');

var paths = {
  scripts: ['scripts/**/*.{js,ts,*.map}'],
  libs: ['node_modules/angular2/bundles/angular2.js',
    'node_modules/angular2/bundles/angular2-polyfills.js',
    'node_modules/systemjs/dist/system.src.js',
    'node_modules/rxjs/bundles/Rx.js']
};

gulp.task('lib', function () {
  gulp.src(paths.libs).pipe(gulp.dest('wwwroot/scripts/lib'));
});
```

```

gulp.task('clean', function () {
  return del(['wwwroot/scripts/**/*']);
});

gulp.task('default', ['lib'], function () {
  gulp.src(paths.scripts).pipe(gulp.dest('wwwroot/scripts'));
});

```

Again, make sure that Task Runner Explorer sees the new `lib` task after you save the gulpfile.

Write a simple Angular app in TypeScript

First, change the code in `app.ts` to:

```

import {Component} from "angular2/core"
import {MyModel} from "./model"

@Component({
  selector: `my-app`,
  template: `<div>Hello from {{getCompiler()}}</div>`
})
export class MyApp {
  model = new MyModel();
  getCompiler() {
    return this.model.compiler;
  }
}

```

Then add another TypeScript file in `scripts` named `model.ts`:

```

export class MyModel {
  compiler = "TypeScript";
}

```

And then another TypeScript file in `scripts` named `main.ts`:

```

import {bootstrap} from "angular2/platform/browser";
import {MyApp} from "./app";
bootstrap(MyApp);

```

Finally, change the code in `index.html` to the following:

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <script src="scripts/lib/angular2-polyfills.js"></script>
  <script src="scripts/lib/system.src.js"></script>
  <script src="scripts/lib/rx.js"></script>
  <script src="scripts/lib/angular2.js"></script>
  <script>
    System.config({
      packages: {
        'scripts': {
          format: 'cjs',
          defaultExtension: 'js'
        }
      }
    });

```

```
System.import('scripts/main').then(null, console.error.bind(console));
</script>
<title></title>
</head>
<body>
<my-app>Loading...</my-app>
</body>
</html>
```

This loads the app. When you run the ASP.NET application you should see a div that says "Loading..." and then updates to say "Hello from TypeScript".

This quick start guide will teach you how to build TypeScript with [gulp](#) and then add [Browserify](#), [uglify](#), or [Watchify](#) to the gulp pipeline. This guide also adds functionality for [Babel](#) functionality using [Babelify](#).

We assume that you're already using [Node.js](#) with [npm](#).

Minimal project

Let's start out with a new directory. We'll name it `proj` for now, but you can change it to whatever you want.

```
mkdir proj  
cd proj
```

To start, we're going to structure our project in the following way:

```
proj/  
  |- src/  
  |  \- dist/
```

TypeScript files will start out in your `src` folder, run through the TypeScript compiler and end up in `dist`.

Let's scaffold this out:

```
mkdir src  
mkdir dist
```

Initialize the project

Now we'll turn this folder into an npm package.

```
npm init
```

You'll be given a series of prompts. You can use the defaults except for your entry point. For your entry point, use `./dist/main.js`. You can always go back and change these in the `package.json` file that's been generated for you.

Install our dependencies

Now we can use `npm install` to install packages. First install `gulp-cli` globally (if you use a Unix system, you may need to prefix the `npm install` commands in this guide with `sudo`).

```
npm install -g gulp-cli
```

Then install `typescript`, `gulp` and `gulp-typescript` in your project's dev dependencies. [Gulp-typescript](#) is a gulp plugin for Typescript.

```
npm install --save-dev typescript gulp gulp-typescript
```

Write a simple example

Let's write a Hello World program. In `src`, create the file `main.ts`:

```
function hello(compiler: string) {
    console.log(`Hello from ${compiler}`);
}
hello("TypeScript");
```

In the project root, `proj`, create the file `tsconfig.json`:

```
{
  "files": [
    "src/main.ts"
  ],
  "compilerOptions": {
    "noImplicitAny": true,
    "target": "es5"
  }
}
```

Create a `gulpfile.js`

In the project root, create the file `gulpfile.js`:

```
var gulp = require("gulp");
var ts = require("gulp-typescript");
var tsProject = ts.createProject("tsconfig.json");

gulp.task("default", function () {
  return tsProject.src()
    .pipe(tsProject())
    .js.pipe(gulp.dest("dist"));
});
```

Test the resulting app

```
gulp
node dist/main.js
```

The program should print "Hello from TypeScript!".

Add modules to the code

Before we get to Browserify, let's build our code out and add modules to the mix. This is the structure you're more likely to use for a real app.

Create a file called `src/greet.ts`:

```
export function sayHello(name: string) {
  return `Hello from ${name}`;
}
```

Now change the code in `src/main.ts` to import `sayHello` from `greet.ts`:

```
import { sayHello } from "./greet";
```

```
console.log(sayHello("TypeScript"));
```

Finally, add `src/greet.ts` to `tsconfig.json`:

```
{
  "files": [
    "src/main.ts",
    "src/greet.ts"
  ],
  "compilerOptions": {
    "noImplicitAny": true,
    "target": "es5"
  }
}
```

Make sure that the modules work by running `gulp` and then testing in Node:

```
gulp
node dist/main.js
```

Notice that even though we used ES2015 module syntax, TypeScript emitted CommonJS modules that Node uses. We'll stick with CommonJS for this tutorial, but you could set `module` in the options object to change this.

Browserify

Now let's move this project from Node to the browser. To do this, we'd like to bundle all our modules into one JavaScript file. Fortunately, that's exactly what Browserify does. Even better, it lets us use the CommonJS module system used by Node, which is the default TypeScript emit. That means our TypeScript and Node setup will transfer to the browser basically unchanged.

First, install browserify, [tsify](#), and vinyl-source-stream. tsify is a Browserify plugin that, like gulp-typescript, gives access to the TypeScript compiler. vinyl-source-stream lets us adapt the file output of Browserify back into a format that gulp understands called [vinyl](#).

```
npm install --save-dev browserify tsify vinyl-source-stream
```

Create a page

Create a file in `src` named `index.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello World!</title>
  </head>
  <body>
    <p id="greeting">Loading ...</p>
    <script src="bundle.js"></script>
  </body>
</html>
```

Now change `main.ts` to update the page:

```

import { sayHello } from "./greet";

function showHello(divName: string, name: string) {
  const elt = document.getElementById(divName);
  elt.innerText = sayHello(name);
}

showHello("greeting", "TypeScript");

```

Calling `showHello` calls `sayHello` to change the paragraph's text. Now change your gulpfile to the following:

```

var gulp = require("gulp");
var browserify = require("browserify");
var source = require('vinyl-source-stream');
var tsify = require("tsify");
var paths = {
  pages: ['src/*.html']
};

gulp.task("copy-html", function () {
  return gulp.src(paths.pages)
    .pipe(gulp.dest("dist"));
});

gulp.task("default", ["copy-html"], function () {
  return browserify({
    basedir: '.',
    debug: true,
    entries: ['src/main.ts'],
    cache: {},
    packageCache: {}
  })
    .plugin(tsify)
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(gulp.dest("dist"));
});

```

This adds the `copy-html` task and adds it as a dependency of `default`. That means any time `default` is run, `copy-html` has to run first. We've also changed `default` to call Browserify with the `tsify` plugin instead of `gulp-typescript`. Conveniently, they both allow us to pass the same options object to the TypeScript compiler.

After calling `bundle` we use `source` (our alias for `vinyl-source-stream`) to name our output bundle `bundle.js`.

Test the page by running `gulp` and then opening `dist/index.html` in a browser. You should see "Hello from TypeScript" on the page.

Notice that we specified `debug: true` to Browserify. This causes `tsify` to emit source maps inside the bundled JavaScript file. Source maps let you debug your original TypeScript code in the browser instead of the bundled JavaScript. You can test that source maps are working by opening the debugger for your browser and putting a breakpoint inside `main.ts`. When you refresh the page the breakpoint should pause the page and let you debug `greet.ts`.

Watchify, Babel, and Uglify

Now that we are bundling our code with Browserify and `tsify`, we can add various features to our build with `browserify` plugins.

- Watchify starts gulp and keeps it running, incrementally compiling whenever you save a file. This lets you keep an edit-save-refresh cycle going in the browser.
- Babel is a hugely flexible compiler that converts ES2015 and beyond into ES5 and ES3. This lets you add extensive and customized transformations that TypeScript doesn't support.
- Uglify compacts your code so that it takes less time to download.

Watchify

We'll start with Watchify to provide background compilation:

```
npm install --save-dev watchify gulp-util
```

Now change your gulpfile to the following:

```
var gulp = require("gulp");
var browserify = require("browserify");
var source = require('vinyl-source-stream');
var watchify = require("watchify");
var tsify = require("tsify");
var gutil = require("gulp-util");
var paths = {
  pages: ['src/*.html']
};

var watchedBrowserify = watchify(browserify({
  basedir: '.',
  debug: true,
  entries: ['src/main.ts'],
  cache: {},
  packageCache: {}
}).plugin(tsify));

gulp.task("copy-html", function () {
  return gulp.src(paths.pages)
    .pipe(gulp.dest("dist"));
});

function bundle() {
  return watchedBrowserify
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(gulp.dest("dist"));
}

gulp.task("default", ["copy-html"], bundle);
watchedBrowserify.on("update", bundle);
watchedBrowserify.on("log", gutil.log);
```

There are basically three changes here, but they require you to refactor your code a bit.

1. We wrapped our `browserify` instance in a call to `watchify`, and then held on to the result.
2. We called `watchedBrowserify.on("update", bundle);` so that Browserify will run the `bundle` function every time one of your TypeScript files changes.
3. We called `watchedBrowserify.on("log", gutil.log);` to log to the console.

Together (1) and (2) mean that we have to move our call to `browserify` out of the `default` task. And we have to give the function for `default` a name since both Watchify and Gulp need to call it. Adding logging with (3) is optional but very useful for debugging your setup.

Now when you run Gulp, it should start and stay running. Try changing the code for `showHello` in `main.ts` and saving it. You should see output that looks like this:

```
proj$ gulp
[10:34:20] Using gulpfile ~/src/proj/gulpfile.js
[10:34:20] Starting 'copy-html'...
[10:34:20] Finished 'copy-html' after 26 ms
[10:34:20] Starting 'default'...
[10:34:21] 2824 bytes written (0.13 seconds)
[10:34:21] Finished 'default' after 1.36 s
[10:35:22] 2261 bytes written (0.02 seconds)
[10:35:24] 2808 bytes written (0.05 seconds)
```

Uglify

First install Uglify. Since the point of Uglify is to mangle your code, we also need to install vinyl-buffer and gulp-sourcemaps to keep sourcemaps working.

```
npm install --save-dev gulp-uglify vinyl-buffer gulp-sourcemaps
```

Now change your gulpfile to the following:

```
var gulp = require("gulp");
var browserify = require("browserify");
var source = require('vinyl-source-stream');
var tsify = require("tsify");
var uglify = require('gulp-uglify');
var sourcemap = require('gulp-sourcemaps');
var buffer = require('vinyl-buffer');
var paths = {
  pages: ['src/*.html']
};

gulp.task("copy-html", function () {
  return gulp.src(paths.pages)
    .pipe(gulp.dest("dist"));
});

gulp.task("default", ["copy-html"], function () {
  return browserify({
    basedir: '.',
    debug: true,
    entries: ['src/main.ts'],
    cache: {},
    packageCache: {}
  })
    .plugin(tsify)
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(buffer())
    .pipe(sourcemap.init({loadMaps: true}))
    .pipe(uglify())
    .pipe(sourcemap.write('./'))
    .pipe(gulp.dest("dist"));
});
```

Notice that `uglify` itself has just one call — the calls to `buffer` and `sourcemap` exist to make sure sourcemaps keep working. These calls give us a separate sourcemap file instead of using inline sourcemaps like before. Now you can run Gulp and check that `bundle.js` does get minified into an unreadable mess:

```
gulp
cat dist/bundle.js
```

Babel

First install Babelify and the Babel preset for ES2015. Like Uglify, Babelify mangles code, so we'll need vinyl-buffer and gulp-sourcemaps. By default Babelify will only process files with extensions of `.js`, `.es`, `.es6` and `.jsx` so we need to add the `.ts` extension as an option to Babelify.

```
npm install --save-dev babelify babel-core babel-preset-es2015 vinyl-buffer gulp-sourcemaps
```

Now change your gulpfile to the following:

```
var gulp = require('gulp');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var tsify = require('tsify');
var sourcemaps = require('gulp-sourcemaps');
var buffer = require('vinyl-buffer');
var paths = {
  pages: ['src/*.html']
};

gulp.task('copyHtml', function () {
  return gulp.src(paths.pages)
    .pipe(gulp.dest('dist'));
});

gulp.task('default', ['copyHtml'], function () {
  return browserify({
    basedir: '.',
    debug: true,
    entries: ['src/main.ts'],
    cache: {},
    packageCache: {}
  })
    .plugin(tsify)
    .transform('babelify', {
      presets: ['es2015'],
      extensions: ['.ts']
    })
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(buffer())
    .pipe(sourcemaps.init({loadMaps: true}))
    .pipe(sourcemaps.write('./'))
    .pipe(gulp.dest('dist'));
});
```

We also need to have TypeScript target ES2015. Babel will then produce ES5 from the ES2015 code that TypeScript emits. Let's modify `tsconfig.json`:

```
{
  "files": [
    "src/main.ts"
  ],
  "compilerOptions": {
    "noImplicitAny": true,
    "target": "es2015"
  }
}
```

Babel's ES5 output should be very similar to TypeScript's output for such a simple script.

TypeScript doesn't exist in a vacuum. It was built with the JavaScript ecosystem in mind, and a lot of JavaScript exists today. Converting a JavaScript codebase over to TypeScript is, while somewhat tedious, usually not challenging. In this tutorial, we're going to look at how you might start out. We assume you've read enough of the handbook to write new TypeScript code.

If you're looking to convert a React project, we recommend looking at the [React Conversion Guide](#) first.

Setting up your Directories

If you're writing in plain JavaScript, it's likely that you're running your JavaScript directly, where your `.js` files are in a `src`, `lib`, or `dist` directory, and then ran as desired.

If that's the case, the files that you've written are going to be used as inputs to TypeScript, and you'll run the outputs it produces. During our JS to TS migration, we'll need to separate our input files to prevent TypeScript from overwriting them. If your output files need to reside in a specific directory, then that will be your output directory.

You might also be running some intermediate steps on your JavaScript, such as bundling or using another transpiler like Babel. In this case, you might already have a folder structure like this set up.

From this point on, we're going to assume that your directory is set up something like this:

```
projectRoot
├── src
|   ├── file1.js
|   └── file2.js
└── built
    └── tsconfig.json
```

If you have a `tests` folder outside of your `src` directory, you might have one `tsconfig.json` in `src`, and one in `tests` as well.

Writing a Configuration File

TypeScript uses a file called `tsconfig.json` for managing your project's options, such as which files you want to include, and what sorts of checking you want to perform. Let's create a bare-bones one for our project:

```
{
  "compilerOptions": {
    "outDir": "./built",
    "allowJs": true,
    "target": "es5"
  },
  "include": [
    "./src/**/*"
  ]
}
```

Here we're specifying a few things to TypeScript:

1. Read in any files it understands in the `src` directory (with `include`).
2. Accept JavaScript files as inputs (with `allowJs`).
3. Emit all of the output files in `built` (with `outDir`).
4. Translate newer JavaScript constructs down to an older version like ECMAScript 5 (using `target`).

At this point, if you try running `tsc` at the root of your project, you should see output files in the `built` directory. The layout of files in `built` should look identical to the layout of `src`. You should now have TypeScript working with your project.

Early Benefits

Even at this point you can get some great benefits from TypeScript understanding your project. If you open up an editor like [VS Code](#) or [Visual Studio](#), you'll see that you can often get some tooling support like completion. You can also catch certain bugs with options like:

- `noImplicitReturns` which prevents you from forgetting to return at the end of a function.
- `noFallthroughCasesInSwitch` which is helpful if you never want to forget a `break` statement between `case`s in a `switch` block.

TypeScript will also warn about unreachable code and labels, which you can disable with `allowUnreachableCode` and `allowUnusedLabels` respectively.

Integrating with Build Tools

You might have some more build steps in your pipeline. Perhaps you concatenate something to each of your files. Each build tool is different, but we'll do our best to cover the gist of things.

Gulp

If you're using Gulp in some fashion, we have a tutorial on [using Gulp](#) with TypeScript, and integrating with common build tools like Browserify, Babelify, and Uglify. You can read more there.

Webpack

Webpack integration is pretty simple. You can use `awesome-typescript-loader`, a TypeScript loader, combined with `source-map-loader` for easier debugging. Simply run

```
npm install awesome-typescript-loader source-map-loader
```

and merge in options from the following into your `webpack.config.js` file:

```
module.exports = {
  entry: "./src/index.ts",
  output: {
    filename: "./dist/bundle.js",
  },
  // Enable sourcemaps for debugging webpack's output.
  devtool: "source-map",

  resolve: {
    // Add '.ts' and '.tsx' as resolvable extensions.
    extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
  },
  module: {
    loaders: [
      // All files with a '.ts' or '.tsx' extension will be handled by 'awesome-typescript-loader'.
    ]
  }
};
```

```

        { test: /\.tsx?$/, loader: "awesome-typescript-loader" }
    ],
    preLoaders: [
        // All output '.js' files will have any sourcemaps re-processed by 'source-map-loader'.
        { test: /\.js$/, loader: "source-map-loader" }
    ]
},
// Other options...
);

```

It's important to note that `awesome-typescript-loader` will need to run before any other loader that deals with `.js` files.

The same goes for `ts-loader`, another TypeScript loader for Webpack. You can read more about the differences between the two [here](#).

You can see an example of using Webpack in our [tutorial on React and Webpack](#).

Moving to TypeScript Files

At this point, you're probably ready to start using TypeScript files. The first step is to rename one of your `.js` files to `.ts`. If your file uses JSX, you'll need to rename it to `.tsx`.

Finished with that step? Great! You've successfully migrated a file from JavaScript to TypeScript!

Of course, that might not feel right. If you open that file in an editor with TypeScript support (or if you run `tsc --pretty`), you might see red squiggles on certain lines. You should think of these the same way you'd think of red squiggles in an editor like Microsoft Word. TypeScript will still translate your code, just like Word will still let you print your documents.

If that sounds too lax for you, you can tighten that behavior up. If, for instance, you *don't* want TypeScript to compile to JavaScript in the face of errors, you can use the `noEmitOnErrors` option. In that sense, TypeScript has a dial on its strictness, and you can turn that knob up as high as you want.

If you plan on using the stricter settings that are available, it's best to turn them on now (see [Getting Stricter Checks](#) below). For instance, if you never want TypeScript to silently infer `any` for a type without you explicitly saying so, you can use `noImplicitAny` before you start modifying your files. While it might feel somewhat overwhelming, the long-term gains become apparent much more quickly.

Weeding out Errors

Like we mentioned, it's not unexpected to get error messages after conversion. The important thing is to actually go one by one through these and decide how to deal with the errors. Often these will be legitimate bugs, but sometimes you'll have to explain what you're trying to do a little better to TypeScript.

Importing from Modules

You might start out getting a bunch of errors like `Cannot find name 'require'`, and `Cannot find name 'define'`. In these cases, it's likely that you're using modules. While you can just convince TypeScript that these exist by writing out

```
// For Node/CommonJS
declare function require(path: string): any;
```

or

```
// For RequireJS/AMD
declare function define(...args: any[]): any;
```

it's better to get rid of those calls and use TypeScript syntax for imports.

First, you'll need to enable some module system by setting TypeScript's `module` flag. Valid options are `commonjs`, `amd`, `system`, and `umd`.

If you had the following Node/CommonJS code:

```
var foo = require("foo");

foo.doStuff();
```

or the following RequireJS/AMD code:

```
define(["foo"], function(foo) {
  foo.doStuff();
})
```

then you would write the following TypeScript code:

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

foo.doStuff();
```

Getting Declaration Files

If you started converting over to TypeScript imports, you'll probably run into errors like `Cannot find module 'foo'.`. The issue here is that you likely don't have *declaration files* to describe your library. Luckily this is pretty easy. If TypeScript complains about a package like `lodash`, you can just write

```
npm install -S @types/lodash
```

If you're using a module option other than `commonjs`, you'll need to set your `moduleResolution` option to `node`.

After that, you'll be able to import `lodash` with no issues, and get accurate completions.

Exporting from Modules

Typically, exporting from a module involves adding properties to a value like `exports` or `module.exports`. TypeScript allows you to use top-level export statements. For instance, if you exported a function like so:

```
module.exports.feedPets = function(pets) {
  // ...
}
```

you could write that out as the following:

```
export function feedPets(pets) {
  // ...
}
```

Sometimes you'll entirely overwrite the `exports` object. This is a common pattern people use to make their modules immediately callable like in this snippet:

```
var express = require("express");
var app = express();
```

You might have previously written that like so:

```
function foo() {
    // ...
}
module.exports = foo;
```

In TypeScript, you can model this with the `export =` construct.

```
function foo() {
    // ...
}
export = foo;
```

Too many/too few arguments

You'll sometimes find yourself calling a function with too many/few arguments. Typically, this is a bug, but in some cases, you might have declared a function that uses the `arguments` object instead of writing out any parameters:

```
function myCoolFunction() {
    if (arguments.length == 2 && !Array.isArray(arguments[1])) {
        var f = arguments[0];
        var arr = arguments[1];
        // ...
    }
    // ...
}

myCoolFunction(function(x) { console.log(x) }, [1, 2, 3, 4]);
myCoolFunction(function(x) { console.log(x) }, 1, 2, 3, 4);
```

In this case, we need to use TypeScript to tell any of our callers about the ways `myCoolFunction` can be called using function overloads.

```
function myCoolFunction(f: (x: number) => void, nums: number[]): void;
function myCoolFunction(f: (x: number) => void, ...nums: number[]): void;
function myCoolFunction() {
    if (arguments.length == 2 && !Array.isArray(arguments[1])) {
        var f = arguments[0];
        var arr = arguments[1];
        // ...
    }
    // ...
}
```

We added two overload signatures to `myCoolFunction`. The first checks states that `myCoolFunction` takes a function (which takes a `number`), and then a list of `number`s. The second one says that it will take a function as well, and then uses a rest parameter (`...nums`) to state that any number of arguments after that need to be `number`s.

Sequentially Added Properties

Some people find it more aesthetically pleasing to create an object and add properties immediately after like so:

```
var options = {};
options.color = "red";
options.volume = 11;
```

TypeScript will say that you can't assign to `color` and `volume` because it first figured out the type of `options` as `{}` which doesn't have any properties. If you instead moved the declarations into the object literal themselves, you'd get no errors:

```
let options = {
  color: "red",
  volume: 11
};
```

You could also define the type of `options` and add a type assertion on the object literal.

```
interface Options { color: string; volume: number }

let options = {} as Options;
options.color = "red";
options.volume = 11;
```

Alternatively, you can just say `options` has the type `any` which is the easiest thing to do, but which will benefit you the least.

any , Object , and {}

You might be tempted to use `Object` or `{}` to say that a value can have any property on it because `Object` is, for most purposes, the most general type. However `any` is actually the type you want to use in those situations, since it's the most *flexible* type.

For instance, if you have something that's typed as `Object` you won't be able to call methods like `toLowerCase()` on it. Being more general usually means you can do less with a type, but `any` is special in that it is the most general type while still allowing you to do anything with it. That means you can call it, construct it, access properties on it, etc. Keep in mind though, whenever you use `any`, you lose out on most of the error checking and editor support that TypeScript gives you.

If a decision ever comes down to `Object` and `{}`, you should prefer `{}`. While they are mostly the same, technically `{}` is a more general type than `Object` in certain esoteric cases.

Getting Stricter Checks

TypeScript comes with certain checks to give you more safety and analysis of your program. Once you've converted your codebase to TypeScript, you can start enabling these checks for greater safety.

No Implicit any

There are certain cases where TypeScript can't figure out what certain types should be. To be as lenient as possible, it will decide to use the type `any` in its place. While this is great for migration, using `any` means that you're not getting any type safety, and you won't get the same tooling support you'd get elsewhere. You can tell TypeScript to flag these locations down and give an error with the `noImplicitAny` option.

Strict null & undefined Checks

By default, TypeScript assumes that `null` and `undefined` are in the domain of every type. That means anything declared with the type `number` could be `null` or `undefined`. Since `null` and `undefined` are such a frequent source of bugs in JavaScript and TypeScript, TypeScript has the `strictNullChecks` option to spare you the stress of worrying about these issues.

When `strictNullChecks` is enabled, `null` and `undefined` get their own types called `null` and `undefined` respectively. Whenever anything is *possibly* `null`, you can use a union type with the original type. So for instance, if something could be a `number` or `null`, you'd write the type out as `number | null`.

If you ever have a value that TypeScript thinks is possibly `null / undefined`, but you know better, you can use the postfix `!` operator to tell it otherwise.

```
declare var foo: string[] | null;

foo.length; // error - 'foo' is possibly 'null'

foo!.length; // okay - 'foo!' just has type 'string[]'
```

As a heads up, when using `strictNullChecks`, your dependencies may need to be updated to use `strictNullChecks` as well.

No Implicit any for this

When you use the `this` keyword outside of classes, it has the type `any` by default. For instance, imagine a `Point` class, and imagine a function that we wish to add as a method:

```
class Point {
    constructor(public x, public y) {}
    getDistance(p: Point) {
        let dx = p.x - this.x;
        let dy = p.y - this.y;
        return Math.sqrt(dx ** 2 + dy ** 2);
    }
    // ...
}

// Reopen the interface.
interface Point {
    distanceFromOrigin(point: Point): number;
}
Point.prototype.distanceFromOrigin = function(point: Point) {
    return this.getDistance({ x: 0, y: 0 });
}
```

This has the same problems we mentioned above - we could easily have misspelled `getDistance` and not gotten an error. For this reason, TypeScript has the `noImplicitThis` option. When that option is set, TypeScript will issue an error when `this` is used without an explicit (or inferred) type. The fix is to use a `this`-parameter to give an explicit type in the interface or in the function itself:

```
Point.prototype.distanceFromOrigin = function(this: Point, point: Point) {
    return this.getDistance({ x: 0, y: 0 });
}
```


This guide will teach you how to wire up TypeScript with [React](#) and [webpack](#).

If you're starting a brand new project, take a look at the [React Quick Start](#) guide first.

Otherwise, we assume that you're already using [Node.js](#) with [npm](#).

Lay out the project

Let's start out with a new directory. We'll name it `proj` for now, but you can change it to whatever you want.

```
mkdir proj  
cd proj
```

To start, we're going to structure our project in the following way:

```
proj/  
  └ dist/  
    └ src/  
      └ components/
```

TypeScript files will start out in your `src` folder, run through the TypeScript compiler, then webpack, and end up in a `bundle.js` file in `dist`. Any components that we write will go in the `src/components` folder.

Let's scaffold this out:

```
mkdir src  
cd src  
mkdir components  
cd ..
```

Webpack will eventually generate the `dist` directory for us.

Initialize the project

Now we'll turn this folder into an npm package.

```
npm init
```

You'll be given a series of prompts, but you can feel free to use the defaults. You can always go back and change these in the `package.json` file that's been generated for you.

Install our dependencies

First ensure Webpack is installed globally.

```
npm install -g webpack
```

Webpack is a tool that will bundle your code and optionally all of its dependencies into a single `.js` file.

Let's now add React and React-DOM, along with their declaration files, as dependencies to your `package.json` file:

```
npm install --save react react-dom @types/react @types/react-dom
```

That `@types/` prefix means that we also want to get the declaration files for React and React-DOM. Usually when you import a path like `"react"`, it will look inside of the `react` package itself; however, not all packages include declaration files, so TypeScript also looks in the `@types/react` package as well. You'll see that we won't even have to think about this later on.

Next, we'll add development-time dependencies on [awesome-typescript-loader](#) and [source-map-loader](#).

```
npm install --save-dev typescript awesome-typescript-loader source-map-loader
```

Both of these dependencies will let TypeScript and webpack play well together. `awesome-typescript-loader` helps Webpack compile your TypeScript code using the TypeScript's standard configuration file named `tsconfig.json`. `source-map-loader` uses any sourcemap outputs from TypeScript to inform webpack when generating *its own* sourcemaps. This will allow you to debug your final output file as if you were debugging your original TypeScript source code.

Please note that `awesome-typescript-loader` is not the only loader for `typescript`. You could instead use [ts-loader](#). Read about the differences between them [here](#)

Notice that we installed TypeScript as a development dependency. We could also have linked TypeScript to a global copy with `npm link typescript`, but this is a less common scenario.

Add a TypeScript configuration file

You'll want to bring your TypeScript files together - both the code you'll be writing as well as any necessary declaration files.

To do this, you'll need to create a `tsconfig.json` which contains a list of your input files as well as all your compilation settings. Simply create a new file in your project root named `tsconfig.json` and fill it with the following contents:

```
{
  "compilerOptions": {
    "outDir": "./dist/",
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react"
  },
  "include": [
    "./src/**/*"
  ]
}
```

You can learn more about `tsconfig.json` files [here](#).

Write some code

Let's write our first TypeScript file using React. First, create a file named `Hello.tsx` in `src/components` and write the following:

```
import * as React from "react";
```

```
export interface HelloProps { compiler: string; framework: string; }

export const Hello = (props: HelloProps) => <h1>Hello from {props.compiler} and {props.framework}!</h1>;
```

Note that while this example uses [stateless functional components](#), we could also make our example a little *classier* as well.

```
import * as React from "react";

export interface HelloProps { compiler: string; framework: string; }

// 'HelloProps' describes the shape of props.
// State is never set so we use the '{}' type.
export class Hello extends React.Component<HelloProps, {}> {
    render() {
        return <h1>Hello from {this.props.compiler} and {this.props.framework}!</h1>;
    }
}
```

Next, let's create an `index.tsx` in `src` with the following source:

```
import * as React from "react";
import * as ReactDOM from "react-dom";

import { Hello } from "./components/Hello";

ReactDOM.render(
    <Hello compiler="TypeScript" framework="React" />,
    document.getElementById("example")
);
```

We just imported our `Hello` component into `index.tsx`. Notice that unlike with `"react"` or `"react-dom"`, we used a *relative path* to `Hello.tsx` - this is important. If we hadn't, TypeScript would've instead tried looking in our `node_modules` folder.

We'll also need a page to display our `Hello` component. Create a file at the root of `proj` named `index.html` with the following contents:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Hello React!</title>
    </head>
    <body>
        <div id="example"></div>

        <!-- Dependencies -->
        <script src="./node_modules/react/umd/react.development.js"></script>
        <script src="./node_modules/react-dom/umd/react-dom.development.js"></script>

        <!-- Main -->
        <script src="./dist/bundle.js"></script>
    </body>
</html>
```

Notice that we're including files from within `node_modules`. React and React-DOM's npm packages include standalone `.js` files that you can include in a web page, and we're referencing them directly to get things moving faster. Feel free to copy these files to another directory, or alternatively, host them on a content delivery network (CDN). Facebook makes CDN-hosted versions of React available, and you can [read more about that here](#).

Create a webpack configuration file

Create a `webpack.config.js` file at the root of the project directory.

```
module.exports = {
  entry: "./src/index.tsx",
  output: {
    filename: "bundle.js",
    path: __dirname + "/dist"
  },
  // Enable sourcemaps for debugging webpack's output.
  devtool: "source-map",

  resolve: {
    // Add '.ts' and '.tsx' as resolvable extensions.
    extensions: [".ts", ".tsx", ".js", ".json"]
  },
  module: {
    rules: [
      // All files with a '.ts' or '.tsx' extension will be handled by 'awesome-typescript-loader'.
      { test: /\.tsx?$/, loader: "awesome-typescript-loader" },

      // All output '.js' files will have any sourcemaps re-processed by 'source-map-loader'.
      { enforce: "pre", test: /\.js$/, loader: "source-map-loader" }
    ]
  },
  // When importing a module whose path matches one of the following, just
  // assume a corresponding global variable exists and use that instead.
  // This is important because it allows us to avoid bundling all of our
  // dependencies, which allows browsers to cache those libraries between builds.
  externals: {
    "react": "React",
    "react-dom": "ReactDOM"
  }
};
```

You might be wondering about that `externals` field. We want to avoid bundling all of React into the same file, since this increases compilation time and browsers will typically be able to cache a library if it doesn't change.

Ideally, we'd just import the React module from within the browser, but most browsers still don't quite support modules yet. Instead libraries have traditionally made themselves available using a single global variable like `jQuery` or `_`. This is called the "namespace pattern", and webpack allows us to continue leveraging libraries written that way. With our entry for `"react": "React"`, webpack will work its magic to make any import of `"react"` load from the `React` variable.

You can learn more about configuring webpack [here](#).

Putting it all together

Just run:

```
webpack
```

Now open up `index.html` in your favorite browser and everything should be ready to use! You should see a page that says "Hello from TypeScript and React!"

This quick start guide will teach you how to wire up TypeScript with [React](#). By the end, you'll have

- a project with React and TypeScript
- linting with [TSLint](#)
- testing with [Jest](#) and [Enzyme](#), and
- state management with [Redux](#)

We'll use the [create-react-app](#) tool to quickly get set up.

We assume that you're already using [Node.js](#) with [npm](#). You may also want to get a sense of [the basics with React](#).

Install `create-react-app`

We're going to use the `create-react-app` because it sets some useful tools and canonical defaults for React projects. This is just a command-line utility to scaffold out new React projects.

```
npm install -g create-react-app
```

Create our new project

We'll create a new project called `my-app`:

```
create-react-app my-app --scripts-version=react-scripts-ts
```

[react-scripts-ts](#) is a set of adjustments to take the standard `create-react-app` project pipeline and bring TypeScript into the mix.

At this point, your project layout should look like the following:

```
my-app/
├── .gitignore
├── node_modules/
├── public/
└── src/
  └── ...
├── package.json
├── tsconfig.json
└── tslint.json
```

Of note:

- `tsconfig.json` contains TypeScript-specific options for our project.
- `tslint.json` stores the settings that our linter, [TSLint](#), will use.
- `package.json` contains our dependencies, as well as some shortcuts for commands we'd like to run for testing, previewing, and deploying our app.
- `public` contains static assets like the HTML page we're planning to deploy to, or images. You can delete any file in this folder apart from `index.html`.
- `src` contains our TypeScript and CSS code. `index.tsx` is the entry-point for our file, and is mandatory.

Running the project

Running the project is as simple as running

```
npm run start
```

This runs the `start` script specified in our `package.json`, and will spawn off a server which reloads the page as we save our files. Typically the server runs at `http://localhost:3000`, but should be automatically opened for you.

This tightens the iteration loop by allowing us to quickly preview changes.

Testing the project

Testing is also just a command away:

```
npm run test
```

This command runs Jest, an incredibly useful testing utility, against all files whose extensions end in `.test.ts` or `.spec.ts`. Like with the `npm run start` command, Jest will automatically run as soon as it detects changes. If you'd like, you can run `npm run start` and `npm run test` side by side so that you can preview changes and test them simultaneously.

Creating a production build

When running the project with `npm run start`, we didn't end up with an optimized build. Typically, we want the code we ship to users to be as fast and small as possible. Certain optimizations like minification can accomplish this, but often take more time. We call builds like this "production" builds (as opposed to development builds).

To run a production build, just run

```
npm run build
```

This will create an optimized JS and CSS build in `./build/static/js` and `./build/static/css` respectively.

You won't need to run a production build most of the time, but it is useful if you need to measure things like the final size of your app.

Creating a component

We're going to write a `Hello` component. The component will take the name of whatever we want to greet (which we'll call `name`), and optionally the number of exclamation marks to trail with (`enthusiasmLevel`).

When we write something like `<Hello name="Daniel" enthusiasmLevel={3} />`, the component should render to something like `<div>Hello Daniel!!!!</div>`. If `enthusiasmLevel` isn't specified, the component should default to showing one exclamation mark. If `enthusiasmLevel` is `0` or negative, it should throw an error.

We'll write a `Hello.tsx`:

```
// src/components/Hello.tsx

import * as React from 'react';

export interface Props {
  name: string;
  enthusiasmLevel?: number;
```

```

}

function Hello({ name, enthusiasmLevel = 1 }: Props) {
  if (enthusiasmLevel <= 0) {
    throw new Error('You could be a little more enthusiastic. :D');
  }

  return (
    <div className="hello">
      <div className="greeting">
        Hello {name + getExclamationMarks(enthusiasmLevel)}
      </div>
    </div>
  );
}

export default Hello;

// helpers

function getExclamationMarks(numChars: number) {
  return Array(numChars + 1).join('!');
}

```

Notice that we defined a type named `Props` that specifies the properties our component will take. `name` is a required `string`, and `enthusiasmLevel` is an optional `number` (which you can tell from the `?` that we wrote out after its name).

We also wrote `Hello` as a stateless function component (an SFC). To be specific, `Hello` is a function that takes a `Props` object, and destructures it. If `enthusiasmLevel` isn't given in our `Props` object, it will default to `1`.

Writing functions is one of two primary [ways React allows us to make components](#)). If we wanted, we *could* have written it out as a class as follows:

```

class Hello extends React.Component<Props, object> {
  render() {
    const { name, enthusiasmLevel = 1 } = this.props;

    if (enthusiasmLevel <= 0) {
      throw new Error('You could be a little more enthusiastic. :D');
    }

    return (
      <div className="hello">
        <div className="greeting">
          Hello {name + getExclamationMarks(enthusiasmLevel)}
        </div>
      </div>
    );
  }
}

```

Classes are useful [when our component instances have some state](#). But we don't really need to think about state in this example - in fact, we specified it as `object` in `React.Component<Props, object>`, so writing an SFC tends to be shorter. Local component state is more useful at the presentational level when creating generic UI elements that can be shared between libraries. For our application's lifecycle, we will revisit how applications manage general state with Redux in a bit.

Now that we've written our component, let's dive into `index.tsx` and replace our render of `<App />` with a render of `<Hello ... />`.

First we'll import it at the top of the file:

```
import Hello from './components/Hello';
```

and then change up our `render` call:

```
ReactDOM.render(
  <Hello name="TypeScript" enthusiasmLevel={10} />,
  document.getElementById('root') as HTMLElement
);
```

Type assertions

One final thing we'll point out in this section is the line `document.getElementById('root') as HTMLElement`. This syntax is called a *type assertion*, sometimes also called a *cast*. This is a useful way of telling TypeScript what the real type of an expression is when you know better than the type checker.

The reason we need to do so in this case is that `getElementById`'s return type is `HTMLElement | null`. Put simply, `getElementById` returns `null` when it can't find an element with a given `id`. We're assuming that `getElementById` will actually succeed, so we need convince TypeScript of that using the `as` syntax.

TypeScript also has a trailing "bang" syntax (`!`), which removes `null` and `undefined` from the prior expression. So we *could* have written `document.getElementById('root')!`, but in this case we wanted to be a bit more explicit.

Adding style

Styling a component with our setup is easy. To style our `Hello` component, we can create a CSS file at `src/components/Hello.css`.

```
.hello {
  text-align: center;
  margin: 20px;
  font-size: 48px;
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif
}

.hello button {
  margin-left: 25px;
  margin-right: 25px;
  font-size: 40px;
  min-width: 50px;
}
```

The tools that `create-react-app` uses (namely, Webpack and various loaders) allow us to just import the stylesheets we're interested in. When our build runs, any imported `.css` files will be concatenated into an output file. So in `src/components/Hello.tsx`, we'll add the following import.

```
import './Hello.css';
```

Writing tests with Jest

We had a certain set of assumptions about our `Hello` component. Let's reiterate what they were:

- When we write something like `<Hello name="Daniel" enthusiasmLevel={3} />`, the component should render to something like `<div>Hello Daniel!!!</div>`.

- If `enthusiasmLevel` isn't specified, the component should default to showing one exclamation mark.
- If `enthusiasmLevel` is `0` or negative, it should throw an error.

We can use these requirements to write a few tests for our components.

But first, let's install Enzyme. [Enzyme](#) is a common tool in the React ecosystem that makes it easier to write tests for how components will behave. By default, our application includes a library called `jsdom` to allow us to simulate the DOM and test its runtime behavior without a browser. Enzyme is similar, but builds on `jsdom` and makes it easier to make certain queries about our components.

Let's install it as a development-time dependency.

```
npm install -D enzyme @types/enzyme react-addons-test-utils
```

Notice we installed packages `enzyme` as well as `@types/enzyme`. The `enzyme` package refers to the package containing JavaScript code that actually gets run, while `@types/enzyme` is a package that contains declaration files (`.d.ts` files) so that TypeScript can understand how you can use Enzyme. You can learn more about `@types` packages [here](#).

We also had to install `react-addons-test-utils`. This is something `enzyme` expects to be installed.

Now that we've got Enzyme set up, let's start writing our test! Let's create a file named `src/components/Hello.test.tsx`, adjacent to our `Hello.tsx` file from earlier.

```
// src/components/Hello.test.tsx

import * as React from 'react';
import * as enzyme from 'enzyme';
import Hello from './Hello';

it('renders the correct text when no enthusiasm level is given', () => {
  const hello = enzyme.shallow(<Hello name='Daniel' />);
  expect(hello.find('.greeting').text()).toEqual('Hello Daniel!');
});

it('renders the correct text with an explicit enthusiasm of 1', () => {
  const hello = enzyme.shallow(<Hello name='Daniel' enthusiasmLevel={1}>);
  expect(hello.find('.greeting').text()).toEqual('Hello Daniel!');
});

it('renders the correct text with an explicit enthusiasm level of 5', () => {
  const hello = enzyme.shallow(<Hello name='Daniel' enthusiasmLevel={5}>);
  expect(hello.find('.greeting').text()).toEqual('Hello Daniel!!!!');
});

it('throws when the enthusiasm level is 0', () => {
  expect(() => {
    enzyme.shallow(<Hello name='Daniel' enthusiasmLevel={0} />);
  }).toThrow();
});

it('throws when the enthusiasm level is negative', () => {
  expect(() => {
    enzyme.shallow(<Hello name='Daniel' enthusiasmLevel={-1}>);
  }).toThrow();
});
```

These tests are extremely basic, but you should be able to get the gist of things.

Adding state management

At this point, if all you're using React for is fetching data once and displaying it, you can consider yourself done. But if you're developing an app that's more interactive, then you may need to add state management.

State management in general

On its own, React is a useful library for creating composable views. However, React doesn't come with any facility for synchronizing data between your application. As far as a React component is concerned, data flows down through its children through the props you specify on each element.

Because React on its own does not provide built-in support for state management, the React community uses libraries like Redux and MobX.

Redux relies on synchronizing data through a centralized and immutable store of data, and updates to that data will trigger a re-render of our application. State is updated in an immutable fashion by sending explicit action messages which must be handled by functions called reducers. Because of the explicit nature, it is often easier to reason about how an action will affect the state of your program.

MobX relies on functional reactive patterns where state is wrapped through observables and passed through as props. Keeping state fully synchronized for any observers is done by simply marking state as observable. As a nice bonus, the library is already written in TypeScript.

There are various merits and tradeoffs to both. Generally Redux tends to see more widespread usage, so for the purposes of this tutorial, we'll focus on adding Redux; however, you should feel encouraged to explore both.

The following section may have a steep learning curve. We strongly suggest you [familiarize yourself with Redux through its documentation](#).

Setting the stage for actions

It doesn't make sense to add Redux unless the state of our application changes. We need a source of actions that will trigger changes to take place. This can be a timer, or something in the UI like a button.

For our purposes, we're going to add two buttons to control the enthusiasm level for our `Hello` component.

Installing Redux

To add Redux, we'll first install `redux` and `react-redux`, as well as their types, as a dependency.

```
npm install -S redux react-redux @types/react-redux
```

In this case we didn't need to install `@types/redux` because Redux already comes with its own definition files (`.d.ts` files).

Defining our app's state

We need to define the shape of the state which Redux will store. For this, we can create a file called `src/types/index.tsx` which will contain definitions for types that we might use throughout the program.

```
// src/types/index.tsx

export interface StoreState {
  languageName: string;
```

```
    enthusiasmLevel: number;
}
```

Our intention is that `languageName` will be the programming language this app was written in (i.e. TypeScript or JavaScript) and `enthusiasmLevel` will vary. When we write our first container, we'll understand why we intentionally made our state slightly different from our props.

Adding actions

Let's start off by creating a set of message types that our app can respond to in `src/constants/index.tsx`.

```
// src/constants/index.tsx

export const INCREMENT_ENTHUSIASM = 'INCREMENT_ENTHUSIASM';
export type INCREMENT_ENTHUSIASM = typeof INCREMENT_ENTHUSIASM;

export const DECREMENT_ENTHUSIASM = 'DECREMENT_ENTHUSIASM';
export type DECREMENT_ENTHUSIASM = typeof DECREMENT_ENTHUSIASM;
```

This `const / type` pattern allows us to use TypeScript's string literal types in an easily accessible and refactorable way.

Next, we'll create a set of actions and functions that can create these actions in `src/actions/index.tsx`.

```
import * as constants from '../constants'

export interface IncrementEnthusiasm {
  type: constants.INCREMENT_ENTHUSIASM;
}

export interface DecrementEnthusiasm {
  type: constants.DECREMENT_ENTHUSIASM;
}

export type EnthusiasmAction = IncrementEnthusiasm | DecrementEnthusiasm;

export function incrementEnthusiasm(): IncrementEnthusiasm {
  return {
    type: constants.INCREMENT_ENTHUSIASM
  }
}

export function decrementEnthusiasm(): DecrementEnthusiasm {
  return {
    type: constants.DECREMENT_ENTHUSIASM
  }
}
```

We've created two types that describe what increment actions and decrement actions should look like. We also created a type (`EnthusiasmAction`) to describe cases where an action could be an increment or a decrement. Finally, we made two functions that actually manufacture the actions which we can use instead of writing out bulky object literals.

There's clearly boilerplate here, so you should feel free to look into libraries like [redux-actions](#) once you've got the hang of things.

Adding a reducer

We're ready to write our first reducer! Reducers are just functions that generate changes by creating modified copies of our application's state, but that have *no side effects*. In other words, they're what we call *pure functions*.

Our reducer will go under `src/reducers/index.tsx`. Its function will be to ensure that increments raise the enthusiasm level by 1, and that decrements reduce the enthusiasm level by 1, but that the level never falls below 1.

```
// src/reducers/index.tsx

import { EnthusiasmAction } from '../actions';
import { StoreState } from '../types/index';
import { INCREMENT_ENTHUSIASM, DECREMENT_ENTHUSIASM } from '../constants/index';

export function enthusiasm(state: StoreState, action: EnthusiasmAction): StoreState {
  switch (action.type) {
    case INCREMENT_ENTHUSIASM:
      return { ...state, enthusiasmLevel: state.enthusiasmLevel + 1 };
    case DECREMENT_ENTHUSIASM:
      return { ...state, enthusiasmLevel: Math.max(1, state.enthusiasmLevel - 1) };
  }
  return state;
}
```

Notice that we're using the *object spread* (`...state`) which allows us to create a shallow copy of our state, while replacing the `enthusiasmLevel`. It's important that the `enthusiasmLevel` property come last, since otherwise it would be overridden by the property in our old state.

You may want to write a few tests for your reducer. Since reducers are pure functions, they can be passed arbitrary data. For every input, reducers can be tested by checking their newly produced state. Consider looking into Jest's `toEqual` method to accomplish this.

Making a container

When writing with Redux, we will often write components as well as containers. Components are often data-agnostic, and work mostly at a presentational level. *Containers* typically wrap components and feed them any data that is necessary to display and modify state. You can read more about this concept on [Dan Abramov's article Presentational and Container Components](#).

First let's update `src/components>Hello.tsx` so that it can modify state. We'll add two optional callback properties to `Props` named `onIncrement` and `onDecrement`:

```
export interface Props {
  name: string;
  enthusiasmLevel?: number;
  onIncrement?: () => void;
  onDecrement?: () => void;
}
```

Then we'll bind those callbacks to two new buttons that we'll add into our component.

```
function Hello({ name, enthusiasmLevel = 1, onIncrement, onDecrement }: Props) {
  if (enthusiasmLevel <= 0) {
    throw new Error('You could be a little more enthusiastic. :D');
  }

  return (
    <div className="hello">
      <div className="greeting">
        Hello {name + getExclamationMarks(enthusiasmLevel)}
      </div>
    </div>
  );
}
```

```

        <div>
          <button onClick={onDecrement}>-</button>
          <button onClick={onIncrement}>+</button>
        </div>
      );
    }
  
```

In general, it'd be a good idea to write a few tests for `onIncrement` and `onDecrement` being triggered when their respective buttons are clicked. Give it a shot to get the hang of writing tests for your components.

Now that our component is updated, we're ready to wrap it into a container. Let's create a file named `src/containers/Hello.tsx` and start off with the following imports.

```

import Hello from '../components/Hello';
import * as actions from '../actions/';
import { StoreState } from '../types/index';
import { connect, Dispatch } from 'react-redux';

```

The real two key pieces here are the original `Hello` component as well as the `connect` function from `react-redux`. `connect` will be able to actually take our original `Hello` component and turn it into a container using two functions:

- `mapStateToProps` which massages the data from the current store to part of the shape that our component needs.
- `mapDispatchToProps` which uses creates callback props to pump actions to our store using a given `dispatch` function.

If we recall, our application state consists of two properties: `languageName` and `enthusiasmLevel`. Our `Hello` component, on the other hand, expected a `name` and an `enthusiasmLevel`. `mapStateToProps` will get the relevant data from the store, and adjust it if necessary, for our component's props. Let's go ahead and write that.

```

export function mapStateToProps({ enthusiasmLevel, languageName }: StoreState) {
  return {
    enthusiasmLevel,
    name: languageName,
  }
}

```

Note that `mapStateToProps` only creates 2 out of 4 of the properties a `Hello` component expects. Namely, we still want to pass in the `onIncrement` and `onDecrement` callbacks. `mapDispatchToProps` is a function that takes a dispatcher function. This dispatcher function can pass actions into our store to make updates, so we can create a pair of callbacks that will call the dispatcher as necessary.

```

export function mapDispatchToProps(dispatch: Dispatch<actions.EnthusiasmAction>) {
  return {
    onIncrement: () => dispatch(actions.incrementEnthusiasm()),
    onDecrement: () => dispatch(actions.decrementEnthusiasm()),
  }
}

```

Finally, we're ready to call `connect`. `connect` will first take `mapStateToProps` and `mapDispatchToProps`, and then return another function that we can use to wrap our component. Our resulting container is defined with the following line of code:

```

export default connect(mapStateToProps, mapDispatchToProps)(Hello);

```

When we're finished, our file should look like this:

```
// src/containers/Hello.tsx

import Hello from '../components/Hello';
import * as actions from '../actions/';
import { StoreState } from '../types/index';
import { connect, Dispatch } from 'react-redux';

export function mapStateToProps({ enthusiasmLevel, languageName }: StoreState) {
  return {
    enthusiasmLevel,
    name: languageName,
  }
}

export function mapDispatchToProps(dispatch: Dispatch<actions.EnthusiasmAction>) {
  return {
    onIncrement: () => dispatch(actions.incrementEnthusiasm()),
    onDecrement: () => dispatch(actions.decrementEnthusiasm()),
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(Hello);
```

Creating a store

Let's go back to `src/index.tsx`. To put this all together, we need to create a store with an initial state, and set it up with all of our reducers.

```
import { createStore } from 'redux';
import { enthusiasm } from './reducers/index';
import { StoreState } from './types/index';

const store = createStore<StoreState>(enthusiasm, {
  enthusiasmLevel: 1,
  languageName: 'TypeScript',
});
```

`store` is, as you might've guessed, our central store for our application's global state.

Next, we're going to swap our use of `./src/components>Hello` with `./src/containers>Hello` and use `react-redux's Provider` to wire up our props with our container. We'll import each:

```
import Hello from './containers/Hello';
import { Provider } from 'react-redux';
```

and pass our `store` through to the `Provider`'s attributes:

```
ReactDOM.render(
  <Provider store={store}>
    <Hello />
  </Provider>,
  document.getElementById('root') as HTMLElement
);
```

Notice that `Hello` no longer needs props, since we used our `connect` function to adapt our application's state for our wrapped `Hello` component's props.

Ejecting

If at any point, you feel like there are certain customizations that the create-react-app setup has made difficult, you can always opt-out and get the various configuration options you need. For example, if you'd like to add a Webpack plugin, it might be necessary to take advantage of the "eject" functionality that create-react-app provides.

Simply run

```
npm run eject
```

and you should be good to go!

As a heads up, you may want to commit all your work before running an eject. You cannot undo an eject command, so opting out is permanent unless you can recover from a commit prior to running an eject.

Next steps

create-react-app comes with a lot of great stuff. Much of it is documented in the default `README.md` that was generated for our project, so give that a quick read.

If you still want to learn more about Redux, you can [check out the official website](#) for documentation. The same goes for [MobX](#).

If you want to eject at some point, you may need to know a little bit more about Webpack. You can check out our [React & Webpack walkthrough here](#).

At some point you might need routing. There are several solutions, but [react-router](#) is probably the most popular for Redux projects, and is often used in conjunction with [react-router-redux](#).

Let's get started by building a simple web application with TypeScript.

Installing TypeScript

There are two main ways to get the TypeScript tools:

- Via npm (the Node.js package manager)
- By installing TypeScript's Visual Studio plugins

Visual Studio 2017 and Visual Studio 2015 Update 3 include TypeScript by default. If you didn't install TypeScript with Visual Studio, you can still [download it](#).

For NPM users:

```
> npm install -g typescript
```

Building your first TypeScript file

In your editor, type the following JavaScript code in `greeter.ts`:

```
function greeter(person) {
    return "Hello, " + person;
}

let user = "Jane User";

document.body.innerHTML = greeter(user);
```

Compiling your code

We used a `.ts` extension, but this code is just JavaScript. You could have copy/pasted this straight out of an existing JavaScript app.

At the command line, run the TypeScript compiler:

```
tsc greeter.ts
```

The result will be a file `greeter.js` which contains the same JavaScript that you fed in. We're up and running using TypeScript in our JavaScript app!

Now we can start taking advantage of some of the new tools TypeScript offers. Add a `: string` type annotation to the 'person' function argument as shown here:

```
function greeter(person: string) {
    return "Hello, " + person;
}

let user = "Jane User";

document.body.innerHTML = greeter(user);
```

Type annotations

Type annotations in TypeScript are lightweight ways to record the intended contract of the function or variable. In this case, we intend the greeter function to be called with a single string parameter. We can try changing the call greeter to pass an array instead:

```
function greeter(person: string) {
    return "Hello, " + person;
}

let user = [0, 1, 2];

document.body.innerHTML = greeter(user);
```

Re-compiling, you'll now see an error:

```
error TS2345: Argument of type 'number[]' is not assignable to parameter of type 'string'.
```

Similarly, try removing all the arguments to the greeter call. TypeScript will let you know that you have called this function with an unexpected number of parameters. In both cases, TypeScript can offer static analysis based on both the structure of your code, and the type annotations you provide.

Notice that although there were errors, the `greeter.js` file is still created. You can use TypeScript even if there are errors in your code. But in this case, TypeScript is warning that your code will likely not run as expected.

Interfaces

Let's develop our sample further. Here we use an interface that describes objects that have a `firstName` and `lastName` field. In TypeScript, two types are compatible if their internal structure is compatible. This allows us to implement an interface just by having the shape the interface requires, without an explicit `implements` clause.

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

function greeter(person: Person) {
    return "Hello, " + person.firstName + " " + person.lastName;
}

let user = { firstName: "Jane", lastName: "User" };

document.body.innerHTML = greeter(user);
```

Classes

Finally, let's extend the example one last time with classes. TypeScript supports new features in JavaScript, like support for class-based object-oriented programming.

Here we're going to create a `Student` class with a constructor and a few public fields. Notice that classes and interfaces play well together, letting the programmer decide on the right level of abstraction.

Also of note, the use of `public` on arguments to the constructor is a shorthand that allows us to automatically create properties with that name.

```
class Student {
    fullName: string;
```

```

constructor(public firstName: string, public middleInitial: string, public lastName: string) {
    this.fullName = firstName + " " + middleInitial + " " + lastName;
}
}

interface Person {
    firstName: string;
    lastName: string;
}

function greeter(person : Person) {
    return "Hello, " + person.firstName + " " + person.lastName;
}

let user = new Student("Jane", "M.", "User");

document.body.innerHTML = greeter(user);

```

Re-run `tsc greeter.ts` and you'll see the generated JavaScript is the same as the earlier code. Classes in TypeScript are just a shorthand for the same prototype-based OO that is frequently used in JavaScript.

Running your TypeScript web app

Now type the following in `greeter.html`:

```

<!DOCTYPE html>
<html>
    <head><title>TypeScript Greeter</title></head>
    <body>
        <script src="greeter.js"></script>
    </body>
</html>

```

Open `greeter.html` in the browser to run your first simple TypeScript web application!

Optional: Open `greeter.ts` in Visual Studio, or copy the code into the TypeScript playground. You can hover over identifiers to see their types. Notice that in some cases these types are inferred automatically for you. Re-type the last line, and see completion lists and parameter help based on the types of the DOM elements. Put your cursor on the reference to the greeter function, and hit F12 to go to its definition. Notice, too, that you can right-click on a symbol and use refactoring to rename it.

The type information provided works together with the tools to work with JavaScript at application scale. For more examples of what's possible in TypeScript, see the Samples section of the website.



Introduction

For programs to be useful, we need to be able to work with some of the simplest units of data: numbers, strings, structures, boolean values, and the like. In TypeScript, we support much the same types as you would expect in JavaScript, with a convenient enumeration type thrown in to help things along.

Boolean

The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a `boolean` value.

```
let isDone: boolean = false;
```

Number

As in JavaScript, all numbers in TypeScript are floating point values. These floating point numbers get the type `number`. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals introduced in ECMAScript 2015.

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

String

Another fundamental part of creating programs in JavaScript for webpages and servers alike is working with textual data. As in other languages, we use the type `string` to refer to these textual datatypes. Just like JavaScript, TypeScript also uses double quotes (`"`) or single quotes (`'`) to surround string data.

```
let color: string = "blue";
color = 'red';
```

You can also use *template strings*, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (```) character, and embedded expressions are of the form `${ expr }` .

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`;
```

This is equivalent to declaring `sentence` like so:

```
let sentence: string = "Hello, my name is " + fullName + ".\n\n" +
"I'll be " + (age + 1) + " years old next month.";
```

Array

TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by `[]` to denote an array of that element type:

```
let list: number[] = [1, 2, 3];
```

The second way uses a generic array type, `Array<elemType>`:

```
let list: Array<number> = [1, 2, 3];
```

Tuple

Tuple types allow you to express an array where the type of a fixed number of elements is known, but need not be the same. For example, you may want to represent a value as a pair of a `string` and a `number`:

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

When accessing an element with a known index, the correct type is retrieved:

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'
```

When accessing an element outside the set of known indices, a union type is used instead:

```
x[3] = "world"; // OK, 'string' can be assigned to 'string | number'

console.log(x[5].toString()); // OK, 'string' and 'number' both have 'toString'

x[6] = true; // Error, 'boolean' isn't 'string | number'
```

Union types are an advanced topic that we'll cover in a later chapter.

Enum

A helpful addition to the standard set of datatypes from JavaScript is the `enum`. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

By default, enums begin numbering their members starting at `0`. You can change this by manually setting the value of one of its members. For example, we can start the previous example at `1` instead of `0`:

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green;
```

Or, even manually set all the values in the enum:

```
enum Color {Red = 1, Green = 2, Blue = 4}
let c: Color = Color.Green;
```

A handy feature of enums is that you can also go from a numeric value to the name of that value in the enum. For example, if we had the value `2` but weren't sure what that mapped to in the `Color` enum above, we could look up the corresponding name:

```
enum Color {Red = 1, Green, Blue}
let colorName: string = Color[2];

console.log(colorName); // Displays 'Green' as its value is 2 above
```

Any

We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content, e.g. from the user or a 3rd party library. In these cases, we want to opt-out of type-checking and let the values pass through compile-time checks. To do so, we label these with the `any` type:

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

The `any` type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation. You might expect `Object` to play a similar role, as it does in other languages. But variables of type `Object` only allow you to assign any value to them - you can't call arbitrary methods on them, even ones that actually exist:

```
let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.
```

The `any` type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:

```
let list: any[] = [1, true, "free"];

list[1] = 100;
```

Void

`void` is a little like the opposite of `any`: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

```
function warnUser(): void {
    console.log("This is my warning message");
}
```

Declaring variables of type `void` is not useful because you can only assign `undefined` or `null` to them:

```
let unusable: void = undefined;
```

Null and Undefined

In TypeScript, both `undefined` and `null` actually have their own types named `undefined` and `null` respectively. Much like `void`, they're not extremely useful on their own:

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

By default `null` and `undefined` are subtypes of all other types. That means you can assign `null` and `undefined` to something like `number`.

However, when using the `--strictNullChecks` flag, `null` and `undefined` are only assignable to `void` and their respective types. This helps avoid *many* common errors. In cases where you want to pass in either a `string` or `null` or `undefined`, you can use the union type `string | null | undefined`. Once again, more on union types later on.

As a note: we encourage the use of `--strictNullChecks` when possible, but for the purposes of this handbook, we will assume it is turned off.

Never

The `never` type represents the type of values that never occur. For instance, `never` is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns; Variables also acquire the type `never` when narrowed by any type guards that can never be true.

The `never` type is a subtype of, and assignable to, every type; however, no type is a subtype of, or assignable to, `never` (except `never` itself). Even `any` isn't assignable to `never`.

Some examples of functions returning `never`:

```
// Function returning never must have unreachable end point
function error(message: string): never {
    throw new Error(message);
}

// Inferred return type is never
function fail() {
    return error("Something failed");
}

// Function returning never must have unreachable end point
function infiniteLoop(): never {
    while (true) {
    }
}
```

Object

`object` is a type that represents the non-primitive type, i.e. any thing that is not `number`, `string`, `boolean`, `symbol`, `null`, or `undefined`.

With `object` type, APIs like `Object.create` can be better represented. For example:

```
declare function create(o: object | null): void;

create({ prop: 0 }); // OK
create(null); // OK

create(42); // Error
create("string"); // Error
create(false); // Error
create(undefined); // Error
```

Type assertions

Sometimes you'll end up in a situation where you'll know more about a value than TypeScript does. Usually this will happen when you know the type of some entity could be more specific than its current type.

Type assertions are a way to tell the compiler "trust me, I know what I'm doing." A type assertion is like a type cast in other languages, but performs no special checking or restructuring of data. It has no runtime impact, and is used purely by the compiler. TypeScript assumes that you, the programmer, have performed any special checks that you need.

Type assertions have two forms. One is the "angle-bracket" syntax:

```
let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;
```

And the other is the `as`-syntax:

```
let someValue: any = "this is a string";

let strLength: number = (someValue as string).length;
```

The two samples are equivalent. Using one over the other is mostly a choice of preference; however, when using TypeScript with JSX, only `as`-style assertions are allowed.

A note about `let`

You may've noticed that so far, we've been using the `let` keyword instead of JavaScript's `var` keyword which you might be more familiar with. The `let` keyword is actually a newer JavaScript construct that TypeScript makes available. We'll discuss the details later, but many common problems in JavaScript are alleviated by using `let`, so you should use it instead of `var` whenever possible.

Variable Declarations

`let` and `const` are two relatively new types of variable declarations in JavaScript. As we mentioned earlier, `let` is similar to `var` in some respects, but allows users to avoid some of the common "gotchas" that users run into in JavaScript. `const` is an augmentation of `let` in that it prevents re-assignment to a variable.

With TypeScript being a superset of JavaScript, the language naturally supports `let` and `const`. Here we'll elaborate more on these new declarations and why they're preferable to `var`.

If you've used JavaScript offhandedly, the next section might be a good way to refresh your memory. If you're intimately familiar with all the quirks of `var` declarations in JavaScript, you might find it easier to skip ahead.

var declarations

Declaring a variable in JavaScript has always traditionally been done with the `var` keyword.

```
var a = 10;
```

As you might've figured out, we just declared a variable named `a` with the value `10`.

We can also declare a variable inside of a function:

```
function f() {
    var message = "Hello, world!";
    return message;
}
```

and we can also access those same variables within other functions:

```
function f() {
    var a = 10;
    return function g() {
        var b = a + 1;
        return b;
    }
}

var g = f();
g(); // returns '11'
```

In this above example, `g` captured the variable `a` declared in `f`. At any point that `g` gets called, the value of `a` will be tied to the value of `a` in `f`. Even if `g` is called once `f` is done running, it will be able to access and modify `a`.

```
function f() {
    var a = 1;

    a = 2;
    var b = g();
    a = 3;

    return b;

    function g() {
```

```

        return a;
    }
}

f(); // returns '2'

```

Scoping rules

`var` declarations have some odd scoping rules for those used to other languages. Take the following example:

```

function f(shouldInitialize: boolean) {
    if (shouldInitialize) {
        var x = 10;
    }

    return x;
}

f(true); // returns '10'
f(false); // returns 'undefined'

```

Some readers might do a double-take at this example. The variable `x` was declared *within the `if` block*, and yet we were able to access it from outside that block. That's because `var` declarations are accessible anywhere within their containing function, module, namespace, or global scope - all which we'll go over later on - regardless of the containing block. Some people call this `var`-scoping or `function-scoping`. Parameters are also function scoped.

These scoping rules can cause several types of mistakes. One problem they exacerbate is the fact that it is not an error to declare the same variable multiple times:

```

function sumMatrix(matrix: number[][]): number {
    var sum = 0;
    for (var i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (var i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }

    return sum;
}

```

Maybe it was easy to spot out for some, but the inner `for`-loop will accidentally overwrite the variable `i` because `i` refers to the same function-scoped variable. As experienced developers know by now, similar sorts of bugs slip through code reviews and can be an endless source of frustration.

Variable capturing quirks

Take a quick second to guess what the output of the following snippet is:

```

for (var i = 0; i < 10; i++) {
    setTimeout(function() { console.log(i); }, 100 * i);
}

```

For those unfamiliar, `setTimeout` will try to execute a function after a certain number of milliseconds (though waiting for anything else to stop running).

Ready? Take a look:

```
10  
10  
10  
10  
10  
10  
10  
10  
10  
10
```

Many JavaScript developers are intimately familiar with this behavior, but if you're surprised, you're certainly not alone. Most people expect the output to be

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

Remember what we mentioned earlier about variable capturing? Every function expression we pass to `setTimeout` actually refers to the same `i` from the same scope.

Let's take a minute to consider what that means. `setTimeout` will run a function after some number of milliseconds, *but only* after the `for` loop has stopped executing; By the time the `for` loop has stopped executing, the value of `i` is `10`. So each time the given function gets called, it will print out `10`!

A common work around is to use an IIFE - an Immediately Invoked Function Expression - to capture `i` at each iteration:

```
for (var i = 0; i < 10; i++) {  
    // capture the current state of 'i'  
    // by invoking a function with its current value  
    (function(i) {  
        setTimeout(function() { console.log(i); }, 100 * i);  
    })(i);  
}
```

This odd-looking pattern is actually pretty common. The `i` in the parameter list actually shadows the `i` declared in the `for` loop, but since we named them the same, we didn't have to modify the loop body too much.

let declarations

By now you've figured out that `var` has some problems, which is precisely why `let` statements were introduced. Apart from the keyword used, `let` statements are written the same way `var` statements are.

```
let hello = "Hello!";
```

The key difference is not in the syntax, but in the semantics, which we'll now dive into.

Block-scoping

When a variable is declared using `let`, it uses what some call *lexical-scoping* or *block-scoping*. Unlike variables declared with `var` whose scopes leak out to their containing function, block-scoped variables are not visible outside of their nearest containing block or `for`-loop.

```
function f(input: boolean) {
    let a = 100;

    if (input) {
        // Still okay to reference 'a'
        let b = a + 1;
        return b;
    }

    // Error: 'b' doesn't exist here
    return b;
}
```

Here, we have two local variables `a` and `b`. `a`'s scope is limited to the body of `f` while `b`'s scope is limited to the containing `if` statement's block.

Variables declared in a `catch` clause also have similar scoping rules.

```
try {
    throw "oh no!";
}
catch (e) {
    console.log("Oh well.");
}

// Error: 'e' doesn't exist here
console.log(e);
```

Another property of block-scoped variables is that they can't be read or written to before they're actually declared. While these variables are "present" throughout their scope, all points up until their declaration are part of their *temporal dead zone*. This is just a sophisticated way of saying you can't access them before the `let` statement, and luckily TypeScript will let you know that.

```
a++; // illegal to use 'a' before it's declared;
let a;
```

Something to note is that you can still *capture* a block-scoped variable before it's declared. The only catch is that it's illegal to call that function before the declaration. If targeting ES2015, a modern runtime will throw an error; however, right now TypeScript is permissive and won't report this as an error.

```
function foo() {
    // okay to capture 'a'
    return a;
}

// illegal call 'foo' before 'a' is declared
// runtimes should throw an error here
foo();

let a;
```

For more information on temporal dead zones, see relevant content on the [Mozilla Developer Network](#).

Re-declarations and Shadowing

With `var` declarations, we mentioned that it didn't matter how many times you declared your variables; you just got one.

```
function f(x) {
    var x;
    var x;

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

In the above example, all declarations of `x` actually refer to the *same* `x`, and this is perfectly valid. This often ends up being a source of bugs. Thankfully, `let` declarations are not as forgiving.

```
let x = 10;
let x = 20; // error: can't re-declare 'x' in the same scope
```

The variables don't necessarily need to both be block-scoped for TypeScript to tell us that there's a problem.

```
function f(x) {
    let x = 100; // error: interferes with parameter declaration
}

function g() {
    let x = 100;
    var x = 100; // error: can't have both declarations of 'x'
}
```

That's not to say that block-scoped variable can never be declared with a function-scoped variable. The block-scoped variable just needs to be declared within a distinctly different block.

```
function f(condition, x) {
    if (condition) {
        let x = 100;
        return x;
    }

    return x;
}

f(false, 0); // returns '0'
f(true, 0); // returns '100'
```

The act of introducing a new name in a more nested scope is called *shadowing*. It is a bit of a double-edged sword in that it can introduce certain bugs on its own in the event of accidental shadowing, while also preventing certain bugs. For instance, imagine we had written our earlier `sumMatrix` function using `let` variables.

```
function sumMatrix(matrix: number[][]) {
    let sum = 0;
    for (let i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (let i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }
}
```

```
    return sum;
}
```

This version of the loop will actually perform the summation correctly because the inner loop's `i` shadows `i` from the outer loop.

Shadowing should *usually* be avoided in the interest of writing clearer code. While there are some scenarios where it may be fitting to take advantage of it, you should use your best judgement.

Block-scoped variable capturing

When we first touched on the idea of variable capturing with `var` declaration, we briefly went into how variables act once captured. To give a better intuition of this, each time a scope is run, it creates an "environment" of variables. That environment and its captured variables can exist even after everything within its scope has finished executing.

```
function theCityThatAlwaysSleeps() {
  let getCity;

  if (true) {
    let city = "Seattle";
    getCity = function() {
      return city;
    }
  }

  return getCity();
}
```

Because we've captured `city` from within its environment, we're still able to access it despite the fact that the `if` block finished executing.

Recall that with our earlier `setTimeout` example, we ended up needing to use an IIFE to capture the state of a variable for every iteration of the `for` loop. In effect, what we were doing was creating a new variable environment for our captured variables. That was a bit of a pain, but luckily, you'll never have to do that again in TypeScript.

`let` declarations have drastically different behavior when declared as part of a loop. Rather than just introducing a new environment to the loop itself, these declarations sort of create a new scope *per iteration*. Since this is what we were doing anyway with our IIFE, we can change our old `setTimeout` example to just use a `let` declaration.

```
for (let i = 0; i < 10 ; i++) {
  setTimeout(function() { console.log(i); }, 100 * i);
}
```

and as expected, this will print out

```
0
1
2
3
4
5
6
7
8
9
```

const declarations

`const` declarations are another way of declaring variables.

```
const numLivesForCat = 9;
```

They are like `let` declarations but, as their name implies, their value cannot be changed once they are bound. In other words, they have the same scoping rules as `let`, but you can't re-assign to them.

This should not be confused with the idea that the values they refer to are *immutable*.

```
const numLivesForCat = 9;
const kitty = {
  name: "Aurora",
  numLives: numLivesForCat,
}

// Error
kitty = {
  name: "Danielle",
  numLives: numLivesForCat
};

// all "okay"
kitty.name = "Rory";
kitty.name = "Kitty";
kitty.name = "Cat";
kitty.numLives--;
```

Unless you take specific measures to avoid it, the internal state of a `const` variable is still modifiable. Fortunately, TypeScript allows you to specify that members of an object are `readonly`. The [chapter on Interfaces](#) has the details.

let vs. const

Given that we have two types of declarations with similar scoping semantics, it's natural to find ourselves asking which one to use. Like most broad questions, the answer is: it depends.

Applying the [principle of least privilege](#), all declarations other than those you plan to modify should use `const`. The rationale is that if a variable didn't need to get written to, others working on the same codebase shouldn't automatically be able to write to the object, and will need to consider whether they really need to reassign to the variable. Using `const` also makes code more predictable when reasoning about flow of data.

Use your best judgement, and if applicable, consult the matter with the rest of your team.

The majority of this handbook uses `let` declarations.

Destructuring

Another ECMAScript 2015 feature that TypeScript has is destructuring. For a complete reference, see the article on [the Mozilla Developer Network](#). In this section, we'll give a short overview.

Array destructuring

The simplest form of destructuring is array destructuring assignment:

```
let input = [1, 2];
let [first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2
```

This creates two new variables named `first` and `second`. This is equivalent to using indexing, but is much more convenient:

```
first = input[0];
second = input[1];
```

Destructuring works with already-declared variables as well:

```
// swap variables
[first, second] = [second, first];
```

And with parameters to a function:

```
function f([first, second]: [number, number]) {
  console.log(first);
  console.log(second);
}
f([1, 2]);
```

You can create a variable for the remaining items in a list using the syntax `...rest`:

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(first); // outputs 1
console.log(rest); // outputs [ 2, 3, 4 ]
```

Of course, since this is JavaScript, you can just ignore trailing elements you don't care about:

```
let [first] = [1, 2, 3, 4];
console.log(first); // outputs 1
```

Or other elements:

```
let [, second, , fourth] = [1, 2, 3, 4];
```

Object destructuring

You can also destructure objects:

```
let o = {
  a: "foo",
  b: 12,
  c: "bar"
};
let { a, b } = o;
```

This creates new variables `a` and `b` from `o.a` and `o.b`. Notice that you can skip `c` if you don't need it.

Like array destructuring, you can have assignment without declaration:

```
({ a, b } = { a: "baz", b: 101 });
```

Notice that we had to surround this statement with parentheses. JavaScript normally parses a `{` as the start of block.

You can create a variable for the remaining items in an object using the syntax `... :`

```
let { a, ...passthrough } = o;
let total = passthrough.b + passthrough.c.length;
```

Property renaming

You can also give different names to properties:

```
let { a: newName1, b: newName2 } = o;
```

Here the syntax starts to get confusing. You can read `a: newName1` as "`a` as `newName1`". The direction is left-to-right, as if you had written:

```
let newName1 = o.a;
let newName2 = o.b;
```

Confusingly, the colon here does *not* indicate the type. The type, if you specify it, still needs to be written after the entire destructuring:

```
let { a, b }: { a: string, b: number } = o;
```

Default values

Default values let you specify a default value in case a property is undefined:

```
function keepWholeObject(wholeObject: { a: string, b?: number }) {
  let { a, b = 1001 } = wholeObject;
}
```

`keepWholeObject` now has a variable for `wholeObject` as well as the properties `a` and `b`, even if `b` is undefined.

Function declarations

Destructuring also works in function declarations. For simple cases this is straightforward:

```
type C = { a: string, b?: number }
function f({ a, b }: C): void {
  // ...
}
```

But specifying defaults is more common for parameters, and getting defaults right with destructuring can be tricky. First of all, you need to remember to put the pattern before the default value.

```
function f({ a="", b=0 } = {}): void {
  // ...
```

```
}
```

The snippet above is an example of type inference, explained later in the handbook.

Then, you need to remember to give a default for optional properties on the destructured property instead of the main initializer. Remember that `c` was defined with `b` optional:

```
function f({ a, b = 0 } = { a: "" }): void {
    // ...
}

f({ a: "yes" }); // ok, default b = 0
f(); // ok, default to { a: "" }, which then defaults b = 0
f({}); // error, 'a' is required if you supply an argument
```

Use destructuring with care. As the previous example demonstrates, anything but the simplest destructuring expression is confusing. This is especially true with deeply nested destructuring, which gets *really* hard to understand even without piling on renaming, default values, and type annotations. Try to keep destructuring expressions small and simple. You can always write the assignments that destructuring would generate yourself.

Spread

The spread operator is the opposite of destructuring. It allows you to spread an array into another array, or an object into another object. For example:

```
let first = [1, 2];
let second = [3, 4];
let bothPlus = [0, ...first, ...second, 5];
```

This gives `bothPlus` the value `[0, 1, 2, 3, 4, 5]`. Spreading creates a shallow copy of `first` and `second`. They are not changed by the spread.

You can also spread objects:

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };
let search = { ...defaults, food: "rich" };
```

Now `search` is `{ food: "rich", price: "$$", ambiance: "noisy" }`. Object spreading is more complex than array spreading. Like array spreading, it proceeds from left-to-right, but the result is still an object. This means that properties that come later in the spread object overwrite properties that come earlier. So if we modify the previous example to spread at the end:

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };
let search = { food: "rich", ...defaults };
```

Then the `food` property in `defaults` overwrites `food: "rich"`, which is not what we want in this case.

Object spread also has a couple of other surprising limits. First, it only includes an objects' own, enumerable properties. Basically, that means you lose methods when you spread instances of an object:

```
class C {
  p = 12;
  m() {
  }
}
```

```
let c = new C();
let clone = { ...c };
clone.p; // ok
clone.m(); // error!
```

Second, the Typescript compiler doesn't allow spreads of type parameters from generic functions. That feature is expected in future versions of the language.

Introduction

One of TypeScript's core principles is that type-checking focuses on the *shape* that values have. This is sometimes called "duck typing" or "structural subtyping". In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.

Our First Interface

The easiest way to see how interfaces work is to start with a simple example:

```
function printLabel(labelledObj: { label: string }) {
    console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

The type-checker checks the call to `printLabel`. The `printLabel` function has a single parameter that requires that the object passed in has a property called `label` of type `string`. Notice that our object actually has more properties than this, but the compiler only checks that *at least* the ones required are present and match the types required. There are some cases where TypeScript isn't as lenient, which we'll cover in a bit.

We can write the same example again, this time using an interface to describe the requirement of having the `label` property that is a string:

```
interface LabelledValue {
    label: string;
}

function printLabel(labelledObj: LabelledValue) {
    console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

The interface `LabelledValue` is a name we can now use to describe the requirement in the previous example. It still represents having a single property called `label` that is of type `string`. Notice we didn't have to explicitly say that the object we pass to `printLabel` implements this interface like we might have to in other languages. Here, it's only the shape that matters. If the object we pass to the function meets the requirements listed, then it's allowed.

It's worth pointing out that the type-checker does not require that these properties come in any sort of order, only that the properties the interface requires are present and have the required type.

Optional Properties

Not all properties of an interface may be required. Some exist under certain conditions or may not be there at all. These optional properties are popular when creating patterns like "option bags" where you pass an object to a function that only has a couple of properties filled in.

Here's an example of this pattern:

```

interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): {color: string; area: number} {
  let newSquare = {color: "white", area: 100};
  if (config.color) {
    newSquare.color = config.color;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({color: "black"});

```

Interfaces with optional properties are written similar to other interfaces, with each optional property denoted by a `?` at the end of the property name in the declaration.

The advantage of optional properties is that you can describe these possibly available properties while still also preventing use of properties that are not part of the interface. For example, had we mistyped the name of the `color` property in `createSquare`, we would get an error message letting us know:

```

interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  let newSquare = {color: "white", area: 100};
  if (config.clor) {
    // Error: Property 'clor' does not exist on type 'SquareConfig'
    newSquare.color = config.clor;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({color: "black"});

```

Readonly properties

Some properties should only be modifiable when an object is first created. You can specify this by putting `readonly` before the name of the property:

```

interface Point {
  readonly x: number;
  readonly y: number;
}

```

You can construct a `Point` by assigning an object literal. After the assignment, `x` and `y` can't be changed.

```

let p1: Point = { x: 10, y: 20 };
p1.x = 5; // error!

```

TypeScript comes with a `ReadonlyArray<T>` type that is the same as `Array<T>` with all mutating methods removed, so you can make sure you don't change your arrays after creation:

```
let a: number[] = [1, 2, 3, 4];
let ro: ReadonlyArray<number> = a;
ro[0] = 12; // error!
ro.push(5); // error!
ro.length = 100; // error!
a = ro; // error!
```

On the last line of the snippet you can see that even assigning the entire `ReadonlyArray` back to a normal array is illegal. You can still override it with a type assertion, though:

```
a = ro as number[];
```

readonly vs const

The easiest way to remember whether to use `readonly` or `const` is to ask whether you're using it on a variable or a property. Variables use `const` whereas properties use `readonly`.

Excess Property Checks

In our first example using interfaces, TypeScript lets us pass `{ size: number; label: string; }` to something that only expected a `{ label: string; }`. We also just learned about optional properties, and how they're useful when describing so-called "option bags".

However, combining the two naively would let you shoot yourself in the foot the same way you might in JavaScript. For example, taking our last example using `createSquare`:

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  // ...
}

let mySquare = createSquare({ colour: "red", width: 100 });
```

Notice the given argument to `createSquare` is spelled `colour` instead of `color`. In plain JavaScript, this sort of thing fails silently.

You could argue that this program is correctly typed, since the `width` properties are compatible, there's no `color` property present, and the extra `colour` property is insignificant.

However, TypeScript takes the stance that there's probably a bug in this code. Object literals get special treatment and undergo *excess property checking* when assigning them to other variables, or passing them as arguments. If an object literal has any properties that the "target type" doesn't have, you'll get an error.

```
// error: 'colour' not expected in type 'SquareConfig'
let mySquare = createSquare({ colour: "red", width: 100 });
```

Getting around these checks is actually really simple. The easiest method is to just use a type assertion:

```
let mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig);
```

However, a better approach might be to add a string index signature if you're sure that the object can have some extra properties that are used in some special way. If `SquareConfig` can have `color` and `width` properties with the above types, but could also have any number of other properties, then we could define it like so:

```
interface SquareConfig {
  color?: string;
  width?: number;
  [propName: string]: any;
}
```

We'll discuss index signatures in a bit, but here we're saying a `SquareConfig` can have any number of properties, and as long as they aren't `color` or `width`, their types don't matter.

One final way to get around these checks, which might be a bit surprising, is to assign the object to another variable: Since `squareOptions` won't undergo excess property checks, the compiler won't give you an error.

```
let squareOptions = { colour: "red", width: 100 };
let mySquare = createSquare(squareOptions);
```

Keep in mind that for simple code like above, you probably shouldn't be trying to "get around" these checks. For more complex object literals that have methods and hold state, you might need to keep these techniques in mind, but a majority of excess property errors are actually bugs. That means if you're running into excess property checking problems for something like option bags, you might need to revise some of your type declarations. In this instance, if it's okay to pass an object with both a `color` or `colour` property to `createSquare`, you should fix up the definition of `SquareConfig` to reflect that.

Function Types

Interfaces are capable of describing the wide range of shapes that JavaScript objects can take. In addition to describing an object with properties, interfaces are also capable of describing function types.

To describe a function type with an interface, we give the interface a call signature. This is like a function declaration with only the parameter list and return type given. Each parameter in the parameter list requires both name and type.

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
```

Once defined, we can use this function type interface like we would other interfaces. Here, we show how you can create a variable of a function type and assign it a function value of the same type.

```
let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
  let result = source.search(subString);
  return result > -1;
}
```

For function types to correctly type-check, the names of the parameters do not need to match. We could have, for example, written the above example like this:

```
let mySearch: SearchFunc;
```

```
mySearch = function(src: string, sub: string): boolean {
    let result = src.search(sub);
    return result > -1;
}
```

Function parameters are checked one at a time, with the type in each corresponding parameter position checked against each other. If you do not want to specify types at all, TypeScript's contextual typing can infer the argument types since the function value is assigned directly to a variable of type `SearchFunc`. Here, also, the return type of our function expression is implied by the values it returns (here `false` and `true`). Had the function expression returned numbers or strings, the type-checker would have warned us that return type doesn't match the return type described in the `SearchFunc` interface.

```
let mySearch: SearchFunc;
mySearch = function(src, sub) {
    let result = src.search(sub);
    return result > -1;
}
```

Indexable Types

Similarly to how we can use interfaces to describe function types, we can also describe types that we can "index into" like `a[10]`, or `ageMap["daniel"]`. Indexable types have an *index signature* that describes the types we can use to index into the object, along with the corresponding return types when indexing. Let's take an example:

```
interface StringArray {
    [index: number]: string;
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];
```

Above, we have a `StringArray` interface that has an index signature. This index signature states that when a `StringArray` is indexed with a `number`, it will return a `string`.

There are two types of supported index signatures: `string` and `number`. It is possible to support both types of indexers, but the type returned from a numeric indexer must be a subtype of the type returned from the string indexer. This is because when indexing with a `number`, JavaScript will actually convert that to a `string` before indexing into an object. That means that indexing with `100` (a `number`) is the same thing as indexing with `"100"` (a `string`), so the two need to be consistent.

```
class Animal {
    name: string;
}
class Dog extends Animal {
    breed: string;
}

// Error: indexing with a numeric string might get you a completely separate type of Animal!
interface NotOkay {
    [x: number]: Animal;
    [x: string]: Dog;
}
```

While string index signatures are a powerful way to describe the "dictionary" pattern, they also enforce that all properties match their return type. This is because a string index declares that `obj.property` is also available as `obj["property"]`. In the following example, `name`'s type does not match the string index's type, and the type-checker gives an error:

```
interface NumberDictionary {
  [index: string]: number;
  length: number; // ok, length is a number
  name: string; // error, the type of 'name' is not a subtype of the indexer
}
```

Finally, you can make index signatures readonly in order to prevent assignment to their indices:

```
interface ReadonlyStringArray {
  readonly [index: number]: string;
}
let myArray: ReadonlyStringArray = ["Alice", "Bob"];
myArray[2] = "Mallory"; // error!
```

You can't set `myArray[2]` because the index signature is readonly.

Class Types

Implementing an interface

One of the most common uses of interfaces in languages like C# and Java, that of explicitly enforcing that a class meets a particular contract, is also possible in TypeScript.

```
interface ClockInterface {
  currentTime: Date;
}

class Clock implements ClockInterface {
  currentTime: Date;
  constructor(h: number, m: number) { }
}
```

You can also describe methods in an interface that are implemented in the class, as we do with `setTime` in the below example:

```
interface ClockInterface {
  currentTime: Date;
  setTime(d: Date);
}

class Clock implements ClockInterface {
  currentTime: Date;
  setTime(d: Date) {
    this.currentTime = d;
  }
  constructor(h: number, m: number) { }
}
```

Interfaces describe the public side of the class, rather than both the public and private side. This prohibits you from using them to check that a class also has particular types for the private side of the class instance.

Difference between the static and instance sides of classes

When working with classes and interfaces, it helps to keep in mind that a class has *two* types: the type of the static side and the type of the instance side. You may notice that if you create an interface with a construct signature and try to create a class that implements this interface you get an error:

```
interface ClockConstructor {
    new (hour: number, minute: number);
}

class Clock implements ClockConstructor {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

This is because when a class implements an interface, only the instance side of the class is checked. Since the constructor sits in the static side, it is not included in this check.

Instead, you would need to work with the static side of the class directly. In this example, we define two interfaces, `ClockConstructor` for the constructor and `ClockInterface` for the instance methods. Then for convenience we define a constructor function `createClock` that creates instances of the type that is passed to it.

```
interface ClockConstructor {
    new (hour: number, minute: number): ClockInterface;
}

interface ClockInterface {
    tick();
}

function createClock(ctor: ClockConstructor, hour: number, minute: number): ClockInterface {
    return new ctor(hour, minute);
}

class DigitalClock implements ClockInterface {
    constructor(h: number, m: number) { }
    tick() {
        console.log("beep beep");
    }
}
class AnalogClock implements ClockInterface {
    constructor(h: number, m: number) { }
    tick() {
        console.log("tick tock");
    }
}

let digital = createClock(DigitalClock, 12, 17);
let analog = createClock(AnalogClock, 7, 32);
```

Because `createClock`'s first parameter is of type `ClockConstructor`, in `createClock(AnalogClock, 7, 32)`, it checks that `AnalogClock` has the correct constructor signature.

Extending Interfaces

Like classes, interfaces can extend each other. This allows you to copy the members of one interface into another, which gives you more flexibility in how you separate your interfaces into reusable components.

```
interface Shape {
    color: string;
```

```

}

interface Square extends Shape {
    sideLength: number;
}

let square = <Square>{};
square.color = "blue";
square.sideLength = 10;

```

An interface can extend multiple interfaces, creating a combination of all of the interfaces.

```

interface Shape {
    color: string;
}

interface PenStroke {
    penWidth: number;
}

interface Square extends Shape, PenStroke {
    sideLength: number;
}

let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;

```

Hybrid Types

As we mentioned earlier, interfaces can describe the rich types present in real world JavaScript. Because of JavaScript's dynamic and flexible nature, you may occasionally encounter an object that works as a combination of some of the types described above.

One such example is an object that acts as both a function and an object, with additional properties:

```

interface Counter {
    (start: number): string;
    interval: number;
    reset(): void;
}

function getCounter(): Counter {
    let counter = <Counter>function (start: number) { };
    counter.interval = 123;
    counter.reset = function () { };
    return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;

```

When interacting with 3rd-party JavaScript, you may need to use patterns like the above to fully describe the shape of the type.

Interfaces Extending Classes

When an interface type extends a class type it inherits the members of the class but not their implementations. It is as if the interface had declared all of the members of the class without providing an implementation. Interfaces inherit even the private and protected members of a base class. This means that when you create an interface that extends a class with private or protected members, that interface type can only be implemented by that class or a subclass of it.

This is useful when you have a large inheritance hierarchy, but want to specify that your code works with only subclasses that have certain properties. The subclasses don't have to be related besides inheriting from the base class. For example:

```
class Control {  
    private state: any;  
}  
  
interface SelectableControl extends Control {  
    select(): void;  
}  
  
class Button extends Control implements SelectableControl {  
    select() {}  
}  
  
class TextBox extends Control {  
    select() {}  
}  
  
// Error: Property 'state' is missing in type 'Image'.  
class Image implements SelectableControl {  
    select() {}  
}  
  
class Location {  
}
```

In the above example, `SelectableControl` contains all of the members of `Control`, including the private `state` property. Since `state` is a private member it is only possible for descendants of `Control` to implement `SelectableControl`. This is because only descendants of `Control` will have a `state` private member that originates in the same declaration, which is a requirement for private members to be compatible.

Within the `Control` class it is possible to access the `state` private member through an instance of `SelectableControl`. Effectively, a `SelectableControl` acts like a `Control` that is known to have a `select` method. The `Button` and `TextBox` classes are subtypes of `SelectableControl` (because they both inherit from `Control` and have a `select` method), but the `Image` and `Location` classes are not.

Introduction

Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components, but this may feel a bit awkward to programmers more comfortable with an object-oriented approach, where classes inherit functionality and objects are built from these classes. Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers will be able to build their applications using this object-oriented class-based approach. In TypeScript, we allow developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

Classes

Let's take a look at a simple class-based example:

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");
```

The syntax should look familiar if you've used C# or Java before. We declare a new class `Greeter`. This class has three members: a property called `greeting`, a constructor, and a method `greet`.

You'll notice that in the class when we refer to one of the members of the class we prepend `this.`. This denotes that it's a member access.

In the last line we construct an instance of the `Greeter` class using `new`. This calls into the constructor we defined earlier, creating a new object with the `Greeter` shape, and running the constructor to initialize it.

Inheritance

In TypeScript, we can use common object-oriented patterns. One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance.

Let's take a look at an example:

```
class Animal {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}m.`);
    }
}

class Dog extends Animal {
    bark() {
        console.log('Woof! Woof!');
    }
}

const dog = new Dog();
```

```
dog.bark();
dog.move(10);
dog.bark();
```

This example shows the most basic inheritance feature: classes inherit properties and methods from base classes. Here, `Dog` is a *derived* class that derives from the `Animal` *base* class using the `extends` keyword. Derived classes are often called *subclasses*, and base classes are often called *superclasses*.

Because `Dog` extends the functionality from `Animal`, we were able to create an instance of `Dog` that could both `bark()` and `move()`.

Let's now look at a more complex example.

```
class Animal {
    name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0) {
        console.log(`\${this.name} moved \${distanceInMeters}m.`);
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}

class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 45) {
        console.log("Galloping...");
        super.move(distanceInMeters);
    }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

This example covers a few other features we didn't previously mention. Again, we see the `extends` keywords used to create two new subclasses of `Animal`: `Horse` and `Snake`.

One difference from the prior example is that each derived class that contains a constructor function *must* call `super()` which will execute the constructor of the base class. What's more, before we ever access a property on `this` in a constructor body, we *have* to call `super()`. This is an important rule that TypeScript will enforce.

The example also shows how to override methods in the base class with methods that are specialized for the subclass. Here both `Snake` and `Horse` create a `move` method that overrides the `move` from `Animal`, giving it functionality specific to each class. Note that even though `tom` is declared as an `Animal`, since its value is a `Horse`, calling `tom.move(34)` will call the overriding method in `Horse`:

```
Slithering...
Sammy the Python moved 5m.
Galloping...
Tommy the Palomino moved 34m.
```

Public, private, and protected modifiers

Public by default

In our examples, we've been able to freely access the members that we declared throughout our programs. If you're familiar with classes in other languages, you may have noticed in the above examples we haven't had to use the word `public` to accomplish this; for instance, C# requires that each member be explicitly labeled `public` to be visible. In TypeScript, each member is `public` by default.

You may still mark a member `public` explicitly. We could have written the `Animal` class from the previous section in the following way:

```
class Animal {
  public name: string;
  public constructor(theName: string) { this.name = theName; }
  public move(distanceInMeters: number) {
    console.log(` ${this.name} moved ${distanceInMeters}m.`);
  }
}
```

Understanding `private`

When a member is marked `private`, it cannot be accessed from outside of its containing class. For example:

```
class Animal {
  private name: string;
  constructor(theName: string) { this.name = theName; }
}

new Animal("Cat").name; // Error: 'name' is private;
```

TypeScript is a structural type system. When we compare two different types, regardless of where they came from, if the types of all members are compatible, then we say the types themselves are compatible.

However, when comparing types that have `private` and `protected` members, we treat these types differently. For two types to be considered compatible, if one of them has a `private` member, then the other must have a `private` member that originated in the same declaration. The same applies to `protected` members.

Let's look at an example to better see how this plays out in practice:

```
class Animal {
  private name: string;
  constructor(theName: string) { this.name = theName; }
}

class Rhino extends Animal {
  constructor() { super("Rhino"); }
}

class Employee {
  private name: string;
  constructor(theName: string) { this.name = theName; }
}

let animal = new Animal("Goat");
let rhino = new Rhino();
let employee = new Employee("Bob");
```

```
animal = rhino;
animal = employee; // Error: 'Animal' and 'Employee' are not compatible
```

In this example, we have an `Animal` and a `Rhino`, with `Rhino` being a subclass of `Animal`. We also have a new class `Employee` that looks identical to `Animal` in terms of shape. We create some instances of these classes and then try to assign them to each other to see what will happen. Because `Animal` and `Rhino` share the `private` side of their shape from the same declaration of `private name: string` in `Animal`, they are compatible. However, this is not the case for `Employee`. When we try to assign from an `Employee` to `Animal` we get an error that these types are not compatible. Even though `Employee` also has a `private` member called `name`, it's not the one we declared in `Animal`.

Understanding `protected`

The `protected` modifier acts much like the `private` modifier with the exception that members declared `protected` can also be accessed within deriving classes. For example,

```
class Person {
    protected name: string;
    constructor(name: string) { this.name = name; }

class Employee extends Person {
    private department: string;

    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name); // error
```

Notice that while we can't use `name` from outside of `Person`, we can still use it from within an instance method of `Employee` because `Employee` derives from `Person`.

A constructor may also be marked `protected`. This means that the class cannot be instantiated outside of its containing class, but can be extended. For example,

```
class Person {
    protected name: string;
    protected constructor(theName: string) { this.name = theName; }

// Employee can extend Person
class Employee extends Person {
    private department: string;

    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }

    public getElevatorPitch() {
```

```

        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
let john = new Person("John"); // Error: The 'Person' constructor is protected

```

Readonly modifier

You can make properties readonly by using the `readonly` keyword. Readonly properties must be initialized at their declaration or in the constructor.

```

class Octopus {
    readonly name: string;
    readonly numberofLegs: number = 8;
    constructor (theName: string) {
        this.name = theName;
    }
}
let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit"; // error! name is readonly.

```

Parameter properties

In our last example, we had to declare a readonly member `name` and a constructor parameter `theName` in the `Octopus` class, and we then immediately set `name` to `theName`. This turns out to be a very common practice. *Parameter properties* let you create and initialize a member in one place. Here's a further revision of the previous `Octopus` class using a parameter property:

```

class Octopus {
    readonly numberofLegs: number = 8;
    constructor(readonly name: string) {
    }
}

```

Notice how we dropped `theName` altogether and just use the shortened `readonly name: string` parameter on the constructor to create and initialize the `name` member. We've consolidated the declarations and assignment into one location.

Parameter properties are declared by prefixing a constructor parameter with an accessibility modifier or `readonly`, or both. Using `private` for a parameter property declares and initializes a private member; likewise, the same is done for `public`, `protected`, and `readonly`.

Accessors

TypeScript supports getters/setters as a way of intercepting accesses to a member of an object. This gives you a way of having finer-grained control over how a member is accessed on each object.

Let's convert a simple class to use `get` and `set`. First, let's start with an example without getters and setters.

```

class Employee {
    fullName: string;
}

```

```
let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
  console.log(employee.fullName);
}
```

While allowing people to randomly set `fullName` directly is pretty handy, this might get us in trouble if people can change names on a whim.

In this version, we check to make sure the user has a secret passcode available before we allow them to modify the employee. We do this by replacing the direct access to `fullName` with a `set` that will check the passcode. We add a corresponding `get` to allow the previous example to continue to work seamlessly.

```
let passcode = "secret passcode";

class Employee {
  private _fullName: string;

  get fullName(): string {
    return this._fullName;
  }

  set fullName(newName: string) {
    if (passcode && passcode == "secret passcode") {
      this._fullName = newName;
    }
    else {
      console.log("Error: Unauthorized update of employee!");
    }
  }
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
  console.log(employee.fullName);
}
```

To prove to ourselves that our accessor is now checking the passcode, we can modify the passcode and see that when it doesn't match we instead get the message warning us we don't have access to update the employee.

A couple of things to note about accessors:

First, accessors require you to set the compiler to output ECMAScript 5 or higher. Downlevelling to ECMAScript 3 is not supported. Second, accessors with a `get` and no `set` are automatically inferred to be `readonly`. This is helpful when generating a `.d.ts` file from your code, because users of your property can see that they can't change it.

Static Properties

Up to this point, we've only talked about the `instance` members of the class, those that show up on the object when it's instantiated. We can also create `static` members of a class, those that are visible on the class itself rather than on the instances. In this example, we use `static` on the origin, as it's a general value for all grids. Each instance accesses this value through prepending the name of the class. Similarly to prepending `this.` in front of instance accesses, here we prepend `Grid.` in front of static accesses.

```
class Grid {
  static origin = {x: 0, y: 0};
  calculateDistanceFromOrigin(point: {x: number; y: number;}) {
    let xDist = (point.x - Grid.origin.x);
```

```

        let yDist = (point.y - Grid.origin.y);
        return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
    }
    constructor (public scale: number) { }
}

let grid1 = new Grid(1.0); // 1x scale
let grid2 = new Grid(5.0); // 5x scale

console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));

```

Abstract Classes

Abstract classes are base classes from which other classes may be derived. They may not be instantiated directly. Unlike an interface, an abstract class may contain implementation details for its members. The `abstract` keyword is used to define abstract classes as well as abstract methods within an abstract class.

```

abstract class Animal {
    abstract makeSound(): void;
    move(): void {
        console.log("roaming the earth...");
    }
}

```

Methods within an abstract class that are marked as abstract do not contain an implementation and must be implemented in derived classes. Abstract methods share a similar syntax to interface methods. Both define the signature of a method without including a method body. However, abstract methods must include the `abstract` keyword and may optionally include access modifiers.

```

abstract class Department {

    constructor(public name: string) {}

    printName(): void {
        console.log("Department name: " + this.name);
    }

    abstract printMeeting(): void; // must be implemented in derived classes
}

class AccountingDepartment extends Department {

    constructor() {
        super("Accounting and Auditing"); // constructors in derived classes must call super()
    }

    printMeeting(): void {
        console.log("The Accounting Department meets each Monday at 10am.");
    }

    generateReports(): void {
        console.log("Generating accounting reports...");
    }
}

let department: Department; // ok to create a reference to an abstract type
department = new Department(); // error: cannot create an instance of an abstract class
department = new AccountingDepartment(); // ok to create and assign a non-abstract subclass
department.printName();
department.printMeeting();

```

```
department.generateReports(); // error: method doesn't exist on declared abstract type
```

Advanced Techniques

Constructor functions

When you declare a class in TypeScript, you are actually creating multiple declarations at the same time. The first is the type of the *instance* of the class.

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter: Greeter;
greeter = new Greeter("world");
console.log(greeter.greet());
```

Here, when we say `let greeter: Greeter`, we're using `Greeter` as the type of instances of the class `Greeter`. This is almost second nature to programmers from other object-oriented languages.

We're also creating another value that we call the *constructor function*. This is the function that is called when we `new` up instances of the class. To see what this looks like in practice, let's take a look at the JavaScript created by the above example:

```
let Greeter = (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
})();

let greeter;
greeter = new Greeter("world");
console.log(greeter.greet());
```

Here, `let Greeter` is going to be assigned the constructor function. When we call `new` and run this function, we get an instance of the class. The constructor function also contains all of the static members of the class. Another way to think of each class is that there is an *instance* side and a *static* side.

Let's modify the example a bit to show this difference:

```
class Greeter {
    static standardGreeting = "Hello, there";
    greeting: string;
    greet() {
        if (this.greeting) {
            return "Hello, " + this.greeting;
        }
        else {
```

```

        return Greeter.standardGreeting;
    }
}

let greeter1: Greeter;
greeter1 = new Greeter();
console.log(greeter1.greet());

let greeterMaker: typeof Greeter = Greeter;
greeterMaker.standardGreeting = "Hey there!";

let greeter2: Greeter = new greeterMaker();
console.log(greeter2.greet());

```

In this example, `greeter1` works similarly to before. We instantiate the `Greeter` class, and use this object. This we have seen before.

Next, we then use the class directly. Here we create a new variable called `greeterMaker`. This variable will hold the class itself, or said another way its constructor function. Here we use `typeof Greeter`, that is "give me the type of the `Greeter` class itself" rather than the instance type. Or, more precisely, "give me the type of the symbol called `Greeter`," which is the type of the constructor function. This type will contain all of the static members of `Greeter` along with the constructor that creates instances of the `Greeter` class. We show this by using `new` on `greeterMaker`, creating new instances of `Greeter` and invoking them as before.

Using a class as an interface

As we said in the previous section, a class declaration creates two things: a type representing instances of the class and a constructor function. Because classes create types, you can use them in the same places you would be able to use interfaces.

```

class Point {
    x: number;
    y: number;
}

interface Point3d extends Point {
    z: number;
}

let point3d: Point3d = {x: 1, y: 2, z: 3};

```

Introduction

Functions are the fundamental building block of any applications in JavaScript. They're how you build up layers of abstraction, mimicking classes, information hiding, and modules. In TypeScript, while there are classes, namespaces, and modules, functions still play the key role in describing how to *do* things. TypeScript also adds some new capabilities to the standard JavaScript functions to make them easier to work with.

Functions

To begin, just as in JavaScript, TypeScript functions can be created both as a named function or as an anonymous function. This allows you to choose the most appropriate approach for your application, whether you're building a list of functions in an API or a one-off function to hand off to another function.

To quickly recap what these two approaches look like in JavaScript:

```
// Named function
function add(x, y) {
    return x + y;
}

// Anonymous function
let myAdd = function(x, y) { return x + y; };
```

Just as in JavaScript, functions can refer to variables outside of the function body. When they do so, they're said to capture these variables. While understanding how this works, and the trade-offs when using this technique, are outside of the scope of this article, having a firm understanding how this mechanic is an important piece of working with JavaScript and TypeScript.

```
let z = 100;

function addToZ(x, y) {
    return x + y + z;
}
```

Function Types

Typing the function

Let's add types to our simple examples from earlier:

```
function add(x: number, y: number): number {
    return x + y;
}

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

We can add types to each of the parameters and then to the function itself to add a return type. TypeScript can figure the return type out by looking at the return statements, so we can also optionally leave this off in many cases.

Writing the function type

Now that we've typed the function, let's write the full type of the function out by looking at each piece of the function type.

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

A function's type has the same two parts: the type of the arguments and the return type. When writing out the whole function type, both parts are required. We write out the parameter types just like a parameter list, giving each parameter a name and a type. This name is just to help with readability. We could have instead written:

```
let myAdd: (baseValue: number, increment: number) => number =
    function(x: number, y: number): number { return x + y; };
```

As long as the parameter types line up, it's considered a valid type for the function, regardless of the names you give the parameters in the function type.

The second part is the return type. We make it clear which is the return type by using a fat arrow (`=>`) between the parameters and the return type. As mentioned before, this is a required part of the function type, so if the function doesn't return a value, you would use `void` instead of leaving it off.

Of note, only the parameters and the return type make up the function type. Captured variables are not reflected in the type. In effect, captured variables are part of the "hidden state" of any function and do not make up its API.

Inferring the types

In playing with the example, you may notice that the TypeScript compiler can figure out the type if you have types on one side of the equation but not the other:

```
// myAdd has the full function type
let myAdd = function(x: number, y: number): number { return x + y; };

// The parameters 'x' and 'y' have the type number
let myAdd: (baseValue: number, increment: number) => number =
    function(x, y) { return x + y; };
```

This is called "contextual typing", a form of type inference. This helps cut down on the amount of effort to keep your program typed.

Optional and Default Parameters

In TypeScript, every parameter is assumed to be required by the function. This doesn't mean that it can't be given `null` or `undefined`, but rather, when the function is called the compiler will check that the user has provided a value for each parameter. The compiler also assumes that these parameters are the only parameters that will be passed to the function. In short, the number of arguments given to a function has to match the number of parameters the function expects.

```
function buildName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob"); // error, too few parameters
```

```
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result3 = buildName("Bob", "Adams"); // ah, just right
```

In JavaScript, every parameter is optional, and users may leave them off as they see fit. When they do, their value is `undefined`. We can get this functionality in TypeScript by adding a `?` to the end of parameters we want to be optional. For example, let's say we want the last name parameter from above to be optional:

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

let result1 = buildName("Bob"); // works correctly now
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result3 = buildName("Bob", "Adams"); // ah, just right
```

Any optional parameters must follow required parameters. Had we wanted to make the first name optional rather than the last name, we would need to change the order of parameters in the function, putting the first name last in the list.

In TypeScript, we can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes `undefined` in its place. These are called default-initialized parameters. Let's take the previous example and default the last name to `"Smith"`.

```
function buildName(firstName: string, lastName = "Smith") {
  return firstName + " " + lastName;
}

let result1 = buildName("Bob"); // works correctly now, returns "Bob Smith"
let result2 = buildName("Bob", undefined); // still works, also returns "Bob Smith"
let result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result4 = buildName("Bob", "Adams"); // ah, just right
```

Default-initialized parameters that come after all required parameters are treated as optional, and just like optional parameters, can be omitted when calling their respective function. This means optional parameters and trailing default parameters will share commonality in their types, so both

```
function buildName(firstName: string, lastName?: string) {
  // ...
}
```

and

```
function buildName(firstName: string, lastName = "Smith") {
  // ...
}
```

share the same type `(firstName: string, lastName?: string) => string`. The default value of `lastName` disappears in the type, only leaving behind the fact that the parameter is optional.

Unlike plain optional parameters, default-initialized parameters don't *need* to occur after required parameters. If a default-initialized parameter comes before a required parameter, users need to explicitly pass `undefined` to get the default initialized value. For example, we could write our last example with only a default initializer on `firstName`:

```
function buildName(firstName = "Will", lastName: string) {
  return firstName + " " + lastName;
}
```

```
let result1 = buildName("Bob");           // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result3 = buildName("Bob", "Adams");      // okay and returns "Bob Adams"
let result4 = buildName(undefined, "Adams"); // okay and returns "Will Adams"
```

Rest Parameters

Required, optional, and default parameters all have one thing in common: they talk about one parameter at a time. Sometimes, you want to work with multiple parameters as a group, or you may not know how many parameters a function will ultimately take. In JavaScript, you can work with the arguments directly using the `arguments` variable that is visible inside every function body.

In TypeScript, you can gather these arguments together into a variable:

```
function buildName(firstName: string, ...restOfName: string[]) {
  return firstName + " " + restOfName.join(" ");
}

let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

Rest parameters are treated as a boundless number of optional parameters. When passing arguments for a rest parameter, you can use as many as you want; you can even pass none. The compiler will build an array of the arguments passed in with the name given after the ellipsis (`...`), allowing you to use it in your function.

The ellipsis is also used in the type of the function with rest parameters:

```
function buildName(firstName: string, ...restOfName: string[]) {
  return firstName + " " + restOfName.join(" ");
}

let buildNameFun: (fname: string, ...rest: string[]) => string = buildName;
```

this

Learning how to use `this` in JavaScript is something of a rite of passage. Since TypeScript is a superset of JavaScript, TypeScript developers also need to learn how to use `this` and how to spot when it's not being used correctly. Fortunately, TypeScript lets you catch incorrect uses of `this` with a couple of techniques. If you need to learn how `this` works in JavaScript, though, first read Yehuda Katz's [Understanding JavaScript Function Invocation and "this"](#). Yehuda's article explains the inner workings of `this` very well, so we'll just cover the basics here.

this and arrow functions

In JavaScript, `this` is a variable that's set when a function is called. This makes it a very powerful and flexible feature, but it comes at the cost of always having to know about the context that a function is executing in. This is notoriously confusing, especially when returning a function or passing a function as an argument.

Let's look at an example:

```
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function() {
```

```

        return function() {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);

```

Notice that `createCardPicker` is a function that itself returns a function. If we tried to run the example, we would get an error instead of the expected alert box. This is because the `this` being used in the function created by `createCardPicker` will be set to `window` instead of our `deck` object. That's because we call `cardPicker()` on its own. A top-level non-method syntax call like this will use `window` for `this`. (Note: under strict mode, `this` will be `undefined` rather than `window`).

We can fix this by making sure the function is bound to the correct `this` before we return the function to be used later. This way, regardless of how it's later used, it will still be able to see the original `deck` object. To do this, we change the function expression to use the ECMAScript 6 arrow syntax. Arrow functions capture the `this` where the function is created rather than where it is invoked:

```

let deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        // NOTE: the line below is now an arrow function, allowing us to capture 'this' right here
        return () => {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);

```

Even better, TypeScript will warn you when you make this mistake if you pass the `--noImplicitThis` flag to the compiler. It will point out that `this` in `this.suits[pickedSuit]` is of type `any`.

this parameters

Unfortunately, the type of `this.suits[pickedSuit]` is still `any`. That's because `this` comes from the function expression inside the object literal. To fix this, you can provide an explicit `this` parameter. `this` parameters are fake parameters that come first in the parameter list of a function:

```

function f(this: void) {
    // make sure `this` is unusable in this standalone function
}

```

Let's add a couple of interfaces to our example above, `Card` and `Deck`, to make the types clearer and easier to reuse:

```
interface Card {
    suit: string;
    card: number;
}

interface Deck {
    suits: string[];
    cards: number[];
    createCardPicker(this: Deck): () => Card;
}

let deck: Deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    // NOTE: The function now explicitly specifies that its callee must be of type Deck
    createCardPicker: function(this: Deck) {
        return () => {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Now TypeScript knows that `createCardPicker` expects to be called on a `Deck` object. That means that `this` is of type `Deck` now, not `any`, so `--noImplicitThis` will not cause any errors.

this parameters in callbacks

You can also run into errors with `this` in callbacks, when you pass functions to a library that will later call them. Because the library that calls your callback will call it like a normal function, `this` will be `undefined`. With some work you can use `this` parameters to prevent errors with callbacks too. First, the library author needs to annotate the callback type with `this`:

```
interface UIElement {
    addClickListener(onclick: (this: void, e: Event) => void): void;
}
```

`this: void` means that `addClickListener` expects `onclick` to be a function that does not require a `this` type. Second, annotate your calling code with `this`:

```
class Handler {
    info: string;
    onClickBad(this: Handler, e: Event) {
        // oops, used this here. using this callback would crash at runtime
        this.info = e.message;
    }
}
let h = new Handler();
uiElement.addClickListener(h.onClickBad); // error!
```

With `this` annotated, you make it explicit that `onClickBad` must be called on an instance of `Handler`. Then TypeScript will detect that `addClickListener` requires a function that has `this: void`. To fix the error, change the type of `this`:

```
class Handler {
    info: string;
    onClickGood(this: void, e: Event) {
        // can't use this here because it's of type void!
        console.log('clicked!');
    }
}
let h = new Handler();
uiElement.addClickListener(h.onClickGood);
```

Because `onClickGood` specifies its `this` type as `void`, it is legal to pass to `addClickListener`. Of course, this also means that it can't use `this.info`. If you want both then you'll have to use an arrow function:

```
class Handler {
    info: string;
    onClickGood = (e: Event) => { this.info = e.message }
}
```

This works because arrow functions don't capture `this`, so you can always pass them to something that expects `this: void`. The downside is that one arrow function is created per object of type `Handler`. Methods, on the other hand, are only created once and attached to `Handler`'s prototype. They are shared between all objects of type `Handler`.

Overloads

JavaScript is inherently a very dynamic language. It's not uncommon for a single JavaScript function to return different types of objects based on the shape of the arguments passed in.

```
let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x): any {
    // Check to see if we're working with an object/array
    // if so, they gave us the deck and we'll pick the card
    if (typeof x == "object") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { suit: "hearts", card: 4 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

Here the `pickCard` function will return two different things based on what the user has passed in. If the user passes in an object that represents the deck, the function will pick the card. If the user picks the card, we tell them which card they've picked. But how do we describe this to the type system?

The answer is to supply multiple function types for the same function as a list of overloads. This list is what the compiler will use to resolve function calls. Let's create a list of overloads that describe what our `pickCard` accepts and what it returns.

```
let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
    // Check to see if we're working with an object/array
    // if so, they gave us the deck and we'll pick the card
    if (typeof x == "object") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { suit: "hearts", card: 4 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

With this change, the overloads now give us type-checked calls to the `pickCard` function.

In order for the compiler to pick the correct typecheck, it follows a similar process to the underlying JavaScript. It looks at the overload list, and proceeding with the first overload attempts to call the function with the provided parameters. If it finds a match, it picks this overload as the correct overload. For this reason, it's customary to order overloads from most specific to least specific.

Note that the `function pickCard(x): any` piece is not part of the overload list, so it only has two overloads: one that takes an object and one that takes a number. Calling `pickCard` with any other parameter types would cause an error.

Introduction

A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable. Components that are capable of working on the data of today as well as the data of tomorrow will give you the most flexible capabilities for building up large software systems.

In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is *generics*, that is, being able to create a component that can work over a variety of types rather than a single one. This allows users to consume these components and use their own types.

Hello World of Generics

To start off, let's do the "hello world" of generics: the identity function. The identity function is a function that will return back whatever is passed in. You can think of this in a similar way to the `echo` command.

Without generics, we would either have to give the identity function a specific type:

```
function identity(arg: number): number {
    return arg;
}
```

Or, we could describe the identity function using the `any` type:

```
function identity(arg: any): any {
    return arg;
}
```

While using `any` is certainly generic in that it will cause the function to accept any and all types for the type of `arg`, we actually are losing the information about what that type was when the function returns. If we passed in a number, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned. Here, we will use a *type variable*, a special kind of variable that works on types rather than values.

```
function identity<T>(arg: T): T {
    return arg;
}
```

We've now added a type variable `T` to the identity function. This `T` allows us to capture the type the user provides (e.g. `number`), so that we can use that information later. Here, we use `T` again as the return type. On inspection, we can now see the same type is used for the argument and the return type. This allows us to traffic that type information in one side of the function and out the other.

We say that this version of the `identity` function is generic, as it works over a range of types. Unlike using `any`, it's also just as precise (ie, it doesn't lose any information) as the first `identity` function that used numbers for the argument and return type.

Once we've written the generic identity function, we can call it in one of two ways. The first way is to pass all of the arguments, including the type argument, to the function:

```
let output = identity<string>("myString"); // type of output will be 'string'
```

Here we explicitly set `T` to be `string` as one of the arguments to the function call, denoted using the `< >` around the arguments rather than `()`.

The second way is also perhaps the most common. Here we use *type argument inference* -- that is, we want the compiler to set the value of `T` for us automatically based on the type of the argument we pass in:

```
let output = identity("myString"); // type of output will be 'string'
```

Notice that we didn't have to explicitly pass the type in the angle brackets (`< >`); the compiler just looked at the value `"myString"`, and set `T` to its type. While type argument inference can be a helpful tool to keep code shorter and more readable, you may need to explicitly pass in the type arguments as we did in the previous example when the compiler fails to infer the type, as may happen in more complex examples.

Working with Generic Type Variables

When you begin to use generics, you'll notice that when you create generic functions like `identity`, the compiler will enforce that you use any generically typed parameters in the body of the function correctly. That is, that you actually treat these parameters as if they could be any and all types.

Let's take our `identity` function from earlier:

```
function identity<T>(arg: T): T {
    return arg;
}
```

What if we want to also log the length of the argument `arg` to the console with each call? We might be tempted to write this:

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length); // Error: T doesn't have .length
    return arg;
}
```

When we do, the compiler will give us an error that we're using the `.length` member of `arg`, but nowhere have we said that `arg` has this member. Remember, we said earlier that these type variables stand in for any and all types, so someone using this function could have passed in a `number` instead, which does not have a `.length` member.

Let's say that we've actually intended this function to work on arrays of `T` rather than `T` directly. Since we're working with arrays, the `.length` member should be available. We can describe this just like we would create arrays of other types:

```
function loggingIdentity<T>(arg: T[]): T[] {
    console.log(arg.length); // Array has a .length, so no more error
    return arg;
}
```

You can read the type of `loggingIdentity` as "the generic function `loggingIdentity` takes a type parameter `T`, and an argument `arg` which is an array of `T`s, and returns an array of `T`s." If we passed in an array of numbers, we'd get an array of numbers back out, as `T` would bind to `number`. This allows us to use our generic type variable `T` as part of the types we're working with, rather than the whole type, giving us greater flexibility.

We can alternatively write the sample example this way:

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {
```

```

    console.log(arg.length); // Array has a .length, so no more error
    return arg;
}

```

You may already be familiar with this style of type from other languages. In the next section, we'll cover how you can create your own generic types like `Array<T>`.

Generic Types

In previous sections, we created generic identity functions that worked over a range of types. In this section, we'll explore the type of the functions themselves and how to create generic interfaces.

The type of generic functions is just like those of non-generic functions, with the type parameters listed first, similarly to function declarations:

```

function identity<T>(arg: T): T {
  return arg;
}

let myIdentity: <T>(arg: T) => T = identity;

```

We could also have used a different name for the generic type parameter in the type, so long as the number of type variables and how the type variables are used line up.

```

function identity<T>(arg: T): T {
  return arg;
}

let myIdentity: <U>(arg: U) => U = identity;

```

We can also write the generic type as a call signature of an object literal type:

```

function identity<T>(arg: T): T {
  return arg;
}

let myIdentity: {<T>(arg: T): T} = identity;

```

Which leads us to writing our first generic interface. Let's take the object literal from the previous example and move it to an interface:

```

interface GenericIdentityFn {
  <T>(arg: T): T;
}

function identity<T>(arg: T): T {
  return arg;
}

let myIdentity: GenericIdentityFn = identity;

```

In a similar example, we may want to move the generic parameter to be a parameter of the whole interface. This lets us see what type(s) we're generic over (e.g. `Dictionary<string>` rather than just `Dictionary`). This makes the type parameter visible to all the other members of the interface.

```

interface GenericIdentityFn<T> {

```

```

        (arg: T): T;
    }

    function identity<T>(arg: T): T {
        return arg;
    }

    let myIdentity: GenericIdentityFn<number> = identity;

```

Notice that our example has changed to be something slightly different. Instead of describing a generic function, we now have a non-generic function signature that is a part of a generic type. When we use `GenericIdentityFn`, we now will also need to specify the corresponding type argument (here: `number`), effectively locking in what the underlying call signature will use. Understanding when to put the type parameter directly on the call signature and when to put it on the interface itself will be helpful in describing what aspects of a type are generic.

In addition to generic interfaces, we can also create generic classes. Note that it is not possible to create generic enums and namespaces.

Generic Classes

A generic class has a similar shape to a generic interface. Generic classes have a generic type parameter list in angle brackets (`<>`) following the name of the class.

```

class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };

```

This is a pretty literal use of the `GenericNumber` class, but you may have noticed that nothing is restricting it to only use the `number` type. We could have instead used `string` or even more complex objects.

```

let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };

console.log(stringNumeric.add(stringNumeric.zeroValue, "test"));

```

Just as with interface, putting the type parameter on the class itself lets us make sure all of the properties of the class are working with the same type.

As we covered in [our section on classes](#), a class has two sides to its type: the static side and the instance side. Generic classes are only generic over their instance side rather than their static side, so when working with classes, static members can not use the class's type parameter.

Generic Constraints

If you remember from an earlier example, you may sometimes want to write a generic function that works on a set of types where you have some knowledge about what capabilities that set of types will have. In our `loggingIdentity` example, we wanted to be able to access the `.length` property of `arg`, but the compiler could not prove that every type had a `.length` property, so it warns us that we can't make this assumption.

```
function loggingIdentity<T>(arg: T) {
    console.log(arg.length); // Error: T doesn't have .length
    return arg;
}
```

Instead of working with any and all types, we'd like to constrain this function to work with any and all types that also have the `.length` property. As long as the type has this member, we'll allow it, but it's required to have at least this member. To do so, we must list our requirement as a constraint on what `T` can be.

To do so, we'll create an interface that describes our constraint. Here, we'll create an interface that has a single `.length` property and then we'll use this interface and the `extends` keyword to denote our constraint:

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T) {
    console.log(arg.length); // Now we know it has a .length property, so no more error
    return arg;
}
```

Because the generic function is now constrained, it will no longer work over any and all types:

```
loggingIdentity(3); // Error, number doesn't have a .length property
```

Instead, we need to pass in values whose type has all the required properties:

```
loggingIdentity({length: 10, value: 3});
```

Using Type Parameters in Generic Constraints

You can declare a type parameter that is constrained by another type parameter. For example, here we'd like to get a property from an object given its name. We'd like to ensure that we're not accidentally grabbing a property that does not exist on the `obj`, so we'll place a constraint between the two types:

```
function getProperty<T, K extends keyof T>(obj: T, key: K) {
    return obj[key];
}

let x = { a: 1, b: 2, c: 3, d: 4 };

getProperty(x, "a"); // okay
getProperty(x, "m"); // error: Argument of type 'm' isn't assignable to 'a' | 'b' | 'c' | 'd'.
```

Using Class Types in Generics

When creating factories in TypeScript using generics, it is necessary to refer to class types by their constructor functions. For example,

```
function create<T>(c: {new(): T; }): T {
    return new c();
}
```

A more advanced example uses the prototype property to infer and constrain relationships between the constructor function and the instance side of class types.

```
class BeeKeeper {
    hasMask: boolean;
}

class ZooKeeper {
    nametag: string;
}

class Animal {
    numLegs: number;
}

class Bee extends Animal {
    keeper: BeeKeeper;
}

class Lion extends Animal {
    keeper: ZooKeeper;
}

function createInstance<A extends Animal>(c: new () => A): A {
    return new c();
}

createInstance(Lion).keeper.nametag; // typechecks!
createInstance(Bee).keeper.hasMask; // typechecks!
```

Enums

Enums allow us to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases. TypeScript provides both numeric and string-based enums.

Numeric enums

We'll first start off with numeric enums, which are probably more familiar if you're coming from other languages. An enum can be defined using the `enum` keyword.

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right,
}
```

Above, we have a numeric enum where `Up` is initialized with `1`. All of the following members are auto-incremented from that point on. In other words, `Direction.Up` has the value `1`, `Down` has `2`, `Left` has `3`, and `Right` has `4`.

If we wanted, we could leave off the initializers entirely:

```
enum Direction {
  Up,
  Down,
  Left,
  Right,
}
```

Here, `Up` would have the value `0`, `Down` would have `1`, etc. This auto-incrementing behavior is useful for cases where we might not care about the member values themselves, but do care that each value is distinct from other values in the same enum.

Using an enum is simple: just access any member as a property off of the enum itself, and declare types using the name of the enum:

```
enum Response {
  No = 0,
  Yes = 1,
}

function respond(recipient: string, message: Response): void {
  // ...
}

respond("Princess Caroline", Response.Yes)
```

Numeric enums can be mixed in [computed and constant members \(see below\)](#). The short story is, enums without initializers either need to be first, or have to come after numeric enums initialized with numeric constants or other constant enum members. In other words, the following isn't allowed:

```
enum E {
  A = getSomeValue(),
```

```
B, // error! 'A' is not constant-initialized, so 'B' needs an initializer
}
```

String enums

String enums are a similar concept, but have some subtle [runtime differences](#) as documented below. In a string enum, each member has to be constant-initialized with a string literal, or with another string enum member.

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT",
}
```

While string enums don't have auto-incrementing behavior, string enums have the benefit that they "serialize" well. In other words, if you were debugging and had to read the runtime value of a numeric enum, the value is often opaque - it doesn't convey any useful meaning on its own (though [reverse mapping](#) can often help), string enums allow you to give a meaningful and readable value when your code runs, independent of the name of the enum member itself.

Heterogeneous enums

Technically enums can be mixed with string and numeric members, but it's not clear why you would ever want to do so:

```
enum BooleanLikeHeterogeneousEnum {
  No = 0,
  Yes = "YES",
}
```

Unless you're really trying to take advantage of JavaScript's runtime behavior in a clever way, it's advised that you don't do this.

Computed and constant members

Each enum member has a value associated with it which can be either *constant* or *computed*. An enum member is considered constant if:

- It is the first member in the enum and it has no initializer, in which case it's assigned the value `0`:

```
// E.X is constant:
enum E { X }
```

- It does not have an initializer and the preceding enum member was a *numeric* constant. In this case the value of the current enum member will be the value of the preceding enum member plus one.

```
// All enum members in 'E1' and 'E2' are constant.

enum E1 { X, Y, Z }

enum E2 {
  A = 1, B, C
}
```

- The enum member is initialized with a constant enum expression. A constant enum expression is a subset of TypeScript expressions that can be fully evaluated at compile time. An expression is a constant enum expression if it is:
 - a literal enum expression (basically a string literal or a numeric literal)
 - a reference to previously defined constant enum member (which can originate from a different enum).
 - a parenthesized constant enum expression
 - one of the `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^` unary operators applied to constant enum expression
 - `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^` binary operators with constant enum expressions as operands It is a compile time error for constant enum expressions to be evaluated to `Nan` or `Infinity`.

In all other cases enum member is considered computed.

```
enum FileAccess {
    // constant members
    None,
    Read = 1 << 1,
    Write = 1 << 2,
    ReadWrite = Read | Write,
    // computed member
    G = "123".length
}
```

Union enums and enum member types

There is a special subset of constant enum members that aren't calculated: literal enum members. A literal enum member is a constant enum member with no initialized value, or with values that are initialized to

- any string literal (e.g. `"foo"`, `"bar"`, `"baz"`)
- any numeric literal (e.g. `1`, `100`)
- a unary minus applied to any numeric literal (e.g. `-1`, `-100`)

When all members in an enum have literal enum values, some special semantics come to play.

The first is that enum members also become types as well! For example, we can say that certain members can *only* have the value of an enum member:

```
enum ShapeKind {
    Circle,
    Square,
}

interface Circle {
    kind: ShapeKind.Circle;
    radius: number;
}

interface Square {
    kind: ShapeKind.Square;
    sideLength: number;
}

let c: Circle = {
    kind: ShapeKind.Square,
    // ~~~~~ Error!
    radius: 100,
}
```

The other change is that enum types themselves effectively become a *union* of each enum member. While we haven't discussed [union types](#) yet, all that you need to know is that with union enums, the type system is able to leverage the fact that it knows the exact set of values that exist in the enum itself. Because of that, TypeScript can catch silly bugs where we might be comparing values incorrectly. For example:

```
enum E {
  Foo,
  Bar,
}

function f(x: E) {
  if (x !== E.Foo || x !== E.Bar) {
    // -----
    // Error! Operator '!==' cannot be applied to types 'E.Foo' and 'E.Bar'.
  }
}
```

In that example, we first checked whether `x` was *not* `E.Foo`. If that check succeeds, then our `||` will short-circuit, and the body of the 'if' will get run. However, if the check didn't succeed, then `x` can *only* be `E.Foo`, so it doesn't make sense to see whether it's equal to `E.Bar`.

Enums at runtime

Enums are real objects that exist at runtime. For example, the following enum

```
enum E {
  X, Y, Z
}
```

can actually be passed around to functions

```
function f(obj: { X: number }) {
  return obj.X;
}

// Works, since 'E' has a property named 'X' which is a number.
f(E);
```

Reverse mappings

In addition to creating an object with property names for members, numeric enums members also get a *reverse mapping* from enum values to enum names. For example, in this example:

```
enum Enum {
  A
}
let a = Enum.A;
let nameOfA = Enum[a]; // "A"
```

TypeScript might compile this down to something like the the following JavaScript:

```
var Enum;
(function (Enum) {
  Enum[Enum["A"] = 0] = "A";
})(Enum || (Enum = {}));
var a = Enum.A;
var nameOfA = Enum[a]; // "A"
```

In this generated code, an enum is compiled into an object that stores both forward (`name -> value`) and reverse (`value -> name`) mappings. References to other enum members are always emitted as property accesses and never inlined.

Keep in mind that string enum members *do not* get a reverse mapping generated at all.

const enums

In most cases, enums are a perfectly valid solution. However sometimes requirements are tighter. To avoid paying the cost of extra generated code and additional indirection when accessing enum values, it's possible to use `const` enums. Const enums are defined using the `const` modifier on our enums:

```
const enum Enum {
  A = 1,
  B = A * 2
}
```

Const enums can only use constant enum expressions and unlike regular enums they are completely removed during compilation. Const enum members are inlined at use sites. This is possible since const enums cannot have computed members.

```
const enum Directions {
  Up,
  Down,
  Left,
  Right
}

let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right]
```

in generated code will become

```
var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* Right */];
```

Ambient enums

Ambient enums are used to describe the shape of already existing enum types.

```
declare enum Enum {
  A = 1,
  B,
  C = 2
}
```

One important difference between ambient and non-ambient enums is that, in regular enums, members that don't have an initializer will be considered constant if its preceding enum member is considered constant. In contrast, an ambient (and non-const) enum member that does not have initializer is *always* considered computed.

Introduction

In this section, we will cover type inference in TypeScript. Namely, we'll discuss where and how types are inferred.

Basics

In TypeScript, there are several places where type inference is used to provide type information when there is no explicit type annotation. For example, in this code

```
let x = 3;
```

The type of the `x` variable is inferred to be `number`. This kind of inference takes place when initializing variables and members, setting parameter default values, and determining function return types.

In most cases, type inference is straightforward. In the following sections, we'll explore some of the nuances in how types are inferred.

Best common type

When a type inference is made from several expressions, the types of those expressions are used to calculate a "best common type". For example,

```
let x = [0, 1, null];
```

To infer the type of `x` in the example above, we must consider the type of each array element. Here we are given two choices for the type of the array: `number` and `null`. The best common type algorithm considers each candidate type, and picks the type that is compatible with all the other candidates.

Because the best common type has to be chosen from the provided candidate types, there are some cases where types share a common structure, but no one type is the super type of all candidate types. For example:

```
let zoo = [new Rhino(), new Elephant(), new Snake()];
```

Ideally, we may want `zoo` to be inferred as an `Animal[]`, but because there is no object that is strictly of type `Animal` in the array, we make no inference about the array element type. To correct this, instead explicitly provide the type when no one type is a super type of all other candidates:

```
let zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
```

When no best common type is found, the resulting inference is the union array type, `(Rhino | Elephant | Snake)[]`.

Contextual Type

Type inference also works in "the other direction" in some cases in TypeScript. This is known as "contextual typing". Contextual typing occurs when the type of an expression is implied by its location. For example:

```
window.onmousedown = function(mouseEvent) {
```

```
    console.log(mouseEvent.clickTime); //<- Error
};
```

For the code above to give the type error, the TypeScript type checker used the type of the `window.onmousedown` function to infer the type of the function expression on the right hand side of the assignment. When it did so, it was able to infer the type of the `mouseEvent` parameter. If this function expression were not in a contextually typed position, the `mouseEvent` parameter would have type `any`, and no error would have been issued.

If the contextually typed expression contains explicit type information, the contextual type is ignored. Had we written the above example:

```
window.onmousedown = function(mouseEvent: any) {
    console.log(mouseEvent.clickTime); //<- Now, no error is given
};
```

The function expression with an explicit type annotation on the parameter will override the contextual type. Once it does so, no error is given as no contextual type applies.

Contextual typing applies in many cases. Common cases include arguments to function calls, right hand sides of assignments, type assertions, members of object and array literals, and return statements. The contextual type also acts as a candidate type in best common type. For example:

```
function createZoo(): Animal[] {
    return [new Rhino(), new Elephant(), new Snake()];
}
```

In this example, best common type has a set of four candidates: `Animal`, `Rhino`, `Elephant`, and `Snake`. Of these, `Animal` can be chosen by the best common type algorithm.

Introduction

Type compatibility in TypeScript is based on structural subtyping. Structural typing is a way of relating types based solely on their members. This is in contrast with nominal typing. Consider the following code:

```
interface Named {
    name: string;
}

class Person {
    name: string;
}

let p: Named;
// OK, because of structural typing
p = new Person();
```

In nominally-typed languages like C# or Java, the equivalent code would be an error because the `Person` class does not explicitly describe itself as being an implementer of the `Named` interface.

TypeScript's structural type system was designed based on how JavaScript code is typically written. Because JavaScript widely uses anonymous objects like function expressions and object literals, it's much more natural to represent the kinds of relationships found in JavaScript libraries with a structural type system instead of a nominal one.

A Note on Soundness

TypeScript's type system allows certain operations that can't be known at compile-time to be safe. When a type system has this property, it is said to not be "sound". The places where TypeScript allows unsound behavior were carefully considered, and throughout this document we'll explain where these happen and the motivating scenarios behind them.

Starting out

The basic rule for TypeScript's structural type system is that `x` is compatible with `y` if `y` has at least the same members as `x`. For example:

```
interface Named {
    name: string;
}

let x: Named;
// y's inferred type is { name: string; location: string; }
let y = { name: "Alice", location: "Seattle" };
x = y;
```

To check whether `y` can be assigned to `x`, the compiler checks each property of `x` to find a corresponding compatible property in `y`. In this case, `y` must have a member called `name` that is a string. It does, so the assignment is allowed.

The same rule for assignment is used when checking function call arguments:

```

function greet(n: Named) {
  console.log("Hello, " + n.name);
}
greet(y); // OK

```

Note that `y` has an extra `location` property, but this does not create an error. Only members of the target type (`Named` in this case) are considered when checking for compatibility.

This comparison process proceeds recursively, exploring the type of each member and sub-member.

Comparing two functions

While comparing primitive types and object types is relatively straightforward, the question of what kinds of functions should be considered compatible is a bit more involved. Let's start with a basic example of two functions that differ only in their parameter lists:

```

let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error

```

To check if `x` is assignable to `y`, we first look at the parameter list. Each parameter in `x` must have a corresponding parameter in `y` with a compatible type. Note that the names of the parameters are not considered, only their types. In this case, every parameter of `x` has a corresponding compatible parameter in `y`, so the assignment is allowed.

The second assignment is an error, because `y` has a required second parameter that `x` does not have, so the assignment is disallowed.

You may be wondering why we allow 'discarding' parameters like in the example `y = x`. The reason for this assignment to be allowed is that ignoring extra function parameters is actually quite common in JavaScript. For example, `Array#forEach` provides three parameters to the callback function: the array element, its index, and the containing array. Nevertheless, it's very useful to provide a callback that only uses the first parameter:

```

let items = [1, 2, 3];

// Don't force these extra parameters
items.forEach((item, index, array) => console.log(item));

// Should be OK!
items.forEach(item => console.log(item));

```

Now let's look at how return types are treated, using two functions that differ only by their return type:

```

let x = () => ({name: "Alice"});
let y = () => ({name: "Alice", location: "Seattle"});

x = y; // OK
y = x; // Error, because x() lacks a location property

```

The type system enforces that the source function's return type be a subtype of the target type's return type.

Function Parameter Bivariance

When comparing the types of function parameters, assignment succeeds if either the source parameter is assignable to the target parameter, or vice versa. This is unsound because a caller might end up being given a function that takes a more specialized type, but invokes the function with a less specialized type. In practice, this sort of error is rare, and allowing this enables many common JavaScript patterns. A brief example:

```
enum EventType { Mouse, Keyboard }

interface Event { timestamp: number; }
interface MouseEvent extends Event { x: number; y: number }
interface KeyEvent extends Event { keyCode: number }

function listenEvent(eventType: EventType, handler: (n: Event) => void) {
    /* ... */
}

// Unsound, but useful and common
listenEvent(EventType.Mouse, (e: MouseEvent) => console.log(e.x + "," + e.y));

// Undesirable alternatives in presence of soundness
listenEvent(EventType.Mouse, (e: Event) => console.log((<MouseEvent>e).x + "," + (<MouseEvent>e).y));
listenEvent(EventType.Mouse, <(e: Event) => void>((e: MouseEvent) => console.log(e.x + "," + e.y)));

// Still disallowed (clear error). Type safety enforced for wholly incompatible types
listenEvent(EventType.Mouse, (e: number) => console.log(e));
```

Optional Parameters and Rest Parameters

When comparing functions for compatibility, optional and required parameters are interchangeable. Extra optional parameters of the source type are not an error, and optional parameters of the target type without corresponding parameters in the source type are not an error.

When a function has a rest parameter, it is treated as if it were an infinite series of optional parameters.

This is unsound from a type system perspective, but from a runtime point of view the idea of an optional parameter is generally not well-enforced since passing `undefined` in that position is equivalent for most functions.

The motivating example is the common pattern of a function that takes a callback and invokes it with some predictable (to the programmer) but unknown (to the type system) number of arguments:

```
function invokeLater(args: any[], callback: (...args: any[]) => void) {
    /* ... Invoke callback with 'args' ... */
}

// Unsound - invokeLater "might" provide any number of arguments
invokeLater([1, 2], (x, y) => console.log(x + ", " + y));

// Confusing (x and y are actually required) and undiscoverable
invokeLater([1, 2], (x?, y?) => console.log(x + ", " + y));
```

Functions with overloads

When a function has overloads, each overload in the source type must be matched by a compatible signature on the target type. This ensures that the target function can be called in all the same situations as the source function.

Enums

Enums are compatible with numbers, and numbers are compatible with enums. Enum values from different enum types are considered incompatible. For example,

```
enum Status { Ready, Waiting };
enum Color { Red, Blue, Green };

let status = Status.Ready;
status = Color.Green; // Error
```

Classes

Classes work similarly to object literal types and interfaces with one exception: they have both a static and an instance type. When comparing two objects of a class type, only members of the instance are compared. Static members and constructors do not affect compatibility.

```
class Animal {
    feet: number;
    constructor(name: string, numFeet: number) { }
}

class Size {
    feet: number;
    constructor(numFeet: number) { }
}

let a: Animal;
let s: Size;

a = s; // OK
s = a; // OK
```

Private and protected members in classes

Private and protected members in a class affect their compatibility. When an instance of a class is checked for compatibility, if the target type contains a private member, then the source type must also contain a private member that originated from the same class. Likewise, the same applies for an instance with a protected member. This allows a class to be assignment compatible with its super class, but *not* with classes from a different inheritance hierarchy which otherwise have the same shape.

Generics

Because TypeScript is a structural type system, type parameters only affect the resulting type when consumed as part of the type of a member. For example,

```
interface Empty<T> {
}

let x: Empty<number>;
let y: Empty<string>;

x = y; // OK, because y matches structure of x
```

In the above, `x` and `y` are compatible because their structures do not use the type argument in a differentiating way. Changing this example by adding a member to `Empty<T>` shows how this works:

```
interface NotEmpty<T> {
  data: T;
}
let x: NotEmpty<number>;
let y: NotEmpty<string>;

x = y; // Error, because x and y are not compatible
```

In this way, a generic type that has its type arguments specified acts just like a non-generic type.

For generic types that do not have their type arguments specified, compatibility is checked by specifying `any` in place of all unspecified type arguments. The resulting types are then checked for compatibility, just as in the non-generic case.

For example,

```
let identity = function<T>(x: T): T {
  // ...
}

let reverse = function<U>(y: U): U {
  // ...
}

identity = reverse; // OK, because (x: any) => any matches (y: any) => any
```

Advanced Topics

Subtype vs Assignment

So far, we've used "compatible", which is not a term defined in the language spec. In TypeScript, there are two kinds of compatibility: subtype and assignment. These differ only in that assignment extends subtype compatibility with rules to allow assignment to and from `any`, and to and from `enum` with corresponding numeric values.

Different places in the language use one of the two compatibility mechanisms, depending on the situation. For practical purposes, type compatibility is dictated by assignment compatibility, even in the cases of the `implements` and `extends` clauses.

For more information, see the [TypeScript spec](#).

Intersection Types

An intersection type combines multiple types into one. This allows you to add together existing types to get a single type that has all the features you need. For example, `Person & Serializable & Loggable` is a `Person` *and* `Serializable` *and* `Loggable`. That means an object of this type will have all members of all three types.

You will mostly see intersection types used for mixins and other concepts that don't fit in the classic object-oriented mold. (There are a lot of these in JavaScript!) Here's a simple example that shows how to create a mixin:

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

class Person {
  constructor(public name: string) { }
}
interface Loggable {
  log(): void;
}
class ConsoleLogger implements Loggable {
  log() {
    // ...
  }
}
var jim = extend(new Person("Jim"), new ConsoleLogger());
var n = jim.name;
jim.log();
```

Union Types

Union types are closely related to intersection types, but they are used very differently. Occasionally, you'll run into a library that expects a parameter to be either a `number` or a `string`. For instance, take the following function:

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: any) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${padding}'.`);
}

padLeft("Hello world", 4); // returns "    Hello world"
```

The problem with `padLeft` is that its `padding` parameter is typed as `any`. That means that we can call it with an argument that's neither a `number` nor a `string`, but TypeScript will be okay with it.

```
let indentedString = padLeft("Hello world", true); // passes at compile time, fails at runtime.
```

In traditional object-oriented code, we might abstract over the two types by creating a hierarchy of types. While this is much more explicit, it's also a little bit overkill. One of the nice things about the original version of `padLeft` was that we were able to just pass in primitives. That meant that usage was simple and concise. This new approach also wouldn't help if we were just trying to use a function that already exists elsewhere.

Instead of `any`, we can use a *union type* for the `padding` parameter:

```
/**  
 * Takes a string and adds "padding" to the left.  
 * If 'padding' is a string, then 'padding' is appended to the left side.  
 * If 'padding' is a number, then that number of spaces is added to the left side.  
 */  
function padLeft(value: string, padding: string | number) {  
    // ...  
}  
  
let indentedString = padLeft("Hello world", true); // errors during compilation
```

A union type describes a value that can be one of several types. We use the vertical bar (`|`) to separate each type, so `number | string | boolean` is the type of a value that can be a `number`, a `string`, or a `boolean`.

If we have a value that has a union type, we can only access members that are common to all types in the union.

```
interface Bird {  
    fly();  
    layEggs();  
}  
  
interface Fish {  
    swim();  
    layEggs();  
}  
  
function getSmallPet(): Fish | Bird {  
    // ...  
}  
  
let pet = getSmallPet();  
pet.layEggs(); // okay  
pet.swim(); // errors
```

Union types can be a bit tricky here, but it just takes a bit of intuition to get used to. If a value has the type `A | B`, we only know for *certain* that it has members that both `A` and `B` have. In this example, `Bird` has a member named `fly`. We can't be sure whether a variable typed as `Bird | Fish` has a `fly` method. If the variable is really a `Fish` at runtime, then calling `pet.fly()` will fail.

Type Guards and Differentiating Types

Union types are useful for modeling situations when values can overlap in the types they can take on. What happens when we need to know specifically whether we have a `Fish`? A common idiom in JavaScript to differentiate between two possible values is to check for the presence of a member. As we mentioned, you can only access members that are guaranteed to be in all the constituents of a union type.

```
let pet = getSmallPet();

// Each of these property accesses will cause an error
if (pet.swim) {
    pet.swim();
}
else if (pet.fly) {
    pet.fly();
}
```

To get the same code working, we'll need to use a type assertion:

```
let pet = getSmallPet();

if ((<Fish>pet).swim) {
    (<Fish>pet).swim();
}
else {
    (<Bird>pet).fly();
}
```

User-Defined Type Guards

Notice that we had to use type assertions several times. It would be much better if once we performed the check, we could know the type of `pet` within each branch.

It just so happens that TypeScript has something called a *type guard*. A type guard is some expression that performs a runtime check that guarantees the type in some scope. To define a type guard, we simply need to define a function whose return type is a *type predicate*:

```
function isFish(pet: Fish | Bird): pet is Fish {
    return (<Fish>pet).swim !== undefined;
}
```

`pet is Fish` is our type predicate in this example. A predicate takes the form `parameterName is Type`, where `parameterName` must be the name of a parameter from the current function signature.

Any time `isFish` is called with some variable, TypeScript will *narrow* that variable to that specific type if the original type is compatible.

```
// Both calls to 'swim' and 'fly' are now okay.

if (isFish(pet)) {
    pet.swim();
}
else {
    pet.fly();
}
```

Notice that TypeScript not only knows that `pet` is a `Fish` in the `if` branch; it also knows that in the `else` branch, you *don't have a* `Fish`, so you must have a `Bird`.

typeof type guards

Let's go back and write the code for the version of `padLeft` that uses union types. We could write it with type predicates as follows:

```

function isNumber(x: any): x is number {
    return typeof x === "number";
}

function isString(x: any): x is string {
    return typeof x === "string";
}

function padLeft(value: string, padding: string | number) {
    if (isNumber(padding)) {
        return Array(padding + 1).join(" ") + value;
    }
    if (isString(padding)) {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'`);
}

```

However, having to define a function to figure out if a type is a primitive is kind of a pain. Luckily, you don't need to abstract `typeof x === "number"` into its own function because TypeScript will recognize it as a type guard on its own. That means we could just write these checks inline.

```

function padLeft(value: string, padding: string | number) {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'`);
}

```

These `typeof` type guards are recognized in two different forms: `typeof v === "typename"` and `typeof v !== "typename"`, where `"typename"` must be `"number"`, `"string"`, `"boolean"`, or `"symbol"`. While TypeScript won't stop you from comparing to other strings, the language won't recognize those expressions as type guards.

instanceof type guards

If you've read about `typeof` type guards and are familiar with the `instanceof` operator in JavaScript, you probably have some idea of what this section is about.

`instanceof` type guards are a way of narrowing types using their constructor function. For instance, let's borrow our industrial string-padder example from earlier:

```

interface Padder {
    getPaddingString(): string
}

class SpaceRepeatingPadder implements Padder {
    constructor(private numSpaces: number) { }
    getPaddingString() {
        return Array(this.numSpaces + 1).join(" ");
    }
}

class StringPadder implements Padder {
    constructor(private value: string) { }
    getPaddingString() {
        return this.value;
    }
}

```

```

function getRandomPadder() {
    return Math.random() < 0.5 ?
        new SpaceRepeatingPadder(4) :
        new StringPadder(" ");
}

// Type is 'SpaceRepeatingPadder | StringPadder'
let padder: Padder = getRandomPadder();

if (padder instanceof SpaceRepeatingPadder) {
    padder; // type narrowed to 'SpaceRepeatingPadder'
}
if (padder instanceof StringPadder) {
    padder; // type narrowed to 'StringPadder'
}

```

The right side of the `instanceof` needs to be a constructor function, and TypeScript will narrow down to:

1. the type of the function's `prototype` property if its type is not `any`
2. the union of types returned by that type's construct signatures

in that order.

Nullable types

TypeScript has two special types, `null` and `undefined`, that have the values `null` and `undefined` respectively. We mentioned these briefly in [the Basic Types section](#). By default, the type checker considers `null` and `undefined` assignable to anything. Effectively, `null` and `undefined` are valid values of every type. That means it's not possible to stop them from being assigned to any type, even when you would like to prevent it. The inventor of `null`, Tony Hoare, calls this his "[billion dollar mistake](#)".

The `--strictNullChecks` flag fixes this: when you declare a variable, it doesn't automatically include `null` or `undefined`. You can include them explicitly using a union type:

```

let s = "foo";
s = null; // error, 'null' is not assignable to 'string'
let sn: string | null = "bar";
sn = null; // ok

sn = undefined; // error, 'undefined' is not assignable to 'string | null'

```

Note that TypeScript treats `null` and `undefined` differently in order to match JavaScript semantics. `string | null` is a different type than `string | undefined` and `string | undefined | null`.

Optional parameters and properties

With `--strictNullChecks`, an optional parameter automatically adds `| undefined`:

```

function f(x: number, y?: number) {
    return x + (y || 0);
}
f(1, 2);
f(1);
f(1, undefined);
f(1, null); // error, 'null' is not assignable to 'number | undefined'

```

The same is true for optional properties:

```
class C {
    a: number;
    b?: number;
}
let c = new C();
c.a = 12;
c.a = undefined; // error, 'undefined' is not assignable to 'number'
c.b = 13;
c.b = undefined; // ok
c.b = null; // error, 'null' is not assignable to 'number | undefined'
```

Type guards and type assertions

Since nullable types are implemented with a union, you need to use a type guard to get rid of the `null`. Fortunately, this is the same code you'd write in JavaScript:

```
function f(sn: string | null): string {
    if (sn == null) {
        return "default";
    }
    else {
        return sn;
    }
}
```

The `null` elimination is pretty obvious here, but you can use terser operators too:

```
function f(sn: string | null): string {
    return sn || "default";
}
```

In cases where the compiler can't eliminate `null` or `undefined`, you can use the type assertion operator to manually remove them. The syntax is postfix `! : identifier!` removes `null` and `undefined` from the type of `identifier`:

```
function broken(name: string | null): string {
    function postfix(epithet: string) {
        return name.charAt(0) + '. the ' + epithet; // error, 'name' is possibly null
    }
    name = name || "Bob";
    return postfix("great");
}

function fixed(name: string | null): string {
    function postfix(epithet: string) {
        return name!.charAt(0) + '. the ' + epithet; // ok
    }
    name = name || "Bob";
    return postfix("great");
}
```

The example uses a nested function here because the compiler can't eliminate nulls inside a nested function (except immediately-invoked function expressions). That's because it can't track all calls to the nested function, especially if you return it from the outer function. Without knowing where the function is called, it can't know what the type of `name` will be at the time the body executes.

Type Aliases

Type aliases create a new name for a type. Type aliases are sometimes similar to interfaces, but can name primitives, unions, tuples, and any other types that you'd otherwise have to write by hand.

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
  if (typeof n === "string") {
    return n;
  }
  else {
    return n();
  }
}
```

Aliasing doesn't actually create a new type - it creates a new *name* to refer to that type. Aliasing a primitive is not terribly useful, though it can be used as a form of documentation.

Just like interfaces, type aliases can also be generic - we can just add type parameters and use them on the right side of the alias declaration:

```
type Container<T> = { value: T };
```

We can also have a type alias refer to itself in a property:

```
type Tree<T> = {
  value: T;
  left: Tree<T>;
  right: Tree<T>;
}
```

Together with intersection types, we can make some pretty mind-bending types:

```
type LinkedList<T> = T & { next: LinkedList<T> };

interface Person {
  name: string;
}

var people: LinkedList<Person>;
var s = people.name;
var s = people.next.name;
var s = people.next.next.name;
var s = people.next.next.next.name;
```

However, it's not possible for a type alias to appear anywhere else on the right side of the declaration:

```
type Yikes = Array<Yikes>; // error
```

Interfaces vs. Type Aliases

As we mentioned, type aliases can act sort of like interfaces; however, there are some subtle differences.

One difference is that interfaces create a new name that is used everywhere. Type aliases don't create a new name — for instance, error messages won't use the alias name. In the code below, hovering over `interfaced` in an editor will show that it returns an `Interface`, but will show that `aliased` returns object literal type.

```
type Alias = { num: number }
interface Interface {
    num: number;
}
declare function aliased(arg: Alias): Alias;
declare function interfaced(arg: Interface): Interface;
```

A second more important difference is that type aliases cannot be extended or implemented from (nor can they extend/implement other types). Because [an ideal property of software is being open to extension](#), you should always use an interface over a type alias if possible.

On the other hand, if you can't express some shape with an interface and you need to use a union or tuple type, type aliases are usually the way to go.

String Literal Types

String literal types allow you to specify the exact value a string must have. In practice string literal types combine nicely with union types, type guards, and type aliases. You can use these features together to get enum-like behavior with strings.

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
class UIElement {
    animate(dx: number, dy: number, easing: Easing) {
        if (easing === "ease-in") {
            // ...
        }
        else if (easing === "ease-out") {
        }
        else if (easing === "ease-in-out") {
        }
        else {
            // error! should not pass null or undefined.
        }
    }
}

let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy"); // error: "uneasy" is not allowed here
```

You can pass any of the three allowed strings, but any other string will give the error

```
Argument of type '"uneasy"' is not assignable to parameter of type '"ease-in" | "ease-out" | "ease-in-out"'
```

String literal types can be used in the same way to distinguish overloads:

```
function createElement(tagName: "img"): HTMLImageElement;
function createElement(tagName: "input"): HTMLInputElement;
// ... more overloads ...
function createElement(tagName: string): Element {
    // ... code goes here ...
}
```

Numeric Literal Types

TypeScript also has numeric literal types.

```
function rollDie(): 1 | 2 | 3 | 4 | 5 | 6 {
    // ...
}
```

These are seldom written explicitly, they can be useful when narrowing can catch bugs:

```
function foo(x: number) {
    if (x !== 1 || x !== 2) {
        // ~~~~~
        // Operator '!==' cannot be applied to types '1' and '2'.
    }
}
```

In other words, `x` must be `1` when it gets compared to `2`, meaning that the above check is making an invalid comparison.

Enum Member Types

As mentioned in [our section on enums](#), enum members have types when every member is literal-initialized.

Much of the time when we talk about "singleton types", we're referring to both enum member types as well as numeric/string literal types, though many users will use "singleton types" and "literal types" interchangeably.

Discriminated Unions

You can combine singleton types, union types, type guards, and type aliases to build an advanced pattern called *discriminated unions*, also known as *tagged unions* or *algebraic data types*. Discriminated unions are useful in functional programming. Some languages automatically discriminate unions for you; TypeScript instead builds on JavaScript patterns as they exist today. There are three ingredients:

1. Types that have a common, singleton type property — the *discriminant*.
2. A type alias that takes the union of those types — the *union*.
3. Type guards on the common property.

```
interface Square {
    kind: "square";
    size: number;
}
interface Rectangle {
    kind: "rectangle";
    width: number;
    height: number;
}
interface Circle {
    kind: "circle";
    radius: number;
}
```

First we declare the interfaces we will union. Each interface has a `kind` property with a different string literal type. The `kind` property is called the *discriminant* or *tag*. The other properties are specific to each interface. Notice that the interfaces are currently unrelated. Let's put them into a union:

```
type Shape = Square | Rectangle | Circle;
```

Now let's use the discriminated union:

```
function area(s: Shape) {
  switch (s.kind) {
    case "square": return s.size * s.size;
    case "rectangle": return s.height * s.width;
    case "circle": return Math.PI * s.radius ** 2;
  }
}
```

Exhaustiveness checking

We would like the compiler to tell us when we don't cover all variants of the discriminated union. For example, if we add `Triangle` to `Shape`, we need to update `area` as well:

```
type Shape = Square | Rectangle | Circle | Triangle;
function area(s: Shape) {
  switch (s.kind) {
    case "square": return s.size * s.size;
    case "rectangle": return s.height * s.width;
    case "circle": return Math.PI * s.radius ** 2;
  }
  // should error here - we didn't handle case "triangle"
}
```

There are two ways to do this. The first is to turn on `--strictNullChecks` and specify a return type:

```
function area(s: Shape): number { // error: returns number | undefined
  switch (s.kind) {
    case "square": return s.size * s.size;
    case "rectangle": return s.height * s.width;
    case "circle": return Math.PI * s.radius ** 2;
  }
}
```

Because the `switch` is no longer exhaustive, TypeScript is aware that the function could sometimes return `undefined`. If you have an explicit return type `number`, then you will get an error that the return type is actually `number | undefined`. However, this method is quite subtle and, besides, `--strictNullChecks` does not always work with old code.

The second method uses the `never` type that the compiler uses to check for exhaustiveness:

```
function assertNever(x: never): never {
  throw new Error("Unexpected object: " + x);
}
function area(s: Shape) {
  switch (s.kind) {
    case "square": return s.size * s.size;
    case "rectangle": return s.height * s.width;
    case "circle": return Math.PI * s.radius ** 2;
    default: return assertNever(s); // error here if there are missing cases
  }
}
```

```
}
```

Here, `assertNever` checks that `s` is of type `never` — the type that's left after all other cases have been removed. If you forget a case, then `s` will have a real type and you will get a type error. This method requires you to define an extra function, but it's much more obvious when you forget it.

Polymorphic `this` types

A polymorphic `this` type represents a type that is the *subtype* of the containing class or interface. This is called *F*-bounded polymorphism. This makes hierarchical fluent interfaces much easier to express, for example. Take a simple calculator that returns `this` after each operation:

```
class BasicCalculator {
    public constructor(protected value: number = 0) { }
    public currentValue(): number {
        return this.value;
    }
    public add(operand: number): this {
        this.value += operand;
        return this;
    }
    public multiply(operand: number): this {
        this.value *= operand;
        return this;
    }
    // ... other operations go here ...
}

let v = new BasicCalculator(2)
    .multiply(5)
    .add(1)
    .currentValue();
```

Since the class uses `this` types, you can extend it and the new class can use the old methods with no changes.

```
class ScientificCalculator extends BasicCalculator {
    public constructor(value = 0) {
        super(value);
    }
    public sin() {
        this.value = Math.sin(this.value);
        return this;
    }
    // ... other operations go here ...
}

let v = new ScientificCalculator(2)
    .multiply(5)
    .sin()
    .add(1)
    .currentValue();
```

Without `this` types, `ScientificCalculator` would not have been able to extend `BasicCalculator` and keep the fluent interface. `multiply` would have returned `BasicCalculator`, which doesn't have the `sin` method. However, with `this` types, `multiply` returns `this`, which is `ScientificCalculator` here.

Index types

With index types, you can get the compiler to check code that uses dynamic property names. For example, a common Javascript pattern is to pick a subset of properties from an object:

```
function pluck(o, names) {
    return names.map(n => o[n]);
}
```

Here's how you would write and use this function in TypeScript, using the **index type query** and **indexed access** operators:

```
function pluck<T, K extends keyof T>(o: T, names: K[]): T[K][] {
    return names.map(n => o[n]);
}

interface Person {
    name: string;
    age: number;
}
let person: Person = {
    name: 'Jarid',
    age: 35
};
let strings: string[] = pluck(person, ['name']); // ok, string[]
```

The compiler checks that `name` is actually a property on `Person`. The example introduces a couple of new type operators. First is `keyof T`, the **index type query operator**. For any type `T`, `keyof T` is the union of known, public property names of `T`. For example:

```
let personProps: keyof Person; // 'name' | 'age'
```

`keyof Person` is completely interchangeable with `'name' | 'age'`. The difference is that if you add another property to `Person`, say `address: string`, then `keyof Person` will automatically update to be `'name' | 'age' | 'address'`. And you can use `keyof` in generic contexts like `pluck`, where you can't possibly know the property names ahead of time. That means the compiler will check that you pass the right set of property names to `pluck`:

```
pluck(person, ['age', 'unknown']); // error, 'unknown' is not in 'name' | 'age'
```

The second operator is `T[K]`, the **indexed access operator**. Here, the type syntax reflects the expression syntax. That means that `person['name']` has the type `Person['name']` — which in our example is just `string`. However, just like index type queries, you can use `T[K]` in a generic context, which is where its real power comes to life. You just have to make sure that the type variable `K extends keyof T`. Here's another example with a function named `getProperty`.

```
function getProperty<T, K extends keyof T>(o: T, name: K): T[K] {
    return o[name]; // o[name] is of type T[K]
}
```

In `getProperty`, `o: T` and `name: K`, so that means `o[name]: T[K]`. Once you return the `T[K]` result, the compiler will instantiate the actual type of the key, so the return type of `getProperty` will vary according to which property you request.

```
let name: string = getProperty(person, 'name');
let age: number = getProperty(person, 'age');
let unknown = getProperty(person, 'unknown'); // error, 'unknown' is not in 'name' | 'age'
```

Index types and string index signatures

`keyof T[K]` interact with string index signatures. If you have a type with a string index signature, `keyof T` will just be `string`. And `T[string]` is just the type of the index signature:

```
interface Map<T> {
  [key: string]: T;
}
let keys: keyof Map<number>; // string
let value: Map<number>['foo']; // number
```

Mapped types

A common task is to take an existing type and make each of its properties optional:

```
interface PersonPartial {
  name?: string;
  age?: number;
}
```

Or we might want a readonly version:

```
interface PersonReadonly {
  readonly name: string;
  readonly age: number;
}
```

This happens often enough in Javascript that TypeScript provides a way to create new types based on old types — **mapped types**. In a mapped type, the new type transforms each property in the old type in the same way. For example, you can make all properties of a type `readonly` or optional. Here are a couple of examples:

```
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
}
type Partial<T> = {
  [P in keyof T]?: T[P];
}
```

And to use it:

```
type PersonPartial = Partial<Person>;
type ReadonlyPerson = Readonly<Person>;
```

Let's take a look at the simplest mapped type and its parts:

```
type Keys = 'option1' | 'option2';
type Flags = { [K in Keys]: boolean };
```

The syntax resembles the syntax for index signatures with a `for .. in` inside. There are three parts:

1. The type variable `K`, which gets bound to each property in turn.
2. The string literal union `Keys`, which contains the names of properties to iterate over.
3. The resulting type of the property.

In this simple example, `Keys` is a hard-coded list of property names and the property type is always `boolean`, so this mapped type is equivalent to writing:

```
type Flags = {
    option1: boolean;
    option2: boolean;
}
```

Real applications, however, look like `Readonly` or `Partial` above. They're based on some existing type, and they transform the properties in some way. That's where `keyof` and indexed access types come in:

```
type NullablePerson = { [P in keyof Person]: Person[P] | null }
type PartialPerson = { [P in keyof Person]?: Person[P] }
```

But it's more useful to have a general version.

```
type Nullable<T> = { [P in keyof T]: T[P] | null }
type Partial<T> = { [P in keyof T]?: T[P] }
```

In these examples, the properties list is `keyof T` and the resulting type is some variant of `T[P]`. This is a good template for any general use of mapped types. That's because this kind of transformation is **homomorphic**, which means that the mapping applies only to properties of `T` and no others. The compiler knows that it can copy all the existing property modifiers before adding any new ones. For example, if `Person.name` was readonly, `Partial<Person>.name` would be readonly and optional.

Here's one more example, in which `T[P]` is wrapped in a `Proxy<T>` class:

```
type Proxy<T> = {
    get(): T;
    set(value: T): void;
}
type Proxify<T> = {
    [P in keyof T]: Proxy<T[P]>;
}
function proxify<T>(o: T): Proxify<T> {
    // ... wrap proxies ...
}
let proxyProps = proxify(props);
```

Note that `Readonly<T>` and `Partial<T>` are so useful, they are included in TypeScript's standard library along with `Pick` and `Record`:

```
type Pick<T, K extends keyof T> = {
    [P in K]: T[P];
}
type Record<K extends string, T> = {
    [P in K]: T;
}
```

`Readonly`, `Partial` and `Pick` are homomorphic whereas `Record` is not. One clue that `Record` is not homomorphic is that it doesn't take an input type to copy properties from:

```
type ThreeStringProps = Record<'prop1' | 'prop2' | 'prop3', string>
```

Non-homomorphic types are essentially creating new properties, so they can't copy property modifiers from anywhere.

Inference from mapped types

Now that you know how to wrap the properties of a type, the next thing you'll want to do is unwrap them. Fortunately, that's pretty easy:

```
function unproxify<T>(t: Proxify<T>): T {
  let result = {} as T;
  for (const k in t) {
    result[k] = t[k].get();
  }
  return result;
}

let originalProps = unproxify(proxyProps);
```

Note that this unwrapping inference only works on homomorphic mapped types. If the mapped type is not homomorphic you'll have to give an explicit type parameter to your unwrapping function.

Conditional Types

TypeScript 2.8 introduces *conditional types* which add the ability to express non-uniform type mappings. A conditional type selects one of two possible types based on a condition expressed as a type relationship test:

```
T extends U ? X : Y
```

The type above means when `T` is assignable to `U` the type is `X`, otherwise the type is `Y`.

A conditional type `T extends U ? X : Y` is either *resolved* to `X` or `Y`, or *deferred* because the condition depends on one or more type variables. When `T` or `U` contains type variables, whether to resolve to `X` or `Y`, or to defer, is determined by whether or not the type system has enough information to conclude that `T` is always assignable to `U`.

As an example of some types that are immediately resolved, we can take a look at the following example:

```
declare function f<T extends boolean>(x: T): T extends true ? string : number;

// Type is 'string' | number
let x = f(Math.random() < 0.5)
```

Another example would be the `TypeName` type alias, which uses nested conditional types:

```
type TypeName<T> =
  T extends string ? "string" :
  T extends number ? "number" :
  T extends boolean ? "boolean" :
  T extends undefined ? "undefined" :
  T extends Function ? "function" :
  "object";

type T0 = TypeName<string>; // "string"
type T1 = TypeName<"a">; // "string"
type T2 = TypeName<true>; // "boolean"
type T3 = TypeName<() => void>; // "function"
type T4 = TypeName<string[]>; // "object"
```

But as an example of a place where conditional types are deferred - where they stick around instead of picking a branch - would be in the following:

```
interface Foo {
    propA: boolean;
    propB: boolean;
}

declare function f<T>(x: T): T extends Foo ? string : number;

function foo<U>(x: U) {
    // Has type 'U extends Foo ? string : number'
    let a = f(x);

    // This assignment is allowed though!
    let b: string | number = a;
}
```

In the above, the variable `a` has a conditional type that hasn't yet chosen a branch. When another piece of code ends up calling `foo`, it will substitute in `U` with some other type, and TypeScript will re-evaluate the conditional type, deciding whether it can actually pick a branch.

In the meantime, we can assign a conditional type to any other target type as long as each branch of the conditional is assignable to that target. So in our example above we were able to assign `U extends Foo ? string : number` to `string | number` since no matter what the conditional evaluates to, it's known to be either `string` or `number`.

Distributive conditional types

Conditional types in which the checked type is a naked type parameter are called *distributive conditional types*. Distributive conditional types are automatically distributed over union types during instantiation. For example, an instantiation of `T extends U ? X : Y` with the type argument `A | B | C` for `T` is resolved as `(A extends U ? X : Y) | (B extends U ? X : Y) | (C extends U ? X : Y)`.

Example

```
type T10 = TypeName<string | (() => void)>; // "string" | "function"
type T12 = TypeName<string | string[] | undefined>; // "string" | "object" | "undefined"
type T11 = TypeName<string[] | number[]>; // "object"
```

In instantiations of a distributive conditional type `T extends U ? X : Y`, references to `T` within the conditional type are resolved to individual constituents of the union type (i.e. `T` refers to the individual constituents *after* the conditional type is distributed over the union type). Furthermore, references to `T` within `X` have an additional type parameter constraint `U` (i.e. `T` is considered assignable to `U` within `X`).

Example

```
type BoxedValue<T> = { value: T };
type BoxedArray<T> = { array: T[] };
type Boxed<T> = T extends any[] ? BoxedArray<T[number]> : BoxedValue<T>;

type T20 = Boxed<string>; // BoxedValue<string>;
type T21 = Boxed<number[]>; // BoxedArray<number>;
type T22 = Boxed<string | number[]>; // BoxedValue<string> | BoxedArray<number>;
```

Notice that `T` has the additional constraint `any[]` within the true branch of `Boxed<T>` and it is therefore possible to refer to the element type of the array as `T[number]`. Also, notice how the conditional type is distributed over the union type in the last example.

The distributive property of conditional types can conveniently be used to *filter* union types:

```
type Diff<T, U> = T extends U ? never : T; // Remove types from T that are assignable to U
type Filter<T, U> = T extends U ? T : never; // Remove types from T that are not assignable to U

type T30 = Diff<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "b" | "d"
type T31 = Filter<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "a" | "c"
type T32 = Diff<string | number | (() => void), Function>; // string | number
type T33 = Filter<string | number | (() => void), Function>; // () => void

type NonNullable<T> = Diff<T, null | undefined>; // Remove null and undefined from T

type T34 = NonNullable<string | number | undefined>; // string | number
type T35 = NonNullable<string | string[] | null | undefined>; // string | string[]

function f1<T>(x: T, y: NonNullable<T>) {
  x = y; // Ok
  y = x; // Error
}

function f2<T extends string | undefined>(x: T, y: NonNullable<T>) {
  x = y; // Ok
  y = x; // Error
  let s1: string = x; // Error
  let s2: string = y; // Ok
}
```

Conditional types are particularly useful when combined with mapped types:

```
type FunctionPropertyNames<T> = { [K in keyof T]: T[K] extends Function ? K : never }[keyof T];
type FunctionProperties<T> = Pick<T, FunctionPropertyNames<T>>;

type NonFunctionPropertyNames<T> = { [K in keyof T]: T[K] extends Function ? never : K }[keyof T];
type NonFunctionProperties<T> = Pick<T, NonFunctionPropertyNames<T>>;

interface Part {
  id: number;
  name: string;
  subparts: Part[];
  updatePart(newName: string): void;
}

type T40 = FunctionPropertyNames<Part>; // "updatePart"
type T41 = NonFunctionPropertyNames<Part>; // "id" | "name" | "subparts"
type T42 = FunctionProperties<Part>; // { updatePart(newName: string): void }
type T43 = NonFunctionProperties<Part>; // { id: number, name: string, subparts: Part[] }
```

Similar to union and intersection types, conditional types are not permitted to reference themselves recursively. For example the following is an error.

Example

```
type ElementType<T> = T extends any[] ? ElementType<T[number]> : T; // Error
```

Type inference in conditional types

Within the `extends` clause of a conditional type, it is now possible to have `infer` declarations that introduce a type variable to be inferred. Such inferred type variables may be referenced in the true branch of the conditional type. It is possible to have multiple `infer` locations for the same type variable.

For example, the following extracts the return type of a function type:

```
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;
```

Conditional types can be nested to form a sequence of pattern matches that are evaluated in order:

```
type Unpacked<T> =
  T extends (infer U)[] ? U :
  T extends (...args: any[]) => infer U ? U :
  T extends Promise<infer U> ? U :
  T;

type T0 = Unpacked<string>; // string
type T1 = Unpacked<string[]>; // string
type T2 = Unpacked<() => string>; // string
type T3 = Unpacked<Promise<string>>; // string
type T4 = Unpacked<Promise<string>[]>; // Promise<string>
type T5 = Unpacked<Unpacked<Promise<string>[]>>; // string
```

The following example demonstrates how multiple candidates for the same type variable in co-variant positions causes a union type to be inferred:

```
type Foo<T> = T extends { a: infer U, b: infer U } ? U : never;
type T10 = Foo<{ a: string, b: string }>; // string
type T11 = Foo<{ a: string, b: number }>; // string | number
```

Likewise, multiple candidates for the same type variable in contra-variant positions causes an intersection type to be inferred:

```
type Bar<T> = T extends { a: (x: infer U) => void, b: (x: infer U) => void } ? U : never;
type T20 = Bar<{ a: (x: string) => void, b: (x: string) => void }>; // string
type T21 = Bar<{ a: (x: string) => void, b: (x: number) => void }>; // string & number
```

When inferring from a type with multiple call signatures (such as the type of an overloaded function), inferences are made from the *last* signature (which, presumably, is the most permissive catch-all case). It is not possible to perform overload resolution based on a list of argument types.

```
declare function foo(x: string): number;
declare function foo(x: number): string;
declare function foo(x: string | number): string | number;
type T30 = ReturnType<typeof foo>; // string | number
```

It is not possible to use `infer` declarations in constraint clauses for regular type parameters:

```
type ReturnType<T extends (...args: any[]) => infer R> = R; // Error, not supported
```

However, much the same effect can be obtained by erasing the type variables in the constraint and instead specifying a conditional type:

```
type AnyFunction = (...args: any[]) => any;
type ReturnType<T extends AnyFunction> = T extends (...args: any[]) => infer R ? R : any;
```

Predefined conditional types

TypeScript 2.8 adds several predefined conditional types to `lib.d.ts`:

- `Exclude<T, U>` -- Exclude from `T` those types that are assignable to `U`.
- `Extract<T, U>` -- Extract from `T` those types that are assignable to `U`.
- `NonNullable<T>` -- Exclude `null` and `undefined` from `T`.
- `ReturnType<T>` -- Obtain the return type of a function type.
- `InstanceType<T>` -- Obtain the instance type of a constructor function type.

Example

```
type T00 = Exclude<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "b" | "d"
type T01 = Extract<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "a" | "c"

type T02 = Exclude<string | number | (() => void), Function>; // string | number
type T03 = Extract<string | number | (() => void), Function>; // () => void

type T04 = NonNullable<string | number | undefined>; // string | number
type T05 = NonNullable<(() => string) | string[] | null | undefined>; // () => string) | string[]

function f1(s: string) {
    return { a: 1, b: s };
}

class C {
    x = 0;
    y = 0;
}

type T10 = ReturnType<() => string>; // string
type T11 = ReturnType<(s: string) => void>; // void
type T12 = ReturnType<(<T>() => T)>; // {}
type T13 = ReturnType<(<T extends U, U extends number[]>() => T)>; // number[]
type T14 = ReturnType<typeof f1>; // { a: number, b: string }
type T15 = ReturnType<any>; // any
type T16 = ReturnType<never>; // any
type T17 = ReturnType<string>; // Error
type T18 = ReturnType<Function>; // Error

type T20 = InstanceType<typeof C>; // C
type T21 = InstanceType<any>; // any
type T22 = InstanceType<never>; // any
type T23 = InstanceType<string>; // Error
type T24 = InstanceType<Function>; // Error
```

Note: The `Exclude` type is a proper implementation of the `Diff` type suggested [here](#). We've used the name `Exclude` to avoid breaking existing code that defines a `Diff`, plus we feel that name better conveys the semantics of the type. We did not include the `Omit<T, K>` type because it is trivially written as `Pick<T, Exclude<keyof T, K>>`.

Introduction

Starting with ECMAScript 2015, `symbol` is a primitive data type, just like `number` and `string`.

`symbol` values are created by calling the `Symbol` constructor.

```
let sym1 = Symbol();
let sym2 = Symbol("key"); // optional string key
```

Symbols are immutable, and unique.

```
let sym2 = Symbol("key");
let sym3 = Symbol("key");

sym2 === sym3; // false, symbols are unique
```

Just like strings, symbols can be used as keys for object properties.

```
let sym = Symbol();

let obj = {
  [sym]: "value"
};

console.log(obj[sym]); // "value"
```

Symbols can also be combined with computed property declarations to declare object properties and class members.

```
const getClassnameSymbol = Symbol();

class C {
  [getClassnameSymbol](){
    return "C";
  }
}

let c = new C();
let className = c[getClassnameSymbol](); // "C"
```

Well-known Symbols

In addition to user-defined symbols, there are well-known built-in symbols. Built-in symbols are used to represent internal language behaviors.

Here is a list of well-known symbols:

`Symbol.hasInstance`

A method that determines if a constructor object recognizes an object as one of the constructor's instances. Called by the semantics of the `instanceof` operator.

Symbol.isConcatSpreadable

A Boolean value indicating that an object should be flattened to its array elements by `Array.prototype.concat`.

Symbol.iterator

A method that returns the default iterator for an object. Called by the semantics of the `for-of` statement.

Symbol.match

A regular expression method that matches the regular expression against a string. Called by the `String.prototype.match` method.

Symbol.replace

A regular expression method that replaces matched substrings of a string. Called by the `String.prototype.replace` method.

Symbol.search

A regular expression method that returns the index within a string that matches the regular expression. Called by the `String.prototype.search` method.

Symbol.species

A function valued property that is the constructor function that is used to create derived objects.

Symbol.split

A regular expression method that splits a string at the indices that match the regular expression. Called by the `String.prototype.split` method.

Symbol.toPrimitive

A method that converts an object to a corresponding primitive value. Called by the `ToPrimitive` abstract operation.

Symbol.toStringTag

A String value that is used in the creation of the default string description of an object. Called by the built-in method `Object.prototype.toString`.

Symbol.unscopables

An Object whose own property names are property names that are excluded from the 'with' environment bindings of the associated objects.

Iterables

An object is deemed iterable if it has an implementation for the `Symbol.iterator` property. Some built-in types like `Array`, `Map`, `Set`, `String`, `Int32Array`, `Uint32Array`, etc. have their `Symbol.iterator` property already implemented. `Symbol.iterator` function on an object is responsible for returning the list of values to iterate on.

for..of statements

`for..of` loops over an iterable object, invoking the `Symbol.iterator` property on the object. Here is a simple `for..of` loop on an array:

```
let someArray = [1, "string", false];

for (let entry of someArray) {
  console.log(entry); // 1, "string", false
}
```

for..of vs. for..in statements

Both `for..of` and `for..in` statements iterate over lists; the values iterated on are different though, `for..in` returns a list of *keys* on the object being iterated, whereas `for..of` returns a list of *values* of the numeric properties of the object being iterated.

Here is an example that demonstrates this distinction:

```
let list = [4, 5, 6];

for (let i in list) {
  console.log(i); // "0", "1", "2",
}

for (let i of list) {
  console.log(i); // "4", "5", "6"
}
```

Another distinction is that `for..in` operates on any object; it serves as a way to inspect properties on this object. `for..of` on the other hand, is mainly interested in values of iterable objects. Built-in objects like `Map` and `Set` implement `Symbol.iterator` property allowing access to stored values.

```
let pets = new Set(["Cat", "Dog", "Hamster"]);
pets["species"] = "mammals";

for (let pet in pets) {
  console.log(pet); // "species"
}

for (let pet of pets) {
  console.log(pet); // "Cat", "Dog", "Hamster"
}
```

Code generation

Targeting ES5 and ES3

When targeting an ES5 or ES3, iterators are only allowed on values of `Array` type. It is an error to use `for..of` loops on non-`Array` values, even if these non-`Array` values implement the `Symbol.iterator` property.

The compiler will generate a simple `for` loop for a `for..of` loop, for instance:

```
let numbers = [1, 2, 3];
for (let num of numbers) {
    console.log(num);
}
```

will be generated as:

```
var numbers = [1, 2, 3];
for (var _i = 0; _i < numbers.length; _i++) {
    var num = numbers[_i];
    console.log(num);
}
```

Targeting ECMAScript 2015 and higher

When targeting an ECMAScript 2015-compliant engine, the compiler will generate `for..of` loops to target the built-in iterator implementation in the engine.

A note about terminology: It's important to note that in TypeScript 1.5, the nomenclature has changed. "Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with [ECMAScript 2015](#)'s terminology, (namely that `module x { }` is equivalent to the now-preferred `namespace x { }`).

Introduction

Starting with ECMAScript 2015, JavaScript has a concept of modules. TypeScript shares this concept.

Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the `export` forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the `import` forms.

Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.

Modules import one another using a module loader. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it. Well-known modules loaders used in JavaScript are the [CommonJS](#) module loader for Node.js and [require.js](#) for Web applications.

In TypeScript, just as in ECMAScript 2015, any file containing a top-level `import` or `export` is considered a module. Conversely, a file without any top-level `import` or `export` declarations is treated as a script whose contents are available in the global scope (and therefore to modules as well).

Export

Exporting a declaration

Any declaration (such as a variable, function, class, type alias, or interface) can be exported by adding the `export` keyword.

Validation.ts

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

ZipCodeValidator.ts

```
export const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Export statements

Export statements are handy when exports need to be renamed for consumers, so the above example can be written as:

```

class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export { ZipCodeValidator };
export { ZipCodeValidator as mainValidator };

```

Re-exports

Often modules extend other modules, and partially expose some of their features. A re-export does not import it locally, or introduce a local variable.

`ParseIntBasedZipCodeValidator.ts`

```

export class ParseIntBasedZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && parseInt(s).toString() === s;
    }
}

// Export original validator but rename it
export {ZipCodeValidator as RegExpBasedZipCodeValidator} from "./ZipCodeValidator";

```

Optionally, a module can wrap one or more modules and combine all their exports using `export * from "module"` syntax.

`AllValidators.ts`

```

export * from "./StringValidator"; // exports interface 'StringValidator'
export * from "./LettersOnlyValidator"; // exports class 'LettersOnlyValidator'
export * from "./ZipCodeValidator"; // exports class 'ZipCodeValidator'

```

Import

Importing is just about as easy as exporting from a module. Importing an exported declaration is done through using one of the `import` forms below:

Import a single export from a module

```

import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();

```

imports can also be renamed

```

import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();

```

Import the entire module into a single variable, and use it to access the module exports

```
import * as validator from "./ZipCodeValidator";
let myValidator = new validator.ZipCodeValidator();
```

Import a module for side-effects only

Though not recommended practice, some modules set up some global state that can be used by other modules. These modules may not have any exports, or the consumer is not interested in any of their exports. To import these modules, use:

```
import "./my-module.js";
```

Default exports

Each module can optionally export a `default` export. Default exports are marked with the keyword `default`; and there can only be one `default` export per module. `default` exports are imported using a different import form.

`default` exports are really handy. For instance, a library like JQuery might have a default export of `jquery` or `$`, which we'd probably also import under the name `$` or `jquery`.

JQuery.d.ts

```
declare let $: JQuery;
export default $;
```

App.ts

```
import $ from "JQuery";

$("button.continue").html( "Next Step..." );
```

Classes and function declarations can be authored directly as default exports. Default export class and function declaration names are optional.

ZipCodeValidator.ts

```
export default class ZipCodeValidator {
    static numberRegexp = /^[0-9]+$/;
    isAcceptable(s: string) {
        return s.length === 5 && ZipCodeValidator.numberRegexp.test(s);
    }
}
```

Test.ts

```
import validator from "./ZipCodeValidator";

let myValidator = new validator();
```

or

StaticZipCodeValidator.ts

```
const numberRegexp = /^[0-9]+$/;

export default function (s: string) {
    return s.length === 5 && numberRegexp.test(s);
}
```

Test.ts

```
import validate from "./StaticZipCodeValidator";

let strings = ["Hello", "98052", "101"];

// Use function validate
strings.forEach(s => {
    console.log(`"${s}" ${validate(s) ? "matches" : "does not match"}`);
});
```

`default exports` can also be just values:

OneTwoThree.ts

```
export default "123";
```

Log.ts

```
import num from "./OneTwoThree";

console.log(num); // "123"
```

export = and import = require()

Both CommonJS and AMD generally have the concept of an `exports` object which contains all exports from a module.

They also support replacing the `exports` object with a custom single object. Default exports are meant to act as a replacement for this behavior; however, the two are incompatible. TypeScript supports `export =` to model the traditional CommonJS and AMD workflow.

The `export =` syntax specifies a single object that is exported from the module. This can be a class, interface, namespace, function, or enum.

When exporting a module using `export =`, TypeScript-specific `import module = require("module")` must be used to import the module.

ZipCodeValidator.ts

```
let numberRegexp = /^[0-9]+$/;
class ZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

```
export = ZipCodeValidator;
```

Test.ts

```
import zip = require("./ZipCodeValidator");

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validator = new zip();

// Show whether each string passed each validator
strings.forEach(s => {
  console.log(`"${s}" - ${validator.isAcceptable(s) ? "matches" : "does not match"}`);
});
```

Code Generation for Modules

Depending on the module target specified during compilation, the compiler will generate appropriate code for Node.js ([CommonJS](#)), [require.js \(AMD\)](#), [UMD](#), [SystemJS](#), or [ECMAScript 2015 native modules](#) (ES6) module-loading systems. For more information on what the `define`, `require` and `register` calls in the generated code do, consult the documentation for each module loader.

This simple example shows how the names used during importing and exporting get translated into the module loading code.

SimpleModule.ts

```
import m = require("mod");
export let t = m.something + 1;
```

AMD / RequireJS SimpleModule.js

```
define(["require", "exports", "./mod"], function (require, exports, mod_1) {
  exports.t = mod_1.something + 1;
});
```

CommonJS / Node SimpleModule.js

```
var mod_1 = require("./mod");
exports.t = mod_1.something + 1;
```

UMD SimpleModule.js

```
(function (factory) {
  if (typeof module === "object" && typeof module.exports === "object") {
    var v = factory(require, exports); if (v !== undefined) module.exports = v;
  }
  else if (typeof define === "function" && define.amd) {
    define(["require", "exports", "./mod"], factory);
  }
})(function (require, exports) {
  var mod_1 = require("./mod");
  exports.t = mod_1.something + 1;
});
```

System SimpleModule.js

```
System.register(["./mod"], function(exports_1) {
    var mod_1;
    var t;
    return {
        setters:[
            function (mod_1_1) {
                mod_1 = mod_1_1;
            }],
        execute: function() {
            exports_1("t", t = mod_1.something + 1);
        }
    });
});
```

Native ECMAScript 2015 modules SimpleModule.js

```
import { something } from "./mod";
export var t = something + 1;
```

Simple Example

Below, we've consolidated the Validator implementations used in previous examples to only export a single named export from each module.

To compile, we must specify a module target on the command line. For Node.js, use `--module commonjs`; for require.js, use `--module amd`. For example:

```
tsc --module commonjs Test.ts
```

When compiled, each module will become a separate `.js` file. As with reference tags, the compiler will follow `import` statements to compile dependent files.

Validation.ts

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

LettersOnlyValidator.ts

```
import { StringValidator } from "./Validation";

const lettersRegexp = /^[A-Za-z]+$/;

export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}
```

ZipCodeValidator.ts

```
import { StringValidator } from "./Validation";

const numberRegexp = /^[0-9]+$/;
```

```
export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Test.ts

```
import { StringValidator } from "./Validation";
import { ZipCodeValidator } from "./ZipCodeValidator";
import { LettersOnlyValidator } from "./LettersOnlyValidator";

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// Show whether each string passed each validator
strings.forEach(s => {
    for (let name in validators) {
        console.log(`"${s}" - ${validators[name].isAcceptable(s) ? "matches" : "does not match"} ${name}`);
    }
});
```

Optional Module Loading and Other Advanced Loading Scenarios

In some cases, you may want to only load a module under some conditions. In TypeScript, we can use the pattern shown below to implement this and other advanced loading scenarios to directly invoke the module loaders without losing type safety.

The compiler detects whether each module is used in the emitted JavaScript. If a module identifier is only ever used as part of a type annotations and never as an expression, then no `require` call is emitted for that module. This elision of unused references is a good performance optimization, and also allows for optional loading of those modules.

The core idea of the pattern is that the `import id = require("...")` statement gives us access to the types exposed by the module. The module loader is invoked (through `require`) dynamically, as shown in the `if` blocks below. This leverages the reference-elision optimization so that the module is only loaded when needed. For this pattern to work, it's important that the symbol defined via an `import` is only used in type positions (i.e. never in a position that would be emitted into the JavaScript).

To maintain type safety, we can use the `typeof` keyword. The `typeof` keyword, when used in a type position, produces the type of a value, in this case the type of the module.

Dynamic Module Loading in Node.js

```
declare function require(moduleName: string): any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    let ZipCodeValidator: typeof Zip = require("./ZipCodeValidator");
    let validator = new ZipCodeValidator();
    if (validator.isAcceptable("...")) { /* ... */ }
```

```
}
```

Sample: Dynamic Module Loading in require.js

```
declare function require(moduleNames: string[], onLoad: (...args: any[]) => void): void;

import * as Zip from "./ZipCodeValidator";

if (needZipValidation) {
    require(["./ZipCodeValidator"], (ZipCodeValidator: typeof Zip) => {
        let validator = new ZipCodeValidator.ZipCodeValidator();
        if (validator.isAcceptable("...")) { /* ... */ }
    });
}
```

Sample: Dynamic Module Loading in System.js

```
declare const System: any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    System.import("./ZipCodeValidator").then((ZipCodeValidator: typeof Zip) => {
        var x = new ZipCodeValidator();
        if (x.isAcceptable("...")) { /* ... */ }
    });
}
```

Working with Other JavaScript Libraries

To describe the shape of libraries not written in TypeScript, we need to declare the API that the library exposes.

We call declarations that don't define an implementation "ambient". Typically, these are defined in `.d.ts` files. If you're familiar with C/C++, you can think of these as `.h` files. Let's look at a few examples.

Ambient Modules

In Node.js, most tasks are accomplished by loading one or more modules. We could define each module in its own `.d.ts` file with top-level export declarations, but it's more convenient to write them as one larger `.d.ts` file. To do so, we use a construct similar to ambient namespaces, but we use the `module` keyword and the quoted name of the module which will be available to a later import. For example:

node.d.ts (simplified excerpt)

```
declare module "url" {
    export interface Url {
        protocol?: string;
        hostname?: string;
        pathname?: string;
    }

    export function parse(urlStr: string, parseQueryString?, slashesDenoteHost?): Url;
}

declare module "path" {
    export function normalize(p: string): string;
    export function join(...paths: any[]): string;
    export var sep: string;
```

```
}
```

Now we can `/// <reference>` `node.d.ts` and then load the modules using `import url = require("url");` or `import * as URL from "url".`

```
/// <reference path="node.d.ts"/>
import * as URL from "url";
let myUrl = URL.parse("http://www.typescriptlang.org");
```

Shorthand ambient modules

If you don't want to take the time to write out declarations before using a new module, you can use a shorthand declaration to get started quickly.

declarations.d.ts

```
declare module "hot-new-module";
```

All imports from a shorthand module will have the `any` type.

```
import x, {y} from "hot-new-module";
x(y);
```

Wildcard module declarations

Some module loaders such as [SystemJS](#) and [AMD](#) allow non-JavaScript content to be imported. These typically use a prefix or suffix to indicate the special loading semantics. Wildcard module declarations can be used to cover these cases.

```
declare module "*!text" {
    const content: string;
    export default content;
}
// Some do it the other way around.
declare module "json!*" {
    const value: any;
    export default value;
}
```

Now you can import things that match `"*!text"` or `"json!*"`.

```
import fileContent from "./xyz.txt!text";
import data from "json!http://example.com/data.json";
console.log(data, fileContent);
```

UMD modules

Some libraries are designed to be used in many module loaders, or with no module loading (global variables). These are known as [UMD](#) modules. These libraries can be accessed through either an import or a global variable. For example:

math-lib.d.ts

```
export function isPrime(x: number): boolean;
```

```
export as namespace mathLib;
```

The library can then be used as an import within modules:

```
import { isPrime } from "math-lib";
isPrime(2);
mathLib.isPrime(2); // ERROR: can't use the global definition from inside a module
```

It can also be used as a global variable, but only inside of a script. (A script is a file with no imports or exports.)

```
mathLib.isPrime(2);
```

Guidance for structuring modules

Export as close to top-level as possible

Consumers of your module should have as little friction as possible when using things that you export. Adding too many levels of nesting tends to be cumbersome, so think carefully about how you want to structure things.

Exporting a namespace from your module is an example of adding too many layers of nesting. While namespaces sometime have their uses, they add an extra level of indirection when using modules. This can quickly become a pain point for users, and is usually unnecessary.

Static methods on an exported class have a similar problem - the class itself adds a layer of nesting. Unless it increases expressivity or intent in a clearly useful way, consider simply exporting a helper function.

If you're only exporting a single `class` or `function`, use `export default`

Just as "exporting near the top-level" reduces friction on your module's consumers, so does introducing a default export. If a module's primary purpose is to house one specific export, then you should consider exporting it as a default export. This makes both importing and actually using the import a little easier. For example:

MyClass.ts

```
export default class SomeType {
  constructor() { ... }
}
```

MyFunc.ts

```
export default function getThing() { return "thing"; }
```

Consumer.ts

```
import t from "./MyClass";
import f from "./MyFunc";
let x = new t();
console.log(f());
```

This is optimal for consumers. They can name your type whatever they want (`t` in this case) and don't have to do any excessive dotting to find your objects.

If you're exporting multiple objects, put them all at top-level

MyThings.ts

```
export class SomeType { /* ... */ }
export function someFunc() { /* ... */ }
```

Conversely when importing:

Explicitly list imported names

Consumer.ts

```
import { SomeType, someFunc } from "./MyThings";
let x = new SomeType();
let y = someFunc();
```

Use the namespace import pattern if you're importing a large number of things

MyLargeModule.ts

```
export class Dog { ... }
export class Cat { ... }
export class Tree { ... }
export class Flower { ... }
```

Consumer.ts

```
import * as myLargeModule from "./MyLargeModule.ts";
let x = new myLargeModule.Dog();
```

Re-export to extend

Often you will need to extend functionality on a module. A common JS pattern is to augment the original object with *extensions*, similar to how JQuery extensions work. As we've mentioned before, modules do not *merge* like global namespace objects would. The recommended solution is to *not* mutate the original object, but rather export a new entity that provides the new functionality.

Consider a simple calculator implementation defined in module `calculator.ts`. The module also exports a helper function to test the calculator functionality by passing a list of input strings and writing the result at the end.

Calculator.ts

```
export class Calculator {
    private current = 0;
    private memory = 0;
```

```

private operator: string;

protected processDigit(digit: string, currentValue: number) {
    if (digit >= "0" && digit <= "9") {
        return currentValue * 10 + (digit.charCodeAt(0) - "0".charCodeAt(0));
    }
}

protected processOperator(operator: string) {
    if ("+", "-", "*", "/".indexOf(operator) >= 0) {
        return operator;
    }
}

protected evaluateOperator(operator: string, left: number, right: number): number {
    switch (this.operator) {
        case "+": return left + right;
        case "-": return left - right;
        case "*": return left * right;
        case "/": return left / right;
    }
}

private evaluate() {
    if (this.operator) {
        this.memory = this.evaluateOperator(this.operator, this.memory, this.current);
    } else {
        this.memory = this.current;
    }
    this.current = 0;
}

public handleChar(char: string) {
    if (char === "=") {
        this.evaluate();
        return;
    } else {
        let value = this.processDigit(char, this.current);
        if (value !== undefined) {
            this.current = value;
            return;
        } else {
            let value = this.processOperator(char);
            if (value !== undefined) {
                this.evaluate();
                this.operator = value;
                return;
            }
        }
    }
    throw new Error(`Unsupported input: '${char}'`);
}

public getResult() {
    return this.memory;
}
}

export function test(c: Calculator, input: string) {
    for (let i = 0; i < input.length; i++) {
        c.handleChar(input[i]);
    }

    console.log(`result of '${input}' is '${c.getResult()}'`);
}

```

Here is a simple test for the calculator using the exposed `test` function.

TestCalculator.ts

```
import { Calculator, test } from "./Calculator";

let c = new Calculator();
test(c, "1+2*33/11="); // prints 9
```

Now to extend this to add support for input with numbers in bases other than 10, let's create `ProgrammerCalculator.ts`

ProgrammerCalculator.ts

```
import { Calculator } from "./Calculator";

class ProgrammerCalculator extends Calculator {
    static digits = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E", "F"];

    constructor(public base: number) {
        super();
        const maxBase = ProgrammerCalculator.digits.length;
        if (base <= 0 || base > maxBase) {
            throw new Error(`base has to be within 0 to ${maxBase} inclusive.`);
        }
    }

    protected processDigit(digit: string, currentValue: number) {
        if (ProgrammerCalculator.digits.indexOf(digit) >= 0) {
            return currentValue * this.base + ProgrammerCalculator.digits.indexOf(digit);
        }
    }
}

// Export the new extended calculator as Calculator
export { ProgrammerCalculator as Calculator };

// Also, export the helper function
export { test } from "./Calculator";
```

The new module `ProgrammerCalculator` exports an API shape similar to that of the original `calculator` module, but does not augment any objects in the original module. Here is a test for our `ProgrammerCalculator` class:

TestProgrammerCalculator.ts

```
import { Calculator, test } from "./ProgrammerCalculator";

let c = new Calculator(2);
test(c, "001+010="); // prints 3
```

Do not use namespaces in modules

When first moving to a module-based organization, a common tendency is to wrap exports in an additional layer of namespaces. Modules have their own scope, and only exported declarations are visible from outside the module. With this in mind, namespaces provide very little, if any, value when working with modules.

On the organization front, namespaces are handy for grouping together logically-related objects and types in the global scope. For example, in C#, you're going to find all the collection types in `System.Collections`. By organizing our types into hierarchical namespaces, we provide a good "discovery" experience for users of those types. Modules, on the other hand, are already present in a file system, necessarily. We have to resolve them by path and filename, so there's a logical organization scheme for us to use. We can have a `/collections/generic/` folder with a list module in it.

Namespaces are important to avoid naming collisions in the global scope. For example, you might have

`My.Application.Customer.AddForm` and `My.Application.Order.AddForm` -- two types with the same name, but a different namespace. This, however, is not an issue with modules. Within a module, there's no plausible reason to have two objects with the same name. From the consumption side, the consumer of any given module gets to pick the name that they will use to refer to the module, so accidental naming conflicts are impossible.

For more discussion about modules and namespaces see [Namespaces and Modules](#).

Red Flags

All of the following are red flags for module structuring. Double-check that you're not trying to namespace your external modules if any of these apply to your files:

- A file whose only top-level declaration is `export namespace Foo { ... }` (remove `Foo` and move everything 'up' a level)
- A file that has a single `export class` or `export function` (consider using `export default`)
- Multiple files that have the same `export namespace Foo {` at top-level (don't think that these are going to combine into one `Foo !`)

A note about terminology: It's important to note that in TypeScript 1.5, the nomenclature has changed. "Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with [ECMAScript 2015](#)'s terminology, (namely that `module x { }` is equivalent to the now-preferred `namespace x { }`).

Introduction

This post outlines the various ways to organize your code using namespaces (previously "internal modules") in TypeScript. As we alluded in our note about terminology, "internal modules" are now referred to as "namespaces". Additionally, anywhere the `module` keyword was used when declaring an internal module, the `namespace` keyword can and should be used instead. This avoids confusing new users by overloading them with similarly named terms.

First steps

Let's start with the program we'll be using as our example throughout this page. We've written a small set of simplistic string validators, as you might write to check a user's input on a form in a webpage or check the format of an externally-provided data file.

Validators in a single file

```
interface StringValidator {
    isAcceptable(s: string): boolean;
}

let lettersRegexp = /^[A-Za-z]+$/;
let numberRegexp = /^[0-9]+$/;

class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}

class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        let isMatch = validators[name].isAcceptable(s);
        console.log(`#${s} ${isMatch ? "matches" : "does not match"}#${name}`);
    }
}
```

Namespacing

As we add more validators, we're going to want to have some kind of organization scheme so that we can keep track of our types and not worry about name collisions with other objects. Instead of putting lots of different names into the global namespace, let's wrap up our objects into a namespace.

In this example, we'll move all validator-related entities into a namespace called `Validation`. Because we want the interfaces and classes here to be visible outside the namespace, we preface them with `export`. Conversely, the variables `lettersRegexp` and `numberRegexp` are implementation details, so they are left unexported and will not be visible to code outside the namespace. In the test code at the bottom of the file, we now need to qualify the names of the types when used outside the namespace, e.g. `Validation.LettersOnlyValidator`.

Namespaced Validators

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }

    const lettersRegexp = /^[A-Za-z]+$/;
    const numberRegexp = /^[0-9]+$/;

    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }

    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log(`"${s}" - ${validators[name].isAcceptable(s) ? "matches" : "does not match"} ${name}`);
    }
}
```

Splitting Across Files

As our application grows, we'll want to split the code across multiple files to make it easier to maintain.

Multi-file namespaces

Here, we'll split our `Validation` namespace across many files. Even though the files are separate, they can each contribute to the same namespace and can be consumed as if they were all defined in one place. Because there are dependencies between files, we'll add reference tags to tell the compiler about the relationships between the files. Our test code is otherwise unchanged.

Validation.ts

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }
}
```

LattersOnlyValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
    const lettersRegexp = /^[A-Za-z]+$/;
    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }
}
```

ZipCodeValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
    const numberRegexp = /^[0-9]+$/;
    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}
```

Test.ts

```
/// <reference path="Validation.ts" />
/// <reference path="LettersOnlyValidator.ts" />
/// <reference path="ZipCodeValidator.ts" />

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log(`"${s}" - ${validators[name].isAcceptable(s) ? "matches" : "does not match"} ${name}`);
    }
}
```

Once there are multiple files involved, we'll need to make sure all of the compiled code gets loaded. There are two ways of doing this.

First, we can use concatenated output using the `--outFile` flag to compile all of the input files into a single JavaScript output file:

```
tsc --outFile sample.js Test.ts
```

The compiler will automatically order the output file based on the reference tags present in the files. You can also specify each file individually:

```
tsc --outFile sample.js Validation.ts LettersOnlyValidator.ts ZipCodeValidator.ts Test.ts
```

Alternatively, we can use per-file compilation (the default) to emit one JavaScript file for each input file. If multiple JS files get produced, we'll need to use `<script>` tags on our webpage to load each emitted file in the appropriate order, for example:

MyTestPage.html (excerpt)

```
<script src="Validation.js" type="text/javascript" />
<script src="LettersOnlyValidator.js" type="text/javascript" />
<script src="ZipCodeValidator.js" type="text/javascript" />
<script src="Test.js" type="text/javascript" />
```

Aliases

Another way that you can simplify working with namespaces is to use `import q = x.y.z` to create shorter names for commonly-used objects. Not to be confused with the `import x = require("name")` syntax used to load modules, this syntax simply creates an alias for the specified symbol. You can use these sorts of imports (commonly referred to as aliases) for any kind of identifier, including objects created from module imports.

```
namespace Shapes {
    export namespace Polygons {
        export class Triangle { }
        export class Square { }
    }
}

import polygons = Shapes.Polygons;
let sq = new polygons.Square(); // Same as 'new Shapes.Polygons.Square()'
```

Notice that we don't use the `require` keyword; instead we assign directly from the qualified name of the symbol we're importing. This is similar to using `var`, but also works on the type and namespace meanings of the imported symbol. Importantly, for values, `import` is a distinct reference from the original symbol, so changes to an aliased `var` will not be reflected in the original variable.

Working with Other JavaScript Libraries

To describe the shape of libraries not written in TypeScript, we need to declare the API that the library exposes. Because most JavaScript libraries expose only a few top-level objects, namespaces are a good way to represent them.

We call declarations that don't define an implementation "ambient". Typically these are defined in `.d.ts` files. If you're familiar with C/C++, you can think of these as `.h` files. Let's look at a few examples.

Ambient Namespaces

The popular library D3 defines its functionality in a global object called `d3`. Because this library is loaded through a `<script>` tag (instead of a module loader), its declaration uses namespaces to define its shape. For the TypeScript compiler to see this shape, we use an ambient namespace declaration. For example, we could begin writing it as follows:

D3.d.ts (simplified excerpt)

```
declare namespace D3 {
    export interface Selectors {
        select: {
            (selector: string): Selection;
            (element: EventTarget): Selection;
        };
    }

    export interface Event {
        x: number;
        y: number;
    }

    export interface Base extends Selectors {
        event: Event;
    }
}

declare var d3: D3.Base;
```

This section assumes some basic knowledge about modules. Please see the [Modules](#) documentation for more information.

Module resolution is the process the compiler uses to figure out what an import refers to. Consider an import statement like `import { a } from "moduleA"`; in order to check any use of `a`, the compiler needs to know exactly what it represents, and will need to check its definition `moduleA`.

At this point, the compiler will ask "what's the shape of `moduleA`?" While this sounds straightforward, `moduleA` could be defined in one of your own `.ts` / `.tsx` files, or in a `.d.ts` that your code depends on.

First, the compiler will try to locate a file that represents the imported module. To do so the compiler follows one of two different strategies: [Classic](#) or [Node](#). These strategies tell the compiler *where* to look for `moduleA`.

If that didn't work and if the module name is non-relative (and in the case of `"moduleA"`, it is), then the compiler will attempt to locate an [ambient module declaration](#). We'll cover non-relative imports next.

Finally, if the compiler could not resolve the module, it will log an error. In this case, the error would be something like `error TS2307: Cannot find module 'moduleA'.`

Relative vs. Non-relative module imports

Module imports are resolved differently based on whether the module reference is relative or non-relative.

A *relative import* is one that starts with `/`, `./` or `../`. Some examples include:

- `import Entry from "./components/Entry";`
- `import { DefaultHeaders } from "../constants/http";`
- `import "/mod";`

Any other import is considered **non-relative**. Some examples include:

- `import * as $ from "jquery";`
- `import { Component } from "@angular/core";`

A relative import is resolved relative to the importing file and *cannot* resolve to an ambient module declaration. You should use relative imports for your own modules that are guaranteed to maintain their relative location at runtime.

A non-relative import can be resolved relative to `baseUrl`, or through path mapping, which we'll cover below. They can also resolve to [ambient module declarations](#). Use non-relative paths when importing any of your external dependencies.

Module Resolution Strategies

There are two possible module resolution strategies: [Node](#) and [Classic](#). You can use the `--moduleResolution` flag to specify the module resolution strategy. If not specified, the default is [Classic](#) for `--module AMD | System | ES2015` or [Node](#) otherwise.

Classic

This used to be TypeScript's default resolution strategy. Nowadays, this strategy is mainly present for backward compatibility.

A relative import will be resolved relative to the importing file. So `import { b } from "./moduleB"` in source file `/root/src/folder/A.ts` would result in the following lookups:

1. `/root/src/folder/moduleB.ts`

2. /root/src/folder/moduleB.d.ts

For non-relative module imports, however, the compiler walks up the directory tree starting with the directory containing the importing file, trying to locate a matching definition file.

For example:

A non-relative import to `moduleB` such as `import { b } from "moduleB"`, in a source file `/root/src/folder/A.ts`, would result in attempting the following locations for locating `"moduleB"`:

1. /root/src/folder/moduleB.ts
2. /root/src/folder/moduleB.d.ts
3. /root/src/moduleB.ts
4. /root/src/moduleB.d.ts
5. /root/moduleB.ts
6. /root/moduleB.d.ts
7. /moduleB.ts
8. /moduleB.d.ts

Node

This resolution strategy attempts to mimic the [Node.js](#) module resolution mechanism at runtime. The full Node.js resolution algorithm is outlined in [Node.js module documentation](#).

How Node.js resolves modules

To understand what steps the TS compiler will follow, it is important to shed some light on Node.js modules.

Traditionally, imports in Node.js are performed by calling a function named `require`. The behavior Node.js takes will differ depending on if `require` is given a relative path or a non-relative path.

Relative paths are fairly straightforward. As an example, let's consider a file located at `/root/src/moduleA.js`, which contains the import `var x = require("./moduleB");`. Node.js resolves that import in the following order:

1. Ask the file named `/root/src/moduleB.js`, if it exists.
2. Ask the folder `/root/src/moduleB` if it contains a file named `package.json` that specifies a `"main"` module. In our example, if Node.js found the file `/root/src/moduleB/package.json` containing `{ "main": "lib/mainModule.js" }`, then Node.js will refer to `/root/src/moduleB/lib/mainModule.js`.
3. Ask the folder `/root/src/moduleB` if it contains a file named `index.js`. That file is implicitly considered that folder's `"main"` module.

You can read more about this in Node.js documentation on [file modules](#) and [folder modules](#).

However, resolution for a [non-relative module name](#) is performed differently. Node will look for your modules in special folders named `node_modules`. A `node_modules` folder can be on the same level as the current file, or higher up in the directory chain. Node will walk up the directory chain, looking through each `node_modules` until it finds the module you tried to load.

Following up our example above, consider if `/root/src/moduleA.js` instead used a non-relative path and had the import `var x = require("moduleB");`. Node would then try to resolve `moduleB` to each of the locations until one worked.

1. /root/src/node_modules/moduleB.js
2. /root/src/node_modules/moduleB/package.json (if it specifies a `"main"` property)
3. /root/src/node_modules/moduleB/index.js

4. /root/node_modules/moduleB.js
5. /root/node_modules/moduleB/package.json (if it specifies a "main" property)
6. /root/node_modules/moduleB/index.js

7. /node_modules/moduleB.js
8. /node_modules/moduleB/package.json (if it specifies a "main" property)
9. /node_modules/moduleB/index.js

Notice that Node.js jumped up a directory in steps (4) and (7).

You can read more about the process in Node.js documentation on [loading modules from node_modules](#).

How TypeScript resolves modules

TypeScript will mimic the Node.js run-time resolution strategy in order to locate definition files for modules at compile-time. To accomplish this, TypeScript overlays the TypeScript source file extensions (`.ts` , `.tsx` , and `.d.ts`) over the Node's resolution logic. TypeScript will also use a field in `package.json` named `"types"` to mirror the purpose of `"main"` - the compiler will use it to find the "main" definition file to consult.

For example, an import statement like `import { b } from "./moduleB"` in `/root/src/moduleA.ts` would result in attempting the following locations for locating `"./moduleB"`:

1. /root/src/moduleB.ts
2. /root/src/moduleB.tsx
3. /root/src/moduleB.d.ts
4. /root/src/moduleB/package.json (if it specifies a "types" property)
5. /root/src/moduleB/index.ts
6. /root/src/moduleB/index.tsx
7. /root/src/moduleB/index.d.ts

Recall that Node.js looked for a file named `moduleB.js` , then an applicable `package.json` , and then for an `index.js` .

Similarly a non-relative import will follow the Node.js resolution logic, first looking up a file, then looking up an applicable folder. So `import { b } from "moduleB"` in source file `/root/src/moduleA.ts` would result in the following lookups:

1. /root/src/node_modules/moduleB.ts
2. /root/src/node_modules/moduleB.tsx
3. /root/src/node_modules/moduleB.d.ts
4. /root/src/node_modules/moduleB/package.json (if it specifies a "types" property)
5. /root/src/node_modules/@types/moduleB.d.ts
6. /root/src/node_modules/moduleB/index.ts
7. /root/src/node_modules/moduleB/index.tsx
8. /root/src/node_modules/moduleB/index.d.ts

9. /root/node_modules/moduleB.ts
10. /root/node_modules/moduleB.tsx
11. /root/node_modules/moduleB.d.ts
12. /root/node_modules/moduleB/package.json (if it specifies a "types" property)
13. /root/node_modules/@types/moduleB.d.ts
14. /root/node_modules/moduleB/index.ts
15. /root/node_modules/moduleB/index.tsx
16. /root/node_modules/moduleB/index.d.ts

17. `/node_modules/moduleB.ts`
18. `/node_modules/moduleB.tsx`
19. `/node_modules/moduleB.d.ts`
20. `/node_modules/moduleB/package.json` (if it specifies a `"types"` property)
21. `/node_modules/@types/moduleB.d.ts`
22. `/node_modules/moduleB/index.ts`
23. `/node_modules/moduleB/index.tsx`
24. `/node_modules/moduleB/index.d.ts`

Don't be intimidated by the number of steps here - TypeScript is still only jumping up directories twice at steps (9) and (17). This is really no more complex than what Node.js itself is doing.

Additional module resolution flags

A project source layout sometimes does not match that of the output. Usually a set of build steps result in generating the final output. These include compiling `.ts` files into `.js`, and copying dependencies from different source locations to a single output location. The net result is that modules at runtime may have different names than the source files containing their definitions. Or module paths in the final output may not match their corresponding source file paths at compile time.

The TypeScript compiler has a set of additional flags to *inform* the compiler of transformations that are expected to happen to the sources to generate the final output.

It is important to note that the compiler will *not* perform any of these transformations; it just uses these pieces of information to guide the process of resolving a module import to its definition file.

Base URL

Using a `baseUrl` is a common practice in applications using AMD module loaders where modules are "deployed" to a single folder at run-time. The sources of these modules can live in different directories, but a build script will put them all together.

Setting `baseUrl` informs the compiler where to find modules. All module imports with non-relative names are assumed to be relative to the `baseUrl`.

Value of `baseUrl` is determined as either:

- value of `baseUrl` command line argument (if given path is relative, it is computed based on current directory)
- value of `baseUrl` property in `'tsconfig.json'` (if given path is relative, it is computed based on the location of `'tsconfig.json'`)

Note that relative module imports are not impacted by setting the `baseUrl`, as they are always resolved relative to their importing files.

You can find more documentation on `baseUrl` in [RequireJS](#) and [SystemJS](#) documentation.

Path mapping

Sometimes modules are not directly located under `baseUrl`. For instance, an import to a module `"jquery"` would be translated at runtime to `"node_modules/jquery/dist/jquery.slim.min.js"`. Loaders use a mapping configuration to map module names to files at run-time, see [RequireJs documentation](#) and [SystemJS documentation](#).

The TypeScript compiler supports the declaration of such mappings using `"paths"` property in `tsconfig.json` files. Here is an example for how to specify the `"paths"` property for `jquery`.

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "jquery": ["node_modules/jquery/dist/jquery"]
    }
  }
}
```

Please notice that `"paths"` are resolved relative to `"baseUrl"`. When setting `"baseUrl"` to another value than `".."`, i.e. the directory of `tsconfig.json`, the mappings must be changed accordingly. Say, you set `"baseUrl": "./src"` in the above example, then `jquery` should be mapped to `"/node_modules/jquery/dist/jquery"`.

Using `"paths"` also allows for more sophisticated mappings including multiple fall back locations. Consider a project configuration where only some modules are available in one location, and the rest are in another. A build step would put them all together in one place. The project layout may look like:

```
projectRoot
└── folder1
  ├── file1.ts (imports 'folder1/file2' and 'folder2/file3')
  └── file2.ts
└── generated
  ├── folder1
  └── folder2
    └── file3.ts
└── tsconfig.json
```

The corresponding `tsconfig.json` would look like:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "*": [
        "*",
        "generated/*"
      ]
    }
  }
}
```

This tells the compiler for any module import that matches the pattern `"*"` (i.e. all values), to look in two locations:

1. `"*"` : meaning the same name unchanged, so map `<moduleName> => <baseUrl>/<moduleName>`
2. `"generated/*"` meaning the module name with an appended prefix "generated", so map `<moduleName> => <baseUrl>/generated/<moduleName>`

Following this logic, the compiler will attempt to resolve the two imports as such:

- import 'folder1/file2'
 1. pattern `"*"` is matched and wildcard captures the whole module name
 2. try first substitution in the list: `"*"` -> `folder1/file2`
 3. result of substitution is non-relative name - combine it with `baseUrl` -> `projectRoot/folder1/file2.ts`.
 4. File exists. Done.
- import 'folder2/file3'
 1. pattern `"*"` is matched and wildcard captures the whole module name
 2. try first substitution in the list: `"*"` -> `folder2/file3`
 3. result of substitution is non-relative name - combine it with `baseUrl` -> `projectRoot/folder2/file3.ts`.

4. File does not exist, move to the second substitution
5. second substitution 'generated/*' -> generated/folder2/file3
6. result of substitution is non-relative name - combine it with `baseUrl` ->
`projectRoot/generated/folder2/file3.ts` .
7. File exists. Done.

Virtual Directories with `rootDirs`

Sometimes the project sources from multiple directories at compile time are all combined to generate a single output directory. This can be viewed as a set of source directories create a "virtual" directory.

Using '`rootDirs`', you can inform the compiler of the *roots* making up this "virtual" directory; and thus the compiler can resolve relative modules imports within these "virtual" directories as *if* were merged together in one directory.

For example consider this project structure:

```

src
└── views
    └── view1.ts (imports './template1')
    └── view2.ts

generated
└── templates
    └── views
        └── template1.ts (imports './view2')

```

Files in `src/views` are user code for some UI controls. Files in `generated/templates` are UI template binding code auto-generated by a template generator as part of the build. A build step will copy the files in `/src/views` and `/generated/templates/views` to the same directory in the output. At run-time, a view can expect its template to exist next to it, and thus should import it using a relative name as `"./template"`.

To specify this relationship to the compiler, use "`rootDirs`". "`rootDirs`" specify a list of *roots* whose contents are expected to merge at run-time. So following our example, the `tsconfig.json` file should look like:

```
{
  "compilerOptions": {
    "rootDirs": [
      "src/views",
      "generated/templates/views"
    ]
  }
}
```

Every time the compiler sees a relative module import in a subfolder of one of the `rootDirs`, it will attempt to look for this import in each of the entries of `rootDirs`.

The flexibility of `rootDirs` is not limited to specifying a list of physical source directories that are logically merged. The supplied array may include any number of ad hoc, arbitrary directory names, regardless of whether they exist or not. This allows the compiler to capture sophisticated bundling and runtime features such as conditional inclusion and project specific loader plugins in a type safe way.

Consider an internationalization scenario where a build tool automatically generates locale specific bundles by interpolating a special path token, say `#{{locale}}`, as part of a relative module path such as `./#{{locale}}/messages`. In this hypothetical setup the tool enumerates supported locales, mapping the abstracted path into `./zh/messages`, `./de/messages`, and so forth.

Assume that each of these modules exports an array of strings. For example `./zh/messages` might contain:

```
export default [
  "您好吗",
  "很高兴认识你"
];
```

By leveraging `rootDirs` we can inform the compiler of this mapping and thereby allow it to safely resolve `./#{locale}/messages`, even though the directory will never exist. For example, with the following `tsconfig.json`:

```
{
  "compilerOptions": {
    "rootDirs": [
      "src/zh",
      "src/de",
      "src/#{locale}"
    ]
  }
}
```

The compiler will now resolve `import messages from './#{locale}/messages'` to `import messages from './zh/messages'` for tooling purposes, allowing development in a locale agnostic manner without compromising design time support.

Tracing module resolution

As discussed earlier, the compiler can visit files outside the current folder when resolving a module. This can be hard when diagnosing why a module is not resolved, or is resolved to an incorrect definition. Enabling the compiler module resolution tracing using `--traceResolution` provides insight in what happened during the module resolution process.

Let's say we have a sample application that uses the `typescript` module. `app.ts` has an import like `import * as ts from "typescript"`.

```
|
|   tsconfig.json
|   └── node_modules
|       └── typescript
|           └── lib
|               └── typescript.d.ts
|
└── src
    └── app.ts
```

Invoking the compiler with `--traceResolution`

```
tsc --traceResolution
```

Results in an output such as:

```
===== Resolving module 'typescript' from 'src/app.ts'. =====
Module resolution kind is not specified, using 'NodeJs'.
Loading module 'typescript' from 'node_modules' folder.
File 'src/node_modules/typescript.ts' does not exist.
File 'src/node_modules/typescript.tsx' does not exist.
File 'src/node_modules/typescript.d.ts' does not exist.
File 'src/node_modules/typescript/package.json' does not exist.
File 'node_modules/typescript.ts' does not exist.
File 'node_modules/typescript.tsx' does not exist.
File 'node_modules/typescript.d.ts' does not exist.
Found 'package.json' at 'node_modules/typescript/package.json'.
'package.json' has 'types' field './lib/typescript.d.ts' that references 'node_modules/typescript/lib/typescript.d.ts'.
File 'node_modules/typescript/lib/typescript.d.ts' exist - use it as a module resolution result.
```

```
===== Module name 'typescript' was successfully resolved to 'node_modules/typescript/lib/typescript.d.ts'. =
=====
```

Things to look out for

- Name and location of the import


```
===== Resolving module 'typescript' from 'src/app.ts'. =====
```
- The strategy the compiler is following


```
Module resolution kind is not specified, using 'NodeJs'.
```
- Loading of types from npm packages


```
'package.json' has 'types' field './lib/typescript.d.ts' that references
'node_modules/typescript/lib/typescript.d.ts'.
```
- Final result


```
===== Module name 'typescript' was successfully resolved to
'node_modules/typescript/lib/typescript.d.ts'. =====
```

Using `--noResolve`

Normally the compiler will attempt to resolve all module imports before it starts the compilation process. Every time it successfully resolves an `import` to a file, the file is added to the set of files the compiler will process later on.

The `--noResolve` compiler option instructs the compiler not to "add" any files to the compilation that were not passed on the command line. It will still try to resolve the module to files, but if the file is not specified, it will not be included.

For instance:

`app.ts`

```
import * as A from "moduleA" // OK, 'moduleA' passed on the command-line
import * as B from "moduleB" // Error TS2307: Cannot find module 'moduleB'.
```

```
tsc app.ts moduleA.ts --noResolve
```

Compiling `app.ts` using `--noResolve` should result in:

- Correctly finding `moduleA` as it was passed on the command-line.
- Error for not finding `moduleB` as it was not passed.

Common Questions

Why does a module in the exclude list still get picked up by the compiler?

`tsconfig.json` turns a folder into a "project". Without specifying any `"exclude"` or `"files"` entries, all files in the folder containing the `tsconfig.json` and all its sub-directories are included in your compilation. If you want to exclude some of the files use `"exclude"`, if you would rather specify all the files instead of letting the compiler look them up, use `"files"`.

That was `tsconfig.json` automatic inclusion. That does not embed module resolution as discussed above. If the compiler identified a file as a target of a module import, it will be included in the compilation regardless if it was excluded in the previous steps.

So to exclude a file from the compilation, you need to exclude it and **all** files that have an `import` or `/// <reference path="..." />` directive to it.

Introduction

Some of the unique concepts in TypeScript describe the shape of JavaScript objects at the type level. One example that is especially unique to TypeScript is the concept of 'declaration merging'. Understanding this concept will give you an advantage when working with existing JavaScript. It also opens the door to more advanced abstraction concepts.

For the purposes of this article, "declaration merging" means that the compiler merges two separate declarations declared with the same name into a single definition. This merged definition has the features of both of the original declarations. Any number of declarations can be merged; it's not limited to just two declarations.

Basic Concepts

In TypeScript, a declaration creates entities in at least one of three groups: namespace, type, or value. Namespace-creating declarations create a namespace, which contains names that are accessed using a dotted notation. Type-creating declarations do just that: they create a type that is visible with the declared shape and bound to the given name. Lastly, value-creating declarations create values that are visible in the output JavaScript.

Declaration Type	Namespace	Type	Value
Namespace	X		X
Class		X	X
Enum		X	X
Interface		X	
Type Alias		X	
Function			X
Variable			X

Understanding what is created with each declaration will help you understand what is merged when you perform a declaration merge.

Merging Interfaces

The simplest, and perhaps most common, type of declaration merging is interface merging. At the most basic level, the merge mechanically joins the members of both declarations into a single interface with the same name.

```
interface Box {
    height: number;
    width: number;
}

interface Box {
    scale: number;
}

let box: Box = {height: 5, width: 6, scale: 10};
```

Non-function members of the interfaces should be unique. If they are not unique, they must be of the same type. The compiler will issue an error if the interfaces both declare a non-function member of the same name, but of different types.

For function members, each function member of the same name is treated as describing an overload of the same function. Of note, too, is that in the case of interface `A` merging with later interface `A`, the second interface will have a higher precedence than the first.

That is, in the example:

```
interface Cloner {
    clone(animal: Animal): Animal;
}

interface Cloner {
    clone(animal: Sheep): Sheep;
}

interface Cloner {
    clone(animal: Dog): Dog;
    clone(animal: Cat): Cat;
}
```

The three interfaces will merge to create a single declaration as so:

```
interface Cloner {
    clone(animal: Dog): Dog;
    clone(animal: Cat): Cat;
    clone(animal: Sheep): Sheep;
    clone(animal: Animal): Animal;
}
```

Notice that the elements of each group maintains the same order, but the groups themselves are merged with later overload sets ordered first.

One exception to this rule is specialized signatures. If a signature has a parameter whose type is a *single* string literal type (e.g. not a union of string literals), then it will be bubbled toward the top of its merged overload list.

For instance, the following interfaces will merge together:

```
interface Document {
    createElement(tagName: any): Element;
}
interface Document {
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
}
interface Document {
    createElement(tagName: string): HTMLElement;
    createElement(tagName: "canvas"): HTMLCanvasElement;
}
```

The resulting merged declaration of `Document` will be the following:

```
interface Document {
    createElement(tagName: "canvas"): HTMLCanvasElement;
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
    createElement(tagName: string): HTMLElement;
    createElement(tagName: any): Element;
}
```

Merging Namespaces

Similarly to interfaces, namespaces of the same name will also merge their members. Since namespaces create both a namespace and a value, we need to understand how both merge.

To merge the namespaces, type definitions from exported interfaces declared in each namespace are themselves merged, forming a single namespace with merged interface definitions inside.

To merge the namespace value, at each declaration site, if a namespace already exists with the given name, it is further extended by taking the existing namespace and adding the exported members of the second namespace to the first.

The declaration merge of `Animals` in this example:

```
namespace Animals {
    export class Zebra { }
}

namespace Animals {
    export interface Legged { numberOfLegs: number; }
    export class Dog { }
}
```

is equivalent to:

```
namespace Animals {
    export interface Legged { numberOfLegs: number; }

    export class Zebra { }
    export class Dog { }
}
```

This model of namespace merging is a helpful starting place, but we also need to understand what happens with non-exported members. Non-exported members are only visible in the original (un-merged) namespace. This means that after merging, merged members that came from other declarations cannot see non-exported members.

We can see this more clearly in this example:

```
namespace Animal {
    let haveMuscles = true;

    export function animalsHaveMuscles() {
        return haveMuscles;
    }
}

namespace Animal {
    export function doAnimalsHaveMuscles() {
        return haveMuscles; // Error, because haveMuscles is not accessible here
    }
}
```

Because `haveMuscles` is not exported, only the `animalsHaveMuscles` function that shares the same un-merged namespace can see the symbol. The `doAnimalsHaveMuscles` function, even though it's part of the merged `Animal` namespace can not see this un-exported member.

Merging Namespaces with Classes, Functions, and Enums

Namespaces are flexible enough to also merge with other types of declarations. To do so, the namespace declaration must follow the declaration it will merge with. The resulting declaration has properties of both declaration types. TypeScript uses this capability to model some of the patterns in JavaScript as well as other programming languages.

Merging Namespaces with Classes

This gives the user a way of describing inner classes.

```
class Album {
    label: Album.AlbumLabel;
}
namespace Album {
    export class AlbumLabel { }
}
```

The visibility rules for merged members is the same as described in the 'Merging Namespaces' section, so we must export the `AlbumLabel` class for the merged class to see it. The end result is a class managed inside of another class. You can also use namespaces to add more static members to an existing class.

In addition to the pattern of inner classes, you may also be familiar with JavaScript practice of creating a function and then extending the function further by adding properties onto the function. TypeScript uses declaration merging to build up definitions like this in a type-safe way.

```
function buildLabel(name: string): string {
    return buildLabel.prefix + name + buildLabel.suffix;
}

namespace buildLabel {
    export let suffix = "";
    export let prefix = "Hello, ";
}

console.log(buildLabel("Sam Smith"));
```

Similarly, namespaces can be used to extend enums with static members:

```
enum Color {
    red = 1,
    green = 2,
    blue = 4
}

namespace Color {
    export function mixColor(colorName: string) {
        if (colorName == "yellow") {
            return Color.red + Color.green;
        }
        else if (colorName == "white") {
            return Color.red + Color.green + Color.blue;
        }
        else if (colorName == "magenta") {
            return Color.red + Color.blue;
        }
        else if (colorName == "cyan") {
            return Color.green + Color.blue;
        }
    }
}
```

```

        }
    }
}
```

Disallowed Merges

Not all merges are allowed in TypeScript. Currently, classes can not merge with other classes or with variables. For information on mimicking class merging, see the [Mixins in TypeScript](#) section.

Module Augmentation

Although JavaScript modules do not support merging, you can patch existing objects by importing and then updating them. Let's look at a toy Observable example:

```

// observable.js
export class Observable<T> {
    // ... implementation left as an exercise for the reader ...
}

// map.js
import { Observable } from "./observable";
Observable.prototype.map = function (f) {
    // ... another exercise for the reader
}
```

This works fine in TypeScript too, but the compiler doesn't know about `observable.prototype.map`. You can use module augmentation to tell the compiler about it:

```

// observable.ts stays the same
// map.ts
import { Observable } from "./observable";
declare module "./observable" {
    interface Observable<T> {
        map<U>(f: (x: T) => U): Observable<U>;
    }
}
Observable.prototype.map = function (f) {
    // ... another exercise for the reader
}

// consumer.ts
import { Observable } from "./observable";
import "./map";
let o: Observable<number>;
o.map(x => x.toFixed());
```

The module name is resolved the same way as module specifiers in `import / export`. See [Modules](#) for more information. Then the declarations in an augmentation are merged as if they were declared in the same file as the original. However, you can't declare new top-level declarations in the augmentation -- just patches to existing declarations.

Global augmentation

You can also add declarations to the global scope from inside a module:

```
// observable.ts
export class Observable<T> {
    // ... still no implementation ...
}

declare global {
    interface Array<T> {
        toObservable(): Observable<T>;
    }
}

Array.prototype.toObservable = function () {
    // ...
}
```

Global augmentations have the same behavior and limits as module augmentations.

Introduction

JSX is an embeddable XML-like syntax. It is meant to be transformed into valid JavaScript, though the semantics of that transformation are implementation-specific. JSX rose to popularity with the [React](#) framework, but has since seen other implementations as well. TypeScript supports embedding, type checking, and compiling JSX directly to JavaScript.

Basic usage

In order to use JSX you must do two things.

1. Name your files with a `.tsx` extension
2. Enable the `jsx` option

TypeScript ships with three JSX modes: `preserve`, `react`, and `react-native`. These modes only affect the emit stage - type checking is unaffected. The `preserve` mode will keep the JSX as part of the output to be further consumed by another transform step (e.g. [Babel](#)). Additionally the output will have a `.jsx` file extension. The `react` mode will emit `React.createElement`, does not need to go through a JSX transformation before use, and the output will have a `.js` file extension. The `react-native` mode is the equivalent of `preserve` in that it keeps all JSX, but the output will instead have a `.js` file extension.

Mode	Input	Output	Output File Extension
<code>preserve</code>	<code><div /></code>	<code><div /></code>	<code>.jsx</code>
<code>react</code>	<code><div /></code>	<code>React.createElement("div")</code>	<code>.js</code>
<code>react-native</code>	<code><div /></code>	<code><div /></code>	<code>.js</code>

You can specify this mode using either the `--jsx` command line flag or the corresponding option in your [tsconfig.json](#) file.

Note: The identifier `React` is hard-coded, so you must make React available with an uppercase R.

The `as` operator

Recall how to write a type assertion:

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

This asserts the variable `bar` to have the type `foo`. Since TypeScript also uses angle brackets for type assertions, combining it with JSX's syntax would introduce certain parsing difficulties. As a result, TypeScript disallows angle bracket type assertions in `.tsx` files.

Since the above syntax cannot be used in `.tsx` files, an alternate type assertion operator should be used: `as`. The example can easily be rewritten with the `as` operator.

```
var foo = bar as foo;
```

The `as` operator is available in both `.ts` and `.tsx` files, and is identical in behavior to the angle-bracket type assertion style.

Type Checking

In order to understand type checking with JSX, you must first understand the difference between intrinsic elements and value-based elements. Given a JSX expression `<expr />`, `expr` may either refer to something intrinsic to the environment (e.g. a `div` or `span` in a DOM environment) or to a custom component that you've created. This is important for two reasons:

1. For React, intrinsic elements are emitted as strings (`React.createElement("div")`), whereas a component you've created is not (`React.createElement(MyComponent)`).
2. The types of the attributes being passed in the JSX element should be looked up differently. Intrinsic element attributes should be known *intrinsically* whereas components will likely want to specify their own set of attributes.

TypeScript uses the [same convention that React does](#) for distinguishing between these. An intrinsic element always begins with a lowercase letter, and a value-based element always begins with an uppercase letter.

Intrinsic elements

Intrinsic elements are looked up on the special interface `JSX.IntrinsicElements`. By default, if this interface is not specified, then anything goes and intrinsic elements will not be type checked. However, if this interface *is* present, then the name of the intrinsic element is looked up as a property on the `JSX.IntrinsicElements` interface. For example:

```
declare namespace JSX {
    interface IntrinsicElements {
        foo: any
    }
}

<foo />; // ok
<bar />; // error
```

In the above example, `<foo />` will work fine but `<bar />` will result in an error since it has not been specified on `JSX.IntrinsicElements`.

Note: You can also specify a catch-all string indexer on `JSX.IntrinsicElements` as follows:

```
declare namespace JSX {
    interface IntrinsicElements {
        [elemName: string]: any;
    }
}
```

Value-based elements

Value based elements are simply looked up by identifiers that are in scope.

```
import MyComponent from "./myComponent";

<MyComponent />; // ok
<SomeOtherComponent />; // error
```

There are two ways to define a value-based element:

1. Stateless Functional Component (SFC)

2. Class Component

Because these two types of value-based elements are indistinguishable from each other in a JSX expression, first TS tries to resolve the expression as Stateless Functional Component using overload resolution. If the process succeeds, then TS finishes resolving the expression to its declaration. If the value fails to resolve as SFC, TS will then try to resolve it as a class component. If that fails, TS will report an error.

Stateless Functional Component

As the name suggests, the component is defined as JavaScript function where its first argument is a `props` object. TS enforces that its return type must be assignable to `JSX.Element`.

```
interface FooProp {
  name: string;
  X: number;
  Y: number;
}

declare function AnotherComponent(prop: {name: string});
function ComponentFoo(prop: FooProp) {
  return <AnotherComponent name={prop.name} />;
}

const Button = (prop: {value: string}, context: {color: string}) => <button>
```

Because an SFC is simply a JavaScript function, function overloads may be used here as well:

```
interface ClickableProps {
  children: JSX.Element[] | JSX.Element
}

interface HomeProps extends ClickableProps {
  home: JSX.Element;
}

interface SideProps extends ClickableProps {
  side: JSX.Element | string;
}

function MainButton(prop: HomeProps): JSX.Element;
function MainButton(prop: SideProps): JSX.Element {
  ...
}
```

Class Component

It is possible to define the type of a class component. However, to do so it is best to understand two new terms: the *element class type* and the *element instance type*.

Given `<Expr />`, the *element class type* is the type of `Expr`. So in the example above, if `MyComponent` was an ES6 class the class type would be that class's constructor and statics. If `MyComponent` was a factory function, the class type would be that function.

Once the class type is established, the instance type is determined by the union of the return types of the class type's construct or call signatures (whichever is present). So again, in the case of an ES6 class, the instance type would be the type of an instance of that class, and in the case of a factory function, it would be the type of the value returned from the function.

```
class MyComponent {
```

```

    render() {}
}

// use a construct signature
var myComponent = new MyComponent();

// element class type => MyComponent
// element instance type => { render: () => void }

function MyFactoryFunction() {
  return {
    render: () => {
    }
  }
}

// use a call signature
var myComponent = MyFactoryFunction();

// element class type => FactoryFunction
// element instance type => { render: () => void }

```

The element instance type is interesting because it must be assignable to `JSX.ElementClass` or it will result in an error. By default `JSX.ElementClass` is `{}`, but it can be augmented to limit the use of JSX to only those types that conform to the proper interface.

```

declare namespace JSX {
  interface ElementClass {
    render: any;
  }
}

class MyComponent {
  render() {}
}

function MyFactoryFunction() {
  return { render: () => {} }
}

<MyComponent />; // ok
<MyFactoryFunction />; // ok

class NotAValidComponent {}
function NotAValidFactoryFunction() {
  return {};
}

<NotAValidComponent />; // error
<NotAValidFactoryFunction />; // error

```

Attribute type checking

The first step to type checking attributes is to determine the *element attributes type*. This is slightly different between intrinsic and value-based elements.

For intrinsic elements, it is the type of the property on `JSX.IntrinsicElements`

```

declare namespace JSX {
  interface IntrinsicElements {
    foo: { bar?: boolean }
  }
}

```

```
// element attributes type for 'foo' is '{bar?: boolean}'
<foo bar />
```

For value-based elements, it is a bit more complex. It is determined by the type of a property on the `element instance type` that was previously determined. Which property to use is determined by `JSX.ElementAttributesProperty`. It should be declared with a single property. The name of that property is then used. As of TypeScript 2.8, if `JSX.ElementAttributesProperty` is not provided, the type of first parameter of the class element's constructor or SFC's call will be used instead.

```
declare namespace JSX {
    interface ElementAttributesProperty {
        props; // specify the property name to use
    }
}

class MyComponent {
    // specify the property on the element instance type
    props: {
        foo?: string;
    }
}

// element attributes type for 'MyComponent' is '{foo?: string}'
<MyComponent foo="bar" />
```

The element attribute type is used to type check the attributes in the JSX. Optional and required properties are supported.

```
declare namespace JSX {
    interface IntrinsicElements {
        foo: { requiredProp: string; optionalProp?: number }
    }
}

<foo requiredProp="bar" />; // ok
<foo requiredProp="bar" optionalProp={0} />; // ok
<foo />; // error, requiredProp is missing
<foo requiredProp={0} />; // error, requiredProp should be a string
<foo requiredProp="bar" unknownProp />; // error, unknownProp does not exist
<foo requiredProp="bar" some-unknown-prop />; // ok, because 'some-unknown-prop' is not a valid identifier
```

Note: If an attribute name is not a valid JS identifier (like a `data-*` attribute), it is not considered to be an error if it is not found in the element attributes type.

Additionally, the `JSX.IntrinsicAttributes` interface can be used to specify extra properties used by the JSX framework which are not generally used by the components' props or arguments - for instance `key` in React. Specializing further, the generic `JSX.IntrinsicClassAttributes<T>` type may also be used to specify the same kind of extra attributes just for class components (and not SFCs). In this type, the generic parameter corresponds to the class instance type. In React, this is used to allow the `ref` attribute of type `Ref<T>`. Generally speaking, all of the properties on these interfaces should be optional, unless you intend that users of your JSX framework need to provide some attribute on every tag.

The spread operator also works:

```
var props = { requiredProp: "bar" };
<foo {...props} />; // ok

var badProps = {};
<foo {...badProps} />; // error
```

Children Type Checking

In TypeScript 2.3, TS introduced type checking of `children`. `children` is a special property in an `element attributes type` where child `JSXExpressions` are taken to be inserted into the attributes. Similar to how TS uses `JSX.ElementAttributesProperty` to determine the name of `props`, TS uses `JSX.ElementChildrenAttribute` to determine the name of `children` within those props. `JSX.ElementChildrenAttribute` should be declared with a single property.

```
declare namespace JSX {
    interface ElementChildrenAttribute {
        children: {}; // specify children name to use
    }
}
```

```
<div>
  <h1>Hello</h1>
</div>

<div>
  <h1>Hello</h1>
  World
</div>

const CustomComp = (props) => <div>props.children</div>
<CustomComp>
  <div>Hello World</div>
  {"This is just a JS expression..." + 1000}
</CustomComp>
```

You can specify the type of `children` like any other attribute. This will override the default type from, eg the [React typings](#) if you use them.

```
interface PropsType {
    children: JSX.Element
    name: string
}

class Component extends React.Component<PropsType, {}> {
    render() {
        return (
            <h2>
                {this.props.children}
            </h2>
        )
    }
}

// OK
<Component>
  <h1>Hello World</h1>
</Component>

// Error: children is of type JSX.Element not array of JSX.Element
<Component>
  <h1>Hello World</h1>
  <h2>Hello World</h2>
</Component>

// Error: children is of type JSX.Element not array of JSX.Element or string.
<Component>
  <h1>Hello</h1>
  World
</Component>
```

The JSX result type

By default the result of a JSX expression is typed as `any`. You can customize the type by specifying the `JSX.Element` interface. However, it is not possible to retrieve type information about the element, attributes or children of the JSX from this interface. It is a black box.

Embedding Expressions

JSX allows you to embed expressions between tags by surrounding the expressions with curly braces (`{ }`).

```
var a = <div>
  [{"foo", "bar"].map(i => <span>{i / 2}</span>)
</div>
```

The above code will result in an error since you cannot divide a string by a number. The output, when using the `preserve` option, looks like:

```
var a = <div>
  {"foo", "bar"].map(function (i) { return <span>{i / 2}</span>; })
</div>
```

React integration

To use JSX with React you should use the [React typings](#). These typings define the `jsx` namespace appropriately for use with React.

```
/// <reference path="react.d.ts" />

interface Props {
  foo: string;
}

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

<MyComponent foo="bar" />; // ok
<MyComponent foo={0} />; // error
```

Factory Functions

The exact factory function used by the `jsx: react` compiler option is configurable. It may be set using either the `jsxFactory` command line option, or an inline `@jsx` comment pragma to set it on a per-file basis. For example, if you set `jsxFactory` to `createElement`, `<div />` will emit as `createElement("div")` instead of `React.createElement("div")`.

The comment pragma version may be used like so (in TypeScript 2.8):

```
import preact = require("preact");
/* @jsx preact.h */
```

```
const x = <div />;
```

emits as:

```
const preact = require("preact");
const x = preact.h("div", null);
```

The factory chosen will also affect where the `jsx` namespace is looked up (for type checking information) before falling back to the global one. If the factory is defined as `React.createElement` (the default), the compiler will check for `React.JSX` before checking for a global `jsx`. If the factory is defined as `h`, it will check for `h.jsx` before a global `JSX`.

Introduction

With the introduction of Classes in TypeScript and ES6, there now exist certain scenarios that require additional features to support annotating or modifying classes and class members. Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members. Decorators are a [stage 2 proposal](#) for JavaScript and are available as an experimental feature of TypeScript.

NOTE Decorators are an experimental feature that may change in future releases.

To enable experimental support for decorators, you must enable the `experimentalDecorators` compiler option either on the command line or in your `tsconfig.json`:

Command Line:

```
tsc --target ES5 --experimentalDecorators
```

tsconfig.json:

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true
  }
}
```

Decorators

A *Decorator* is a special kind of declaration that can be attached to a [class declaration](#), [method](#), [accessor](#), [property](#), or [parameter](#). Decorators use the form `@expression`, where `expression` must evaluate to a function that will be called at runtime with information about the decorated declaration.

For example, given the decorator `@sealed` we might write the `sealed` function as follows:

```
function sealed(target) {
  // do something with 'target' ...
}
```

NOTE You can see a more detailed example of a decorator in [Class Decorators](#), below.

Decorator Factories

If we want to customize how a decorator is applied to a declaration, we can write a decorator factory. A *Decorator Factory* is simply a function that returns the expression that will be called by the decorator at runtime.

We can write a decorator factory in the following fashion:

```
function color(value: string) { // this is the decorator factory
  return function (target) { // this is the decorator
    // do something with 'target' and 'value'...
  }
}
```

NOTE You can see a more detailed example of a decorator factory in [Method Decorators](#), below.

Decorator Composition

Multiple decorators can be applied to a declaration, as in the following examples:

- On a single line:

```
@f @g x
```

- On multiple lines:

```
@f  
@g  
x
```

When multiple decorators apply to a single declaration, their evaluation is similar to [function composition in mathematics](#). In this model, when composing functions f and g , the resulting composite $(f \circ g)(x)$ is equivalent to $f(g(x))$.

As such, the following steps are performed when evaluating multiple decorators on a single declaration in TypeScript:

1. The expressions for each decorator are evaluated top-to-bottom.
2. The results are then called as functions from bottom-to-top.

If we were to use [decorator factories](#), we can observe this evaluation order with the following example:

```
function f() {
    console.log("f(): evaluated");
    return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("f(): called");
    }
}

function g() {
    console.log("g(): evaluated");
    return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("g(): called");
    }
}

class C {
    @f()
    @g()
    method() {}
}
```

Which would print this output to the console:

```
f(): evaluated
g(): evaluated
g(): called
f(): called
```

Decorator Evaluation

There is a well defined order to how decorators applied to various declarations inside of a class are applied:

1. *Parameter Decorators*, followed by *Method*, *Accessor*, or *Property Decorators* are applied for each instance member.
2. *Parameter Decorators*, followed by *Method*, *Accessor*, or *Property Decorators* are applied for each static member.
3. *Parameter Decorators* are applied for the constructor.
4. *Class Decorators* are applied for the class.

Class Decorators

A *Class Decorator* is declared just before a class declaration. The class decorator is applied to the constructor of the class and can be used to observe, modify, or replace a class definition. A class decorator cannot be used in a declaration file, or in any other ambient context (such as on a `declare class`).

The expression for the class decorator will be called as a function at runtime, with the constructor of the decorated class as its only argument.

If the class decorator returns a value, it will replace the class declaration with the provided constructor function.

NOTE Should you chose to return a new constructor function, you must take care to maintain the original prototype. The logic that applies decorators at runtime will **not** do this for you.

The following is an example of a class decorator (`@sealed`) applied to the `Greeter` class:

```
@sealed
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

We can define the `@sealed` decorator using the following function declaration:

```
function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}
```

When `@sealed` is executed, it will seal both the constructor and its prototype.

Next we have an example of how to override the constructor.

```
function classDecorator<T extends {new(...args:any[]):{}}>(constructor:T) {
  return class extends constructor {
    newProperty = "new property";
    hello = "override";
  }
}

@classDecorator
class Greeter {
  property = "property";
  hello: string;
  constructor(m: string) {
    this.hello = m;
  }
}
```

```

    }

console.log(new Greeter("world"));

```

Method Decorators

A *Method Decorator* is declared just before a method declaration. The decorator is applied to the *Property Descriptor* for the method, and can be used to observe, modify, or replace a method definition. A method decorator cannot be used in a declaration file, on an overload, or in any other ambient context (such as in a `declare class`).

The expression for the method decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The *Property Descriptor* for the member.

NOTE The *Property Descriptor* will be `undefined` if your script target is less than `ES5`.

If the method decorator returns a value, it will be used as the *Property Descriptor* for the method.

NOTE The return value is ignored if your script target is less than `ES5`.

The following is an example of a method decorator (`@enumerable`) applied to a method on the `Greeter` class:

```

class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }

  @enumerable(false)
  greet() {
    return "Hello, " + this.greeting;
  }
}

```

We can define the `@enumerable` decorator using the following function declaration:

```

function enumerable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    descriptor.enumerable = value;
  };
}

```

The `@enumerable(false)` decorator here is a [decorator factory](#). When the `@enumerable(false)` decorator is called, it modifies the `enumerable` property of the property descriptor.

Accessor Decorators

An *Accessor Decorator* is declared just before an accessor declaration. The accessor decorator is applied to the *Property Descriptor* for the accessor and can be used to observe, modify, or replace an accessor's definitions. An accessor decorator cannot be used in a declaration file, or in any other ambient context (such as in a `declare class`).

NOTE TypeScript disallows decorating both the `get` and `set` accessor for a single member. Instead, all decorators for the member must be applied to the first accessor specified in document order. This is because decorators apply to a *Property Descriptor*, which combines both the `get` and `set` accessor, not each declaration separately.

The expression for the accessor decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The *Property Descriptor* for the member.

NOTE The *Property Descriptor* will be `undefined` if your script target is less than `ES5`.

If the accessor decorator returns a value, it will be used as the *Property Descriptor* for the member.

NOTE The return value is ignored if your script target is less than `ES5`.

The following is an example of an accessor decorator (`@configurable`) applied to a member of the `Point` class:

```
class Point {
    private _x: number;
    private _y: number;
    constructor(x: number, y: number) {
        this._x = x;
        this._y = y;
    }
    @configurable(false)
    get x() { return this._x; }

    @configurable(false)
    get y() { return this._y; }
}
```

We can define the `@configurable` decorator using the following function declaration:

```
function configurable(value: boolean) {
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        descriptor.configurable = value;
    };
}
```

Property Decorators

A *Property Decorator* is declared just before a property declaration. A property decorator cannot be used in a declaration file, or in any other ambient context (such as in a `declare` class).

The expression for the property decorator will be called as a function at runtime, with the following two arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.

NOTE A *Property Descriptor* is not provided as an argument to a property decorator due to how property decorators are initialized in TypeScript. This is because there is currently no mechanism to describe an instance property when defining members of a prototype, and no way to observe or modify the initializer for a

property. The return value is ignored too. As such, a property decorator can only be used to observe that a property of a specific name has been declared for a class.

We can use this information to record metadata about the property, as in the following example:

```
class Greeter {
    @format("Hello, %s")
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        let formatString = getFormat(this, "greeting");
        return formatString.replace("%s", this.greeting);
    }
}
```

We can then define the `@format` decorator and `getFormat` functions using the following function declarations:

```
import "reflect-metadata";

const formatMetadataKey = Symbol("format");

function format(formatString: string) {
    return Reflect.metadata(formatMetadataKey, formatString);
}

function getFormat(target: any, propertyKey: string) {
    return Reflect.getMetadata(formatMetadataKey, target, propertyKey);
}
```

The `@format("Hello, %s")` decorator here is a [decorator factory](#). When `@format("Hello, %s")` is called, it adds a metadata entry for the property using the `Reflect.metadata` function from the `reflect-metadata` library. When `getFormat` is called, it reads the metadata value for the format.

NOTE This example requires the `reflect-metadata` library. See [Metadata](#) for more information about the `reflect-metadata` library.

Parameter Decorators

A *Parameter Decorator* is declared just before a parameter declaration. The parameter decorator is applied to the function for a class constructor or method declaration. A parameter decorator cannot be used in a declaration file, an overload, or in any other ambient context (such as in a `declare class`).

The expression for the parameter decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The ordinal index of the parameter in the function's parameter list.

NOTE A parameter decorator can only be used to observe that a parameter has been declared on a method.

The return value of the parameter decorator is ignored.

The following is an example of a parameter decorator (`@required`) applied to parameter of a member of the `Greeter` class:

```

class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    @validate
    greet(@required name: string) {
        return "Hello " + name + ", " + this.greeting;
    }
}

```

We can then define the `@required` and `@validate` decorators using the following function declarations:

```

import "reflect-metadata";

const requiredMetadataKey = Symbol("required");

function required(target: Object, propertyKey: string | symbol, parameterIndex: number) {
    let existingRequiredParameters: number[] = Reflect.getOwnMetadata(requiredMetadataKey, target, propertyKey)
    || [];
    existingRequiredParameters.push(parameterIndex);
    Reflect.defineMetadata(requiredMetadataKey, existingRequiredParameters, target, propertyKey);
}

function validate(target: any, propertyName: string, descriptor: TypedPropertyDescriptor<Function>) {
    let method = descriptor.value;
    descriptor.value = function () {
        let requiredParameters: number[] = Reflect.getOwnMetadata(requiredMetadataKey, target, propertyName);
        if (requiredParameters) {
            for (let parameterIndex of requiredParameters) {
                if (parameterIndex >= arguments.length || arguments[parameterIndex] === undefined) {
                    throw new Error("Missing required argument.");
                }
            }
        }
        return method.apply(this, arguments);
    }
}

```

The `@required` decorator adds a metadata entry that marks the parameter as required. The `@validate` decorator then wraps the existing `greet` method in a function that validates the arguments before invoking the original method.

NOTE This example requires the `reflect-metadata` library. See [Metadata](#) for more information about the `reflect-metadata` library.

Metadata

Some examples use the `reflect-metadata` library which adds a polyfill for an [experimental metadata API](#). This library is not yet part of the ECMAScript (JavaScript) standard. However, once decorators are officially adopted as part of the ECMAScript standard these extensions will be proposed for adoption.

You can install this library via npm:

```
npm i reflect-metadata --save
```

TypeScript includes experimental support for emitting certain types of metadata for declarations that have decorators. To enable this experimental support, you must set the `emitDecoratorMetadata` compiler option either on the command line or in your `tsconfig.json`:

Command Line:

```
tsc --target ES5 --experimentalDecorators --emitDecoratorMetadata
```

tsconfig.json:

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

When enabled, as long as the `reflect-metadata` library has been imported, additional design-time type information will be exposed at runtime.

We can see this in action in the following example:

```
import "reflect-metadata";

class Point {
  x: number;
  y: number;
}

class Line {
  private _p0: Point;
  private _p1: Point;

  @validate
  set p0(value: Point) { this._p0 = value; }
  get p0() { return this._p0; }

  @validate
  set p1(value: Point) { this._p1 = value; }
  get p1() { return this._p1; }
}

function validate<T>(target: any, propertyKey: string, descriptor: TypedPropertyDescriptor<T>) {
  let set = descriptor.set;
  descriptor.set = function (value: T) {
    let type = Reflect.getMetadata("design:type", target, propertyKey);
    if (!(value instanceof type)) {
      throw new TypeError("Invalid type.");
    }
    set(value);
  }
}
```

The TypeScript compiler will inject design-time type information using the `@Reflect.metadata` decorator. You could consider it the equivalent of the following TypeScript:

```
class Line {
  private _p0: Point;
  private _p1: Point;

  @validate
```

```
@Reflect.metadata("design:type", Point)
set p0(value: Point) { this._p0 = value; }
get p0() { return this._p0; }

@validate
@Reflect.metadata("design:type", Point)
set p1(value: Point) { this._p1 = value; }
get p1() { return this._p1; }
}
```

NOTE Decorator metadata is an experimental feature and may introduce breaking changes in future releases.

Introduction

Along with traditional OO hierarchies, another popular way of building up classes from reusable components is to build them by combining simpler partial classes. You may be familiar with the idea of mixins or traits for languages like Scala, and the pattern has also reached some popularity in the JavaScript community.

Mixin sample

In the code below, we show how you can model mixins in TypeScript. After the code, we'll break down how it works.

```
// Disposable Mixin
class Disposable {
    isDisposed: boolean;
    dispose() {
        this.isDisposed = true;
    }
}

// Activatable Mixin
class Activatable {
    isActive: boolean;
    activate() {
        this.isActive = true;
    }
    deactivate() {
        this.isActive = false;
    }
}

class SmartObject implements Disposable, Activatable {
    constructor() {
        setInterval(() => console.log(this.isActive + " : " + this.isDisposed), 500);
    }

    interact() {
        this.activate();
    }

    // Disposable
    isDisposed: boolean = false;
    dispose: () => void;
    // Activatable
    isActive: boolean = false;
    activate: () => void;
    deactivate: () => void;
}
applyMixins(SmartObject, [Disposable, Activatable]);

let smartObj = new SmartObject();
setTimeout(() => smartObj.interact(), 1000);

///////////////////////////////
// In your runtime library somewhere
///////////////////////////////

function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        });
    });
}
```

```
    });
}
```

Understanding the sample

The code sample starts with the two classes that will act as our mixins. You can see each one is focused on a particular activity or capability. We'll later mix these together to form a new class from both capabilities.

```
// Disposable Mixin
class Disposable {
    isDisposed: boolean;
    dispose() {
        this.isDisposed = true;
    }
}

// Activatable Mixin
class Activatable {
    isActive: boolean;
    activate() {
        this.isActive = true;
    }
    deactivate() {
        this.isActive = false;
    }
}
```

Next, we'll create the class that will handle the combination of the two mixins. Let's look at this in more detail to see how it does this:

```
class SmartObject implements Disposable, Activatable {
```

The first thing you may notice in the above is that instead of using `extends`, we use `implements`. This treats the classes as interfaces, and only uses the types behind `Disposable` and `Activatable` rather than the implementation. This means that we'll have to provide the implementation in class. Except, that's exactly what we want to avoid by using mixins.

To satisfy this requirement, we create stand-in properties and their types for the members that will come from our mixins. This satisfies the compiler that these members will be available at runtime. This lets us still get the benefit of the mixins, albeit with some bookkeeping overhead.

```
// Disposable
isDisposed: boolean = false;
dispose: () => void;
// Activatable
isActive: boolean = false;
activate: () => void;
deactivate: () => void;
```

Finally, we mix our mixins into the class, creating the full implementation.

```
applyMixins(SmartObject, [Disposable, Activatable]);
```

Lastly, we create a helper function that will do the mixing for us. This will run through the properties of each of the mixins and copy them over to the target of the mixins, filling out the stand-in properties with their implementations.

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
  baseCtors.forEach(baseCtor => {
    Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
      derivedCtor.prototype[name] = baseCtor.prototype[name];
    });
  });
}
```

Introduction

TypeScript provides several utility types to facilitate common type transformations. These utilities are available globally.

Table of contents

- [Partial<T>](#)
- [Readonly<T>](#)
- [Record<K, T>](#)
- [Pick<T, K>](#)
- [Exclude<T, U>](#)
- [Extract<T, U>](#)
- [NonNullable<T>](#)
- [ReturnType<T>](#)
- [InstanceType<T>](#)
- [Required<T>](#)
- [ThisType<T>](#)

Partial<T>

Constructs a type with all properties of `T` set to optional. This utility will return a type that represents all subsets of a given type.

Example

```
interface Todo {  
    title: string;  
    description: string;  
}  
  
function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {  
    return { ...todo, ...fieldsToUpdate };  
}  
  
const todo1 = {  
    title: 'organize desk',  
    description: 'clear clutter',  
};  
  
const todo2 = updateTodo(todo1, {  
    description: 'throw out trash',  
});
```

Readonly<T>

Constructs a type with all properties of `T` set to `readonly`, meaning the properties of the constructed type cannot be reassigned.

Example

```

interface Todo {
    title: string;
}

const todo: Readonly<Todo> = {
    title: 'Delete inactive users',
};

todo.title = 'Hello'; // Error: cannot reassign a readonly property

```

This utility is useful for representing assignment expressions that will fail at runtime (i.e. when attempting to reassign properties of a [frozen object](#)).

`Object.freeze`

```
function freeze<T>(obj: T): Readonly<T>;
```

Record<K, T>

Constructs a type with a set of properties `k` of type `T`. This utility can be used to map the properties of a type to another type.

Example

```

interface PageInfo {
    title: string;
}

type Page = 'home' | 'about' | 'contact';

const x: Record<Page, PageInfo> = {
    about: { title: 'about' },
    contact: { title: 'contact' },
    home: { title: 'home' },
};

```

Pick<T, K>

Constructs a type by picking the set of properties `k` from `T`.

Example

```

interface Todo {
    title: string;
    description: string;
    completed: boolean;
}

type TodoPreview = Pick<Todo, 'title' | 'completed'>;

const todo: TodoPreview = {
    title: 'Clean room',
    completed: false,
};

```

Exclude<T, U>

Constructs a type by excluding from `T` all properties that are assignable to `U`.

Example

```
type T0 = Exclude<"a" | "b" | "c", "a">; // "b" | "c"
type T1 = Exclude<"a" | "b" | "c", "a" | "b">; // "c"
type T2 = Exclude<string | number | ((() => void), Function>; // string | number
```

Extract<T, U>

Constructs a type by extracting from `T` all properties that are assignable to `U`.

Example

```
type T0 = Extract<"a" | "b" | "c", "a" | "f">; // "a"
type T1 = Extract<string | number | ((() => void), Function>; // () => void
```

NonNullable<T>

Constructs a type by excluding `null` and `undefined` from `T`.

Example

```
type T0 = NonNullable<string | number | undefined>; // string | number
type T1 = NonNullable<string[] | null | undefined>; // string[]
```

ReturnType<T>

Constructs a type consisting of the return type of function `T`.

Example

```
type T0 = ReturnType<() => string>; // string
type T1 = ReturnType<(s: string) => void>; // void
type T2 = ReturnType<(<T>() => T)>; // {}
type T3 = ReturnType<(<T extends U, U extends number[]>() => T)>; // number[]
type T4 = ReturnType<typeof f1>; // { a: number, b: string }
type T5 = ReturnType<any>; // any
type T6 = ReturnType<never>; // any
type T7 = ReturnType<string>; // Error
type T8 = ReturnType<Function>; // Error
```

InstanceType<T>

Constructs a type consisting of the instance type of a constructor function type `T`.

Example

```

class C {
  x = 0;
  y = 0;
}

type T0 = InstanceType<typeof C>; // C
type T1 = InstanceType<any>; // any
type T2 = InstanceType<never>; // any
type T3 = InstanceType<string>; // Error
type T4 = InstanceType<Function>; // Error

```

Required<T>

Constructs a type consisting of all properties of `T` set to required.

Example

```

interface Props {
  a?: number;
  b?: string;
};

const obj: Props = { a: 5 }; // OK

const obj2: Required<Props> = { a: 5 }; // Error: property 'b' missing

```

ThisType<T>

This utility does not return a transformed type. Instead, it serves a marker for a contextual `this` type. Note that the `--noImplicitThis` flag must be enabled to use this utility.

Example

```

// Compile with --noImplicitThis

type ObjectDescriptor<D, M> = {
  data?: D;
  methods?: M & ThisType<D & M>; // Type of 'this' in methods is D & M
}

function makeObject<D, M>(desc: ObjectDescriptor<D, M>): D & M {
  let data: object = desc.data || {};
  let methods: object = desc.methods || {};
  return { ...data, ...methods } as D & M;
}

let obj = makeObject({
  data: { x: 0, y: 0 },
  methods: {
    moveBy(dx: number, dy: number) {
      this.x += dx; // Strongly typed this
      this.y += dy; // Strongly typed this
    }
  }
);

obj.x = 10;
obj.y = 20;
obj.moveBy(5, 5);

```

In the example above, the `methods` object in the argument to `makeObject` has a contextual type that includes `ThisType<D & M>` and therefore the type of `this` in methods within the `methods` object is `{ x: number, y: number } & { moveBy(dx: number, dy: number): number }`. Notice how the type of the `methods` property simultaneously is an inference target and a source for the `this` type in methods.

The `ThisType<T>` marker interface is simply an empty interface declared in `lib.d.ts`. Beyond being recognized in the contextual type of an object literal, the interface acts like any empty interface.

Triple-slash directives are single-line comments containing a single XML tag. The contents of the comment are used as compiler directives.

Triple-slash directives are **only** valid at the top of their containing file. A triple-slash directive can only be preceded by single or multi-line comments, including other triple-slash directives. If they are encountered following a statement or a declaration they are treated as regular single-line comments, and hold no special meaning.

```
/// <reference path="..." />
```

The `/// <reference path="..." />` directive is the most common of this group. It serves as a declaration of *dependency* between files.

Triple-slash references instruct the compiler to include additional files in the compilation process.

They also serve as a method to order the output when using `--out` or `--outFile`. Files are emitted to the output file location in the same order as the input after preprocessing pass.

Preprocessing input files

The compiler performs a preprocessing pass on input files to resolve all triple-slash reference directives. During this process, additional files are added to the compilation.

The process starts with a set of *root files*; these are the file names specified on the command-line or in the `"files"` list in the `tsconfig.json` file. These root files are preprocessed in the same order they are specified. Before a file is added to the list, all triple-slash references in it are processed, and their targets included. Triple-slash references are resolved in a depth first manner, in the order they have been seen in the file.

A triple-slash reference path is resolved relative to the containing file, if unrooted.

Errors

It is an error to reference a file that does not exist. It is an error for a file to have a triple-slash reference to itself.

Using `--noResolve`

If the compiler flag `--noResolve` is specified, triple-slash references are ignored; they neither result in adding new files, nor change the order of the files provided.

```
/// <reference types="..." />
```

Similar to a `/// <reference path="..." />` directive, this directive serves as a declaration of *dependency*; a `/// <reference types="..." />` directive, however, declares a dependency on a package.

The process of resolving these package names is similar to the process of resolving module names in an `import` statement. An easy way to think of triple-slash-reference-types directives are as an `import` for declaration packages.

For example, including `/// <reference types="node" />` in a declaration file declares that this file uses names declared in `@types/node/index.d.ts`; and thus, this package needs to be included in the compilation along with the declaration file.

Use these directives only when you're authoring a `d.ts` file by hand.

For declaration files generated during compilation, the compiler will automatically add `/// <reference types="..." />` for you; A `/// <reference types="..." />` in a generated declaration file is added *if and only if* the resulting file uses any declarations from the referenced package.

For declaring a dependency on an `@types` package in a `.ts` file, use `--types` on the command line or in your `tsconfig.json` instead. See using `@types`, `typeRoots` and `types` in `tsconfig.json` files for more details.

`/// <reference lib="..." />`

This directive allows a file to explicitly include an existing built-in `/lib` file.

Built-in `/lib` files are referenced in the same fashion as the `"lib"` compiler option in `tsconfig.json` (e.g. use `lib="es2015"` and not `lib="lib.es2015.d.ts"`, etc.).

For declaration file authors who rely on built-in types, e.g. DOM APIs or built-in JS run-time constructors like `Symbol` or `Iterable`, triple-slash-reference `lib` directives are the recommended. Previously these `.d.ts` files had to add forward/duplicate declarations of such types.

For example, adding `/// <reference lib="es2017.string" />` to one of the files in a compilation is equivalent to compiling with `--lib es2017.string`.

```
//<reference lib="es2017.string" />
"foo".padStart(4);
```

`/// <reference no-default-lib="true"/>`

This directive marks a file as a *default library*. You will see this comment at the top of `lib.d.ts` and its different variants.

This directive instructs the compiler to *not* include the default library (i.e. `lib.d.ts`) in the compilation. The impact here is similar to passing `--noLib` on the command line.

Also note that when passing `--skipDefaultLibCheck`, the compiler will only skip checking files with `/// <reference no-default-lib="true"/>`.

`/// <amd-module />`

By default AMD modules are generated anonymous. This can lead to problems when other tools are used to process the resulting modules, such as bundlers (e.g. `r.js`).

The `amd-module` directive allows passing an optional module name to the compiler:

amdModule.ts

```
//<amd-module name="NamedModule"/>
export class C { }
```

Will result in assigning the name `NamedModule` to the module as part of calling the AMD `define`:

amdModule.js

```
define("NamedModule", ["require", "exports"], function (require, exports) {
  var C = (function () {
    function C() {
    }
    return C;
  })();
})()
```

```
    exports.C = C;
});
```

/// <amd-dependency />

Note: this directive has been deprecated. Use `import "moduleName";` statements instead.

`/// <amd-dependency path="x" />` informs the compiler about a non-TS module dependency that needs to be injected in the resulting module's require call.

The `amd-dependency` directive can also have an optional `name` property; this allows passing an optional name for an `amd-dependency`:

```
/// <amd-dependency path="legacy/moduleA" name="moduleA"/>
declare var moduleA:MyType
moduleA.callStuff()
```

Generated JS code:

```
define(["require", "exports", "legacy/moduleA"], function (require, exports, moduleA) {
  moduleA.callStuff()
});
```

This guide is designed to teach you how to write a high-quality TypeScript Declaration File.

In this guide, we'll assume basic familiarity with the TypeScript language. If you haven't already, you should read the [TypeScript Handbook](#) to familiarize yourself with basic concepts, especially types and namespaces.

Sections

The guide is broken down into the following sections.

Library Structures

The [Library Structures](#) guide helps you understand common library formats and how to write a correct declaration file for each format. If you're editing an existing file, you probably don't need to read this section. Authors of new declaration files must read this section to properly understand how the format of the library influences the writing of the declaration file.

By Example

Many times, we are faced with writing a declaration file when we only have examples of the underlying library to guide us. The [By Example](#) section shows many common API patterns and how to write declarations for each of them. This guide is aimed at the TypeScript novice who may not yet be familiar with every language construct in TypeScript.

"Do's and "Don't"s

Many common mistakes in declaration files can be easily avoided. The [Do's and Don'ts](#) section identifies common errors, describes how to detect them, and how to fix them. Everyone should read this section to help themselves avoid common mistakes.

Deep Dive

For seasoned authors interested in the underlying mechanics of how declaration files work, the [Deep Dive](#) section explains many advanced concepts in declaration writing, and shows how to leverage these concepts to create cleaner and more intuitive declaration files.

Templates

In [Templates](#) you'll find a number of declaration files that serve as a useful starting point when writing a new file. Refer to the documentation in [Library Structures](#) to figure out which template file to use.

Publish to npm

The [Publishing](#) section explains how to publish your declaration files to an npm package, and shows how to manage your dependent packages.

Find and Install Declaration Files

For JavaScript library users, the [Consumption](#) section offers a few simple steps to locate and install corresponding declaration files.

Overview

Broadly speaking, the way you *structure* your declaration file depends on how the library is consumed. There are many ways of offering a library for consumption in JavaScript, and you'll need to write your declaration file to match it. This guide covers how to identify common library patterns, and how to write declaration files which correspond to that pattern.

Each type of major library structuring pattern has a corresponding file in the [Templates](#) section. You can start with these templates to help you get going faster.

Identifying Kinds of Libraries

First, we'll review the kinds of libraries TypeScript declaration files can represent. We'll briefly show how each kind of library is *used*, how it is *written*, and list some example libraries from the real world.

Identifying the structure of a library is the first step in writing its declaration file. We'll give hints on how to identify structure both based on its *usage* and its *code*. Depending on the library's documentation and organization, one might be easier than the other. We recommend using whichever is more comfortable to you.

Global Libraries

A *global* library is one that can be accessed from the global scope (i.e. without using any form of `import`). Many libraries simply expose one or more global variables for use. For example, if you were using [jQuery](#), the `\$` variable can be used by simply referring to it:

```
$(() => { console.log('hello!'); } );
```

You'll usually see guidance in the documentation of a global library of how to use the library in an HTML script tag:

```
<script src="http://a.great.cdn.for/someLib.js"></script>
```

Today, most popular globally-accessible libraries are actually written as UMD libraries (see below). UMD library documentation is hard to distinguish from global library documentation. Before writing a global declaration file, make sure the library isn't actually UMD.

Identifying a Global Library from Code

Global library code is usually extremely simple. A global "Hello, world" library might look like this:

```
function createGreeting(s) {
  return "Hello, " + s;
}
```

or like this:

```
window.createGreeting = function(s) {
  return "Hello, " + s;
}
```

When looking at the code of a global library, you'll usually see:

- Top-level `var` statements or `function` declarations
- One or more assignments to `window.someName`
- Assumptions that DOM primitives like `document` OR `window` exist

You *won't* see:

- Checks for, or usage of, module loaders like `require` or `define`
- CommonJS/Node.js-style imports of the form `var fs = require("fs");`
- Calls to `define(...)`
- Documentation describing how to `require` or import the library

Examples of Global Libraries

Because it's usually easy to turn a global library into a UMD library, very few popular libraries are still written in the global style. However, libraries that are small and require the DOM (or have *no* dependencies) may still be global.

Global Library Template

The template file `global.d.ts` defines an example library `myLib`. Be sure to read the "[Preventing Name Conflicts](#)" footnote.

Modular Libraries

Some libraries only work in a module loader environment. For example, because `express` only works in Node.js and must be loaded using the CommonJS `require` function.

ECMAScript 2015 (also known as ES2015, ECMAScript 6, and ES6), CommonJS, and RequireJS have similar notions of *importing a module*. In JavaScript CommonJS (Node.js), for example, you would write

```
var fs = require("fs");
```

In TypeScript or ES6, the `import` keyword serves the same purpose:

```
import fs = require("fs");
```

You'll typically see modular libraries include one of these lines in their documentation:

```
var someLib = require('someLib');
```

or

```
define(..., ['someLib'], function(someLib) {  
});
```

As with global modules, you might see these examples in the documentation of a UMD module, so be sure to check the code or documentation.

Identifying a Module Library from Code

Modular libraries will typically have at least some of the following:

- Unconditional calls to `require` or `define`
- Declarations like `import * as a from 'b';` or `export c;`
- Assignments to `exports` or `module.exports`

They will rarely have:

- Assignments to properties of `window` or `global`

Examples of Modular Libraries

Many popular Node.js libraries are in the module family, such as `express`, `gulp`, and `request`.

UMD

A *UMD* module is one that can *either* be used as module (through an import), or as a global (when run in an environment without a module loader). Many popular libraries, such as [Moment.js](#), are written this way. For example, in Node.js or using RequireJS, you would write:

```
import moment = require("moment");
console.log(moment.format());
```

whereas in a vanilla browser environment you would write:

```
console.log(moment.format());
```

Identifying a UMD library

[UMD modules](#) check for the existence of a module loader environment. This is an easy-to-spot pattern that looks something like this:

```
(function (root, factory) {
    if (typeof define === "function" && define.amd) {
        define(["libName"], factory);
    } else if (typeof module === "object" && module.exports) {
        module.exports = factory(require("libName"));
    } else {
        root.returnExports = factory(root.libName);
    }
})(this, function (b) {
```

If you see tests for `typeof define`, `typeof window`, or `typeof module` in the code of a library, especially at the top of the file, it's almost always a UMD library.

Documentation for UMD libraries will also often demonstrate a "Using in Node.js" example showing `require`, and a "Using in the browser" example showing using a `<script>` tag to load the script.

Examples of UMD libraries

Most popular libraries are now available as UMD packages. Examples include [jQuery](#), [Moment.js](#), [lodash](#), and many more.

Template

There are three templates available for modules, `module.d.ts`, `module-class.d.ts` and `module-function.d.ts`.

Use `module-function.d.ts` if your module can be *called* like a function:

```
var x = require("foo");
// Note: calling 'x' as a function
var y = x(42);
```

Be sure to read the [footnote "The Impact of ES6 on Module Call Signatures"](#)

Use `module-class.d.ts` if your module can be *constructed* using `new`:

```
var x = require("bar");
// Note: using 'new' operator on the imported variable
var y = new x("hello");
```

The same [footnote](#) applies to these modules.

If your module is not callable or constructable, use the `module.d.ts` file.

Module Plugin or UMD Plugin

A *module plugin* changes the shape of another module (either UMD or module). For example, in `Moment.js`, `moment-range` adds a new `range` method to the `moment` object.

For the purposes of writing a declaration file, you'll write the same code whether the module being changed is a plain module or UMD module.

Template

Use the `module-plugin.d.ts` template.

Global Plugin

A *global plugin* is global code that changes the shape of some global. As with *global-modifying modules*, these raise the possibility of runtime conflict.

For example, some libraries add new functions to `Array.prototype` or `String.prototype`.

Identifying global plugins

Global plugins are generally easy to identify from their documentation.

You'll see examples that look like this:

```
var x = "hello, world";
// Creates new methods on built-in types
console.log(x.startsWithHello());

var y = [1, 2, 3];
// Creates new methods on built-in types
console.log(y.reverseAndSort());
```

Template

Use the `global-plugin.d.ts` template.

Global-modifying Modules

A *global-modifying module* alters existing values in the global scope when they are imported. For example, there might exist a library which adds new members to `String.prototype` when imported. This pattern is somewhat dangerous due to the possibility of runtime conflicts, but we can still write a declaration file for it.

Identifying global-modifying modules

Global-modifying modules are generally easy to identify from their documentation. In general, they're similar to global plugins, but need a `require` call to activate their effects.

You might see documentation like this:

```
// 'require' call that doesn't use its return value
var unused = require("magic-string-time");
/* or */
require("magic-string-time");

var x = "hello, world";
// Creates new methods on built-in types
console.log(x.startsWithHello());

var y = [1, 2, 3];
// Creates new methods on built-in types
console.log(y.reverseAndSort());
```

Template

Use the `global-modifying-module.d.ts` template.

Consuming Dependencies

There are several kinds of dependencies you might have.

Dependencies on Global Libraries

If your library depends on a global library, use a `/// <reference types="..." />` directive:

```
/// <reference types="someLib" />

function getThing(): someLib.thing;
```

Dependencies on Modules

If your library depends on a module, use an `import` statement:

```
import * as moment from "moment";

function getThing(): moment;
```

Dependencies on UMD libraries

From a Global Library

If your global library depends on a UMD module, use a `/// <reference types` directive:

```
/// <reference types="moment" />

function getThing(): moment;
```

From a Module or UMD Library

If your module or UMD library depends on a UMD library, use an `import` statement:

```
import * as someLib from 'someLib';
```

Do *not* use a `/// <reference` directive to declare a dependency to a UMD library!

Footnotes

Preventing Name Conflicts

Note that it's possible to define many types in the global scope when writing a global declaration file. We strongly discourage this as it leads to possible unresolvable name conflicts when many declaration files are in a project.

A simple rule to follow is to only declare types *namespaced* by whatever global variable the library defines. For example, if the library defines the global value 'cats', you should write

```
declare namespace cats {
    interface KittySettings { }
}
```

But *not*

```
// at top-level
interface CatsKittySettings { }
```

This guidance also ensures that the library can be transitioned to UMD without breaking declaration file users.

The Impact of ES6 on Module Plugins

Some plugins add or modify top-level exports on existing modules. While this is legal in CommonJS and other loaders, ES6 modules are considered immutable and this pattern will not be possible. Because TypeScript is loader-agnostic, there is no compile-time enforcement of this policy, but developers intending to transition to an ES6 module loader should be aware of this.

The Impact of ES6 on Module Call Signatures

Many popular libraries, such as Express, expose themselves as a callable function when imported. For example, the typical Express usage looks like this:

```
import exp = require("express");
var app = exp();
```

In ES6 module loaders, the top-level object (here imported as `exp`) can only have properties; the top-level module object is *never* callable. The most common solution here is to define a `default` export for a callable/constructable object; some module loader shims will automatically detect this situation and replace the top-level object with the `default export`.

Introduction

The purpose of this guide is to teach you how to write a high-quality definition file. This guide is structured by showing documentation for some API, along with sample usage of that API, and explaining how to write the corresponding declaration.

These examples are ordered in approximately increasing order of complexity.

- [Global Variables](#)
- [Global Functions](#)
- [Objects with Properties](#)
- [Overloaded Function](#)
- [Reusable Types \(Interfaces\)](#)
- [Reusable Types \(Type Aliases\)](#)
- [Organizing Types](#)
- [Classes](#)

The Examples

Global Variables

Documentation

The global variable `foo` contains the number of widgets present.

Code

```
console.log("Half the number of widgets is " + (foo / 2));
```

Declaration

Use `declare var` to declare variables. If the variable is read-only, you can use `declare const`. You can also use `declare let` if the variable is block-scoped.

```
/** The number of widgets present */
declare var foo: number;
```

Global Functions

Documentation

You can call the function `greet` with a string to show a greeting to the user.

Code

```
greet("hello, world");
```

Declaration

Use `declare function` to declare functions.

```
declare function greet(greeting: string): void;
```

Objects with Properties

Documentation

The global variable `myLib` has a function `makeGreeting` for creating greetings, and a property `numberOfGreetings` indicating the number of greetings made so far.

Code

```
let result = myLib.makeGreeting("hello, world");
console.log("The computed greeting is:" + result);

let count = myLib.numberOfGreetings;
```

Declaration

Use `declare namespace` to describe types or values accessed by dotted notation.

```
declare namespace myLib {
    function makeGreeting(s: string): string;
    let numberOfGreetings: number;
}
```

Overloaded Functions

Documentation

The `getWidget` function accepts a number and returns a Widget, or accepts a string and returns a Widget array.

Code

```
let x: Widget = getWidget(43);

let arr: Widget[] = getWidget("all of them");
```

Declaration

```
declare function getWidget(n: number): Widget;
declare function getWidget(s: string): Widget[];
```

Reusable Types (Interfaces)

Documentation

When specifying a greeting, you must pass a `GreetingSettings` object. This object has the following properties:

- 1 - greeting: Mandatory string
- 2 - duration: Optional length of time (in milliseconds)
- 3 - color: Optional string, e.g. '#ff00ff'

Code

```
greet({
  greeting: "hello world",
  duration: 4000
});
```

Declaration

Use an `interface` to define a type with properties.

```
interface GreetingSettings {
  greeting: string;
  duration?: number;
  color?: string;
}

declare function greet(setting: GreetingSettings): void;
```

Reusable Types (Type Aliases)

Documentation

Anywhere a greeting is expected, you can provide a `string`, a function returning a `string`, or a `Greeter` instance.

Code

```
function getGreeting() {
  return "howdy";
}
class MyGreeter extends Greeter { }

greet("hello");
greet(getGreeting);
greet(new MyGreeter());
```

Declaration

You can use a type alias to make a shorthand for a type:

```
type GreetingLike = string | (() => string) | MyGreeter;

declare function greet(g: GreetingLike): void;
```

Organizing Types

Documentation

The `greeter` object can log to a file or display an alert. You can provide LogOptions to `.log(...)` and alert options to `.alert(...)`

Code

```
const g = new Greeter("Hello");
g.log({ verbose: true });
g.alert({ modal: false, title: "Current Greeting" });
```

Declaration

Use namespaces to organize types.

```
declare namespace GreetingLib {
    interface LogOptions {
        verbose?: boolean;
    }
    interface AlertOptions {
        modal: boolean;
        title?: string;
        color?: string;
    }
}
```

You can also create nested namespaces in one declaration:

```
declare namespace GreetingLib.Options {
    // Refer to via GreetingLib.Options.Log
    interface Log {
        verbose?: boolean;
    }
    interface Alert {
        modal: boolean;
        title?: string;
        color?: string;
    }
}
```

Classes

Documentation

You can create a greeter by instantiating the `Greeter` object, or create a customized greeter by extending from it.

Code

```
const myGreeter = new Greeter("hello, world");
myGreeter.greeting = "howdy";
myGreeter.showGreeting();

class SpecialGreeter extends Greeter {
    constructor() {
        super("Very special greetings");
    }
}
```

Declaration

Use `declare class` to describe a class or class-like object. Classes can have properties and methods as well as a constructor.

```
declare class Greeter {
    constructor(greeting: string);

    greeting: string;
    showGreeting(): void;
}
```


General Types

Number , String , Boolean , and Object

Don't ever use the types `Number` , `String` , `Boolean` , or `Object` . These types refer to non-primitive boxed objects that are almost never used appropriately in JavaScript code.

```
/* WRONG */
function reverse(s: String): String;
```

Do use the types `number` , `string` , and `boolean` .

```
/* OK */
function reverse(s: string): string;
```

Instead of `Object` , use the non-primitive `object` type ([added in TypeScript 2.2](#)).

Generics

Don't ever have a generic type which doesn't use its type parameter. See more details in [TypeScript FAQ page](#).

Callback Types

Return Types of Callbacks

Don't use the return type `any` for callbacks whose value will be ignored:

```
/* WRONG */
function fn(x: () => any) {
    x();
}
```

Do use the return type `void` for callbacks whose value will be ignored:

```
/* OK */
function fn(x: () => void) {
    x();
}
```

Why: Using `void` is safer because it prevents you from accidentally using the return value of `x` in an unchecked way:

```
function fn(x: () => void) {
    var k = x(); // oops! meant to do something else
    k.doSomething(); // error, but would be OK if the return type had been 'any'
}
```

Optional Parameters in Callbacks

Don't use optional parameters in callbacks unless you really mean it:

```
/* WRONG */
interface Fetcher {
    getObject(done: (data: any, elapsedTime?: number) => void): void;
}
```

This has a very specific meaning: the `done` callback might be invoked with 1 argument or might be invoked with 2 arguments. The author probably intended to say that the callback might not care about the `elapsedTime` parameter, but there's no need to make the parameter optional to accomplish this -- it's always legal to provide a callback that accepts fewer arguments.

Do write callback parameters as non-optional:

```
/* OK */
interface Fetcher {
    getObject(done: (data: any, elapsedTime: number) => void): void;
}
```

Overloads and Callbacks

Don't write separate overloads that differ only on callback arity:

```
/* WRONG */
declare function beforeAll(action: () => void, timeout?: number): void;
declare function beforeAll(action: (done: DoneFn) => void, timeout?: number): void;
```

Do write a single overload using the maximum arity:

```
/* OK */
declare function beforeAll(action: (done: DoneFn) => void, timeout?: number): void;
```

Why: It's always legal for a callback to disregard a parameter, so there's no need for the shorter overload. Providing a shorter callback first allows incorrectly-typed functions to be passed in because they match the first overload.

Function Overloads

Ordering

Don't put more general overloads before more specific overloads:

```
/* WRONG */
declare function fn(x: any): any;
declare function fn(x: HTMLElement): number;
declare function fn(x: HTMLDivElement): string;

var myElem: HTMLDivElement;
var x = fn(myElem); // x: any, wat?
```

Do sort overloads by putting the more general signatures after more specific signatures:

```
/* OK */
declare function fn(x: HTMLDivElement): string;
```

```
declare function fn(x: HTMLElement): number;
declare function fn(x: any): any;

var myElem: HTMLDivElement;
var x = fn(myElem); // x: string, :)
```

Why: TypeScript chooses the *first matching overload* when resolving function calls. When an earlier overload is "more general" than a later one, the later one is effectively hidden and cannot be called.

Use Optional Parameters

Don't write several overloads that differ only in trailing parameters:

```
/* WRONG */
interface Example {
    diff(one: string): number;
    diff(one: string, two: string): number;
    diff(one: string, two: string, three: boolean): number;
}
```

Do use optional parameters whenever possible:

```
/* OK */
interface Example {
    diff(one: string, two?: string, three?: boolean): number;
}
```

Note that this collapsing should only occur when all overloads have the same return type.

Why: This is important for two reasons.

TypeScript resolves signature compatibility by seeing if any signature of the target can be invoked with the arguments of the source, *and extraneous arguments are allowed*. This code, for example, exposes a bug only when the signature is correctly written using optional parameters:

```
function fn(x: (a: string, b: number, c: number) => void) { }
var x: Example;
// When written with overloads, OK -- used first overload
// When written with optionals, correctly an error
fn(x.diff);
```

The second reason is when a consumer uses the "strict null checking" feature of TypeScript. Because unspecified parameters appear as `undefined` in JavaScript, it's usually fine to pass an explicit `undefined` to a function with optional arguments. This code, for example, should be OK under strict nulls:

```
var x: Example;
// When written with overloads, incorrectly an error because of passing 'undefined' to 'string'
// When written with optionals, correctly OK
x.diff("something", true ? undefined : "hour");
```

Use Union Types

Don't write overloads that differ by type in only one argument position:

```
/* WRONG */
```

```
interface Moment {  
    utcOffset(): number;  
    utcOffset(b: number): Moment;  
    utcOffset(b: string): Moment;  
}
```

Do use union types whenever possible:

```
/* OK */  
interface Moment {  
    utcOffset(): number;  
    utcOffset(b: number|string): Moment;  
}
```

Note that we didn't make `b` optional here because the return types of the signatures differ.

Why: This is important for people who are "passing through" a value to your function:

```
function fn(x: string): void;  
function fn(x: number): void;  
function fn(x: number|string) {  
    // When written with separate overloads, incorrectly an error  
    // When written with union types, correctly OK  
    return moment().utcOffset(x);  
}
```

Definition File Theory: A Deep Dive

Structuring modules to give the exact API shape you want can be tricky. For example, we might want a module that can be invoked with or without `new` to produce different types, has a variety of named types exposed in a hierarchy, and has some properties on the module object as well.

By reading this guide, you'll have the tools to write complex definition files that expose a friendly API surface. This guide focuses on module (or UMD) libraries because the options here are more varied.

Key Concepts

You can fully understand how to make any shape of definition by understanding some key concepts of how TypeScript works.

Types

If you're reading this guide, you probably already roughly know what a type in TypeScript is. To be more explicit, though, a *type* is introduced with:

- A type alias declaration (`type sn = number | string;`)
- An interface declaration (`interface I { x: number[]; }`)
- A class declaration (`class C { }`)
- An enum declaration (`enum E { A, B, C }`)
- An `import` declaration which refers to a type

Each of these declaration forms creates a new type name.

Values

As with types, you probably already understand what a value is. Values are runtime names that we can reference in expressions. For example `let x = 5;` creates a value called `x`.

Again, being explicit, the following things create values:

- `let`, `const`, and `var` declarations
- A `namespace` or `module` declaration which contains a value
- An `enum` declaration
- A `class` declaration
- An `import` declaration which refers to a value
- A `function` declaration

Namespaces

Types can exist in *namespaces*. For example, if we have the declaration `let x: A.B.C`, we say that the type `c` comes from the `A.B` namespace.

This distinction is subtle and important -- here, `A.B` is not necessarily a type or a value.

Simple Combinations: One name, multiple meanings

Given a name `A`, we might find up to three different meanings for `A`: a type, a value or a namespace. How the name is interpreted depends on the context in which it is used. For example, in the declaration `let m: A.A = A;`, `A` is used first as a namespace, then as a type name, then as a value. These meanings might end up referring to entirely different declarations!

This may seem confusing, but it's actually very convenient as long as we don't excessively overload things. Let's look at some useful aspects of this combining behavior.

Built-in Combinations

Astute readers will notice that, for example, `class` appeared in both the *type* and *value* lists. The declaration `class C { }` creates two things: a *type* `C` which refers to the instance shape of the class, and a *value* `C` which refers to the constructor function of the class. Enum declarations behave similarly.

User Combinations

Let's say we wrote a module file `foo.d.ts`:

```
export var SomeVar: { a: SomeType };
export interface SomeType {
  count: number;
}
```

Then consumed it:

```
import * as foo from './foo';
let x: foo.SomeType = foo.SomeVar.a;
console.log(x.count);
```

This works well enough, but we might imagine that `SomeType` and `SomeVar` were very closely related such that you'd like them to have the same name. We can use combining to present these two different objects (the value and the type) under the same name `Bar`:

```
export var Bar: { a: Bar };
export interface Bar {
  count: number;
}
```

This presents a very good opportunity for destructuring in the consuming code:

```
import { Bar } from './foo';
let x: Bar = Bar.a;
console.log(x.count);
```

Again, we've used `Bar` as both a type and a value here. Note that we didn't have to declare the `Bar` value as being of the `Bar` type -- they're independent.

Advanced Combinations

Some kinds of declarations can be combined across multiple declarations. For example, `class C { }` and `interface C { }` can co-exist and both contribute properties to the `C` types.

This is legal as long as it does not create a conflict. A general rule of thumb is that values always conflict with other values of the same name unless they are declared as `namespace S`, types will conflict if they are declared with a type alias declaration (`type S = string`), and namespaces never conflict.

Let's see how this can be used.

Adding using an `interface`

We can add additional members to an `interface` with another `interface` declaration:

```
interface Foo {
  x: number;
}
// ... elsewhere ...
interface Foo {
  y: number;
}
let a: Foo = ...;
console.log(a.x + a.y); // OK
```

This also works with classes:

```
class Foo {
  x: number;
}
// ... elsewhere ...
interface Foo {
  y: number;
}
let a: Foo = ...;
console.log(a.x + a.y); // OK
```

Note that we cannot add to type aliases (`type S = string;`) using an interface.

Adding using a `namespace`

A `namespace` declaration can be used to add new types, values, and namespaces in any way which does not create a conflict.

For example, we can add a static member to a class:

```
class C {
}
// ... elsewhere ...
namespace C {
  export let x: number;
}
let y = C.x; // OK
```

Note that in this example, we added a value to the `static` side of `C` (its constructor function). This is because we added a `value`, and the container for all values is another value (types are contained by namespaces, and namespaces are contained by other namespaces).

We could also add a namespaced type to a class:

```
class C {
}
// ... elsewhere ...
namespace C {
```

```
export interface D { }
}
let y: C.D; // OK
```

In this example, there wasn't a namespace `c` until we wrote the `namespace` declaration for it. The meaning `c` as a namespace doesn't conflict with the value or type meanings of `c` created by the class.

Finally, we could perform many different merges using `namespace` declarations. This isn't a particularly realistic example, but shows all sorts of interesting behavior:

```
namespace X {
  export interface Y { }
  export class Z { }
}

// ... elsewhere ...
namespace X {
  export var Y: number;
  export namespace Z {
    export class C { }
  }
}
type X = string;
```

In this example, the first block creates the following name meanings:

- A value `x` (because the `namespace` declaration contains a value, `z`)
- A namespace `x` (because the `namespace` declaration contains a type, `y`)
- A type `y` in the `x` namespace
- A type `z` in the `x` namespace (the instance shape of the class)
- A value `z` that is a property of the `x` value (the constructor function of the class)

The second block creates the following name meanings:

- A value `y` (of type `number`) that is a property of the `x` value
- A namespace `z`
- A value `z` that is a property of the `x` value
- A type `c` in the `x.z` namespace
- A value `c` that is a property of the `x.z` value
- A type `x`

Using with `export =` or `import`

An important rule is that `export` and `import` declarations export or import *all meanings* of their targets.

- [global-modifying-module.d.ts](#)
- [global-plugin.d.ts](#)
- [global.d.ts](#)
- [module-class.d.ts](#)
- [module-function.d.ts](#)
- [module-plugin.d.ts](#)
- [module.d.ts](#)

Now that you have authored a declaration file following the steps of this guide, it is time to publish it to npm. There are two main ways you can publish your declaration files to npm:

1. bundling with your npm package, or
2. publishing to the [@types organization](#) on npm.

If your package is written in TypeScript then the first approach is favored. Use the `--declaration` flag to generate declaration files. This way, your declarations and JavaScript always be in sync.

If your package is not written in TypeScript then the second is the preferred approach.

Including declarations in your npm package

If your package has a main `.js` file, you will need to indicate the main declaration file in your `package.json` file as well. Set the `types` property to point to your bundled declaration file. For example:

```
{
  "name": "awesome",
  "author": "Vandelay Industries",
  "version": "1.0.0",
  "main": "./lib/main.js",
  "types": "./lib/main.d.ts"
}
```

Note that the `" typings "` field is synonymous with `" types "`, and could be used as well.

Also note that if your main declaration file is named `index.d.ts` and lives at the root of the package (next to `index.js`) you do not need to mark the `" types "` property, though it is advisable to do so.

Dependencies

All dependencies are managed by npm. Make sure all the declaration packages you depend on are marked appropriately in the `"dependencies"` section in your `package.json`. For example, imagine we authored a package that used Browserify and TypeScript.

```
{
  "name": "browserify-typescript-extension",
  "author": "Vandelay Industries",
  "version": "1.0.0",
  "main": "./lib/main.js",
  "types": "./lib/main.d.ts",
  "dependencies": {
    "browserify": "latest",
    "@types/browserify": "latest",
    "typescript": "next"
  }
}
```

Here, our package depends on the `browserify` and `typescript` packages. `browserify` does not bundle its declaration files with its npm packages, so we needed to depend on `@types/browserify` for its declarations. `typescript`, on the other hand, packages its declaration files, so there was no need for any additional dependencies.

Our package exposes declarations from each of those, so any user of our `browserify-typescript-extension` package needs to have these dependencies as well. For that reason, we used `"dependencies"` and not `"devDependencies"`, because otherwise our consumers would have needed to manually install those packages. If we had just written a

command line application and not expected our package to be used as a library, we might have used `devDependencies`.

Red flags

```
/// <reference path="..." />
```

Don't use `/// <reference path="..." />` in your declaration files.

```
/// <reference path="../typescript/lib/typescriptServices.d.ts" />
....
```

Do use `/// <reference types="..." />` instead.

```
/// <reference types="typescript" />
....
```

Make sure to revisit the [Consuming dependencies](#) section for more information.

Packaging dependent declarations

If your type definitions depend on another package:

- *Don't combine it with yours, keep each in their own file.*
- *Don't copy the declarations in your package either.*
- *Do depend on the npm type declaration package if it doesn't package its declaration files.*

Publish to [@types](#)

Packages under the [@types](#) organization are published automatically from [DefinitelyTyped](#) using the [types-publisher tool](#). To get your declarations published as an @types package, please submit a pull request to <https://github.com/DefinitelyTyped/DefinitelyTyped>. You can find more details in the [contribution guidelines page](#).

In TypeScript 2.0, it has become significantly easier to consume declaration files, in acquiring, using, and finding them. This page details exactly how to do all three.

Downloading

Getting type declarations in TypeScript 2.0 and above requires no tools apart from npm.

As an example, getting the declarations for a library like lodash takes nothing more than the following command

```
npm install --save @types/lodash
```

It is worth noting that if the npm package already includes its declaration file as described in [Publishing](#), downloading the corresponding `@types` package is not needed.

Consuming

From there you'll be able to use lodash in your TypeScript code with no fuss. This works for both modules and global code.

For example, once you've `npm install -ed` your type declarations, you can use imports and write

```
import * as _ from "lodash";
_.padStart("Hello TypeScript!", 20, " ");
```

or if you're not using modules, you can just use the global variable `_`.

```
_.padStart("Hello TypeScript!", 20, " ");
```

Searching

For the most part, type declaration packages should always have the same name as the package name on `npm`, but prefixed with `@types/`, but if you need, you can check out <https://aka.ms/types> to find the package for your favorite library.

Note: if the declaration file you are searching for is not present, you can always contribute one back and help out the next developer looking for it. Please see the DefinitelyTyped [contribution guidelines page](#) for details.

Overview

The presence of a `tsconfig.json` file in a directory indicates that the directory is the root of a TypeScript project. The `tsconfig.json` file specifies the root files and the compiler options required to compile the project. A project is compiled in one of the following ways:

Using tsconfig.json

- By invoking `tsc` with no input files, in which case the compiler searches for the `tsconfig.json` file starting in the current directory and continuing up the parent directory chain.
- By invoking `tsc` with no input files and a `--project` (or just `-p`) command line option that specifies the path of a directory containing a `tsconfig.json` file, or a path to a valid `.json` file containing the configurations.

When input files are specified on the command line, `tsconfig.json` files are ignored.

Examples

Example `tsconfig.json` files:

- Using the `"files"` property

```
{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "sourceMap": true
  },
  "files": [
    "core.ts",
    "sys.ts",
    "types.ts",
    "scanner.ts",
    "parser.ts",
    "utilities.ts",
    "binder.ts",
    "checker.ts",
    "emitter.ts",
    "program.ts",
    "commandLineParser.ts",
    "tsc.ts",
    "diagnosticInformationMap.generated.ts"
  ]
}
```

- Using the `"include"` and `"exclude"` properties

```
{
  "compilerOptions": {
    "module": "system",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "outFile": "../../built/local/tsc.js",
    "sourceMap": true
  },
}
```

```

    "include": [
        "src/**/*"
    ],
    "exclude": [
        "node_modules",
        "**/*.spec.ts"
    ]
}

```

Details

The `"compilerOptions"` property can be omitted, in which case the compiler's defaults are used. See our full list of supported [Compiler Options](#).

The `"files"` property takes a list of relative or absolute file paths. The `"include"` and `"exclude"` properties take a list of glob-like file patterns. The supported glob wildcards are:

- `*` matches zero or more characters (excluding directory separators)
- `?` matches any one character (excluding directory separators)
- `**/` recursively matches any subdirectory

If a segment of a glob pattern includes only `*` or `.*`, then only files with supported extensions are included (e.g. `.ts`, `.tsx`, and `.d.ts` by default with `.js` and `.jsx` if `allowJs` is set to true).

If the `"files"` and `"include"` are both left unspecified, the compiler defaults to including all TypeScript (`.ts`, `.d.ts` and `.tsx`) files in the containing directory and subdirectories except those excluded using the `"exclude"` property. JS files (`.js` and `.jsx`) are also included if `allowJs` is set to true. If the `"files"` or `"include"` properties are specified, the compiler will instead include the union of the files included by those two properties. Files in the directory specified using the `"outDir"` compiler option are excluded as long as `"exclude"` property is not specified.

Files included using `"include"` can be filtered using the `"exclude"` property. However, files included explicitly using the `"files"` property are always included regardless of `"exclude"`. The `"exclude"` property defaults to excluding the `node_modules`, `bower_components`, `jspm_packages` and `<outDir>` directories when not specified.

Any files that are referenced by files included via the `"files"` or `"include"` properties are also included. Similarly, if a file `B.ts` is referenced by another file `A.ts`, then `B.ts` cannot be excluded unless the referencing file `A.ts` is also specified in the `"exclude"` list.

Please note that the compiler does not include files that can be possible outputs; e.g. if the input includes `index.ts`, then `index.d.ts` and `index.js` are excluded. In general, having files that differ only in extension next to each other is not recommended.

A `tsconfig.json` file is permitted to be completely empty, which compiles all files included by default (as described above) with the default compiler options.

Compiler options specified on the command line override those specified in the `tsconfig.json` file.

`@types` , `typeRoots` and `types`

By default all *visible* `"@types"` packages are included in your compilation. Packages in `node_modules/@types` of any enclosing folder are considered *visible*; specifically, that means packages within `./node_modules/@types/`, `../node_modules/@types/`, `.../..node_modules/@types/`, and so on.

If `typeRoots` is specified, *only* packages under `typeRoots` will be included. For example:

```
{
  "compilerOptions": {
    "typeRoots" : ["./typings"]
  }
}
```

This config file will include `all` packages under `./typings`, and no packages from `./node_modules/@types`.

If `types` is specified, only packages listed will be included. For instance:

```
{
  "compilerOptions": {
    "types" : ["node", "lodash", "express"]
  }
}
```

This `tsconfig.json` file will *only* include `./node_modules/@types/node`, `./node_modules/@types/lodash` and `./node_modules/@types/express`. Other packages under `node_modules/@types/*` will not be included.

A types package is a folder with a file called `index.d.ts` or a folder with a `package.json` that has a `types` field.

Specify `"types": []` to disable automatic inclusion of `@types` packages.

Keep in mind that automatic inclusion is only important if you're using files with global declarations (as opposed to files declared as modules). If you use an `import "foo"` statement, for instance, TypeScript may still look through `node_modules` & `node_modules/@types` folders to find the `foo` package.

Configuration inheritance with `extends`

A `tsconfig.json` file can inherit configurations from another file using the `extends` property.

The `extends` is a top-level property in `tsconfig.json` (alongside `compilerOptions`, `files`, `include`, and `exclude`). `extends`' value is a string containing a path to another configuration file to inherit from.

The configuration from the base file are loaded first, then overridden by those in the inheriting config file. If a circularity is encountered, we report an error.

`files`, `include` and `exclude` from the inheriting config file *overwrite* those from the base config file.

All relative paths found in the configuration file will be resolved relative to the configuration file they originated in.

For example:

`configs/base.json`:

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "strictNullChecks": true
  }
}
```

`tsconfig.json`:

```
{
  "extends": "./configs/base",
  "files": [
    "main.ts",
    "supplemental.ts"
  ]
}
```

```
    ]  
}
```

```
tsconfig.nostrictnull.json :
```

```
{  
  "extends": "./tsconfig",  
  "compilerOptions": {  
    "strictNullChecks": false  
  }  
}
```

compileOnSave

Setting a top-level property `compileOnSave` signals to the IDE to generate all files for a given tsconfig.json upon saving.

```
{  
  "compileOnSave": true,  
  "compilerOptions": {  
    "noImplicitAny" : true  
  }  
}
```

This feature is currently supported in Visual Studio 2015 with TypeScript 1.8.4 and above, and [atom-typescript](#) plugin.

Schema

Schema can be found at: <http://json.schemastore.org/tsconfig>

Compiler Options

Option	Type	Default	Description
--allowJs	boolean	false	Allow JavaScript files to be compiled.
--allowSyntheticDefaultImports	boolean	module === "system" or --esModuleInterop is set and module is not es2015 / esnext	Allow default imports from modules with no default export. This does not affect code emit, just typechecking.
--allowUnreachableCode	boolean	false	Do not report errors on unreachable code.
--allowUnusedLabels	boolean	false	Do not report errors on unused labels.
--alwaysStrict	boolean	false	Parse in strict mode and emit "use strict" for each source file
--baseUrl	string		Base directory to resolve non-relative module names. See Module Resolution documentation for more details.
--build -b	boolean	false	Builds this project and all of its dependencies specified by Project References . Note that this flag is not compatible with others on this page. See more here
--charset	string	"utf8"	The character set of the input files.
--checkJs	boolean	false	Report errors in .js files. Use in conjunction with --allowJs.
--declaration -d	boolean	false	Generates corresponding .d.ts file.
--declarationDir	string		Output directory for generated declaration files.
--declarationMap	boolean	false	Generates a sourcemap for each corresponding '.d.ts' file.
--diagnostics	boolean	false	Show diagnostic information.
--disableSizeLimit	boolean	false	Disable size limitation on JavaScript project.
--downlevelIteration	boolean	false	Provide full support for iterables in for..of, spread and destructuring when targeting ES5 or ES3.
--emitBOM	boolean	false	Emit a UTF-8 Byte Order Mark (BOM) in the beginning of output files.
--emitDeclarationOnly	boolean	false	Only emit '.d.ts' declaration files.
--emitDecoratorMetadata [1]	boolean	false	Emit design-type metadata for decorated declarations in source.

--emitDecoratorMetadata [1]			See issue #2577 for details.
--esModuleInterop	boolean	false	Emit <code>__importStar</code> and <code>__importDefault</code> helpers for runtime babel ecosystem compatibility and enable <code>--allowSyntheticDefaultImports</code> for typesystem compatibility.
--experimentalDecorators [1]	boolean	false	Enables experimental support for ES decorators.
--extendedDiagnostics	boolean	false	Show verbose diagnostic information
--forceConsistentCasingInFileNames	boolean	false	Disallow inconsistently-cased references to the same file.
--help -h			Print help message.
--importHelpers	boolean	false	Import emit helpers (e.g. <code>__extends</code> , <code>__rest</code> , etc..) from <code>tslib</code>
--inlineSourceMap	boolean	false	Emit a single file with source maps instead of having a separate file.
--inlineSources	boolean	false	Emit the source alongside the sourcemaps within a single file; requires <code>--inlineSourceMap</code> or <code>-sourceMap</code> to be set.
--init			Initializes a TypeScript project and creates a <code>tsconfig.json</code> file.
--isolatedModules	boolean	false	Transpile each file as a separate module (similar to "ts.transpileModule").
--jsx	string	"Preserve"	Support JSX in <code>.tsx</code> files: "React" or "Preserve". See JSX .
--jsxFactory	string	"React.createElement"	Specify the JSX factory function to use when targeting react JSX emit, e.g. <code>React.createElement</code> or <code>h</code> .
--keyofStringsOnly	boolean	false	Resolve <code>keyof</code> to string valued property names only (no numbers or symbols).
			List of library files to be included in the compilation. Possible values are: <ul style="list-style-type: none">► ES5► ES6► ES2015► ES7► ES2016► ES2017► ES2018► ESNext► DOM► DOM.Iterable► WebWorker► ScriptHost

--lib	string[]	<ul style="list-style-type: none"> ► ES2015.Collection ► ES2015.Generator ► ES2015.Iterable ► ES2015.Promise ► ES2015.Proxy ► ES2015.Reflect ► ES2015.Symbol ► ES2015.Symbol.WellKnown ► ES2016.Array.Include ► ES2017.Object ► ES2017.Intl ► ES2017.SharedMemory ► ES2017.String ► ES2017.TypedArrays ► ES2018.Intl ► ES2018.Promise ► ES2018.RegExp ► ESNext.AsyncIterable ► ESNext.Array ► ESNext.Intl ► ESNext.Symbol <p>Note: If <code>--lib</code> is not specified a default list of libraries are injected. The default libraries injected are:</p> <ul style="list-style-type: none"> ► For <code>--target ES5</code> : DOM, ES5, ScriptHost ► For <code>--target ES6</code> : DOM, ES6, DOM.Iterable, ScriptHost
--listEmittedFiles	boolean	false
--listFiles	boolean	false
--locale	string	(platform specific)
--mapRoot	string	Specifies the location where debugger should locate map files instead of generated locations. Use this flag if the .map files will be located at run-time in a different location than the .js files. The location specified will be embedded in the sourceMap to direct the debugger where the map files will be located.
--maxNodeModuleJsDepth	number	0

			applicable with <code>--allowJs</code> .
<code>--module</code> <code>-m</code>	string	<code>target === "ES3" or "ES5" ? "CommonJS" : "ES6"</code>	Specify module code generation: "None", "CommonJS", "AMD", "System", "UMD", "ES6", "ES2015" OR "ESNext". ► Only "AMD" and "System" can be used in conjunction with <code>--outFile</code> . ► "ES6" and "ES2015" values may be used when targeting "ES5" or lower.
<code>--moduleResolution</code>	string	<code>module === "AMD" or "System" or "ES6" ? "Classic" : "Node"</code>	Determine how modules get resolved. Either "Node" for Node.js/io.js style resolution, or "Classic". See Module Resolution documentation for more details.
<code>--newLine</code>	string	(platform specific)	Use the specified end of line sequence to be used when emitting files: "crlf" (windows) or "lf" (unix)."
<code>--noEmit</code>	boolean	false	Do not emit outputs.
<code>--noEmitHelpers</code>	boolean	false	Do not generate custom helper functions like <code>_extends</code> in compiled output.
<code>--noEmitOnError</code>	boolean	false	Do not emit outputs if any errors were reported.
<code>--noErrorTruncation</code>	boolean	false	Do not truncate error messages.
<code>--noFallthroughCasesInSwitch</code>	boolean	false	Report errors for fallthrough cases in switch statement.
<code>--noImplicitAny</code>	boolean	false	Raise error on expressions and declarations with an implied <code>any</code> type.
<code>--noImplicitReturns</code>	boolean	false	Report error when not all code paths in function return a value.
<code>--noImplicitThis</code>	boolean	false	Raise error on <code>this</code> expressions with an implied <code>any</code> type.
<code>--noImplicitUseStrict</code>	boolean	false	Do not emit "use strict" directives in module output.
<code>--noLib</code>	boolean	false	Do not include the default library file (<code>lib.d.ts</code>).
<code>--noResolve</code>	boolean	false	Do not add triple-slash references or module import targets to the list of compiled files.
<code>--noStrictGenericChecks</code>	boolean	false	Disable strict checking of generic signatures in function types.
<code>--noUnusedLocals</code>	boolean	false	Report errors on unused locals.
<code>--noUnusedParameters</code>	boolean	false	Report errors on unused parameters.
<code>--out</code>	string		DEPRECATED. Use <code>--outFile</code>

--outDir	string		Redirect output structure to the directory.
--outFile	string		Concatenate and emit output to single file. The order of concatenation is determined by the list of files passed to the compiler on the command line along with triple-slash references and imports. See output file order documentation for more details.
paths [2]	Object		List of path mapping entries for module names to locations relative to the <code>baseUrl</code> . See Module Resolution documentation for more details.
--preserveConstEnums	boolean	false	Do not erase const enum declarations in generated code. See const enums documentation for more details.
--preserveSymlinks	boolean	false	Do not resolve symlinks to their real path; treat a symlinked file like a real one.
--preserveWatchOutput	boolean	false	Keep outdated console output in watch mode instead of clearing the screen
--pretty	boolean	true unless piping to another program or redirecting output to a file	Stylize errors and messages using color and context.
--project -p	string		Compile a project given a valid configuration file. The argument can be a file path to a valid JSON configuration file, or a directory path to a directory containing a <code>tsconfig.json</code> file. See tsconfig.json documentation for more details.
--reactNamespace	string	"React"	DEPRECATED. Use <code>--jsxFactory</code> instead. Specifies the object invoked for <code>createElement</code> and <code>__spread</code> when targeting "react" JSX emit.
--removeComments	boolean	false	Remove all comments except copy-right header comments beginning with <code>/*!</code>
--resolveJsonModule	boolean	false	Include modules imported with <code>.json</code> extension.
--rootDir	string	(common root directory is computed from the list of input files)	Specifies the root directory of input files. Only use to control the output directory structure with <code>--outDir</code> .
rootDirs [2]	string[]		List of <code>root</code> folders whose combined content represent the structure of the project at runtime. See Module Resolution documentation for more details.

			documentation for more details.
--skipDefaultLibCheck	boolean	false	DEPRECATED. Use <code>--skipLibCheck</code> instead. Skip type checking of default library declaration files .
--skipLibCheck	boolean	false	Skip type checking of all declaration files (<code>*.d.ts</code>).
--sourceMap	boolean	false	Generates corresponding <code>.map</code> file.
--sourceRoot	string		Specifies the location where debugger should locate TypeScript files instead of source locations. Use this flag if the sources will be located at run-time in a different location than that at design-time. The location specified will be embedded in the sourceMap to direct the debugger where the source files will be located.
--strict	boolean	false	Enable all strict type checking options. Enabling <code>--strict</code> enables <code>--noImplicitAny</code> , <code>--noImplicitThis</code> , <code>--alwaysStrict</code> , <code>--strictNullChecks</code> , <code>--strictFunctionTypes</code> and <code>--strictPropertyInitialization</code> .
--strictFunctionTypes	boolean	false	Disable bivariant parameter checking for function types.
--strictPropertyInitialization	boolean	false	Ensure non-undefined class properties are initialized in the constructor. This option requires <code>--strictNullChecks</code> be enabled in order to take effect.
--strictNullChecks	boolean	false	In strict null checking mode, the <code>null</code> and <code>undefined</code> values are not in the domain of every type and are only assignable to themselves and <code>any</code> (the one exception being that <code>undefined</code> is also assignable to <code>void</code>).
--stripInternal [1]	boolean	false	Do not emit declarations for code that has an <code>/** @internal */</code> JSDoc annotation.
--suppressExcessPropertyErrors	boolean	false	Suppress excess property checks for object literals.
--suppressImplicitAnyIndexErrors	boolean	false	Suppress <code>--noImplicitAny</code> errors for indexing objects lacking index signatures. See issue #1232 for more details.
--target -t	string	"ES3"	Specify ECMAScript target version: "ES3" (default), "ES5", "ES6" / "ES2015", "ES2016", "ES2017" OR "ESNext". Note: "ESNext" targets latest

--traceResolution	boolean	false	Report module resolution log messages.
--types	string[]		List of names of type definitions to include. See @types , --typeRoots and --types for more details.
--typeRoots	string[]		List of folders to include type definitions from. See @types , --typeRoots and --types for more details.
--version -v			Print the compiler's version.
--watch -w			Run the compiler in watch mode. Watch input files and trigger recompilation on changes. The implementation of watching files and directories can be configured using environment variable. See configuring watch for more details.

- [1] These options are experimental.
- [2] These options are only allowed in `tsconfig.json`, and not through command-line switches.

Related

- Setting compiler options in `tsconfig.json` files.
- Setting compiler options in [MSBuild projects](#).

Build tools

- Browserify
- Duo
- Grunt
- Gulp
- Jspm
- Webpack
- MSBuild
- NuGet

Browserify

Install

```
npm install tsify
```

Using Command Line Interface

```
browserify main.ts -p [ tsify --noImplicitAny ] > bundle.js
```

Using API

```
var browserify = require("browserify");
var tsify = require("tsify");

browserify()
  .add("main.ts")
  .plugin("tsify", { noImplicitAny: true })
  .bundle()
  .pipe(process.stdout);
```

More details: [smrq/tsify](#)

Duo

Install

```
npm install duo-typescript
```

Using Command Line Interface

```
duo --use duo-typescript entry.ts
```

Using API

```
var Duo = require("duo");
var fs = require("fs")
```

```

var path = require("path")
var typescript = require("duo-typescript");

var out = path.join(__dirname, "output.js")

Duo(__dirname)
  .entry("entry.ts")
  .use(typescript())
  .run(function (err, results) {
    if (err) throw err;
    // Write compiled result to output file
    fs.writeFileSync(out, results.code);
  });

```

More details: [frankwallis/duo-typescript](#)

Grunt

Install

```
npm install grunt-ts
```

Basic Gruntfile.js

```

module.exports = function(grunt) {
  grunt.initConfig({
    ts: {
      default : {
        src: ["**/*.ts", "!node_modules/**/*.ts"]
      }
    }
  });
  grunt.loadNpmTasks("grunt-ts");
  grunt.registerTask("default", ["ts"]);
};

```

More details: [TypeStrong/grunt-ts](#)

Gulp

Install

```
npm install gulp-typescript
```

Basic gulpfile.js

```

var gulp = require("gulp");
var ts = require("gulp-typescript");

gulp.task("default", function () {
  var tsResult = gulp.src("src/*.ts")
    .pipe(ts({
      noImplicitAny: true,
      out: "output.js"
    }));
}

```

```
    return tsResult.js.pipe(gulp.dest("built/local"));
});
```

More details: [ivogabe/gulp-typescript](#)

Jspm

Install

```
npm install -g jspm@beta
```

Note: Currently TypeScript support in jspm is in 0.16beta

More details: [TypeScriptSamples/jspm](#)

Webpack

Install

```
npm install ts-loader --save-dev
```

Basic webpack.config.js

```
module.exports = {
  entry: "./src/index.tsx",
  output: {
    filename: "bundle.js"
  },
  resolve: {
    // Add '.ts' and '.tsx' as a resolvable extension.
    extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
  },
  module: {
    loaders: [
      // all files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'
      { test: /\.tsx?$/, loader: "ts-loader" }
    ]
  }
}
```

See [more details on ts-loader here](#).

Alternatives:

- [awesome-typescript-loader](#)

MSBuild

Update project file to include locally installed `microsoft.TypeScript.Default.props` (at the top) and `Microsoft.TypeScript.targets` (at the bottom) files:

```
<?xml version="1.0" encoding="utf-8"?>
```

```

<Project ToolsVersion="4.0" DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <!-- Include default props at the top -->
    <Import
        Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.Default.props"
        Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.Default.props')"/>

    <!-- TypeScript configurations go here -->
    <PropertyGroup Condition="'$(Configuration)' == 'Debug'>
        <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
        <TypeScriptSourceMap>true</TypeScriptSourceMap>
    </PropertyGroup>
    <PropertyGroup Condition="'$(Configuration)' == 'Release'>
        <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
        <TypeScriptSourceMap>false</TypeScriptSourceMap>
    </PropertyGroup>

    <!-- Include default targets at the bottom -->
    <Import
        Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.targets"
        Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.targets')"/>
</Project>

```

More details about defining MSBuild compiler options: [Setting Compiler Options in MSBuild projects](#)

NuGet

- Right-Click -> Manage NuGet Packages
- Search for `Microsoft.TypeScript.MSBuild`
- Hit `Install`
- When install is complete, rebuild!

More details can be found at [Package Manager Dialog](#) and [using nightly builds with NuGet](#)

A nightly build from the [TypeScript's master branch](#) is published by midnight PST to NPM and NuGet. Here is how you can get it and use it with your tools.

Using npm

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

Using NuGet with MSBuild

Note: You'll need to configure your project to use the NuGet packages. Please see [Configuring MSBuild projects to use NuGet](#) for more information.

The nightlies are available on www.myget.org.

There are two packages:

- `Microsoft.TypeScript.Compiler` : Tools only (`tsc.exe`, `lib.d.ts`, etc.).
- `Microsoft.TypeScript.MSBuild` : Tools as above, as well as MSBuild tasks and targets (`Microsoft.TypeScript.targets`, `Microsoft.TypeScript.Default.props`, etc.)

Updating your IDE to use the nightly builds

You can also update your IDE to use the nightly drop. First you will need to install the package through npm. You can either install the npm package globally or to a local `node_modules` folder.

The rest of this section assumes `typescript@next` is already installed.

Visual Studio Code

Update `.vscode/settings.json` with the following:

```
"typescript.tsdk": "<path to your folder>/node_modules/typescript/lib"
```

More information is available at [VSCode documentation](#).

Sublime Text

Update the `settings - User` file with the following:

```
"typescript_tsdk": "<path to your folder>/node_modules/typescript/lib"
```

More information is available at the [TypeScript Plugin for Sublime Text installation documentation](#).

Visual Studio 2013 and 2015

Note: Most changes do not require you to install a new version of the VS TypeScript plugin.

The nightly build currently does not include the full plugin setup, but we are working on publishing an installer on a nightly basis as well.

1. Download the [VSDevMode.ps1](#) script.

Also see our wiki page on [using a custom language service file](#).

2. From a PowerShell command window, run:

For VS 2015:

```
VSDevMode.ps1 14 -tsScript <path to your folder>/node_modules/typescript/lib
```

For VS 2013:

```
VSDevMode.ps1 12 -tsScript <path to your folder>/node_modules/typescript/lib
```

IntelliJ IDEA (Mac)

Go to `Preferences > Languages & Frameworks > TypeScript`:

TypeScript Version: If you installed with npm: `/usr/local/lib/node_modules/typescript/lib`

Mapped types on tuples and arrays

In TypeScript 3.1, mapped object types^[1] over tuples and arrays now produce new tuples/arrays, rather than creating a new type where members like `push()`, `pop()`, and `length` are converted. For example:

```
type MapToPromise<T> = { [K in keyof T]: Promise<T[K]> };

type Coordinate = [number, number]

type PromiseCoordinate = MapToPromise<Coordinate>; // [Promise<number>, Promise<number>]
```

`MapToPromise` takes a type `T`, and when that type is a tuple like `Coordinate`, only the numeric properties are converted. In `[number, number]`, there are two numerically named properties: `0` and `1`. When given a tuple like that, `MapToPromise` will create a new tuple where the `0` and `1` properties are `Promise`s of the original type. So the resulting type `PromiseCoordinate` ends up with the type `[Promise<number>, Promise<number>]`.

Properties declarations on functions

TypeScript 3.1 brings the ability to define properties on function declarations and `const`-declared functions, simply by assigning to properties on these functions in the same scope. This allows us to write canonical JavaScript code without resorting to `namespace` hacks. For example:

```
function readImage(path: string, callback: (err: any, image: Image) => void) {
    // ...
}

readImage.sync = (path: string) => {
    const contents = fs.readFileSync(path);
    return decodeImageSync(contents);
}
```

Here, we have a function `readImage` which reads an image in a non-blocking asynchronous way. In addition to `readImage`, we've provided a convenience function on `readImage` itself called `readImage.sync`.

While ECMAScript exports are often a better way of providing this functionality, this new support allows code written in this style to "just work" TypeScript. Additionally, this approach for property declarations allows us to express common patterns like `defaultProps` and `propTypes` on React stateless function components (SFCs).

```
export const FooComponent => ({ name }) => (
    <div>Hello! I am {name}</div>
);

FooComponent.defaultProps = {
    name: "(anonymous)",
};
```

[1] More specifically, homomorphic mapped types like in the above form.

Version selection with `typesVersions`

Feedback from our community, as well as our own experience, has shown us that leveraging the newest TypeScript features while also accomodating users on the older versions are difficult. TypeScript introduces a new feature called `typesVersions` to help accomodate these scenarios.

When using Node module resolution in TypeScript 3.1, when TypeScript cracks open a `package.json` file to figure out which files it needs to read, it first looks at a new field called `typesVersions`. A `package.json` with a `typesVersions` field might look like this:

```
{
  "name": "package-name",
  "version": "1.0",
  "types": "./index.d.ts",
  "typesVersions": {
    ">=3.1": { "*": ["ts3.1/*"] }
  }
}
```

This `package.json` tells TypeScript to check whether the current version of TypeScript is running. If it's 3.1 or later, it figures out the path you've imported relative to the package, and reads from the package's `ts3.1` folder. That's what that `{ "*": ["ts3.1/*"] }` means - if you're familiar with path mapping today, it works exactly like that.

So in the above example, if we're importing from `"package-name"`, we'll try to resolve from `[...]/node_modules/package-name/ts3.1/index.d.ts` (and other relevant paths) when running in TypeScript 3.1. If we import from `package-name/foo`, we'll try to look for `[...]/node_modules/package-name/ts3.1/foo.d.ts` and `[...]/node_modules/package-name/ts3.1/foo/index.d.ts`.

What if we're not running in TypeScript 3.1 in this example? Well, if none of the fields in `typesVersions` get matched, TypeScript falls back to the `types` field, so here TypeScript 3.0 and earlier will be redirected to `[...]/node_modules/package-name/index.d.ts`.

Matching behavior

The way that TypeScript decides on whether a version of the compiler & language matches is by using Node's [semver ranges](#).

Multiple fields

`typesVersions` can support multiple fields where each field name is specified by the range to match on.

```
{
  "name": "package-name",
  "version": "1.0",
  "types": "./index.d.ts",
  "typesVersions": {
    ">=3.2": { "*": ["ts3.2/*"] },
    ">=3.1": { "*": ["ts3.1/*"] }
  }
}
```

Since ranges have the potential to overlap, determining which redirect applies is order-specific. That means in the above example, even though both the `>=3.2` and the `>=3.1` matchers support TypeScript 3.2 and above, reversing the order could have different behavior, so the above sample would not be equivalent to the following.

```
{
  "name": "package-name",
```

```
"version": "1.0",
"types": "./index.d.ts",
"typesVersions": {
  // NOTE: this doesn't work!
  ">=3.1": { "*": ["ts3.1/*"] },
  ">=3.2": { "*": ["ts3.2/*"] }
}
```

Project References

TypeScript 3.0 introduces a new concept of project references. Project references allow TypeScript projects to depend on other TypeScript projects - specifically, allowing `tsconfig.json` files to reference other `tsconfig.json` files.

Specifying these dependencies makes it easier to split your code into smaller projects, since it gives TypeScript (and tools around it) a way to understand build ordering and output structure.

TypeScript 3.0 introduces also introducing a new mode for tsc, the `--build` flag, that works hand-in-hand with project references to enable faster TypeScript builds.

See [Project References handbook page](#) for more documentation.

Tuples in rest parameters and spread expressions

TypeScript 3.0 adds support to multiple new capabilities to interact with function parameter lists as tuple types.

TypeScript 3.0 adds support for:

- Expansion of rest parameters with tuple types into discrete parameters.
- Expansion of spread expressions with tuple types into discrete arguments.
- Generic rest parameters and corresponding inference of tuple types.
- Optional elements in tuple types.
- Rest elements in tuple types.

With these features it becomes possible to strongly type a number of higher-order functions that transform functions and their parameter lists.

Rest parameters with tuple types

When a rest parameter has a tuple type, the tuple type is expanded into a sequence of discrete parameters. For example the following two declarations are equivalent:

```
declare function foo(...args: [number, string, boolean]): void;  
  
declare function foo(args_0: number, args_1: string, args_2: boolean): void;
```

Spread expressions with tuple types

When a function call includes a spread expression of a tuple type as the last argument, the spread expression corresponds to a sequence of discrete arguments of the tuple element types.

Thus, the following calls are equivalent:

```
const args: [number, string, boolean] = [42, "hello", true];  
foo(42, "hello", true);  
foo(args[0], args[1], args[2]);  
foo(...args);
```

Generic rest parameters

A rest parameter is permitted to have a generic type that is constrained to an array type, and type inference can infer tuple types for such generic rest parameters. This enables higher-order capturing and spreading of partial parameter lists:

Example

```
declare function bind<T, U extends any[], V>(f: (x: T, ...args: U) => V, x: T): (...args: U) => V;

declare function f3(x: number, y: string, z: boolean): void;

const f2 = bind(f3, 42); // (y: string, z: boolean) => void
const f1 = bind(f2, "hello"); // (z: boolean) => void
const f0 = bind(f1, true); // () => void

f3(42, "hello", true);
f2("hello", true);
f1(true);
f0();
```

In the declaration of `f2` above, type inference infers types `number`, `[string, boolean]` and `void` for `T`, `U` and `V` respectively.

Note that when a tuple type is inferred from a sequence of parameters and later expanded into a parameter list, as is the case for `U`, the original parameter names are used in the expansion (however, the names have no semantic meaning and are not otherwise observable).

Optional elements in tuple types

Tuple types now permit a `?` postfix on element types to indicate that the element is optional:

Example

```
let t: [number, string?, boolean?];
t = [42, "hello", true];
t = [42, "hello"];
t = [42];
```

In `--strictNullChecks` mode, a `?` modifier automatically includes `undefined` in the element type, similar to optional parameters.

A tuple type permits an element to be omitted if it has a postfix `?` modifier on its type and all elements to the right of it also have `?` modifiers.

When tuple types are inferred for rest parameters, optional parameters in the source become optional tuple elements in the inferred type.

The `length` property of a tuple type with optional elements is a union of numeric literal types representing the possible lengths. For example, the type of the `length` property in the tuple type `[number, string?, boolean?]` is `1 | 2 | 3`.

Rest elements in tuple types

The last element of a tuple type can be a rest element of the form `...x`, where `x` is an array type. A rest element indicates that the tuple type is open-ended and may have zero or more additional elements of the array element type. For example, `[number, ...string[]]` means tuples with a `number` element followed by any number of `string` elements.

Example

```
function tuple<T extends any[]>(...args: T): T {
    return args;
}

const numbers: number[] = getArrayOfNumbers();
const t1 = tuple("foo", 1, true); // [string, number, boolean]
const t2 = tuple("bar", ...numbers); // [string, ...number[]]
```

The type of the `length` property of a tuple type with a rest element is `number`.

New unknown top type

TypeScript 3.0 introduces a new top type `unknown`. `unknown` is the type-safe counterpart of `any`. Anything is assignable to `unknown`, but `unknown` isn't assignable to anything but itself and `any` without a type assertion or a control flow based narrowing. Likewise, no operations are permitted on an `unknown` without first asserting or narrowing to a more specific type.

Example

```
// In an intersection everything absorbs unknown

type T00 = unknown & null; // null
type T01 = unknown & undefined; // undefined
type T02 = unknown & null & undefined; // null & undefined (which becomes never)
type T03 = unknown & string; // string
type T04 = unknown & string[]; // string[]
type T05 = unknown & unknown; // unknown
type T06 = unknown & any; // any

// In a union an unknown absorbs everything

type T10 = unknown | null; // unknown
type T11 = unknown | undefined; // unknown
type T12 = unknown | null | undefined; // unknown
type T13 = unknown | string; // unknown
type T14 = unknown | string[]; // unknown
type T15 = unknown | unknown; // unknown
type T16 = unknown | any; // any

// Type variable and unknown in union and intersection

type T20<T> = T & {};
type T21<T> = T | {};
type T22<T> = T & unknown;
type T23<T> = T | unknown;

// unknown in conditional types

type T30<T> = unknown extends T ? true : false; // Deferred
type T31<T> = T extends unknown ? true : false; // Deferred (so it distributes)
type T32<T> = never extends T ? true : false; // true
type T33<T> = T extends never ? true : false; // Deferred

// keyof unknown

type T40 = keyof any; // string | number | symbol
type T41 = keyof unknown; // never

// Only equality operators are allowed with unknown
```

```

function f10(x: unknown) {
    x == 5;
    x != 10;
    x >= 0; // Error
    x + 1; // Error
    x * 2; // Error
    -x; // Error
    +x; // Error
}

// No property accesses, element accesses, or function calls

function f11(x: unknown) {
    x.foo; // Error
    x[5]; // Error
    x(); // Error
    new x(); // Error
}

// typeof, instanceof, and user defined type predicates

declare function isFunction(x: unknown): x is Function;

function f20(x: unknown) {
    if (typeof x === "string" || typeof x === "number") {
        x; // string | number
    }
    if (x instanceof Error) {
        x; // Error
    }
    if (isFunction(x)) {
        x; // Function
    }
}

// Homomorphic mapped type over unknown

type T50<T> = { [P in keyof T]: number };
type T51 = T50<any>; // { [x: string]: number }
type T52 = T50<unknown>; // {}

// Anything is assignable to unknown

function f21<T>(pAny: any, pNever: never, pT: T) {
    let x: unknown;
    x = 123;
    x = "hello";
    x = [1, 2, 3];
    x = new Error();
    x = x;
    x = pAny;
    x = pNever;
    x = pT;
}

// unknown assignable only to itself and any

function f22(x: unknown) {
    let v1: any = x;
    let v2: unknown = x;
    let v3: object = x; // Error
    let v4: string = x; // Error
    let v5: string[] = x; // Error
    let v6: {} = x; // Error
    let v7: {} | null | undefined = x; // Error
}

// Type parameter 'T' extends 'unknown' not related to object

```

```

function f23<T extends unknown>(x: T) {
    let y: object = x; // Error
}

// Anything but primitive assignable to { [x: string]: unknown }

function f24(x: { [x: string]: unknown }) {
    x = {};
    x = { a: 5 };
    x = [1, 2, 3];
    x = 123; // Error
}

// Locals of type unknown always considered initialized

function f25() {
    let x: unknown;
    let y = x;
}

// Spread of unknown causes result to be unknown

function f26(x: {}, y: unknown, z: any) {
    let o1 = { a: 42, ...x }; // { a: number }
    let o2 = { a: 42, ...x, ...y }; // unknown
    let o3 = { a: 42, ...x, ...y, ...z }; // any
}

// Functions with unknown return type don't need return expressions

function f27(): unknown {
}

// Rest type cannot be created from unknown

function f28(x: unknown) {
    let { ...a } = x; // Error
}

// Class properties of type unknown don't need definite assignment

class C1 {
    a: string; // Error
    b: unknown;
    c: any;
}

```

Support for `defaultProps` in JSX

TypeScript 2.9 and earlier didn't leverage `React.defaultProps` declarations inside JSX components. Users would often have to declare properties optional and use non-null assertions inside of `render`, or they'd use type-assertions to fix up the type of the component before exporting it.

TypeScript 3.0 adds supports a new type alias in the `jsx` namespace called `LibraryManagedAttributes`. This helper type defines a transformation on the component's `Props` type, before using to check a JSX expression targeting it; thus allowing customization like: how conflicts between provided props and inferred props are handled, how inferences are mapped, how optionality is handled, and how inferences from differing places should be combined.

In short using this general type, we can model React's specific behavior for things like `defaultProps` and, to some extent, `propTypes`.

```

export interface Props {
    name: string;
}

```

```

}

export class Greet extends React.Component<Props> {
    render() {
        const { name } = this.props;
        return <div>Hello ${name.toUpperCase()}!</div>;
    }
    static defaultProps = { name: "world" };
}

// Type-checks! No type assertions needed!
let el = <Greet />

```

Caveats

Explicit types on `defaultProps`

The default-ed properties are inferred from the `defaultProps` property type. If an explicit type annotation is added, e.g. `static defaultProps: Partial<Props>;` the compiler will not be able to identify which properties have defaults (since the type of `defaultProps` include all properties of `Props`).

Use `static defaultProps: Pick<Props, "name">;` as an explicit type annotation instead, or do not add a type annotation as done in the example above.

For stateless function components (SFCs) use ES2015 default initializers for SFCs:

```

function Greet({ name = "world" }: Props) {
    return <div>Hello ${name.toUpperCase()}!</div>;
}

```

Changes to `@types/React`

Corresponding changes to add `LibraryManagedAttributes` definition to the `jsx` namespace in `@types/React` are still needed. Keep in mind that there are some limitations.

`/// <reference lib="..." />` reference directives

TypeScript adds a new triple-slash-reference directive (`/// <reference lib="name" />`), allowing a file to explicitly include an existing built-in `/lib` file.

Built-in `/lib` files are referenced in the same fashion as the `"lib"` compiler option in `tsconfig.json` (e.g. use `lib="es2015"` and not `lib="lib.es2015.d.ts"`, etc.).

For declaration file authors who rely on built-in types, e.g. DOM APIs or built-in JS run-time constructors like `Symbol` or `Iterable`, triple-slash-reference lib directives are the recommended. Previously these `.d.ts` files had to add forward/duplicate declarations of such types.

Example

Using `/// <reference lib="es2017.string" />` to one of the files in a compilation is equivalent to compiling with `--lib es2017.string`.

```

/// <reference lib="es2017.string" />

"foo".padStart(4);

```


Support number and symbol named properties with keyof and mapped types

TypeScript 2.9 adds support for `number` and `symbol` named properties in index types and mapped types. Previously, the `keyof` operator and mapped types only supported `string` named properties.

Changes include:

- An index type `keyof T` for some type `T` is a subtype of `string | number | symbol`.
- A mapped type `{ [P in K]: xxx }` permits any `K` assignable to `string | number | symbol`.
- In a `for...in` statement for an object of a generic type `T`, the inferred type of the iteration variable was previously `keyof T` but is now `Extract<keyof T, string>`. (In other words, the subset of `keyof T` that includes only string-like values.)

Given an object type `x`, `keyof x` is resolved as follows:

- If `x` contains a string index signature, `keyof x` is a union of `string`, `number`, and the literal types representing symbol-like properties, otherwise
- If `x` contains a numeric index signature, `keyof x` is a union of `number` and the literal types representing string-like and symbol-like properties, otherwise
- `keyof x` is a union of the literal types representing string-like, number-like, and symbol-like properties.

Where:

- String-like properties of an object type are those declared using an identifier, a string literal, or a computed property name of a string literal type.
- Number-like properties of an object type are those declared using a numeric literal or computed property name of a numeric literal type.
- Symbol-like properties of an object type are those declared using a computed property name of a unique symbol type.

In a mapped type `{ [P in K]: xxx }`, each string literal type in `K` introduces a property with a string name, each numeric literal type in `K` introduces a property with a numeric name, and each unique symbol type in `K` introduces a property with a unique symbol name. Furthermore, if `K` includes type `string`, a string index signature is introduced, and if `K` includes type `number`, a numeric index signature is introduced.

Example

```
const c = "c";
const d = 10;
const e = Symbol();

const enum E1 { A, B, C }
const enum E2 { A = "A", B = "B", C = "C" }

type Foo = {
    a: string;           // String-like name
    5: string;          // Number-like name
    [c]: string;        // String-like name
    [d]: string;        // Number-like name
    [e]: string;        // Symbol-like name
    [E1.A]: string;    // Number-like name
    [E2.A]: string;    // String-like name
}

type K1 = keyof Foo;  // "a" | 5 | "c" | 10 | typeof e | E1.A | E2.A
type K2 = Extract<keyof Foo, string>; // "a" | "c" | E2.A
```

```
type K3 = Extract<keyof Foo, number>; // 5 | 10 | E1.A
type K4 = Extract<keyof Foo, symbol>; // typeof e
```

Since `keyof` now reflects the presence of a numeric index signature by including type `number` in the key type, mapped types such as `Partial<T>` and `Readonly<T>` work correctly when applied to object types with numeric index signatures:

```
type Arrayish<T> = {
    length: number;
    [x: number]: T;
}

type ReadonlyArrayish<T> = Readonly<Arrayish<T>>;

declare const map: ReadonlyArrayish<string>;
let n = map.length;
let x = map[123]; // Previously of type any (or an error with --noImplicitAny)
```

Furthermore, with the `keyof` operator's support for `number` and `symbol` named keys, it is now possible to abstract over access to properties of objects that are indexed by numeric literals (such as numeric enum types) and unique symbols.

```
const enum Enum { A, B, C }

const enumToStringMap = {
    [Enum.A]: "Name A",
    [Enum.B]: "Name B",
    [Enum.C]: "Name C"
}

const sym1 = Symbol();
const sym2 = Symbol();
const sym3 = Symbol();

const symbolToNumberMap = {
    [sym1]: 1,
    [sym2]: 2,
    [sym3]: 3
};

type KE = keyof typeof enumToStringMap; // Enum (i.e. Enum.A | Enum.B | Enum.C)
type KS = keyof typeof symbolToNumberMap; // typeof sym1 | typeof sym2 | typeof sym3

function getValue<T, K extends keyof T>(obj: T, key: K): T[K] {
    return obj[key];
}

let x1 = getValue(enumToStringMap, Enum.C); // Returns "Name C"
let x2 = getValue(symbolToNumberMap, sym3); // Returns 3
```

This is a breaking change; previously, the `keyof` operator and mapped types only supported `string` named properties. Code that assumed values typed with `keyof T` were always `string`s, will now be flagged as error.

Example

```
function useKey<T, K extends keyof T>(o: T, k: K) {
    var name: string = k; // Error: keyof T is not assignable to string
}
```

Recommendations

- If your functions are only able to handle string named property keys, use `Extract<keyof T, string>` in the declaration:

```
function useKey<T, K extends Extract<keyof T, string>>(o: T, k: K) {
  var name: string = k; // OK
}
```

- If your functions are open to handling all property keys, then the changes should be done down-stream:

```
function useKey<T, K extends keyof T>(o: T, k: K) {
  var name: string | number | symbol = k;
}
```

- Otherwise use `--keyofStringsOnly` compiler option to disable the new behavior.

Generic type arguments in JSX elements

JSX elements now allow passing type arguments to generic components.

Example

```
class GenericComponent<P> extends React.Component<P> {
  internalProp: P;
}

type Props = { a: number; b: string; };

const x = <GenericComponent<Props> a={10} b="hi"/>; // OK

const y = <GenericComponent<Props> a={10} b={20} />; // Error
```

Generic type arguments in generic tagged templates

Tagged templates are a form of invocation introduced in ECMAScript 2015. Like call expressions, generic functions may be used in a tagged template and TypeScript will infer the type arguments utilized.

TypeScript 2.9 allows passing generic type arguments to tagged template strings.

Example

```
declare function styledComponent<Props>(strs: TemplateStringsArray): Component<Props>;

interface MyProps {
  name: string;
  age: number;
}

styledComponent<MyProps> `

  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;

declare function tag<T>(strs: TemplateStringsArray, ...args: T[]): T;

// inference fails because 'number' and 'string' are both candidates that conflict
let a = tag<string | number> `${100} ${"hello"}`;
```

import types

Modules can import types declared in other modules. But non-module global scripts cannot access types declared in modules. Enter `import` types.

Using `import("mod")` in a type annotation allows for reaching in a module and accessing its exported declaration without importing it.

Example

Given a declaration of a class `Pet` in a module file:

```
// module.d.ts

export declare class Pet {
    name: string;
}
```

Can be used in a non-module file `global-script.ts`:

```
// global-script.ts

function adopt(p: import("./module").Pet) {
    console.log(`Adopting ${p.name}...`);
}
```

This also works in JSDoc comments to refer to types from other modules in `.js`:

```
// a.js

/**
 * @param p { import("./module").Pet }
 */
function walk(p) {
    console.log(`Walking ${p.name}...`);
}
```

Relaxing declaration emit visibility rules

With `import` types available, many of the visibility errors reported during declaration file generation can be handled by the compiler without the need to change the input.

For instance:

```
import { createHash } from "crypto";

export const hash = createHash("sha256");
//           ^^^^
// Exported variable 'hash' has or is using name 'Hash' from external module "crypto" but cannot be named.
```

With TypeScript 2.9, no errors are reported, and now the generated file looks like:

```
export declare const hash: import("crypto").Hash;
```

Support for `import.meta`

TypeScript 2.9 introduces support for `import.meta`, a new meta-property as described by the current [TC39 proposal](#).

The type of `import.meta` is the global `ImportMeta` type which is defined in `lib.es5.d.ts`. This interface is extremely limited. Adding well-known properties for Node or browsers requires interface merging and possibly a global augmentation depending on the context.

Example

Assuming that `__dirname` is always available on `import.meta`, the declaration would be done through reopening `ImportMeta` interface:

```
// node.d.ts
interface ImportMeta {
    __dirname: string;
}
```

And usage would be:

```
import.meta.__dirname // Has type 'string'
```

`import.meta` is only allowed when targeting `ESNext` modules and ECMAScript targets.

New `--resolveJsonModule`

Often in Node.js applications a `.json` is needed. With TypeScript 2.9, `--resolveJsonModule` allows for importing, extracting types from and generating `.json` files.

Example

```
// settings.json

{
    "repo": "TypeScript",
    "dry": false,
    "debug": false
}

// a.ts

import settings from "./settings.json";

settings.debug === true; // OK
settings.dry === 2; // Error: Operator '===' cannot be applied boolean and number
```

```
// tsconfig.json

{
    "compilerOptions": {
        "module": "commonjs",
        "resolveJsonModule": true,
        "esModuleInterop": true
    }
}
```

--pretty output by default

Starting TypeScript 2.9 errors are displayed under `--pretty` by default if the output device is applicable for colorful text. TypeScript will check if the output stream has `isTty` property set.

Use `--pretty false` on the command line or set `"pretty": false` in your `tsconfig.json` to disable `--pretty` output.

New --declarationMap

Enabling `--declarationMap` alongside `--declaration` causes the compiler to emit `.d.ts.map` files alongside the output `.d.ts` files. Language Services can also now understand these map files, and uses them to map declaration-file based definition locations to their original source, when available.

In other words, hitting go-to-definition on a declaration from a `.d.ts` file generated with `--declarationMap` will take you to the source file (`.ts`) location where that declaration was defined, and not to the `.d.ts`.

Conditional Types

TypeScript 2.8 introduces *conditional types* which add the ability to express non-uniform type mappings. A conditional type selects one of two possible types based on a condition expressed as a type relationship test:

```
T extends U ? X : Y
```

The type above means when `T` is assignable to `U` the type is `X`, otherwise the type is `Y`.

A conditional type `T extends U ? X : Y` is either *resolved* to `X` or `Y`, or *deferred* because the condition depends on one or more type variables. Whether to resolve or defer is determined as follows:

- First, given types `T'` and `U'` that are instantiations of `T` and `U` where all occurrences of type parameters are replaced with `any`, if `T'` is not assignable to `U'`, the conditional type is resolved to `Y`. Intuitively, if the most permissive instantiation of `T` is not assignable to the most permissive instantiation of `U`, we know that no instantiation will be and we can just resolve to `Y`.
- Next, for each type variable introduced by an `infer` (more later) declaration within `U` collect a set of candidate types by inferring from `T` to `U` (using the same inference algorithm as type inference for generic functions). For a given `infer` type variable `v`, if any candidates were inferred from co-variant positions, the type inferred for `v` is a union of those candidates. Otherwise, if any candidates were inferred from contra-variant positions, the type inferred for `v` is an intersection of those candidates. Otherwise, the type inferred for `v` is `never`.
- Then, given a type `T''` that is an instantiation of `T` where all `infer` type variables are replaced with the types inferred in the previous step, if `T''` is *definitely assignable* to `U`, the conditional type is resolved to `X`. The definitely assignable relation is the same as the regular assignable relation, except that type variable constraints are not considered. Intuitively, when a type is definitely assignable to another type, we know that it will be assignable for *all instantiations* of those types.
- Otherwise, the condition depends on one or more type variables and the conditional type is deferred.

Example

```
type TypeName<T> =
  T extends string ? "string" :
  T extends number ? "number" :
  T extends boolean ? "boolean" :
  T extends undefined ? "undefined" :
  T extends Function ? "function" :
  "object";

type T0 = TypeName<string>; // "string"
type T1 = TypeName<"a">; // "string"
type T2 = TypeName<true>; // "boolean"
type T3 = TypeName<() => void>; // "function"
type T4 = TypeName<string[]>; // "object"
```

Distributive conditional types

Conditional types in which the checked type is a naked type parameter are called *distributive conditional types*. Distributive conditional types are automatically distributed over union types during instantiation. For example, an instantiation of `T extends U ? X : Y` with the type argument `A | B | C` for `T` is resolved as `(A extends U ? X : Y) | (B extends U ? X : Y) | (C extends U ? X : Y)`.

Example

```

type T10 = TypeName<string | (() => void)>; // "string" | "function"
type T12 = TypeName<string | string[] | undefined>; // "string" | "object" | "undefined"
type T11 = TypeName<string[] | number[]>; // "object"

```

In instantiations of a distributive conditional type `T extends U ? X : Y`, references to `T` within the conditional type are resolved to individual constituents of the union type (i.e. `T` refers to the individual constituents *after* the conditional type is distributed over the union type). Furthermore, references to `T` within `X` have an additional type parameter constraint `U` (i.e. `T` is considered assignable to `U` within `X`).

Example

```

type BoxedValue<T> = { value: T };
type BoxedArray<T> = { array: T[] };
type Boxed<T> = T extends any[] ? BoxedArray<T[number]> : BoxedValue<T>

type T20 = Boxed<string>; // BoxedValue<string>
type T21 = Boxed<number[]>; // BoxedArray<number>;
type T22 = Boxed<string | number[]>; // BoxedValue<string> | BoxedArray<number>;

```

Notice that `T` has the additional constraint `any[]` within the true branch of `Boxed<T>` and it is therefore possible to refer to the element type of the array as `T[number]`. Also, notice how the conditional type is distributed over the union type in the last example.

The distributive property of conditional types can conveniently be used to *filter* union types:

```

type Diff<T, U> = T extends U ? never : T; // Remove types from T that are assignable to U
type Filter<T, U> = T extends U ? T : never; // Remove types from T that are not assignable to U

type T30 = Diff<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "b" | "d"
type T31 = Filter<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "a" | "c"
type T32 = Diff<string | number | () => void, Function>; // string | number
type T33 = Filter<string | number | () => void, Function>; // () => void

type NonNullable<T> = Diff<T, null | undefined>; // Remove null and undefined from T

type T34 = NonNullable<string | number | undefined>; // string | number
type T35 = NonNullable<string | string[] | null | undefined>; // string | string[]

function f1<T>(x: T, y: NonNullable<T>) {
    x = y; // Ok
    y = x; // Error
}

function f2<T extends string | undefined>(x: T, y: NonNullable<T>) {
    x = y; // Ok
    y = x; // Error
    let s1: string = x; // Error
    let s2: string = y; // Ok
}

```

Conditional types are particularly useful when combined with mapped types:

```

type FunctionPropertyNames<T> = { [K in keyof T]: T[K] extends Function ? K : never }[keyof T];
type FunctionProperties<T> = Pick<T, FunctionPropertyNames<T>>;

type NonFunctionPropertyNames<T> = { [K in keyof T]: T[K] extends Function ? never : K }[keyof T];
type NonFunctionProperties<T> = Pick<T, NonFunctionPropertyNames<T>>;

interface Part {
    id: number;
    name: string;
    subparts: Part[];
}

```

```

        updatePart(newName: string): void;
    }

type T40 = FunctionPropertyNames<Part>; // "updatePart"
type T41 = NonFunctionPropertyNames<Part>; // "id" | "name" | "subparts"
type T42 = FunctionProperties<Part>; // { updatePart(newName: string): void }
type T43 = NonFunctionProperties<Part>; // { id: number, name: string, subparts: Part[] }

```

Similar to union and intersection types, conditional types are not permitted to reference themselves recursively. For example the following is an error.

Example

```
type ElementType<T> = T extends any[] ? ElementType<T[number]> : T; // Error
```

Type inference in conditional types

Within the `extends` clause of a conditional type, it is now possible to have `infer` declarations that introduce a type variable to be inferred. Such inferred type variables may be referenced in the true branch of the conditional type. It is possible to have multiple `infer` locations for the same type variable.

For example, the following extracts the return type of a function type:

```
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;
```

Conditional types can be nested to form a sequence of pattern matches that are evaluated in order:

```

type Unpacked<T> =
    T extends (infer U)[] ? U :
    T extends (...args: any[]) => infer U ? U :
    T extends Promise<infer U> ? U :
    T;

type T0 = Unpacked<string>; // string
type T1 = Unpacked<string[]>; // string
type T2 = Unpacked<() => string>; // string
type T3 = Unpacked<Promise<string>>; // string
type T4 = Unpacked<Promise<string>[]>; // Promise<string>
type T5 = Unpacked<Unpacked<Promise<string>[]>; // string

```

The following example demonstrates how multiple candidates for the same type variable in co-variant positions causes a union type to be inferred:

```

type Foo<T> = T extends { a: infer U, b: infer U } ? U : never;
type T10 = Foo<{ a: string, b: string }>; // string
type T11 = Foo<{ a: string, b: number }>; // string | number

```

Likewise, multiple candidates for the same type variable in contra-variant positions causes an intersection type to be inferred:

```

type Bar<T> = T extends { a: (x: infer U) => void, b: (x: infer U) => void } ? U : never;
type T20 = Bar<{ a: (x: string) => void, b: (x: string) => void }>; // string
type T21 = Bar<{ a: (x: string) => void, b: (x: number) => void }>; // string & number

```

When inferring from a type with multiple call signatures (such as the type of an overloaded function), inferences are made from the *last* signature (which, presumably, is the most permissive catch-all case). It is not possible to perform overload resolution based on a list of argument types.

```
declare function foo(x: string): number;
declare function foo(x: number): string;
declare function foo(x: string | number): string | number;
type T30 = ReturnType<typeof foo>; // string | number
```

It is not possible to use `infer` declarations in constraint clauses for regular type parameters:

```
type ReturnType<T extends (...args: any[]) => infer R> = R; // Error, not supported
```

However, much the same effect can be obtained by erasing the type variables in the constraint and instead specifying a conditional type:

```
type AnyFunction = (...args: any[]) => any;
type ReturnType<T extends AnyFunction> = T extends (...args: any[]) => infer R ? R : any;
```

Predefined conditional types

TypeScript 2.8 adds several predefined conditional types to `lib.d.ts`:

- `Exclude<T, U>` -- Exclude from `T` those types that are assignable to `U`.
- `Extract<T, U>` -- Extract from `T` those types that are assignable to `U`.
- `NonNullable<T>` -- Exclude `null` and `undefined` from `T`.
- `ReturnType<T>` -- Obtain the return type of a function type.
- `InstanceType<T>` -- Obtain the instance type of a constructor function type.

Example

```
type T00 = Exclude<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "b" | "d"
type T01 = Extract<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "a" | "c"

type T02 = Exclude<string | number | (() => void), Function>; // string | number
type T03 = Extract<string | number | (() => void), Function>; // () => void

type T04 = NonNullable<string | number | undefined>; // string | number
type T05 = NonNullable<(() => string) | string[] | null | undefined>; // () => string | string[]

function f1(s: string) {
    return { a: 1, b: s };
}

class C {
    x = 0;
    y = 0;
}

type T10 = ReturnType<() => string>; // string
type T11 = ReturnType<(s: string) => void>; // void
type T12 = ReturnType<(<T>() => T)>; // {}
type T13 = ReturnType<(<T extends U, U extends number[]>() => T)>; // number[]
type T14 = ReturnType<typeof f1>; // { a: number, b: string }
type T15 = ReturnType<any>; // any
type T16 = ReturnType<never>; // any
type T17 = ReturnType<string>; // Error
type T18 = ReturnType<Function>; // Error
```

```
type T20 = InstanceType<typeof C>; // C
type T21 = InstanceType<any>; // any
type T22 = InstanceType<never>; // any
type T23 = InstanceType<string>; // Error
type T24 = InstanceType<Function>; // Error
```

Note: The `Exclude` type is a proper implementation of the `Diff` type suggested [here](#). We've used the name `Exclude` to avoid breaking existing code that defines a `Diff`, plus we feel that name better conveys the semantics of the type. We did not include the `Omit<T, K>` type because it is trivially written as `Pick<T, Exclude<keyof T, K>>`.

Improved control over mapped type modifiers

Mapped types support adding a `readonly` or `?` modifier to a mapped property, but they did not provide support the ability to *remove* modifiers. This matters in [homomorphic mapped types](#) which by default preserve the modifiers of the underlying type.

TypeScript 2.8 adds the ability for a mapped type to either add or remove a particular modifier. Specifically, a `readonly` or `?` property modifier in a mapped type can now be prefixed with either `+` or `-` to indicate that the modifier should be added or removed.

Example

```
type MutableRequired<T> = { -readonly [P in keyof T]-?: T[P] }; // Remove readonly and ?
type ReadonlyPartial<T> = { +readonly [P in keyof T]+?: T[P] }; // Add readonly and ?
```

A modifier with no `+` or `-` prefix is the same as a modifier with a `+` prefix. So, the `ReadonlyPartial<T>` type above corresponds to

```
type ReadonlyPartial<T> = { readonly [P in keyof T]?: T[P] }; // Add readonly and ?
```

Using this ability, `lib.d.ts` now has a new `Required<T>` type. This type strips `?` modifiers from all properties of `T`, thus making all properties required.

Example

```
type Required<T> = { [P in keyof T]-?: T[P] };
```

Note that in `--strictNullChecks` mode, when a homomorphic mapped type removes a `?` modifier from a property in the underlying type it also removes `undefined` from the type of that property:

Example

```
type Foo = { a?: string }; // Same as { a?: string | undefined }
type Bar = Required<Foo>; // Same as { a: string }
```

Improved `keyof` with intersection types

With TypeScript 2.8 `keyof` applied to an intersection type is transformed to a union of `keyof` applied to each intersection constituent. In other words, types of the form `keyof (A & B)` are transformed to be `keyof A | keyof B`. This change should address inconsistencies with inference from `keyof` expressions.

Example

```
type A = { a: string };
type B = { b: string };

type T1 = keyof (A & B); // "a" | "b"
type T2<T> = keyof (T & B); // keyof T | "b"
type T3<U> = keyof (A & U); // "a" | keyof U
type T4<T, U> = keyof (T & U); // keyof T | keyof U
type T5 = T2<A>; // "a" | "b"
type T6 = T3<B>; // "a" | "b"
type T7 = T4<A, B>; // "a" | "b"
```

Better handling for namespace patterns in `.js` files

TypeScript 2.8 adds support for understanding more namespace patterns in `.js` files. Empty object literals declarations on top level, just like functions and classes, are now recognized as as namespace declarations in JavaScript.

```
var ns = {};  
ns.constant = 1;
```

Assignments at the top-level should behave the same way; in other words, a `var` or `const` declaration is not required.

```
app = {};  
app.C = class {};  
app.f = function() {};  
app.prop = 1;
```

IIFEs as namespace declarations

An IIFE returning a function, class or empty object literal, is also recognized as a namespace:

```
var C = (function () {  
    function C(n) {  
        this.p = n;  
    }  
    return C;  
})();  
C.staticProperty = 1;
```

Defaulted declarations

"Defaulted declarations" allow initializers that reference the declared name in the left side of a logical or:

```
my = window.my || {};
```

```
my.app = my.app || {};
```

Prototype assignment

You can assign an object literal directly to the prototype property. Individual prototype assignments still work too:

```
var C = function (p) {
    this.p = p;
};
C.prototype = {
    m() {
        console.log(this.p);
    }
};
C.prototype.q = function(r) {
    return this.p === r;
};
```

Nested and merged declarations

Nesting works to any level now, and merges correctly across files. Previously neither was the case.

```
var app = window.app || {};
app.C = class {};
```

Per-file JSX factories

TypeScript 2.8 adds support for a per-file configurable JSX factory name using `@jsx dom` pragma. JSX factory can be configured for a compilation using `--jsxFactory` (default is `React.createElement`). With TypeScript 2.8 you can override this on a per-file-basis by adding a comment to the beginning of the file.

Example

```
/** @jsx dom */
import { dom } from "./renderer"
<h></h>
```

Generates:

```
var renderer_1 = require("./renderer");
renderer_1.dom("h", null);
```

Locally scoped JSX namespaces

JSX type checking is driven by definitions in a JSX namespace, for instance `JSX.Element` for the type of a JSX element, and `JSX.IntrinsicElements` for built-in elements. Before TypeScript 2.8 the `JSX` namespace was expected to be in the global namespace, and thus only allowing one to be defined in a project. Starting with TypeScript 2.8 the `JSX` namespace will be looked under the `jsxNamespace` (e.g. `React`) allowing for multiple `jsx` factories in one compilation. For backward compatibility the global `JSX` namespace is used as a fallback if none was defined on the factory function. Combined with the per-file `@jsx` pragma, each file can have a different JSX factory.

New `--emitDeclarationOnly`

`--emitDeclarationOnly` allows for *only* generating declaration files; `.js` / `.jsx` output generation will be skipped with this flag. The flag is useful when the `.js` output generation is handled by a different transpiler like Babel.

Constant-named properties

TypeScript 2.7 adds support for declaring const-named properties on types including ECMAScript symbols.

Example

```
// Lib
export const SERIALIZED = Symbol("serialize-method-key");

export interface Serializable {
  [SERIALIZED](obj: {}): string;
}

// consumer

import { SERIALIZED, Serializable } from "lib";

class JSONSerializableItem implements Serializable {
  [SERIALIZED](obj: {}) {
    return JSON.stringify(obj);
  }
}
```

This also applies to numeric and string literals.

Example

```
const Foo = "Foo";
const Bar = "Bar";

let x = {
  [Foo]: 100,
  [Bar]: "hello",
};

let a = x[Bar]; // has type 'string'
let b = x[Bar]; // has type 'string'
```

unique symbol

To enable treating symbols as unique literals a new type `unique symbol` is available. `unique symbol` is a subtype of `symbol`, and are produced only from calling `Symbol()` or `Symbol.for()`, or from explicit type annotations. The new type is only allowed on `const` declarations and `readonly static` properties, and in order to reference a specific unique symbol, you'll have to use the `typeof` operator. Each reference to a `unique symbol` implies a completely unique identity that's tied to a given declaration.

Example

```
// Works
declare const Foo: unique symbol;

// Error! 'Bar' isn't a constant.
let Bar: unique symbol = Symbol();

// Works - refers to a unique symbol, but its identity is tied to 'Foo'.
let Baz: typeof Foo = Foo;
```

```
// Also works.
class C {
    static readonly StaticSymbol: unique symbol = Symbol();
}
```

Because each `unique symbol` has a completely separate identity, no two `unique symbol` types are assignable or comparable to each other.

Example

```
const Foo = Symbol();
const Bar = Symbol();

// Error: can't compare two unique symbols.
if (Foo === Bar) {
    // ...
}
```

Strict Class Initialization

TypeScript 2.7 introduces a new flag called `--strictPropertyInitialization`. This flag performs checks to ensure that each instance property of a class gets initialized in the constructor body, or by a property initializer. For example

```
class C {
    foo: number;
    bar = "hello";
    baz: boolean;
    // ~~~
    // Error! Property 'baz' has no initializer and is not definitely assigned in the
    // constructor.

    constructor() {
        this.foo = 42;
    }
}
```

In the above, if we truly meant for `baz` to potentially be `undefined`, we should have declared it with the type `boolean | undefined`.

There are certain scenarios where properties can be initialized indirectly (perhaps by a helper method or dependency injection library), in which case you can use the new *definite assignment assertion modifiers* for your properties (discussed below).

```
class C {
    foo!: number;
    // ^
    // Notice this '!' modifier.
    // This is the "definite assignment assertion"

    constructor() {
        this.initialize();
    }

    initialize() {
        this.foo = 0;
    }
}
```

Keep in mind that `--strictPropertyInitialization` will be turned on along with other `--strict` mode flags, which can impact your project. You can set the `strictPropertyInitialization` setting to `false` in your `tsconfig.json`'s `compilerOptions`, or `--strictPropertyInitialization false` on the command line to turn off this checking.

Definite Assignment Assertions

The definite assignment assertion is a feature that allows a `!` to be placed after instance property and variable declarations to relay to TypeScript that a variable is indeed assigned for all intents and purposes, even if TypeScript's analyses cannot detect so.

Example

```
let x: number;
initialize();
console.log(x + x);
//      ~   ~
// Error! Variable 'x' is used before being assigned.

function initialize() {
    x = 10;
}
```

With definite assignment assertions, we can assert that `x` is really assigned by appending an `!` to its declaration:

```
// Notice the '!'
let x!: number;
initialize();

// No error!
console.log(x + x);

function initialize() {
    x = 10;
}
```

In a sense, the definite assignment assertion operator is the dual of the non-null assertion operator (in which *expressions* are post-fixed with a `!`), which we could also have used in the example.

```
let x: number;
initialize();

// No error!
console.log(x! + x!);

function initialize() {
    x = 10;
```

In our example, we knew that all uses of `x` would be initialized so it makes more sense to use definite assignment assertions than non-null assertions.

Fixed Length Tuples

In TypeScript 2.6 and earlier, `[number, string, string]` was considered a subtype of `[number, string]`. This was motivated by TypeScript's structural nature; the first and second elements of a `[number, string, string]` are respectively subtypes of the first and second elements of `[number, string]`. However, after examining real world

usage of tuples, we noticed that most situations in which this was permitted was typically undesirable.

In TypeScript 2.7, tuples of different arities are no longer assignable to each other. Thanks to a pull request from [Tycho Grouwstra](#), tuple types now encode their arity into the type of their respective `length` property. This is accomplished by leveraging numeric literal types, which now allow tuples to be distinct from tuples of different arities.

Conceptually, you might consider the type `[number, string]` to be equivalent to the following declaration of

`NumStrTuple`:

```
interface NumStrTuple extends Array<number | string> {
  0: number;
  1: string;
  length: 2; // using the numeric literal type '2'
}
```

Note that this is a breaking change for some code. If you need to resort to the original behavior in which tuples only enforce a minimum length, you can use a similar declaration that does not explicitly define a `length` property, falling back to `number`.

```
interface MinimumNumStrTuple extends Array<number | string> {
  0: number;
  1: string;
}
```

Note that this does not imply tuples represent immutable arrays, but it is an implied convention.

Improved type inference for object literals

TypeScript 2.7 improves type inference for multiple object literals occurring in the same context. When multiple object literal types contribute to a union type, we now *normalize* the object literal types such that all properties are present in each constituent of the union type.

Consider:

```
const obj = test ? { text: "hello" } : {};
// { text: string } | { text?: undefined }

const s = obj.text; // string | undefined
```

Previously type `{}` was inferred for `obj` and the second line subsequently caused an error because `obj` would appear to have no properties. That obviously wasn't ideal.

Example

```
// let obj: { a: number, b: number } |
//   { a: string, b?: undefined } |
//   { a?: undefined, b?: undefined }
let obj = [{ a: 1, b: 2 }, { a: "abc" }, {}][0];
obj.a; // string | number | undefined
obj.b; // number | undefined
```

Multiple object literal type inferences for the same type parameter are similarly collapsed into a single normalized union type:

```
declare function f<T>(...items: T[]): T;
let obj: { a: number, b: number } |
  { a: string, b?: undefined } |
```

```
//      { a?: undefined, b?: undefined }
let obj = f({ a: 1, b: 2 }, { a: "abc" }, {});
obj.a; // string | number | undefined
obj.b; // number | undefined
```

Improved handling of structurally identical classes and `instanceof` expressions

TypeScript 2.7 improves the handling of structurally identical classes in union types and `instanceof` expressions:

- Structurally identical, but distinct, class types are now preserved in union types (instead of eliminating all but one).
- Union type subtype reduction only removes a class type if it is a subclass of *and* derives from another class type in the union.
- Type checking of the `instanceof` operator is now based on whether the type of the left operand *derives from* the type indicated by the right operand (as opposed to a structural subtype check).

This means that union types and `instanceof` properly distinguish between structurally identical classes.

Example

```
class A {}
class B extends A {}
class C extends A {}
class D extends A { c: string }
class E extends D {}

let x1 = !true ? new A() : new B(); // A
let x2 = !true ? new B() : new C(); // B | C (previously B)
let x3 = !true ? new C() : new D(); // C | D (previously C)

let a1 = [new A(), new B(), new C(), new D(), new E()]; // A[]
let a2 = [new B(), new C(), new D(), new E()]; // (B | C | D)[] (previously B[])

function f1(x: B | C | D) {
    if (x instanceof B) {
        x; // B (previously B | D)
    }
    else if (x instanceof C) {
        x; // C
    }
    else {
        x; // D (previously never)
    }
}
```

Type guards inferred from `in` operator

The `in` operator now acts as a narrowing expression for types.

For a `n in x` expression, where `n` is a string literal or string literal type and `x` is a union type, the "true" branch narrows to types which have an optional or required property `n`, and the "false" branch narrows to types which have an optional or missing property `n`.

Example

```

interface A { a: number };
interface B { b: string };

function foo(x: A | B) {
    if ("a" in x) {
        return x.a;
    }
    return x.b;
}

```

Support for `import d from "cjs"` form CommonJS modules with `--esModuleInterop`

TypeScript 2.7 updates CommonJS/AMD/UMD module emit to synthesize namespace records based on the presence of an `__esModule` indicator under `--esModuleInterop`. The change brings the generated output from TypeScript closer to that generated by Babel.

Previously CommonJS/AMD/UMD modules were treated in the same way as ES6 modules, resulting in a couple of problems. Namely:

- TypeScript treats a namespace import (i.e. `import * as foo from "foo"`) for a CommonJS/AMD/UMD module as equivalent to `const foo = require("foo")`. Things are simple here, but they don't work out if the primary object being imported is a primitive or a class or a function. ECMAScript spec stipulates that a namespace record is a plain object, and that a namespace import (`foo` in the example above) is not callable, though allowed by TypeScript
- Similarly a default import (i.e. `import d from "foo"`) for a CommonJS/AMD/UMD module as equivalent to `const d = require("foo").default`. Most of the CommonJS/AMD/UMD modules available today do not have a `default` export, making this import pattern practically unusable to import non-ES modules (i.e. CommonJS/AMD/UMD). For instance `import fs from "fs"` or `import express from "express"` are not allowed.

Under the new `--esModuleInterop` these two issues should be addressed:

- A namespace import (i.e. `import * as foo from "foo"`) is now correctly flagged as uncallable. Calling it will result in an error.
- Default imports to CommonJS/AMD/UMD are now allowed (e.g. `import fs from "fs"`), and should work as expected.

Note: The new behavior is added under a flag to avoid unwarranted breaks to existing code bases. We highly recommend applying it both to new and existing projects. For existing projects, namespace imports (`import * as express from "express"; express();`) will need to be converted to default imports (`import express from "express"; express();`).

Example

With `--esModuleInterop` two new helpers are generated `__importDefault` and `__importStar` for `import *` and `import default` respectively. For instance input like:

```

import * as foo from "foo";
import b from "bar";

```

Will generate:

```

"use strict";
var __importDefault = (this && this.__importDefault) || function (mod) {

```

```
if (mod && mod.__esModule) return mod;
var result = {};
if (mod != null) for (var k in mod) if (Object.hasOwnProperty.call(mod, k)) result[k] = mod[k];
result["default"] = mod;
return result;
}
var __importDefault = (this && this.__importDefault) || function (mod) {
  return (mod && mod.__esModule) ? mod : { "default": mod };
}
exports.__esModule = true;
var foo = __importStar(require("foo"));
var bar_1 = __importDefault(require("bar"));
```

Numeric separators

TypeScript 2.7 brings support for [ES Numeric Separators](#). Numeric literals can now be separated into segments using `_`.

Example

```
const milion = 1_000_000;
const phone = 555_734_2231;
const bytes = 0xFF_0C_00_FF;
const word = 0b1100_0011_1101_0001;
```

Cleaner output in `--watch` mode

TypeScript's `--watch` mode now clears the screen after a re-compilation is requested.

Prettier `--pretty` output

TypeScript's `--pretty` flag can make error messages easier to read and manage. `--pretty` now uses colors for file names, diagnostic codes, and line numbers. File names and positions are now also formatted to allow navigation in common terminals (e.g. Visual Studio Code terminal).

Strict function types

TypeScript 2.6 introduces a new strict checking flag, `--strictFunctionTypes`. The `--strictFunctionTypes` switch is part of the `--strict` family of switches, meaning that it defaults to on in `--strict` mode. You can opt-out by setting `--strictFunctionTypes false` on your command line or in your `tsconfig.json`.

Under `--strictFunctionTypes` function type parameter positions are checked *contravariantly* instead of *bivariantly*. For some background on what variance means for function types check out [What are covariance and contravariance?](#).

The stricter checking applies to all function types, *except* those originating in method or constructor declarations. Methods are excluded specifically to ensure generic classes and interfaces (such as `Array<T>`) continue to mostly relate covariantly.

Consider the following example in which `Animal` is the supertype of `Dog` and `Cat`:

```
declare let f1: (x: Animal) => void;
declare let f2: (x: Dog) => void;
declare let f3: (x: Cat) => void;
f1 = f2; // Error with --strictFunctionTypes
f2 = f1; // Ok
f2 = f3; // Error
```

The first assignment is permitted in default type checking mode, but flagged as an error in strict function types mode. Intuitively, the default mode permits the assignment because it is *possibly* sound, whereas strict function types mode makes it an error because it isn't *provable* sound. In either mode the third assignment is an error because it is *never* sound.

Another way to describe the example is that the type `(x: T) => void` is *bivariant* (i.e. covariant or contravariant) for `T` in default type checking mode, but *contravariant* for `T` in strict function types mode.

Example

```
interface Comparer<T> {
    compare: (a: T, b: T) => number;
}

declare let animalComparer: Comparer<Animal>;
declare let dogComparer: Comparer<Dog>;

animalComparer = dogComparer; // Error
dogComparer = animalComparer; // Ok
```

The first assignment is now an error. Effectively, `T` is contravariant in `comparer<T>` because it is used only in function type parameter positions.

By the way, note that whereas some languages (e.g. C# and Scala) require variance annotations (`out / in` or `+ / -`), variance emerges naturally from the actual use of a type parameter within a generic type due to TypeScript's structural type system.

Note

Under `--strictFunctionTypes` the first assignment is still permitted if `compare` was declared as a method. Effectively, `T` is bivariant in `comparer<T>` because it is used only in method parameter positions.

```
interface Comparer<T> {
```

```

    compare(a: T, b: T): number;
}

declare let animalComparer: Comparer<Animal>;
declare let dogComparer: Comparer<Dog>;

animalComparer = dogComparer; // Ok because of bivariance
dogComparer = animalComparer; // Ok

```

TypeScript 2.6 also improves type inference involving contravariant positions:

```

function combine<T>(...funcs: ((x: T) => void)[]): (x: T) => void {
    return x => {
        for (const f of funcs) f(x);
    }
}

function animalFunc(x: Animal) {}
function dogFunc(x: Dog) {}

let combined = combine(animalFunc, dogFunc); // (x: Dog) => void

```

Above, all inferences for `T` originate in contravariant positions, and we therefore infer the *best common subtype* for `T`. This contrasts with inferences from covariant positions, where we infer the *best common supertype*.

Cache tagged template objects in modules

TypeScript 2.6 fixes the tagged string template emit to align better with the ECMAScript spec. As per the [ECMAScript spec](#), every time a template tag is evaluated, the *same* template strings object (the same `TemplateStringsArray`) should be passed as the first argument. Before TypeScript 2.6, the generated output was a completely new template object each time. Though the string contents are the same, this emit affects libraries that use the identity of the string for cache invalidation purposes, e.g. [lit-html](#).

Example

```

export function id(x: TemplateStringsArray) {
    return x;
}

export function templateObjectFactory() {
    return id`hello world`;
}

let result = templateObjectFactory() === templateObjectFactory(); // true in TS 2.6

```

Results in the following generated code:

```

"use strict";
var __makeTemplateObject = (this && this.__makeTemplateObject) || function (cooked, raw) {
    if (Object.defineProperty) { Object.defineProperty(cooked, "raw", { value: raw }); } else { cooked.raw = raw; }
    return cooked;
};

function id(x) {
    return x;
}

var _a;

```

```

function templateObjectFactory() {
    return id(_a) || (_a = __makeTemplateObject(["hello world"], ["hello world"])));
}

var result = templateObjectFactory() === templateObjectFactory();

```

Note: This change brings a new emit helper, `__makeTemplateObject`; if you are using `--importHelpers` with `tslib`, an update to version 1.8 or later.

Localized diagnostics on the command line

TypeScript 2.6 npm package ships with localized versions of diagnostic messages for 13 languages. The localized messages are available when using `--locale` flag on the command line.

Example

Error messages in Russian:

```

c:\ts>tsc --v
Version 2.6.0-dev.20171003

c:\ts>tsc --locale ru --pretty c:\test\a.ts

..../test/a.ts(1,5): error TS2322: Тип ""string"" не может быть назначен для типа "number".

1 var x: number = "string";
  ~

```

And help in Japanese:

```

PS C:\ts> tsc --v
Version 2.6.0-dev.20171003

PS C:\ts> tsc --locale ja-jp
バージョン 2.6.0-dev.20171003
構文: tsc [オプション] [ファイル ...]

例: tsc hello.ts
    tsc --outFile file.js file.ts
    tsc @args.txt

オプション:
-h, --help                                このメッセージを表示します。
--all                                         コンパイラ オプションをすべて表示します。
-v, --version                               コンパイラのバージョンを表示します。
--init                                       TypeScript プロジェクトを初期化して、tsconfig.json ファイルを作成します。
-p ファイルまたはディレクトリ, --project ファイルまたはディレクトリ 構成ファイルか、'tsconfig.json' を含むフォルダーにパスが
指定されたプロジェクトをコ
ンパイルします。
--pretty                                     色とコンテキストを使用してエラーとメッセージにスタイルを適用します（試験的）。
-w, --watch                                   入力ファイルを監視します。
-t バージョン, --target バージョン          ECMAScript のターゲット バージョンを指定します: 'ES3' (既定)、'ES5'、
'ES2015'、'ES2016'、'ES2017'、'ES
NEXT'。
-m 種類, --module 種類                      モジュール コード生成を指定します: 'none'、'commonjs'、'amd'、'system'、
'umd'、'es2015'、'ESNext'。
--lib                                         コンパイルに含めるライブラリ ファイルを指定します:
                                                'es5' 'es6' 'es2015' 'es7' 'es2016' 'es2017' 'esnext' 'dom' 'dom.
iterable' 'webworker' 'scripthost' 'es201
5.core' 'es2015.collection' 'es2015.generator' 'es2015.iterable' 'es2015.promise' 'es2015.proxy' 'es2015.reflec
t' 'es2015.symbol' 'es2015.symbol.wellkno
wn' 'es2016.array.include' 'es2017.object' 'es2017.sharedmemory' 'es2017.string' 'es2017.intl' 'esnext.asyncite

```

```


|                              |                                                     |
|------------------------------|-----------------------------------------------------|
| --target                     | javascript ファイルのコンパイルを許可します。                        |
| --allowJs                    | JSX コード生成を指定します: 'preserve'、'react-native'、'react'。 |
| --jsx 種類                     | 対応する '.d.ts' ファイルを生成します。                            |
| -d, --declaration            | 対応する '.map' ファイルを生成します。                             |
| --sourceMap                  | 出力を連結して 1 つのファイルを生成します。                             |
| --outFile ファイル               | ディレクトリへ出力構造をリダイレクトします。                              |
| --outDir ディレクトリ              | コメントを出力しないでください。                                    |
| --removeComments             | 出力しないでください。                                         |
| --noEmit                     | strict 型チェックのオプションをすべて有効にします。                       |
| --strict                     | 暗黙的な 'any' 型を含む式と宣言に関するエラーを発生させます。                  |
| --noImplicitAny              | 厳格な null チェックを有効にします。                               |
| --strictNullChecks           | 暗黙的な 'any' 型を持つ 'this' 式でエラーが発生します。                 |
| --noImplicitThis             | 厳格モードで解析してソース ファイルごとに "use strict" を生成します。          |
| --alwaysStrict               | 使用されていないローカルに関するエラーを報告します。                          |
| --noUnusedLocals             | 使用されていないパラメーターに関するエラーを報告します。                        |
| --noUnusedParameters         | 関数の一部のコード パスが値を返さない場合にエラーを報告します。                    |
| --noImplicitReturns          | switch ステートメントに case のフォールスルーがある場合にエラーを報告します。       |
| --noFallthroughCasesInSwitch | コンパイルに含む型宣言ファイル。                                    |
| --types                      |                                                     |
| @<ファイル>                      |                                                     |


```

Suppress errors in .ts files using '>// @ts-ignore' comments

TypeScript 2.6 support suppressing errors in .js files using `// @ts-ignore` comments placed above the offending lines.

Example

```

if (false) {
  // @ts-ignore: Unreachable code error
  console.log("hello");
}

```

A `// @ts-ignore` comment suppresses all errors that originate on the following line. It is recommended practice to have the remainder of the comment following `@ts-ignore` explain which error is being suppressed.

Please note that this comment only suppresses the error reporting, and we recommend you use this comments *very sparingly*.

Faster `tsc --watch`

TypeScript 2.6 brings a faster `--watch` implementation. The new version optimizes code generation and checking for code bases using ES modules. Changes detected in a module file will result in *only* regenerating the changed module, and files that depend on it, instead of the whole project. Projects with large number of files should reap the most benefit from this change.

The new implementation also brings performance enhancements to watching in tsserver. The watcher logic has been completely rewritten to respond faster to change events.

Write-only references now flagged as unused

TypeScript 2.6 adds revised implementation the `--noUnusedLocals` and `--noUnusedParameters` compiler options. Declarations are only written to but never read from are now flagged as unused.

Example

Bellow both `n` and `m` will be marked as unused, because their values are never *read*. Previously TypeScript would only check whether their values were *referenced*.

```
function f(n: number) {
    n = 0;
}

class C {
    private m: number;
    constructor() {
        this.m = 0;
    }
}
```

Also functions that are only called within their own bodies are considered unused.

Example

```
function f() {
    f(); // Error: 'f' is declared but its value is never read
}
```

Optional catch clause variables

Thanks to work done by [@tinganho](#), TypeScript 2.5 implements a new ECMAScript feature that allows users to omit the variable in `catch` clauses. For example, when using `JSON.parse` you may need to wrap calls to the function with a `try / catch`, but you may not end up using the `SyntaxError` that gets thrown when input is erroneous.

```
let input = "...";
try {
    JSON.parse(input);
}
catch {
    // ^ Notice that our `catch` clause doesn't declare a variable.
    console.log("Invalid JSON given\n\n" + input)
}
```

Type assertion/cast syntax in `checkJs` / `@ts-check` mode

TypeScript 2.5 introduces the ability to [assert the type of expressions when using plain JavaScript in your projects](#). The syntax is an `/** @type {...} */` annotation comment followed by a parenthesized expression whose type needs to be re-evaluated. For example:

```
var x = /** @type {SomeType} */ (AnyParenthesizedExpression);
```

Deduplicated and redirected packages

When importing using the `Node` module resolution strategy in TypeScript 2.5, the compiler will now check whether files originate from "identical" packages. If a file originates from a package with a `package.json` containing the same `name` and `version` fields as a previously encountered package, then TypeScript will redirect itself to the top-most package. This helps resolve problems where two packages might contain identical declarations of classes, but which contain `private` members that cause them to be structurally incompatible.

As a nice bonus, this can also reduce the memory and runtime footprint of the compiler and language service by avoiding loading `.d.ts` files from duplicate packages.

The `--preserveSymlinks` compiler flag

TypeScript 2.5 brings the `preserveSymlinks` flag, which parallels the behavior of the [--preserve-symlinks flag in Node.js](#). This flag also exhibits the opposite behavior to Webpack's `resolve.symlinks` option (i.e. setting TypeScript's `preserveSymlinks` to `true` parallels setting Webpack's `resolve.symlinks` to `false`, and vice-versa).

In this mode, references to modules and packages (e.g. `import s` and `/// <reference type="..." />` directives) are all resolved relative to the location of the symbolic link file, rather than relative to the path that the symbolic link resolves to. For a more concrete example, we'll defer to [the documentation on the Node.js website](#).

TypeScript 2.4

Dynamic Import Expressions

Dynamic `import` expressions are a new feature and part of ECMAScript that allows users to asynchronously request a module at any arbitrary point in your program.

This means that you can conditionally and lazily import other modules and libraries. For example, here's an `async` function that only imports a utility library when it's needed:

```
async function getZipFile(name: string, files: File[]): Promise<File> {
    const zipUtil = await import('./utils/create-zip-file');
    const zipContents = await zipUtil.getContentAsBlob(files);
    return new File(zipContents, name);
}
```

Many bundlers have support for automatically splitting output bundles based on these `import` expressions, so consider using this new feature with the `esnext` module target.

String Enums

TypeScript 2.4 now allows enum members to contain string initializers.

```
enum Colors {
    Red = "RED",
    Green = "GREEN",
    Blue = "BLUE",
}
```

The caveat is that string-initialized enums can't be reverse-mapped to get the original enum member name. In other words, you can't write `Colors["RED"]` to get the string `"Red"`.

Improved inference for generics

TypeScript 2.4 introduces a few wonderful changes around the way generics are inferred.

Return types as inference targets

For one, TypeScript can now make inferences for the return type of a call. This can improve your experience and catch errors. Something that now works:

```
function arrayMap<T, U>(f: (x: T) => U): (a: T[]) => U[] {
    return a => a.map(f);
}

const lengths: (a: string[]) => number[] = arrayMap(s => s.length);
```

As an example of new errors you might spot as a result:

```
let x: Promise<string> = new Promise(resolve => {
    resolve(10);
```

```
//      ~~ Error!
});
```

Type parameter inference from contextual types

Prior to TypeScript 2.4, in the following example

```
let f: <T>(x: T) => T = y => y;
```

`y` would have the type `any`. This meant the program would type-check, but you could technically do anything with `y`, such as the following:

```
let f: <T>(x: T) => T = y => y() + y.foo.bar;
```

That last example isn't actually type-safe.

In TypeScript 2.4, the function on the right side implicitly *gains* type parameters, and `y` is inferred to have the type of that type-parameter.

If you use `y` in a way that the type parameter's constraint doesn't support, you'll correctly get an error. In this case, the constraint of `T` was (implicitly) `{}`, so the last example will appropriately fail.

Stricter checking for generic functions

TypeScript now tries to unify type parameters when comparing two single-signature types. As a result, you'll get stricter checks when relating two generic signatures, and may catch some bugs.

```
type A = <T, U>(x: T, y: U) => [T, U];
type B = <S>(x: S, y: S) => [S, S];

function f(a: A, b: B) {
    a = b; // Error
    b = a; // Ok
}
```

Strict contravariance for callback parameters

TypeScript has always compared parameters in a bivariant way. There are a number of reasons for this, but by-and-large this was not been a huge issue for our users until we saw some of the adverse effects it had with `Promise`s and `Observable`s.

TypeScript 2.4 introduces tightens this up when relating two callback types. For example:

```
interface Mappable<T> {
    map<U>(f: (x: T) => U): Mappable<U>;
}

declare let a: Mappable<number>;
declare let b: Mappable<string | number>;

a = b;
b = a;
```

Prior to TypeScript 2.4, this example would succeed. When relating the types of `map`, TypeScript would bidirectionally relate their parameters (i.e. the type of `f`). When relating each `f`, TypeScript would also bidirectionally relate the type of *those* parameters.

When relating the type of `map` in TS 2.4, the language will check whether each parameter is a callback type, and if so, it will ensure that those parameters are checked in a contravariant manner with respect to the current relation.

In other words, TypeScript now catches the above bug, which may be a breaking change for some users, but will largely be helpful.

Weak Type Detection

TypeScript 2.4 introduces the concept of "weak types". Any type that contains nothing but a set of all-optional properties is considered to be *weak*. For example, this `options` type is a weak type:

```
interface Options {
  data?: string,
  timeout?: number,
  maxRetries?: number,
}
```

In TypeScript 2.4, it's now an error to assign anything to a weak type when there's no overlap in properties. For example:

```
function sendMessage(options: Options) {
  // ...
}

const opts = {
  payload: "hello world!",
  retryOnFail: true,
}

// Error!
sendMessage(opts);
// No overlap between the type of 'opts' and 'Options' itself.
// Maybe we meant to use 'data'/'maxRetries' instead of 'payload'/'retryOnFail'.
```

You can think of this as TypeScript "toughening up" the weak guarantees of these types to catch what would otherwise be silent bugs.

Since this is a breaking change, you may need to know about the workarounds which are the same as those for strict object literal checks:

1. Declare the properties if they really do exist.
2. Add an index signature to the weak type (i.e. `[propName: string]: {}`).
3. Use a type assertion (i.e. `opts as Options`).

Generators and Iteration for ES5/ES3

First some ES2016 terminology:

Iterators

ES2015 introduced `Iterator`, which is an object that exposes three methods, `next`, `return`, and `throw`, as per the following interface:

```
interface Iterator<T> {
  next(value?: any): IteratorResult<T>;
  return?(value?: any): IteratorResult<T>;
  throw?(e?: any): IteratorResult<T>;
}
```

This kind of iterator is useful for iterating over synchronously available values, such as the elements of an Array or the keys of a Map. An object that supports iteration is said to be "iterable" if it has a `Symbol.iterator` method that returns an `Iterator` object.

The Iterator protocol also defines the target of some of the ES2015 features like `for..of` and spread operator and the array rest in destructuring assignments.

Generators

ES2015 also introduced "Generators", which are functions that can be used to yield partial computation results via the `Iterator` interface and the `yield` keyword. Generators can also internally delegate calls to another iterable through `yield *`. For example:

```
function* f() {
  yield 1;
  yield* [2, 3];
}
```

NEW `--downlevelIteration`

Previously generators were only supported if the target is ES6/ES2015 or later. Moreover, constructs that operate on the Iterator protocol, e.g. `for..of` were only supported if they operate on arrays for targets below ES6/ES2015.

TypeScript 2.3 adds full support for generators and the Iterator protocol for ES3 and ES5 targets with `--downlevelIteration` flag.

With `--downlevelIteration`, the compiler uses new type check and emit behavior that attempts to call a `[Symbol.iterator]()` method on the iterated object if it is found, and creates a synthetic array iterator over the object if it is not.

Please note that this requires a native `Symbol.iterator` or `Symbol.iterator` shim at runtime for any non-array values.

`for..of` statements, Array Destructuring, and Spread elements in Array, Call, and New expressions support `Symbol.iterator` in ES5/E3 if available when using `--downlevelIteration`, but can be used on an Array even if it does not define `Symbol.iterator` at run time or design time.

Async Iteration

TypeScript 2.3 adds support for the `async` iterators and generators as described by the current [TC39 proposal](#).

Async iterators

The `Async Iteration` introduces an `AsyncIterator`, which is similar to `Iterator`. The difference lies in the fact that the `next`, `return`, and `throw` methods of an `AsyncIterator` return a `Promise` for the iteration result, rather than the result itself. This allows the caller to enlist in an asynchronous notification for the time at which the `AsyncIterator` has advanced to the point of yielding a value. An `AsyncIterator` has the following shape:

```
interface AsyncIterator<T> {
    next(value?: any): Promise<IteratorResult<T>>;
    return?(value?: any): Promise<IteratorResult<T>>;
    throw?(e?: any): Promise<IteratorResult<T>>;
}
```

An object that supports `async` iteration is said to be "iterable" if it has a `Symbol.asyncIterator` method that returns an `AsyncIterator` object.

Async Generators

The [Async Iteration proposal](#) introduces "Async Generators", which are `async` functions that also can be used to yield partial computation results. Async Generators can also delegate calls via `yield*` to either an iterable or `async` iterable:

```
async function* g() {
    yield 1;
    await sleep(100);
    yield* [2, 3];
    yield* (async function* () {
        await sleep(100);
        yield 4;
    })();
}
```

As with Generators, Async Generators can only be function declarations, function expressions, or methods of classes or object literals. Arrow functions cannot be Async Generators. Async Generators require a valid, global `Promise` implementation (either native or an ES2015-compatible polyfill), in addition to a valid `Symbol.asyncIterator` reference (either a native symbol or a shim).

The `for..await..of` Statement

Finally, ES2015 introduced the `for..of` statement as a means of iterating over an iterable. Similarly, the `Async Iteration` proposal introduces the `for..await..of` statement to iterate over an `async` iterable:

```
async function f() {
    for await (const x of g()) {
        console.log(x);
    }
}
```

The `for..await..of` statement is only legal within an `Async Function` or `Async Generator`.

Caveats

- Keep in mind that our support for `async` iterators relies on support for `Symbol.asyncIterator` to exist at runtime. You may need to polyfill `Symbol.asyncIterator`, which for simple purposes can be as simple as: `(Symbol as any).asyncIterator = Symbol.asyncIterator || Symbol.for("Symbol.asyncIterator");`
- You also need to include `esnext` in your `--lib` option, to get the `AsyncIterator` declaration if you do not

already have it.

- Finally, if your target is ES5 or ES3, you'll also need to set the `--downlevelIterators` flag.

Generic parameter defaults

TypeScript 2.3 adds support for declaring defaults for generic type parameters.

Example

Consider a function that creates a new `HTMLElement`, calling it with no arguments generates a `Div`; you can optionally pass a list of children as well. Previously you would have to define it as:

```
declare function create(): Container<HTMLDivElement, HTMLDivElement[]>;
declare function create<T extends HTMLElement>(element: T): Container<T, T[]>;
declare function create<T extends HTMLElement, U extends HTMLElement>(element: T, children: U[]): Container<T, U[]>;
```

With generic parameter defaults we can reduce it to:

```
declare function create<T extends HTMLElement = HTMLDivElement, U = T[]>(element?: T, children?: U): Container<T, U>;
```

A generic parameter default follows the following rules:

- A type parameter is deemed optional if it has a default.
- Required type parameters must not follow optional type parameters.
- Default types for a type parameter must satisfy the constraint for the type parameter, if it exists.
- When specifying type arguments, you are only required to specify type arguments for the required type parameters. Unspecified type parameters will resolve to their default types.
- If a default type is specified and inference cannot choose a candidate, the default type is inferred.
- A class or interface declaration that merges with an existing class or interface declaration may introduce a default for an existing type parameter.
- A class or interface declaration that merges with an existing class or interface declaration may introduce a new type parameter as long as it specifies a default.

New `--strict` master option

New checks added to TypeScript are often off by default to avoid breaking existing projects. While avoiding breakage is a good thing, this strategy has the drawback of making it increasingly complex to choose the highest level of type safety, and doing so requires explicit opt-in action on every TypeScript release. With the `--strict` option it becomes possible to choose maximum type safety with the understanding that additional errors might be reported by newer versions of the compiler as improved type checking features are added.

The new `--strict` compiler option represents the recommended setting of a number of type checking options. Specifically, specifying `--strict` corresponds to specifying all of the following options (and may in the future include more options):

- `--strictNullChecks`
- `--noImplicitAny`
- `--noImplicitThis`
- `--alwaysStrict`

In exact terms, the `--strict` option sets the *default* value for the compiler options listed above. This means it is still possible to individually control the options. For example,

```
--strict --noImplicitThis false
```

has the effect of turning on all strict options *except* the `--noImplicitThis` option. Using this scheme it is possible to express configurations consisting of *all* strict options except some explicitly listed options. In other words, it is now possible to default to the highest level of type safety but opt out of certain checks.

Starting with TypeScript 2.3, the default `tsconfig.json` generated by `tsc --init` includes a `"strict": true` setting in the `"compilerOptions"` section. Thus, new projects started with `tsc --init` will by default have the highest level of type safety enabled.

Enhanced `--init` output

Along with setting `--strict` on by default, `tsc --init` has an enhanced output. Default `tsconfig.json` files generated by `tsc --init` now include a set of the common compiler options along with their descriptions commented out. Just un-comment the configuration you like to set to get the desired behavior; we hope the new output simplifies the setting up new projects and keeps configuration files readable as projects grow.

Errors in `.js` files with `--checkJs`

By default the TypeScript compiler does not report any errors in `.js` files including using `--allowJs`. With TypeScript 2.3 type-checking errors can also be reported in `.js` files with `--checkJs`.

You can skip checking some files by adding `// @ts-nocheck` comment to them; conversely you can choose to check only a few `.js` files by adding `// @ts-check` comment to them without setting `--checkJs`. You can also ignore errors on specific lines by adding `// @ts-ignore` on the preceding line.

`.js` files are still checked to ensure that they only include standard ECMAScript features; type annotations are only allowed in `.ts` files and are flagged as errors in `.js` files. JSDoc comments can be used to add some type information to your JavaScript code, see [JSDoc Support documentation](#) for more details about the supported JSDoc constructs.

See [Type checking JavaScript Files documentation](#) for more details.

Support for Mix-in classes

TypeScript 2.2 adds support for the ECMAScript 2015 mixin class pattern (see [MDN Mixin description](#) and ["Real" Mixins with JavaScript Classes](#) for more details) as well as rules for combining mixin construct signatures with regular construct signatures in intersection types.

First some terminology

A **mixin constructor type** refers to a type that has a single construct signature with a single rest argument of type `any[]` and an object-like return type. For example, given an object-like type `x`, `new (...args: any[]) => x` is a mixin constructor type with an instance type `x`.

A **mixin class** is a class declaration or expression that `extends` an expression of a type parameter type. The following rules apply to mixin class declarations:

- The type parameter type of the `extends` expression must be constrained to a mixin constructor type.
- The constructor of a mixin class (if any) must have a single rest parameter of type `any[]` and must use the spread operator to pass those parameters as arguments in a `super(...args)` call.

Given an expression `Base` of a parametric type `T` with a constraint `x`, a mixin class `class C extends Base {...}` is processed as if `Base` had type `x` and the resulting type is the intersection `typeof C & T`. In other words, a mixin class is represented as an intersection between the mixin class constructor type and the parametric base class constructor type.

When obtaining the construct signatures of an intersection type that contains mixin constructor types, the mixin construct signatures are discarded and their instance types are mixed into the return types of the other construct signatures in the intersection type. For example, the intersection type `{ new(...args: any[]) => A } & { new(s: string) => B }` has a single construct signature `new(s: string) => A & B`.

Putting all of the above rules together in an example

```
class Point {
    constructor(public x: number, public y: number) {}
}

class Person {
    constructor(public name: string) {}
}

type Constructor<T> = new(...args: any[]) => T;

function Tagged<T extends Constructor<{}>>(Base: T) {
    return class extends Base {
        _tag: string;
        constructor(...args: any[]) {
            super(...args);
            this._tag = "";
        }
    }
}

const TaggedPoint = Tagged(Point);

let point = new TaggedPoint(10, 20);
point._tag = "hello";

class Customer extends Tagged(Person) {
    accountBalance: number;
}
```

```
let customer = new Customer("Joe");
customer._tag = "test";
customer.accountBalance = 0;
```

Mixin classes can constrain the types of classes they can mix into by specifying a construct signature return type in the constraint for the type parameter. For example, the following `WithLocation` function implements a subclass factory that adds a `getLocation` method to any class that satisfies the `Point` interface (i.e. that has `x` and `y` properties of type `number`).

```
interface Point {
    x: number;
    y: number;
}

const WithLocation = <T extends Constructor<Point>>(Base: T) =>
    class extends Base {
        getLocation(): [number, number] {
            return [this.x, this.y];
        }
    }
}
```

object type

TypeScript did not have a type that represents the non-primitive type, i.e. any thing that is not `number`, `string`, `boolean`, `symbol`, `null`, or `undefined`. Enter the new `object` type.

With `object` type, APIs like `Object.create` can be better represented. For example:

```
declare function create(o: object | null): void;

create({ prop: 0 }); // OK
create(null); // OK

create(42); // Error
create("string"); // Error
create(false); // Error
create(undefined); // Error
```

Support for `new.target`

The `new.target` meta-property is new syntax introduced in ES2015. When an instance of a constructor is created via `new`, the value of `new.target` is set to be a reference to the constructor function initially used to allocate the instance. If a function is called rather than constructed via `new`, `new.target` is set to `undefined`.

`new.target` comes in handy when `Object.setPrototypeOf` or `__proto__` needs to be set in a class constructor. One such use case is inheriting from `Error` in NodeJS v4 and higher.

Example

```
class CustomError extends Error {
    constructor(message?: string) {
        super(message); // 'Error' breaks prototype chain here
        Object.setPrototypeOf(this, new.target.prototype); // restore prototype chain
    }
}
```

This results in the generated JS

```
var CustomError = (function (_super) {
  __extends(CustomError, _super);
  function CustomError() {
    var _newTarget = this.constructor;
    var _this = _super.apply(this, arguments); // 'Error' breaks prototype chain here
    _this.__proto__ = _newTarget.prototype; // restore prototype chain
    return _this;
  }
  return CustomError;
})(Error);
```

`new.target` also comes in handy for writing constructable functions, for example:

```
function f() {
  if (new.target) { /* called via 'new' */ }
}
```

Which translates to:

```
function f() {
  var _newTarget = this && this instanceof f ? this.constructor : void 0;
  if (_newTarget) { /* called via 'new' */ }
}
```

Better checking for `null` / `undefined` in operands of expressions

TypeScript 2.2 improves checking of nullable operands in expressions. Specifically, these are now flagged as errors:

- If either operand of a `+` operator is nullable, and neither operand is of type `any` or `string`.
- If either operand of a `-`, `*`, `**`, `/`, `%`, `<<`, `>>`, `>>>`, `&`, `|`, or `^` operator is nullable.
- If either operand of a `<`, `>`, `<=`, `>=`, or `in` operator is nullable.
- If the right operand of an `instanceof` operator is nullable.
- If the operand of a `+, -`, `~, ++`, or `--` unary operator is nullable.

An operand is considered nullable if the type of the operand is `null` or `undefined` or a union type that includes `null` or `undefined`. Note that the union type case only occurs in `--strictNullChecks` mode because `null` and `undefined` disappear from unions in classic type checking mode.

Dotted property for types with string index signatures

Types with a string index signature can be indexed using the `[]` notation, but were not allowed to use the `.`. Starting with TypeScript 2.2 using either should be allowed.

```
interface StringMap<T> {
  [x: string]: T;
}

const map: StringMap<number>;

map["prop1"] = 1;
map.prop2 = 2;
```

This only apply to types with an *explicit* string index signature. It is still an error to access unknown properties on a type using `.` notation.

Support for spread operator on JSX element children

TypeScript 2.2 adds support for using spread on a JSX element children. Please see [facebook/jsx#57](#) for more details.

Example

```
function Todo(prop: { key: number, todo: string }) {
    return <div>{prop.key.toString() + prop.todo}</div>;
}

function TodoList({ todos }: TodoListProps) {
    return <div>
        {...todos.map(todo => <Todo key={todo.id} todo={todo.todo} />)}
    </div>;
}

let x: TodoListProps;

<TodoList {...x} />
```

New `jsx: react-native`

React-native build pipeline expects all files to have a `.js` extensions even if the file contains JSX syntax. The new `-jsx` value `react-native` will persevere the JSX syntax in the output file, but give it a `.js` extension.

keyof and Lookup Types

In JavaScript it is fairly common to have APIs that expect property names as parameters, but so far it hasn't been possible to express the type relationships that occur in those APIs.

Enter Index Type Query or `keyof`; An indexed type query `keyof T` yields the type of permitted property names for `T`. A `keyof T` type is considered a subtype of `string`.

Example

```
interface Person {
    name: string;
    age: number;
    location: string;
}

type K1 = keyof Person; // "name" | "age" | "location"
type K2 = keyof Person[]; // "length" | "push" | "pop" | "concat" | ...
type K3 = keyof { [x: string]: Person }; // string
```

The dual of this is *indexed access types*, also called *lookup types*. Syntactically, they look exactly like an element access, but are written as types:

Example

```
type P1 = Person["name"]; // string
type P2 = Person["name" | "age"]; // string | number
type P3 = string["charAt"]; // (pos: number) => string
type P4 = string[]["push"]; // (...items: string[]) => number
type P5 = string[][], // string
```

You can use this pattern with other parts of the type system to get type-safe lookups.

```
function getProperty<T, K extends keyof T>(obj: T, key: K) {
    return obj[key]; // Inferred type is T[K]
}

function setProperty<T, K extends keyof T>(obj: T, key: K, value: T[K]) {
    obj[key] = value;
}

let x = { foo: 10, bar: "hello!" };

let foo = getProperty(x, "foo"); // number
let bar = getProperty(x, "bar"); // string

let oops = getProperty(x, "wargarbl"); // Error! "wargarbl" is not "foo" | "bar"

setProperty(x, "foo", "string"); // Error!, string expected number
```

Mapped Types

One common task is to take an existing type and make each of its properties entirely optional. Let's say we have a `Person`:

```
interface Person {
    name: string;
    age: number;
    location: string;
}
```

A partial version of it would be:

```
interface PartialPerson {
    name?: string;
    age?: number;
    location?: string;
}
```

with Mapped types, `PartialPerson` can be written as a generalized transformation on the type `Person` as:

```
type Partial<T> = {
    [P in keyof T]?: T[P];
};

type PartialPerson = Partial<Person>;
```

Mapped types are produced by taking a union of literal types, and computing a set of properties for a new object type. They're like [list comprehensions in Python](#), but instead of producing new elements in a list, they produce new properties in a type.

In addition to `Partial`, Mapped Types can express many useful transformations on types:

```
// Keep types the same, but make each property to be read-only.
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};

// Same property names, but make the value a promise instead of a concrete one
type Deferred<T> = {
    [P in keyof T]: Promise<T[P]>;
};

// Wrap proxies around properties of T
type Proxify<T> = {
    [P in keyof T]: { get(): T[P]; set(v: T[P]): void }
};
```

Partial , Readonly , Record , and Pick

`Partial` and `Readonly`, as described earlier, are very useful constructs. You can use them to describe some common JS routines like:

```
function assign<T>(obj: T, props: Partial<T>): void;
function freeze<T>(obj: T): Readonly<T>;
```

Because of that, they are now included by default in the standard library.

We're also including two other utility types as well: `Record` and `Pick`.

```
// From T pick a set of properties K
declare function pick<T, K extends keyof T>(obj: T, ...keys: K[]): Pick<T, K>;
```

```
const nameAndAgeOnly = pick(person, "name", "age"); // { name: string, age: number }

// For every properties K of type T, transform it to U
function mapObject<K extends string, T, U>(obj: Record<K, T>, f: (x: T) => U): Record<K, U>

const names = { foo: "hello", bar: "world", baz: "bye" };
const lengths = mapObject(names, s => s.length); // { foo: number, bar: number, baz: number }
```

Object Spread and Rest

TypeScript 2.1 brings support for [ESnext Spread and Rest](#).

Similar to array spread, spreading an object can be handy to get a shallow copy:

```
let copy = { ...original };
```

Similarly, you can merge several different objects. In the following example, `merged` will have properties from `foo`, `bar`, and `baz`.

```
let merged = { ...foo, ...bar, ...baz };
```

You can also override existing properties and add new ones:

```
let obj = { x: 1, y: "string" };
var newObj = { ...obj, z: 3, y: 4 }; // { x: number, y: number, z: number }
```

The order of specifying spread operations determines what properties end up in the resulting object; properties in later spreads "win out" over previously created properties.

Object rests are the dual of object spreads, in that they can extract any extra properties that don't get picked up when destructuring an element:

```
let obj = { x: 1, y: 1, z: 1 };
let { z, ...obj1 } = obj;
obj1; // { x: number, y: number };
```

Downlevel Async Functions

This feature was supported before TypeScript 2.1, but only when targeting ES6/ES2015. TypeScript 2.1 brings the capability to ES3 and ES5 run-times, meaning you'll be free to take advantage of it no matter what environment you're using.

Note: first, we need to make sure our run-time has an ECMAScript-compliant `Promise` available globally. That might involve grabbing [a polyfill](#) for `Promise`, or relying on one that you might have in the run-time that you're targeting. We also need to make sure that TypeScript knows `Promise` exists by setting your `lib` flag to something like `"dom", "es2015"` or `"dom", "es2015.promise", "es5"`

Example

`tsconfig.json`

```
{
  "compilerOptions": {
    "lib": ["dom", "es2015.promise", "es5"]
  }
}
```

dramaticWelcome.ts

```
function delay(milliseconds: number) {
  return new Promise<void>(resolve => {
    setTimeout(resolve, milliseconds);
  });
}

async function dramaticWelcome() {
  console.log("Hello");

  for (let i = 0; i < 3; i++) {
    await delay(500);
    console.log(".");
  }

  console.log("World!");
}

dramaticWelcome();
```

Compiling and running the output should result in the correct behavior on an ES3/ES5 engine.

Support for external helpers library (`tslib`)

TypeScript injects a handful of helper functions such as `__extends` for inheritance, `__assign` for spread operator in object literals and JSX elements, and `__awaiter` for async functions.

Previously there were two options:

1. inject helpers in **every** file that needs them, or
2. no helpers at all with `--noEmitHelpers`.

The two options left more to be desired; bundling the helpers in every file was a pain point for customers trying to keep their package size small. And not including helpers, meant customers had to maintain their own helpers library.

TypeScript 2.1 allows for including these files in your project once in a separate module, and the compiler will emit imports to them as needed.

First, install the `tslib` utility library:

```
npm install tslib
```

Second, compile your files using `--importHelpers`:

```
tsc --module commonjs --importHelpers a.ts
```

So given the following input, the resulting `.js` file will include an import to `tslib` and use the `__assign` helper from it instead of inlining it.

```
export const o = { a: 1, name: "o" };
export const copy = { ...o };
```

```
"use strict";
var tslib_1 = require("tslib");
exports.o = { a: 1, name: "o" };
exports.copy = tslib_1.__assign({}, exports.o);
```

Untyped imports

TypeScript has traditionally been overly strict about how you can import modules. This was to avoid typos and prevent users from using modules incorrectly.

However, a lot of the time, you might just want to import an existing module that may not have its own `.d.ts` file. Previously this was an error. Starting with TypeScript 2.1 this is now much easier.

With TypeScript 2.1, you can import a JavaScript module without needing a type declaration. A type declaration (such as `declare module "foo" { ... }` or `node_modules/@types/foo`) still takes priority if it exists.

An import to a module with no declaration file will still be flagged as an error under `--noImplicitAny`.

Example

```
// Succeeds if `node_modules/asdf/index.js` exists
import { x } from "asdf";
```

Support for `--target ES2016`, `--target ES2017` and `--target ESNext`

TypeScript 2.1 supports three new target values `--target ES2016`, `--target ES2017` and `--target ESNext`.

Using target `--target ES2016` will instruct the compiler not to transform ES2016-specific features, e.g. `**` operator.

Similarly, `--target ES2017` will instruct the compiler not to transform ES2017-specific features like `async` / `await`.

`--target ESNext` targets latest supported [ES proposed features](#).

Improved any Inference

Previously, if TypeScript couldn't figure out the type of a variable, it would choose the `any` type.

```
let x;      // implicitly 'any'
let y = []; // implicitly 'any[]'

let z: any; // explicitly 'any'.
```

With TypeScript 2.1, instead of just choosing `any`, TypeScript will infer types based on what you end up assigning later on.

This is only enabled if `--noImplicitAny` is set.

Example

```
let x;
```

```
// You can still assign anything you want to 'x'.
x = () => 42;

// After that last assignment, TypeScript 2.1 knows that 'x' has type '() => number'.
let y = x();

// Thanks to that, it will now tell you that you can't add a number to a function!
console.log(x + y);
// ~~~~~
// Error! Operator '+' cannot be applied to types '() => number' and 'number'.

// TypeScript still allows you to assign anything you want to 'x'.
x = "Hello world!";

// But now it also knows that 'x' is a 'string'!
x.toLowerCase();
```

The same sort of tracking is now also done for empty arrays.

A variable declared with no type annotation and an initial value of `[]` is considered an implicit `any[]` variable. However, each subsequent `x.push(value)`, `x.unshift(value)` or `x[n] = value` operation evolves the type of the variable in accordance with what elements are added to it.

```
function f1() {
    let x = [];
    x.push(5);
    x[1] = "hello";
    x.unshift(true);
    return x; // (string | number | boolean)[]
}

function f2() {
    let x = null;
    if (cond()) {
        x = [];
        while (cond()) {
            x.push("hello");
        }
    }
    return x; // string[] | null
}
```

Implicit any errors

One great benefit of this is that you'll see way fewer implicit `any` errors when running with `--noImplicitAny`. Implicit `any` errors are only reported when the compiler is unable to know the type of a variable without a type annotation.

Example

```
function f3() {
    let x = []; // Error: Variable 'x' implicitly has type 'any[]' in some locations where its type cannot be determined.
    x.push(5);
    function g() {
        x; // Error: Variable 'x' implicitly has an 'any[]' type.
    }
}
```

Better inference for literal types

String, numeric and boolean literal types (e.g. `"abc"`, `1`, and `true`) were previously inferred only in the presence of an explicit type annotation. Starting with TypeScript 2.1, literal types are *always* inferred for `const` variables and `readonly` properties.

The type inferred for a `const` variable or `readonly` property without a type annotation is the type of the literal initializer. The type inferred for a `let` variable, `var` variable, parameter, or non-`readonly` property with an initializer and no type annotation is the widened literal type of the initializer. Where the widened type for a string literal type is `string`, `number` for numeric literal types, `boolean` for `true` or `false` and the containing enum for enum literal types.

Example

```
const c1 = 1; // Type 1
const c2 = c1; // Type 1
const c3 = "abc"; // Type "abc"
const c4 = true; // Type true
const c5 = cond ? 1 : "abc"; // Type 1 | "abc"

let v1 = 1; // Type number
let v2 = c2; // Type number
let v3 = c3; // Type string
let v4 = c4; // Type boolean
let v5 = c5; // Type number | string
```

Literal type widening can be controlled through explicit type annotations. Specifically, when an expression of a literal type is inferred for a `const` location without a type annotation, that `const` variable gets a widening literal type inferred. But when a `const` location has an explicit literal type annotation, the `const` variable gets a non-widening literal type.

Example

```
const c1 = "hello"; // Widening type "hello"
let v1 = c1; // Type string

const c2: "hello" = "hello"; // Type "hello"
let v2 = c2; // Type "hello"
```

Use returned values from super calls as 'this'

In ES2015, constructors which return an object implicitly substitute the value of `this` for any callers of `super()`. As a result, it is necessary to capture any potential return value of `super()` and replace it with `this`. This change enables working with [Custom Elements](#), which takes advantage of this to initialize browser-allocated elements with user-written constructors.

Example

```
class Base {
    x: number;
    constructor() {
        // return a new object other than `this`
        return {
            x: 1,
        };
    }
}

class Derived extends Base {
    constructor() {
        super();
```

```

        this.x = 2;
    }
}

```

Generates:

```

var Derived = (function (_super) {
    __extends(Derived, _super);
    function Derived() {
        var _this = _super.call(this) || this;
        _this.x = 2;
        return _this;
    }
    return Derived;
})(Base));

```

This change entails a break in the behavior of extending built-in classes like `Error`, `Array`, `Map`, etc.. Please see the [extending built-ins breaking change documentation](#) for more details.

Configuration inheritance

Often a project has multiple output targets, e.g. `ES5` and `ES2015`, `debug` and `production` or `CommonJS` and `System`; Just a few configuration options change between these two targets, and maintaining multiple `tsconfig.json` files can be a hassle.

TypeScript 2.1 supports inheriting configuration using `extends`, where:

- `extends` is a new top-level property in `tsconfig.json` (alongside `compilerOptions`, `files`, `include`, and `exclude`).
- The value of `extends` must be a string containing a path to another configuration file to inherit from.
- The configuration from the base file are loaded first, then overridden by those in the inheriting config file.
- Circularity between configuration files is not allowed.
- `files`, `include` and `exclude` from the inheriting config file *overwrite* those from the base config file.
- All relative paths found in the configuration file will be resolved relative to the configuration file they originated in.

Example

`configs/base.json`:

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "strictNullChecks": true
  }
}
```

`tsconfig.json`:

```
{
  "extends": "./configs/base",
  "files": [
    "main.ts",
    "supplemental.ts"
  ]
}
```

`tsconfig.nostrictnull.json`:

```
{  
  "extends": "./tsconfig",  
  "compilerOptions": {  
    "strictNullChecks": false  
  }  
}
```

New `--alwaysStrict`

Invoking the compiler with `--alwaysStrict` causes:

1. Parses all the code in strict mode.
2. Writes `"use strict";` directive atop every generated file.

Modules are parsed automatically in strict mode. The new flag is recommended for non-module code.

Null- and undefined-aware types

TypeScript has two special types, Null and Undefined, that have the values `null` and `undefined` respectively. Previously it was not possible to explicitly name these types, but `null` and `undefined` may now be used as type names regardless of type checking mode.

The type checker previously considered `null` and `undefined` assignable to anything. Effectively, `null` and `undefined` were valid values of every type and it wasn't possible to specifically exclude them (and therefore not possible to detect erroneous use of them).

--strictNullChecks

`--strictNullChecks` switches to a new strict null checking mode.

In strict null checking mode, the `null` and `undefined` values are *not* in the domain of every type and are only assignable to themselves and `any` (the one exception being that `undefined` is also assignable to `void`). So, whereas `T` and `T | undefined` are considered synonymous in regular type checking mode (because `undefined` is considered a subtype of any `T`), they are different types in strict type checking mode, and only `T | undefined` permits `undefined` values. The same is true for the relationship of `T` to `T | null`.

Example

```
// Compiled with --strictNullChecks
let x: number;
let y: number | undefined;
let z: number | null | undefined;
x = 1; // Ok
y = 1; // Ok
z = 1; // Ok
x = undefined; // Error
y = undefined; // Ok
z = undefined; // Ok
x = null; // Error
y = null; // Error
z = null; // Ok
x = y; // Error
x = z; // Error
y = x; // Ok
y = z; // Error
z = x; // Ok
z = y; // Ok
```

Assigned-before-use checking

In strict null checking mode the compiler requires every reference to a local variable of a type that doesn't include `undefined` to be preceded by an assignment to that variable in every possible preceding code path.

Example

```
// Compiled with --strictNullChecks
let x: number;
let y: number | null;
let z: number | undefined;
x; // Error, reference not preceded by assignment
y; // Error, reference not preceded by assignment
```

```
z; // Ok
x = 1;
y = null;
x; // Ok
y; // Ok
```

The compiler checks that variables are definitely assigned by performing *control flow based type analysis*. See later for further details on this topic.

Optional parameters and properties

Optional parameters and properties automatically have `undefined` added to their types, even when their type annotations don't specifically include `undefined`. For example, the following two types are identical:

```
// Compiled with --strictNullChecks
type T1 = (x?: number) => string; // x has type number | undefined
type T2 = (x?: number | undefined) => string; // x has type number | undefined
```

Non-null and non-undefined type guards

A property access or a function call produces a compile-time error if the object or function is of a type that includes `null` or `undefined`. However, type guards are extended to support non-null and non-undefined checks.

Example

```
// Compiled with --strictNullChecks
declare function f(x: number): string;
let x: number | null | undefined;
if (x) {
    f(x); // Ok, type of x is number here
}
else {
    f(x); // Error, type of x is number? here
}
let a = x != null ? f(x) : ""; // Type of a is string
let b = x && f(x); // Type of b is string | 0 | null | undefined
```

Non-null and non-undefined type guards may use the `==`, `!=`, `===`, or `!==` operator to compare to `null` or `undefined`, as in `x != null` or `x === undefined`. The effects on subject variable types accurately reflect JavaScript semantics (e.g. double-equals operators check for both values no matter which one is specified whereas triple-equals only checks for the specified value).

Dotted names in type guards

Type guards previously only supported checking local variables and parameters. Type guards now support checking "dotted names" consisting of a variable or parameter name followed one or more property accesses.

Example

```
interface Options {
    location?: {
        x?: number;
        y?: number;
    };
}
```

```

    }

function foo(options?: Options) {
    if (options && options.location && options.location.x) {
        const x = options.location.x; // Type of x is number
    }
}

```

Type guards for dotted names also work with user defined type guard functions and the `typeof` and `instanceof` operators and do not depend on the `--strictNullChecks` compiler option.

A type guard for a dotted name has no effect following an assignment to any part of the dotted name. For example, a type guard for `x.y.z` will have no effect following an assignment to `x`, `x.y`, or `x.y.z`.

Expression operators

Expression operators permit operand types to include `null` and/or `undefined` but always produce values of non-null and non-undefined types.

```

// Compiled with --strictNullChecks
function sum(a: number | null, b: number | null) {
    return a + b; // Produces value of type number
}

```

The `&&` operator adds `null` and/or `undefined` to the type of the right operand depending on which are present in the type of the left operand, and the `||` operator removes both `null` and `undefined` from the type of the left operand in the resulting union type.

```

// Compiled with --strictNullChecks
interface Entity {
    name: string;
}

let x: Entity | null;
let s = x && x.name; // s is of type string | null
let y = x || { name: "test" }; // y is of type Entity

```

Type widening

The `null` and `undefined` types are *not* widened to `any` in strict null checking mode.

```
let z = null; // Type of z is null
```

In regular type checking mode the inferred type of `z` is `any` because of widening, but in strict null checking mode the inferred type of `z` is `null` (and therefore, absent a type annotation, `null` is the only possible value for `z`).

Non-null assertion operator

A new `!` post-fix expression operator may be used to assert that its operand is non-null and non-undefined in contexts where the type checker is unable to conclude that fact. Specifically, the operation `x!` produces a value of the type of `x` with `null` and `undefined` excluded. Similar to type assertions of the forms `<T>x` and `x as T`, the `!` non-null assertion operator is simply removed in the emitted JavaScript code.

```
// Compiled with --strictNullChecks
```

```

function validateEntity(e?: Entity) {
    // Throw exception if e is null or invalid entity
}

function processEntity(e?: Entity) {
    validateEntity(e);
    let s = e!.name; // Assert that e is non-null and access name
}

```

Compatibility

The new features are designed such that they can be used in both strict null checking mode and regular type checking mode. In particular, the `null` and `undefined` types are automatically erased from union types in regular type checking mode (because they are subtypes of all other types), and the `!` non-null assertion expression operator is permitted but has no effect in regular type checking mode. Thus, declaration files that are updated to use null- and undefined-aware types can still be used in regular type checking mode for backwards compatibility.

In practical terms, strict null checking mode requires that all files in a compilation are null- and undefined-aware.

Control flow based type analysis

TypeScript 2.0 implements a control flow-based type analysis for local variables and parameters. Previously, the type analysis performed for type guards was limited to `if` statements and `?:` conditional expressions and didn't include effects of assignments and control flow constructs such as `return` and `break` statements. With TypeScript 2.0, the type checker analyses all possible flows of control in statements and expressions to produce the most specific type possible (the *narrowed type*) at any given location for a local variable or parameter that is declared to have a union type.

Example

```

function foo(x: string | number | boolean) {
    if (typeof x === "string") {
        x; // type of x is string here
        x = 1;
        x; // type of x is number here
    }
    x; // type of x is number | boolean here
}

function bar(x: string | number) {
    if (typeof x === "number") {
        return;
    }
    x; // type of x is string here
}

```

Control flow based type analysis is particularly relevant in `--strictNullChecks` mode because nullable types are represented using union types:

```

function test(x: string | null) {
    if (x === null) {
        return;
    }
    x; // type of x is string in remainder of function
}

```

Furthermore, in `--strictNullChecks` mode, control flow based type analysis includes *definite assignment analysis* for local variables of types that don't permit the value `undefined`.

```
function mumble(check: boolean) {
    let x: number; // Type doesn't permit undefined
    x; // Error, x is undefined
    if (check) {
        x = 1;
        x; // Ok
    }
    x; // Error, x is possibly undefined
    x = 2;
    x; // Ok
}
```

Tagged union types

TypeScript 2.0 implements support for tagged (or discriminated) union types. Specifically, the TS compiler now supports type guards that narrow union types based on tests of a discriminant property and furthermore extend that capability to `switch` statements.

Example

```
interface Square {
    kind: "square";
    size: number;
}

interface Rectangle {
    kind: "rectangle";
    width: number;
    height: number;
}

interface Circle {
    kind: "circle";
    radius: number;
}

type Shape = Square | Rectangle | Circle;

function area(s: Shape) {
    // In the following switch statement, the type of s is narrowed in each case clause
    // according to the value of the discriminant property, thus allowing the other properties
    // of that variant to be accessed without a type assertion.
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.width * s.height;
        case "circle": return Math.PI * s.radius * s.radius;
    }
}

function test1(s: Shape) {
    if (s.kind === "square") {
        s; // Square
    }
    else {
        s; // Rectangle | Circle
    }
}

function test2(s: Shape) {
    if (s.kind === "square" || s.kind === "rectangle") {
```

```

        return;
    }
s; // Circle
}

```

A *discriminant property type guard* is an expression of the form `x.p == v`, `x.p === v`, `x.p != v`, or `x.p !== v`, where `p` and `v` are a property and an expression of a string literal type or a union of string literal types. The discriminant property type guard narrows the type of `x` to those constituent types of `x` that have a discriminant property `p` with one of the possible values of `v`.

Note that we currently only support discriminant properties of string literal types. We intend to later add support for boolean and numeric literal types.

The `never` type

TypeScript 2.0 introduces a new primitive type `never`. The `never` type represents the type of values that never occur. Specifically, `never` is the return type for functions that never return and `never` is the type of variables under type guards that are never true.

The `never` type has the following characteristics:

- `never` is a subtype of and assignable to every type.
- No type is a subtype of or assignable to `never` (except `never` itself).
- In a function expression or arrow function with no return type annotation, if the function has no `return` statements, or only `return` statements with expressions of type `never`, and if the end point of the function is not reachable (as determined by control flow analysis), the inferred return type for the function is `never`.
- In a function with an explicit `never` return type annotation, all `return` statements (if any) must have expressions of type `never` and the end point of the function must not be reachable.

Because `never` is a subtype of every type, it is always omitted from union types and it is ignored in function return type inference as long as there are other types being returned.

Some examples of functions returning `never`:

```

// Function returning never must have unreachable end point
function error(message: string): never {
    throw new Error(message);
}

// Inferred return type is never
function fail() {
    return error("Something failed");
}

// Function returning never must have unreachable end point
function infiniteLoop(): never {
    while (true) {
    }
}

```

Some examples of use of functions returning `never`:

```

// Inferred return type is number
function move1(direction: "up" | "down") {
    switch (direction) {
        case "up":
            return 1;
        case "down":

```

```

        return -1;
    }
    return error("Should never get here");
}

// Inferred return type is number
function move2(direction: "up" | "down") {
    return direction === "up" ? 1 :
        direction === "down" ? -1 :
        error("Should never get here");
}

// Inferred return type is T
function check<T>(x: T | undefined) {
    return x || error("Undefined value");
}

```

Because `never` is assignable to every type, a function returning `never` can be used when a callback returning a more specific type is required:

```

function test(cb: () => string) {
    let s = cb();
    return s;
}

test(() => "hello");
test(() => fail());
test(() => { throw new Error(); })

```

Read-only properties and index signatures

A property or index signature can now be declared with the `readonly` modifier is considered read-only.

Read-only properties may have initializers and may be assigned to in constructors within the same class declaration, but otherwise assignments to read-only properties are disallowed.

In addition, entities are *implicitly* read-only in several situations:

- A property declared with a `get` accessor and no `set` accessor is considered read-only.
- In the type of an enum object, enum members are considered read-only properties.
- In the type of a module object, exported `const` variables are considered read-only properties.
- An entity declared in an `import` statement is considered read-only.
- An entity accessed through an ES2015 namespace import is considered read-only (e.g. `foo.x` is read-only when `foo` is declared as `import * as foo from "foo"`).

Example

```

interface Point {
    readonly x: number;
    readonly y: number;
}

var p1: Point = { x: 10, y: 20 };
p1.x = 5; // Error, p1.x is read-only

var p2 = { x: 1, y: 1 };
var p3: Point = p2; // Ok, read-only alias for p2
p3.x = 5; // Error, p3.x is read-only
p2.x = 5; // Ok, but also changes p3.x because of aliasing

```

```
class Foo {
    readonly a = 1;
    readonly b: string;
    constructor() {
        this.b = "hello"; // Assignment permitted in constructor
    }
}
```

```
let a: Array<number> = [0, 1, 2, 3, 4];
let b: ReadonlyArray<number> = a;
b[5] = 5; // Error, elements are read-only
b.push(5); // Error, no push method (because it mutates array)
b.length = 3; // Error, length is read-only
a = b; // Error, mutating methods are missing
```

Specifying the type of `this` for functions

Following up on specifying the type of `this` in a class or an interface, functions and methods can now declare the type of `this` they expect.

By default the type of `this` inside a function is `any`. Starting with TypeScript 2.0, you can provide an explicit `this` parameter. `this` parameters are fake parameters that come first in the parameter list of a function:

```
function f(this: void) {
    // make sure `this` is unusable in this standalone function
}
```

`this` parameters in callbacks

Libraries can also use `this` parameters to declare how callbacks will be invoked.

Example

```
interface UIElement {
    addClickListener(onclick: (this: void, e: Event) => void): void;
}
```

`this: void` means that `addClickListener` expects `onclick` to be a function that does not require a `this` type.

Now if you annotate calling code with `this`:

```
class Handler {
    info: string;
    onClickBad(this: Handler, e: Event) {
        // oops, used this here. using this callback would crash at runtime
        this.info = e.message;
    }
}
let h = new Handler();
uiElement.addClickListener(h.onClickBad); // error!
```

--noImplicitThis

A new flag is also added in TypeScript 2.0 to flag all uses of `this` in functions without an explicit type annotation.

Glob support in `tsconfig.json`

Glob support is here!! Glob support has been [one of the most requested features](#).

Glob-like file patterns are supported two properties `"include"` and `"exclude"`.

Example

```
{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "outFile": "../../built/local/tsc.js",
    "sourceMap": true
  },
  "include": [
    "src/**/*"
  ],
  "exclude": [
    "node_modules",
    "**/*.spec.ts"
  ]
}
```

The supported glob wildcards are:

- `*` matches zero or more characters (excluding directory separators)
- `?` matches any one character (excluding directory separators)
- `**/` recursively matches any subdirectory

If a segment of a glob pattern includes only `*` or `.*`, then only files with supported extensions are included (e.g. `.ts`, `.tsx`, and `.d.ts` by default with `.js` and `.jsx` if `allowJs` is set to true).

If the `"files"` and `"include"` are both left unspecified, the compiler defaults to including all TypeScript (`.ts`, `.d.ts` and `.tsx`) files in the containing directory and subdirectories except those excluded using the `"exclude"` property. JS files (`.js` and `.jsx`) are also included if `allowJs` is set to true.

If the `"files"` or `"include"` properties are specified, the compiler will instead include the union of the files included by those two properties. Files in the directory specified using the `"outDir"` compiler option are always excluded unless explicitly included via the `"files"` property (even when the `"exclude"` property is specified).

Files included using `"include"` can be filtered using the `"exclude"` property. However, files included explicitly using the `"files"` property are always included regardless of `"exclude"`. The `"exclude"` property defaults to excluding the `node_modules`, `bower_components`, and `jspm_packages` directories when not specified.

Module resolution enhancements: `BaseUrl`, `Path mapping`, `rootDirs` and `tracing`

TypeScript 2.0 provides a set of additional module resolution knobs to *inform* the compiler where to find declarations for a given module.

See [Module Resolution](#) documentation for more details.

Base URL

Using a `baseUrl` is a common practice in applications using AMD module loaders where modules are "deployed" to a single folder at run-time. All module imports with non-relative names are assumed to be relative to the `baseUrl`.

Example

```
{
  "compilerOptions": {
    "baseUrl": "./modules"
  }
}
```

Now imports to `"moduleA"` would be looked up in `./modules/moduleA`

```
import A from "moduleA";
```

Path mapping

Sometimes modules are not directly located under `baseUrl`. Loaders use a mapping configuration to map module names to files at run-time, see [RequireJs documentation](#) and [SystemJS documentation](#).

The TypeScript compiler supports the declaration of such mappings using `"paths"` property in `tsconfig.json` files.

Example

For instance, an import to a module `"jquery"` would be translated at runtime to

```
"node_modules/jquery/dist/jquery.slim.min.js".
```

```
{
  "compilerOptions": {
    "baseUrl": "./node_modules",
    "paths": {
      "jquery": ["jquery/dist/jquery.slim.min"]
    }
}
```

Using `"paths"` also allow for more sophisticated mappings including multiple fall back locations. Consider a project configuration where only some modules are available in one location, and the rest are in another.

Virtual Directories with `rootDirs`

Using 'rootDirs', you can inform the compiler of the `roots` making up this "virtual" directory; and thus the compiler can resolve relative modules imports within these "virtual" directories as if were merged together in one directory.

Example

Given this project structure:

```

src
└── views
    └── view1.ts (imports './template1')
    └── view2.ts

generated
└── templates
    └── views
        └── template1.ts (imports './view2')

```

A build step will copy the files in `/src/views` and `/generated/templates/views` to the same directory in the output. At run-time, a view can expect its template to exist next to it, and thus should import it using a relative name as

```
"./template"
```

`"rootDirs"` specify a list of *roots* whose contents are expected to merge at run-time. So following our example, the `tsconfig.json` file should look like:

```
{
  "compilerOptions": {
    "rootDirs": [
      "src/views",
      "generated/templates/views"
    ]
  }
}
```

Tracing module resolution

`--traceResolution` offers a handy way to understand how modules have been resolved by the compiler.

```
tsc --traceResolution
```

Shorthand ambient module declarations

If you don't want to take the time to write out declarations before using a new module, you can now just use a shorthand declaration to get started quickly.

declarations.d.ts

```
declare module "hot-new-module";
```

All imports from a shorthand module will have the any type.

```
import x, {y} from "hot-new-module";
x(y);
```

Wildcard character in module names

Importing none-code resources using module loaders extension (e.g. [AMD](#) or [SystemJS](#)) has not been easy before; previously an ambient module declaration had to be defined for each resource.

TypeScript 2.0 supports the use of the wildcard character (`*`) to declare a "family" of module names; this way, a declaration is only required once for an extension, and not for every resource.

Example

```
declare module "*!text" {
  const content: string;
  export default content;
}
// Some do it the other way around.
```

```
declare module "json!*" {
    const value: any;
    export default value;
}
```

Now you can import things that match `"*!text"` or `"json!*"`.

```
import fileContent from "./xyz.txt!text";
import data from "json!http://example.com/data.json";
console.log(data, fileContent);
```

Wildcard module names can be even more useful when migrating from an un-typed code base. Combined with Shorthand ambient module declarations, a set of modules can be easily declared as `any`.

Example

```
declare module "myLibrary/*";
```

All imports to any module under `myLibrary` would be considered to have the type `any` by the compiler; thus, shutting down any checking on the shapes or types of these modules.

```
import { readFile } from "myLibrary/fileSystem/readFile`;

readFile(); // readFile is 'any'
```

Support for UMD module definitions

Some libraries are designed to be used in many module loaders, or with no module loading (global variables). These are known as [UMD](#) or [Isomorphic](#) modules. These libraries can be accessed through either an import or a global variable.

For example:

math-lib.d.ts

```
export const isPrime(x: number): boolean;
export as namespace mathLib;
```

The library can then be used as an import within modules:

```
import { isPrime } from "math-lib";
isPrime(2);
mathLib.isPrime(2); // ERROR: can't use the global definition from inside a module
```

It can also be used as a global variable, but only inside of a script. (A script is a file with no imports or exports.)

```
mathLib.isPrime(2);
```

Optional class properties

Optional properties and methods can now be declared in classes, similar to what is already permitted in interfaces.

Example

```
class Bar {
    a: number;
    b?: number;
    f() {
        return 1;
    }
    g?(): number; // Body of optional method can be omitted
    h?() {
        return 2;
    }
}
```

When compiled in `--strictNullChecks` mode, optional properties and methods automatically have `undefined` included in their type. Thus, the `b` property above is of type `number | undefined` and the `g` method above is of type `((() => number) | undefined)`. Type guards can be used to strip away the `undefined` part of the type:

```
function test(x: Bar) {
    x.a; // number
    x.b; // number | undefined
    x.f; // () => number
    x.g; // ((() => number) | undefined)
    let f1 = x.f(); // number
    let g1 = x.g && x.g(); // number | undefined
    let g2 = x.g ? x.g() : 0; // number
}
```

Private and Protected Constructors

A class constructor may be marked `private` or `protected`. A class with private constructor cannot be instantiated outside the class body, and cannot be extended. A class with protected constructor cannot be instantiated outside the class body, but can be extended.

Example

```
class Singleton {
    private static instance: Singleton;

    private constructor() { }

    static getInstance() {
        if (!Singleton.instance) {
            Singleton.instance = new Singleton();
        }
        return Singleton.instance;
    }
}

let e = new Singleton(); // Error: constructor of 'Singleton' is private.
let v = Singleton.getInstance();
```

Abstract properties and accessors

An abstract class can declare abstract properties and/or accessors. Any sub class will need to declare the abstract properties or be marked as abstract. Abstract properties cannot have an initializer. Abstract accessors cannot have bodies.

Example

```
abstract class Base {
    abstract name: string;
    abstract get value();
    abstract set value(v: number);
}

class Derived extends Base {
    name = "derived";

    value = 1;
}
```

Implicit index signatures

An object literal type is now assignable to a type with an index signature if all known properties in the object literal are assignable to that index signature. This makes it possible to pass a variable that was initialized with an object literal as parameter to a function that expects a map or dictionary:

```
function httpService(path: string, headers: { [x: string]: string }) { }

const headers = {
    "Content-Type": "application/x-www-form-urlencoded"
};

httpService("", { "Content-Type": "application/x-www-form-urlencoded" }); // Ok
httpService("", headers); // Now ok, previously wasn't
```

Including built-in type declarations with `--lib`

Getting to ES6/ES2015 built-in API declarations were only limited to `target: ES6`. Enter `--lib`; with `--lib` you can specify a list of built-in API declaration groups that you can chose to include in your project. For instance, if you expect your runtime to have support for `Map`, `Set` and `Promise` (e.g. most evergreen browsers today), just include `--lib es2015.collection,es2015.promise`. Similarly you can exclude declarations you do not want to include in your project, e.g. `DOM` if you are working on a node project using `--lib es5,es6`.

Here is a list of available API groups:

- dom
- webworker
- es5
- es6 / es2015
- es2015.core
- es2015.collection
- es2015.iterable
- es2015.promise
- es2015.proxy
- es2015.reflect
- es2015.generator
- es2015.symbol
- es2015.symbol.wellknown
- es2016

- es2016.array.include
- es2017
- es2017.object
- es2017.sharedmemory
- scripthost

Example

```
tsc --target es5 --lib es5,es2015.promise
```

```
"compilerOptions": {
  "lib": ["es5", "es2015.promise"]
}
```

Flag unused declarations with `--noUnusedParameters` and `--noUnusedLocals`

TypeScript 2.0 has two new flags to help you maintain a clean code base. `--noUnusedParameters` flags any unused function or method parameters errors. `--noUnusedLocals` flags any unused local (un-exported) declaration like variables, functions, classes, imports, etc... Also, unused private members of a class would be flagged as errors under `--noUnusedLocals`.

Example

```
import B, { readFile } from "./b";
//      ^ Error: `B` declared but never used
readFile();

export function write(message: string, args: string[]) {
  //          ^^^^^ Error: 'arg' declared but never used.
  console.log(message);
}
```

Parameters declaration with names starting with `_` are exempt from the unused parameter checking. e.g.:

```
function returnNull(_a) { // OK
  return null;
}
```

Module identifiers allow for `.js` extension

Before TypeScript 2.0, a module identifier was always assumed to be extension-less; for instance, given an import as `import d from "./moduleA.js"`, the compiler looked up the definition of `"moduleA.js"` in `./moduleA.js.ts` or `./moduleA.js.d.ts`. This made it hard to use bundling/loading tools like [SystemJS](#) that expect URI's in their module identifier.

With TypeScript 2.0, the compiler will look up definition of `"moduleA.js"` in `./moduleA.ts` or `./moduleA.d.ts`.

Support 'target : es5' with 'module: es6'

Previously flagged as an invalid flag combination, `target: es5` and 'module: es6' is now supported. This should facilitate using ES2015-based tree shakers like [rollup](#).

Trailing commas in function parameter and argument lists

Trailing comma in function parameter and argument lists are now allowed. This is an implementation for a [Stage-3 ECMAScript proposal](#) that emits down to valid ES3/ES5/ES6.

Example

```
function foo(
  bar: Bar,
  baz: Baz, // trailing commas are OK in parameter lists
) {
  // Implementation...
}

foo(
  bar,
  baz, // and in argument lists
);
```

New `--skipLibCheck`

TypeScript 2.0 adds a new `--skipLibCheck` compiler option that causes type checking of declaration files (files with extension `.d.ts`) to be skipped. When a program includes large declaration files, the compiler spends a lot of time type checking declarations that are already known to not contain errors, and compile times may be significantly shortened by skipping declaration file type checks.

Since declarations in one file can affect type checking in other files, some errors may not be detected when `--skipLibCheck` is specified. For example, if a non-declaration file augments a type declared in a declaration file, errors may result that are only reported when the declaration file is checked. However, in practice such situations are rare.

Allow duplicate identifiers across declarations

This has been one common source of duplicate definition errors. Multiple declaration files defining the same members on interfaces.

TypeScript 2.0 relaxes this constraint and allows duplicate identifiers across blocks, as long as they have *identical* types.

Within the same block duplicate definitions are still disallowed.

Example

```
interface Error {
  stack?: string;
}

interface Error {
  code?: string;
```

```
    path?: string;
    stack?: string; // OK
}
```

New `--declarationDir`

`--declarationDir` allows for generating declaration files in a different location than JavaScript files.

Type parameters as constraints

With TypeScript 1.8 it becomes possible for a type parameter constraint to reference type parameters from the same type parameter list. Previously this was an error. This capability is usually referred to as [F-Bounded Polymorphism](#).

Example

```
function assign<T extends U, U>(target: T, source: U): T {
    for (let id in source) {
        target[id] = source[id];
    }
    return target;
}

let x = { a: 1, b: 2, c: 3, d: 4 };
assign(x, { b: 10, d: 20 });
assign(x, { e: 0 }); // Error
```

Control flow analysis errors

TypeScript 1.8 introduces control flow analysis to help catch common errors that users tend to run into. Read on to get more details, and check out these errors in action:

```
function foo(x:boolean):number{
    if(x){
        return 10;
    }
    else{
        throw new Error();
    }
    return 1;
}

function bar(x:number,y:boolean):number{
    switch(x){
        case 3:if(y)return 2;
        case 4:return 3;
        default:return 4;
    }
}
```

Unreachable code

Statements guaranteed to not be executed at run time are now correctly flagged as unreachable code errors. For instance, statements following unconditional `return`, `throw`, `break` or `continue` statements are considered unreachable. Use `--allowUnreachableCode` to disable unreachable code detection and reporting.

Example

Here's a simple example of an unreachable code error:

```
function f(x) {
  if (x) {
    return true;
  }
  else {
    return false;
  }

  x = 0; // Error: Unreachable code detected.
}
```

A more common error that this feature catches is adding a newline after a `return` statement:

```
function f() {
  return          // Automatic Semicolon Insertion triggered at newline
{
  x: "string"   // Error: Unreachable code detected.
}
}
```

Since JavaScript automatically terminates the `return` statement at the end of the line, the object literal becomes a block.

Unused labels

Unused labels are also flagged. Just like unreachable code checks, these are turned on by default; use `--allowUnusedLabels` to stop reporting these errors.

Example

```
loop: while (x > 0) { // Error: Unused label.
  x++;
}
```

Implicit returns

Functions with code paths that do not return a value in JS implicitly return `undefined`. These can now be flagged by the compiler as implicit returns. The check is turned off by default; use `--noImplicitReturns` to turn it on.

Example

```
function f(x) { // Error: Not all code paths return a value.
  if (x) {
    return false;
  }

  // implicitly returns `undefined`
}
```

Case clause fall-throughs

TypeScript can reports errors for fall-through cases in switch statement where the case clause is non-empty. This check is turned off by default, and can be enabled using `--noFallthroughCasesInSwitch`.

Example

With `--noFallthroughCasesInSwitch`, this example will trigger an error:

```
switch (x % 2) {
  case 0: // Error: Fallthrough case in switch.
    console.log("even");

  case 1:
    console.log("odd");
    break;
}
```

However, in the following example, no error will be reported because the fall-through case is empty:

```
switch (x % 3) {
  case 0:
  case 1:
    console.log("Acceptable");
    break;

  case 2:
    console.log("This is *two much*!");
    break;
}
```

Stateless Function Components in React

TypeScript now supports [Stateless Function components](#). These are lightweight components that easily compose other components:

```
// Use parameter destructuring and defaults for easy definition of 'props' type
const Greeter = ({name = 'world'}) => <div>Hello, {name}!</div>;

// Properties get validated
let example = <Greeter name='TypeScript 1.8' />;
```

For this feature and simplified props, be sure to be use the [latest version of react.d.ts](#).

Simplified `props` type management in React

In TypeScript 1.8 with the latest version of `react.d.ts` (see above), we've also greatly simplified the declaration of `props` types.

Specifically:

- You no longer need to either explicitly declare `ref` and `key` or extend `React.PropTypes`
- The `ref` and `key` properties will appear with correct types on all components
- The `ref` property is correctly disallowed on instances of Stateless Function components

Augmenting global/module scope from modules

Users can now declare any augmentations that they want to make, or that any other consumers already have made, to an existing module. Module augmentations look like plain old ambient module declarations (i.e. the `declare module "foo" { }` syntax), and are directly nested either your own modules, or in another top level ambient external module.

Furthermore, TypeScript also has the notion of *global* augmentations of the form `declare global { }`. This allows modules to augment global types such as `Array` if necessary.

The name of a module augmentation is resolved using the same set of rules as module specifiers in `import` and `export` declarations. The declarations in a module augmentation are merged with any existing declarations the same way they would if they were declared in the same file.

Neither module augmentations nor global augmentations can add new items to the top level scope - they can only "patch" existing declarations.

Example

Here `map.ts` can declare that it will internally patch the `observable` type from `observable.ts` and add the `map` method to it.

```
// observable.ts
export class Observable<T> {
    // ...
}

// map.ts
import { Observable } from "./observable";

// Create an augmentation for "./observable"
declare module "./observable" {

    // Augment the 'Observable' class definition with interface merging
    interface Observable<T> {
        map<U>(proj: (el: T) => U): Observable<U>;
    }
}

Observable.prototype.map = /*...*/;
```

```
// consumer.ts
import { Observable } from "./observable";
import "./map";

let o: Observable<number>;
o.map(x => x.toFixed());
```

Similarly, the global scope can be augmented from modules using a `declare global` declarations:

Example

```
// Ensure this is treated as a module.
export {};

declare global {
    interface Array<T> {
        mapToNumbers(): number[];
    }
}

Array.prototype.mapToNumbers = function () { /* ... */ }
```

String literal types

It's not uncommon for an API to expect a specific set of strings for certain values. For instance, consider a UI library that can move elements across the screen while controlling the "easing" of the animation.

```
declare class UIElement {
    animate(options: AnimationOptions): void;
}

interface AnimationOptions {
    deltaX: number;
    deltaY: number;
    easing: string; // Can be "ease-in", "ease-out", "ease-in-out"
}
```

However, this is error prone - there is nothing stopping a user from accidentally misspelling one of the valid easing values:

```
// No errors
new UIElement().animate({ deltaX: 100, deltaY: 100, easing: "ease-inout" });
```

With TypeScript 1.8, we've introduced string literal types. These types are written the same way string literals are, but in type positions.

Users can now ensure that the type system will catch such errors. Here's our new `AnimationOptions` using string literal types:

```
interface AnimationOptions {
    deltaX: number;
    deltaY: number;
    easing: "ease-in" | "ease-out" | "ease-in-out";
}

// Error: Type '"ease-inout"' is not assignable to type '"ease-in" | "ease-out" | "ease-in-out"'
new UIElement().animate({ deltaX: 100, deltaY: 100, easing: "ease-inout" });
```

Improved union/intersection type inference

TypeScript 1.8 improves type inference involving source and target sides that are both union or intersection types. For example, when inferring from `string | string[]` to `string | T`, we reduce the types to `string[]` and `T`, thus inferring `string[]` for `T`.

Example

```
type Maybe<T> = T | void;

function isDefined<T>(x: Maybe<T>): x is T {
    return x !== undefined && x !== null;
}

function isUndefined<T>(x: Maybe<T>): x is void {
    return x === undefined || x === null;
}

function getOrElse<T>(x: Maybe<T>, defaultValue: T): T {
    return isDefined(x) ? x : defaultValue;
}

function test1(x: Maybe<string>) {
    let x1 = getOrElse(x, "Undefined");           // string
    let x2 = isDefined(x) ? x : "Undefined";      // string
```

```
    let x3 = isUndefined(x) ? "Undefined" : x; // string
}

function test2(x: Maybe<number>) {
    let x1 = getOrElse(x, -1); // number
    let x2 = isDefined(x) ? x : -1; // number
    let x3 = isUndefined(x) ? -1 : x; // number
}
```

Concatenate AMD and System modules with -- outFile

Specifying `--outFile` in conjunction with `--module amd` or `--module system` will concatenate all modules in the compilation into a single output file containing multiple module closures.

A module name will be computed for each module based on its relative location to `rootDir`.

Example

```
// file src/a.ts
import * as B from "./lib/b";
export function createA() {
    return B.createB();
}
```

```
// file src/lib/b.ts
export function createB() {
    return {};
}
```

Results in:

```
define("lib/b", ["require", "exports"], function (require, exports) {
    "use strict";
    function createB() {
        return {};
    }
    exports.createB = createB;
});
define("a", ["require", "exports", "lib/b"], function (require, exports, B) {
    "use strict";
    function createA() {
        return B.createB();
    }
    exports.createA = createA;
});
```

Support for default import interop with SystemJS

Module loaders like SystemJS wrap CommonJS modules and expose them as a `default` ES6 import. This makes it impossible to share the definition files between the SystemJS and CommonJS implementation of the module as the module shape looks different based on the loader.

Setting the new compiler flag `--allowSyntheticDefaultImports` indicates that the module loader performs some kind of synthetic default import member creation not indicated in the imported .ts or .d.ts. The compiler will infer the existence of a `default` export that has the shape of the entire module itself.

System modules have this flag on by default.

Allow captured `let / const` in loops

Previously an error, now supported in TypeScript 1.8. `let / const` declarations within loops and captured in functions are now emitted to correctly match `let / const` freshness semantics.

Example

```
let list = [];
for (let i = 0; i < 5; i++) {
    list.push(() => i);
}

list.forEach(f => console.log(f()));
```

is compiled to:

```
var list = [];
var _loop_1 = function(i) {
    list.push(function () { return i; });
};
for (var i = 0; i < 5; i++) {
    _loop_1(i);
}
list.forEach(function (f) { return console.log(f()); });
```

And results in

```
0
1
2
3
4
```

Improved checking for `for..in` statements

Previously the type of a `for..in` variable is inferred to `any`; that allowed the compiler to ignore invalid uses within the `for..in` body.

Starting with TypeScript 1.8:

- The type of a variable declared in a `for..in` statement is implicitly `string`.
- When an object with a numeric index signature of type `T` (such as an array) is indexed by a `for..in` variable of a containing `for..in` statement for an object *with* a numeric index signature and *without* a string index signature (again such as an array), the value produced is of type `T`.

Example

```
var a: MyObject[];
for (var x in a) { // Type of x is implicitly string}
```

```
    var obj = a[x]; // Type of obj is MyObject
}
```

Modules are now emitted with a "use strict"; prologue

Modules were always parsed in strict mode as per ES6, but for non-ES6 targets this was not respected in the generated code. Starting with TypeScript 1.8, emitted modules are always in strict mode. This shouldn't have any visible changes in most code as TS considers most strict mode errors as errors at compile time, but it means that some things which used to silently fail at runtime in your TS code, like assigning to `Nan`, will now loudly fail. You can reference the [MDN Article](#) on strict mode for a detailed list of the differences between strict mode and non-strict mode.

Including .js files with --allowJs

Often there are external source files in your project that may not be authored in TypeScript. Alternatively, you might be in the middle of converting a JS code base into TS, but still want to bundle all your JS code into a single file with the output of your new TS code.

`.js` files are now allowed as input to `tsc`. The TypeScript compiler checks the input `.js` files for syntax errors, and emits valid output based on the `--target` and `--module` flags. The output can be combined with other `.ts` files as well. Source maps are still generated for `.js` files just like with `.ts` files.

Custom JSX factories using --reactNamespace

Passing `--reactNamespace <JSX factory Name>` along with `--jsx react` allows for using a different JSX factory from the default `React`.

The new factory name will be used to call `createElement` and `__spread` functions.

Example

```
import {jsxFactory} from "jsxFactory";

var div = <div>Hello JSX!</div>
```

Compiled with:

```
tsc --jsx react --reactNamespace jsxFactory --m commonJS
```

Results in:

```
"use strict";
var jsxFactory_1 = require("jsxFactory");
var div = jsxFactory_1.jsxFactory.createElement("div", null, "Hello JSX!");
```

this -based type guards

TypeScript 1.8 extends [user-defined type guard functions](#) to class and interface methods.

`this is T` is now valid return type annotation for methods in classes and interfaces. When used in a type narrowing position (e.g. `if` statement), the type of the call expression target object would be narrowed to `T`.

Example

```
class FileSystemObject {
    isFile(): this is File { return this instanceof File; }
    isDirectory(): this is Directory { return this instanceof Directory; }
    isNetworked(): this is (Networked & this) { return this.networked; }
    constructor(public path: string, private networked: boolean) {}
}

class File extends FileSystemObject {
    constructor(path: string, public content: string) { super(path, false); }
}
class Directory extends FileSystemObject {
    children: FileSystemObject[];
}
interface Networked {
    host: string;
}

let fso: FileSystemObject = new File("foo/bar.txt", "foo");
if (fso.isFile()) {
    fso.content; // fso is File
}
else if (fso.isDirectory()) {
    fso.children; // fso is Directory
}
else if (fso.isNetworked()) {
    fso.host; // fso is networked
}
```

Official TypeScript NuGet package

Starting with TypeScript 1.8, official NuGet packages are available for the Typescript Compiler (`tsc.exe`) as well as the MSBuild integration (`Microsoft.TypeScript.targets` and `Microsoft.TypeScript.Tasks.dll`).

Stable packages are available here:

- [Microsoft.TypeScript.Compiler](#)
- [Microsoft.TypeScript.MSBuild](#)

Also, a nightly NuGet package to match the [nightly npm package](#) is available on [myget](#):

- [TypeScript-Preview](#)

Prettier error messages from `tsc`

We understand that a ton of monochrome output can be a little difficult on the eyes. Colors can help discern where a message starts and ends, and these visual clues are important when error output gets overwhelming.

By just passing the `--pretty` command line option, TypeScript gives more colorful output with context about where things are going wrong.

```

Windows PowerShell (Admin)
PS C:\iFeelPretty> tsc --watch --pretty
18      await reticulateSplines();
~~~~~
ohSoPretty.ts(18,5): error TS1308: 'await' expression is only allowed within an async function.

13:32:44 - Compilation complete. Watching for file changes.
|
```

node.exe[*32]:4892 « 150513[64] 1/1 [+] NUM PRInt 99x23 (1,9) 25V 15108 100%

Colorization of JSX code in VS 2015

With TypeScript 1.8, JSX tags are now classified and colorized in Visual Studio 2015.



The classification can be further customized by changing the font and color settings for the `VB XML` color and font settings through `Tools -> Options -> Environment -> Fonts and Colors` page.

The `--project` (`-p`) flag can now take any file path

The `--project` command line option originally could only take paths to a folder containing a `tsconfig.json`. Given the different scenarios for build configurations, it made sense to allow `--project` to point to any other compatible JSON file. For instance, a user might want to target ES2015 with CommonJS modules for Node 5, but ES5 with AMD modules for the browser. With this new work, users can easily manage two separate build targets using `tsc` alone without having to perform hacky workarounds like placing `tsconfig.json` files in separate directories.

The old behavior still remains the same if given a directory - the compiler will try to find a file in the directory named `tsconfig.json`.

Allow comments in tsconfig.json

It's always nice to be able to document your configuration! `tsconfig.json` now accepts single and multi-line comments.

```
{
  "compilerOptions": {
    "target": "ES2015", // running on node v5, yaay!
    "sourceMap": true // makes debugging easier
  },
  /*
   * Excluded files
   */
  "exclude": [
    "file.d.ts"
  ]
}
```

Support output to IPC-driven files

TypeScript 1.8 allows users to use the `--outFile` argument with special file system entities like named pipes, devices, etc.

As an example, on many Unix-like systems, the standard output stream is accessible by the file `/dev/stdout`.

```
tsc foo.ts --outFile /dev/stdout
```

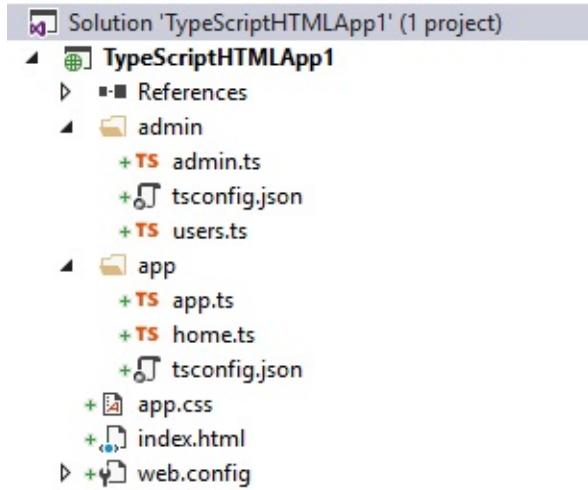
This can be used to pipe output between commands as well.

As an example, we can pipe our emitted JavaScript into a pretty printer like [pretty-js](#):

```
tsc foo.ts --outFile /dev/stdout | pretty-js
```

Improved support for `tsconfig.json` in Visual Studio 2015

TypeScript 1.8 allows `tsconfig.json` files in all project types. This includes ASP.NET v4 projects, *Console Application*, and the *Html Application with TypeScript* project types. Further, you are no longer limited to a single `tsconfig.json` file but can add multiple, and each will be built as part of the project. This allows you to separate the configuration for different parts of your application without having to use multiple different projects.



We also disable the project properties page when you add a `tsconfig.json` file. This means that all configuration changes have to be made in the `tsconfig.json` file itself.

A couple of limitations

- If you add a `tsconfig.json` file, TypeScript files that are not considered part of that context are not compiled.
- Apache Cordova Apps still have the existing limitation of a single `tsconfig.json` file, which must be in either the root or the `scripts` folder.
- There is no template for `tsconfig.json` in most project types.

async / await support in ES6 targets (Node v4+)

TypeScript now supports asynchronous functions for engines that have native support for ES6 generators, e.g. Node v4 and above. Asynchronous functions are prefixed with the `async` keyword; `await` suspends the execution until an asynchronous function return promise is fulfilled and unwraps the value from the `Promise` returned.

Example

In the following example, each input element will be printed out one at a time with a 400ms delay:

```
"use strict";

// printDelayed is a 'Promise<void>'
async function printDelayed(elements: string[]) {
    for (const element of elements) {
        await delay(400);
        console.log(element);
    }
}

async function delay(milliseconds: number) {
    return new Promise<void>(resolve => {
        setTimeout(resolve, milliseconds);
    });
}

printDelayed(["Hello", "beautiful", "asynchronous", "world"]).then(() => {
    console.log();
    console.log("Printed every element!");
});
```

For more information see [Async Functions](#) blog post.

Support for --target ES6 with --module

TypeScript 1.7 adds `ES6` to the list of options available for the `--module` flag and allows you to specify the module output when targeting `ES6`. This provides more flexibility to target exactly the features you want in specific runtimes.

Example

```
{
    "compilerOptions": {
        "module": "amd",
        "target": "es6"
    }
}
```

this -typing

It is a common pattern to return the current object (i.e. `this`) from a method to create [fluent-style APIs](#). For instance, consider the following `BasicCalculator` module:

```
export default class BasicCalculator {
    public constructor(protected value: number = 0) { }
```

```

public currentValue(): number {
    return this.value;
}

public add(operand: number) {
    this.value += operand;
    return this;
}

public subtract(operand: number) {
    this.value -= operand;
    return this;
}

public multiply(operand: number) {
    this.value *= operand;
    return this;
}

public divide(operand: number) {
    this.value /= operand;
    return this;
}
}

```

A user could express `2 * 5 + 1` as

```

import calc from "./BasicCalculator";

let v = new calc(2)
    .multiply(5)
    .add(1)
    .currentValue();

```

This often opens up very elegant ways of writing code; however, there was a problem for classes that wanted to extend from `BasicCalculator`. Imagine a user wanted to start writing a `ScientificCalculator`:

```

import BasicCalculator from "./BasicCalculator";

export default class ScientificCalculator extends BasicCalculator {
    public constructor(value = 0) {
        super(value);
    }

    public square() {
        this.value = this.value ** 2;
        return this;
    }

    public sin() {
        this.value = Math.sin(this.value);
        return this;
    }
}

```

Because TypeScript used to infer the type `BasicCalculator` for each method in `BasicCalculator` that returned `this`, the type system would forget that it had `ScientificCalculator` whenever using a `BasicCalculator` method.

For instance:

```

import calc from "./ScientificCalculator";

let v = new calc(0.5)

```

```
.square()  
.divide(2)  
.sin() // Error: 'BasicCalculator' has no 'sin' method.  
.currentValue();
```

This is no longer the case - TypeScript now infers `this` to have a special type called `this` whenever inside an instance method of a class. The `this` type is written as so, and basically means "the type of the left side of the dot in a method call".

The `this` type is also useful with intersection types in describing libraries (e.g. Ember.js) that use mixin-style patterns to describe inheritance:

```
interface MyType {  
    extend<T>(other: T): this & T;  
}
```

ES7 exponentiation operator

TypeScript 1.7 supports upcoming [ES7/ES2016 exponentiation operators](#): `**` and `**=`. The operators will be transformed in the output to ES3/ES5 using `Math.pow`.

Example

```
var x = 2 ** 3;  
var y = 10;  
y **= 2;  
var z = -(4 ** 3);
```

Will generate the following JavaScript output:

```
var x = Math.pow(2, 3);  
var y = 10;  
y = Math.pow(y, 2);  
var z = -(Math.pow(4, 3));
```

Improved checking for destructuring object literal

TypeScript 1.7 makes checking of destructuring patterns with an object literal or array literal initializers less rigid and more intuitive.

When an object literal is contextually typed by the implied type of an object binding pattern:

- Properties with default values in the object binding pattern become optional in the object literal.
- Properties in the object binding pattern that have no match in the object literal are required to have a default value in the object binding pattern and are automatically added to the object literal type.
- Properties in the object literal that have no match in the object binding pattern are an error.

When an array literal is contextually typed by the implied type of an array binding pattern:

- Elements in the array binding pattern that have no match in the array literal are required to have a default value in the array binding pattern and are automatically added to the array literal type.

Example

```
// Type of f1 is (arg?: { x?: number, y?: number }) => void
function f1({ x = 0, y = 0 } = {}) { }

// And can be called as:
f1();
f1({});
f1({ x: 1 });
f1({ y: 1 });
f1({ x: 1, y: 1 });

// Type of f2 is (arg?: { x: number, y?: number }) => void
function f2({ x, y = 0 } = { x: 0 }) { }

f2();
f2({});           // Error, x not optional
f2({ x: 1 });
f2({ y: 1 });    // Error, x not optional
f2({ x: 1, y: 1 });
```

Support for decorators when targeting ES3

Decorators are now allowed when targeting ES3. TypeScript 1.7 removes the ES5-specific use of `reduceRight` from the `_decorate` helper. The changes also inline calls `Object.getOwnPropertyDescriptor` and `Object.defineProperty` in a backwards-compatible fashion that allows for a to clean up the emit for ES5 and later by removing various repetitive calls to the aforementioned `Object` methods.

JSX support

JSX is an embeddable XML-like syntax. It is meant to be transformed into valid JavaScript, but the semantics of that transformation are implementation-specific. JSX came to popularity with the React library but has since seen other applications. TypeScript 1.6 supports embedding, type checking, and optionally compiling JSX directly into JavaScript.

New `.tsx` file extension and `as` operator

TypeScript 1.6 introduces a new `.tsx` file extension. This extension does two things: it enables JSX inside of TypeScript files, and it makes the new `as` operator the default way to cast (removing any ambiguity between JSX expressions and the TypeScript prefix cast operator). For example:

```
var x = <any> foo;
// is equivalent to:
var x = foo as any;
```

Using React

To use JSX-support with React you should use the [React typings](#). These typings define the `jsx` namespace so that TypeScript can correctly check JSX expressions for React. For example:

```
/// <reference path="react.d.ts" />

interface Props {
    name: string;
}

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

<MyComponent name="bar" />; // OK
<MyComponent name={0} />; // error, `name` is not a number
```

Using other JSX frameworks

JSX element names and properties are validated against the `jsx` namespace. Please see the [\[\[JSX\]\]](#) wiki page for defining the `jsx` namespace for your framework.

Output generation

TypeScript ships with two JSX modes: `preserve` and `react`.

- The `preserve` mode will keep JSX expressions as part of the output to be further consumed by another transform step. *Additionally the output will have a `.jsx` file extension.*
- The `react` mode will emit `React.createElement`, does not need to go through a JSX transformation before use, and the output will have a `.js` file extension.

See the [\[\[JSX\]\]](#) wiki page for more information on using JSX in TypeScript.

Intersection types

TypeScript 1.6 introduces intersection types, the logical complement of union types. A union type `A | B` represents an entity that is either of type `A` or type `B`, whereas an intersection type `A & B` represents an entity that is both of type `A` *and* type `B`.

Example

```
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 });
var s = x.a;
var n = x.b;
```

```
type LinkedList<T> = T & { next: LinkedList<T> };

interface Person {
    name: string;
}

var people: LinkedList<Person>;
var s = people.name;
var s = people.next.name;
var s = people.next.next.name;
var s = people.next.next.next.name;
```

```
interface A { a: string }
interface B { b: string }
interface C { c: string }

var abc: A & B & C;
abc.a = "hello";
abc.b = "hello";
abc.c = "hello";
```

See [issue #1256](#) for more information.

Local type declarations

Local class, interface, enum, and type alias declarations can now appear inside function declarations. Local types are block scoped, similar to variables declared with `let` and `const`. For example:

```
function f() {
    if (true) {
        interface T { x: number }
        let v: T;
        v.x = 5;
    }
    else {
        interface T { x: string }
```

```

        let v: T;
        v.x = "hello";
    }
}

```

The inferred return type of a function may be a type declared locally within the function. It is not possible for callers of the function to reference such a local type, but it can of course be matched structurally. For example:

```

interface Point {
    x: number;
    y: number;
}

function getPointFactory(x: number, y: number) {
    class P {
        x = x;
        y = y;
    }
    return P;
}

var PointZero = getPointFactory(0, 0);
var PointOne = getPointFactory(1, 1);
var p1 = new PointZero();
var p2 = new PointZero();
var p3 = new PointOne();

```

Local types may reference enclosing type parameters and local class and interfaces may themselves be generic. For example:

```

function f3() {
    function f<X, Y>(x: X, y: Y) {
        class C {
            public x = x;
            public y = y;
        }
        return C;
    }
    let C = f(10, "hello");
    let v = new C();
    let x = v.x; // number
    let y = v.y; // string
}

```

Class expressions

TypeScript 1.6 adds support for ES6 class expressions. In a class expression, the class name is optional and, if specified, is only in scope in the class expression itself. This is similar to the optional name of a function expression. It is not possible to refer to the class instance type of a class expression outside the class expression, but the type can of course be matched structurally. For example:

```

let Point = class {
    constructor(public x: number, public y: number) { }
    public length() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
};
var p = new Point(3, 4); // p has anonymous class type
console.log(p.length());

```

Extending expressions

TypeScript 1.6 adds support for classes extending arbitrary expression that computes a constructor function. This means that built-in types can now be extended in class declarations.

The `extends` clause of a class previously required a type reference to be specified. It now accepts an expression optionally followed by a type argument list. The type of the expression must be a constructor function type with at least one construct signature that has the same number of type parameters as the number of type arguments specified in the `extends` clause. The return type of the matching construct signature(s) is the base type from which the class instance type inherits. Effectively, this allows both real classes and "class-like" expressions to be specified in the `extends` clause.

Some examples:

```
// Extend built-in types

class MyArray extends Array<number> { }
class MyError extends Error { }

// Extend computed base class

class ThingA {
    getGreeting() { return "Hello from A"; }
}

class ThingB {
    getGreeting() { return "Hello from B"; }
}

interface Greeter {
    getGreeting(): string;
}

interface GreeterConstructor {
    new (): Greeter;
}

function getGreeterBase(): GreeterConstructor {
    return Math.random() >= 0.5 ? ThingA : ThingB;
}

class Test extends getGreeterBase() {
    sayHello() {
        console.log(this.getGreeting());
    }
}
```

abstract classes and methods

TypeScript 1.6 adds support for `abstract` keyword for classes and their methods. An abstract class is allowed to have methods with no implementation, and cannot be constructed.

Examples

```
abstract class Base {
    abstract getThing(): string;
    getOtherThing() { return 'hello'; }
}

let x = new Base(); // Error, 'Base' is abstract
```

```
// Error, must either be 'abstract' or implement concrete 'getThing'
class Derived1 extends Base { }

class Derived2 extends Base {
    getThing() { return 'hello'; }
    foo() {
        super.getThing(); // Error: cannot invoke abstract members through 'super'
    }
}

var x = new Derived2(); // OK
var y: Base = new Derived2(); // Also OK
y.getThing(); // OK
y.getOtherThing(); // OK
```

Generic type aliases

With TypeScript 1.6, type aliases can be generic. For example:

```
type Lazy<T> = T | (() => T);

var s: Lazy<string>;
s = "eager";
s = () => "lazy";

interface Tuple<A, B> {
    a: A;
    b: B;
}

type Pair<T> = Tuple<T, T>;
```

Stricter object literal assignment checks

TypeScript 1.6 enforces stricter object literal assignment checks for the purpose of catching excess or misspelled properties. Specifically, when a fresh object literal is assigned to a variable or passed as an argument for a non-empty target type, it is an error for the object literal to specify properties that don't exist in the target type.

Examples

```
var x: { foo: number };
x = { foo: 1, baz: 2 }; // Error, excess property `baz`

var y: { foo: number, bar?: number };
y = { foo: 1, baz: 2 }; // Error, excess or misspelled property `baz`
```

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
```

ES6 generators

TypeScript 1.6 adds support for generators when targeting ES6.

A generator function can have a return type annotation, just like a function. The annotation represents the type of the generator returned by the function. Here is an example:

```
function *g(): Iterable<string> {
    for (var i = 0; i < 100; i++) {
        yield ""; // string is assignable to string
    }
    yield * otherStringGenerator(); // otherStringGenerator must be iterable and element type assignable to str
    ing
}
```

A generator function with no type annotation can have the type annotation inferred. So in the following case, the type will be inferred from the `yield` statements:

```
function *g() {
    for (var i = 0; i < 100; i++) {
        yield ""; // infer string
    }
    yield * otherStringGenerator(); // infer element type of otherStringGenerator
}
```

Experimental support for `async` functions

TypeScript 1.6 introduces experimental support of `async` functions when targeting ES6. Async functions are expected to invoke an asynchronous operation and await its result without blocking normal execution of the program. This is accomplished through the use of an ES6-compatible `Promise` implementation, and transposition of the function body into a compatible form to resume execution when the awaited asynchronous operation completes.

An *async function* is a function or method that has been prefixed with the `async` modifier. This modifier informs the compiler that function body transposition is required, and that the keyword `await` should be treated as a unary expression instead of an identifier. An *Async Function* must provide a return type annotation that points to a compatible `Promise` type. Return type inference can only be used if there is a globally defined, compatible `Promise` type.

Example

```
var p: Promise<number> = /* ... */;
async function fn(): Promise<number> {
    var i = await p; // suspend execution until 'p' is settled. 'i' has type "number"
    return 1 + i;
}

var a = async (): Promise<number> => 1 + await p; // suspends execution.
var a = async () => 1 + await p; // suspends execution. return type is inferred as "Promise<number>" when compi
ling with --target ES6
var fe = async function(): Promise<number> {
    var i = await p; // suspend execution until 'p' is settled. 'i' has type "number"
    return 1 + i;
}

class C {
    async m(): Promise<number> {
        var i = await p; // suspend execution until 'p' is settled. 'i' has type "number"
        return 1 + i;
    }

    async get p(): Promise<number> {
        var i = await p; // suspend execution until 'p' is settled. 'i' has type "number"
        return 1 + i;
    }
}
```

```

    }
}
```

Nightly builds

While not strictly a language change, nightly builds are now available by installing with the following command:

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

Adjustments in module resolution logic

Starting from release 1.6 TypeScript compiler will use different set of rules to resolve module names when targeting 'commonjs'. These [rules](#) attempted to model module lookup procedure used by Node. This effectively mean that node modules can include information about its typings and TypeScript compiler will be able to find it. User however can override module resolution rules picked by the compiler by using `--moduleResolution` command line option. Possible values are:

- 'classic' - module resolution rules used by pre 1.6 TypeScript compiler
- 'node' - node-like module resolution

Merging ambient class and interface declaration

The instance side of an ambient class declaration can be extended using an interface declaration. The class constructor object is unmodified. For example:

```

declare class Foo {
  public x : number;
}

interface Foo {
  y : string;
}

function bar(foo : Foo) {
  foo.x = 1; // OK, declared in the class Foo
  foo.y = "1"; // OK, declared in the interface Foo
}
```

User-defined type guard functions

TypeScript 1.6 adds a new way to narrow a variable type inside an `if` block, in addition to `typeof` and `instanceof`. A user-defined type guard function is one with a return type annotation of the form `x is T`, where `x` is a declared parameter in the signature, and `T` is any type. When a user-defined type guard function is invoked on a variable in an `if` block, the type of the variable will be narrowed to `T`.

Examples

```

function isCat(a: any): a is Cat {
  return a.name === 'kitty';
}
```

```
var x: Cat | Dog;
if(isCat(x)) {
  x.meow(); // OK, x is Cat in this block
}
```

exclude property support in tsconfig.json

A tsconfig.json file that doesn't specify a files property (and therefore implicitly references all *.ts files in all subdirectories) can now contain an exclude property that specifies a list of files and/or directories to exclude from the compilation. The exclude property must be an array of strings that each specify a file or folder name relative to the location of the tsconfig.json file. For example:

```
{
  "compilerOptions": {
    "out": "test.js"
  },
  "exclude": [
    "node_modules",
    "test.ts",
    "utils/t2.ts"
  ]
}
```

The `exclude` list does not support wildcards. It must simply be a list of files and/or directories.

--init command line option

Run `tsc --init` in a directory to create an initial `tsconfig.json` in this directory with preset defaults. Optionally pass command line arguments along with `--init` to be stored in your initial `tsconfig.json` on creation.

ES6 Modules

TypeScript 1.5 supports ECMAScript 6 (ES6) modules. ES6 modules are effectively TypeScript external modules with a new syntax: ES6 modules are separately loaded source files that possibly import other modules and provide a number of externally accessible exports. ES6 modules feature several new export and import declarations. It is recommended that TypeScript libraries and applications be updated to use the new syntax, but this is not a requirement. The new ES6 module syntax coexists with TypeScript's original internal and external module constructs and the constructs can be mixed and matched at will.

Export Declarations

In addition to the existing TypeScript support for decorating declarations with `export`, module members can also be exported using separate export declarations, optionally specifying different names for exports using `as` clauses.

```
interface Stream { ... }
function writeToStream(stream: Stream, data: string) { ... }
export { Stream, writeToStream as write }; // writeToStream exported as write
```

Import declarations, as well, can optionally use `as` clauses to specify different local names for the imports. For example:

```
import { read, write, standardOutput as stdout } from "./inout";
var s = read(stdout);
write(stdout, s);
```

As an alternative to individual imports, a namespace import can be used to import an entire module:

```
import * as io from "./inout";
var s = io.read(io.standardOutput);
io.write(io.standardOutput, s);
```

Re-exporting

Using `from` clause a module can copy the exports of a given module to the current module without introducing local names.

```
export { read, write, standardOutput as stdout } from "./inout";
```

`export *` can be used to re-export all exports of another module. This is useful for creating modules that aggregate the exports of several other modules.

```
export function transform(s: string): string { ... }
export * from "./mod1";
export * from "./mod2";
```

Default Export

An export default declaration specifies an expression that becomes the default export of a module:

```
export default class Greeter {
    sayHello() {
```

```
    console.log("Greetings!");
}
}
```

Which in turn can be imported using default imports:

```
import Greeter from "./greeter";
var g = new Greeter();
g.sayHello();
```

Bare Import

A "bare import" can be used to import a module only for its side-effects.

```
import "./polyfills";
```

For more information about module, please see the [ES6 module support spec](#).

Destructuring in declarations and assignments

TypeScript 1.5 adds support to ES6 destructuring declarations and assignments.

Declarations

A destructuring declaration introduces one or more named variables and initializes them with values extracted from properties of an object or elements of an array.

For example, the following sample declares variables `x`, `y`, and `z`, and initializes them to `getSomeObject().x`, `getSomeObject().y` and `getSomeObject().z` respectively:

```
var { x, y, z } = getSomeObject();
```

Destructuring declarations also works for extracting values from arrays:

```
var [x, y, z = 10] = getSomeArray();
```

Similarly, destructuring can be used in function parameter declarations:

```
function drawText({ text = "", location: [x, y] = [0, 0], bold = false }) {
    // Draw text
}

// Call drawText with an object literal
var item = { text: "someText", location: [1,2,3], style: "italics" };
drawText(item);
```

Assignments

Destructuring patterns can also be used in regular assignment expressions. For instance, swapping two variables can be written as a single destructuring assignment:

```
var x = 1;
var y = 2;
```

```
[x, y] = [y, x];
```

namespace keyword

TypeScript used the `module` keyword to define both "internal modules" and "external modules"; this has been a bit of confusion for developers new to TypeScript. "Internal modules" are closer to what most people would call a namespace; likewise, "external modules" in JS speak really just are modules now.

Note: Previous syntax defining internal modules are still supported.

Before:

```
module Math {  
    export function add(x, y) { ... }  
}
```

After:

```
namespace Math {  
    export function add(x, y) { ... }  
}
```

let and const support

ES6 `let` and `const` declarations are now supported when targeting ES3 and ES5.

Const

```
const MAX = 100;  
  
++MAX; // Error: The operand of an increment or decrement  
//          operator cannot be a constant.
```

Block scoped

```
if (true) {  
    let a = 4;  
    // use a  
}  
else {  
    let a = "string";  
    // use a  
}  
  
alert(a); // Error: a is not defined in this scope
```

for..of support

TypeScript 1.5 adds support to ES6 `for..of` loops on arrays for ES3/ES5 as well as full support for Iterator interfaces when targetting ES6.

Example

The TypeScript compiler will transpile for..of arrays to idiomatic ES3/ES5 JavaScript when targeting those versions:

```
for (var v of expr) { }
```

will be emitted as:

```
for (var _i = 0, _a = expr; _i < _a.length; _i++) {
    var v = _a[_i];
}
```

Decorators

TypeScript decorators are based on the [ES7 decorator proposal](#).

A decorator is:

- an expression
- that evaluates to a function
- that takes the target, name, and property descriptor as arguments
- and optionally returns a property descriptor to install on the target object

For more information, please see the [Decorators](#) proposal.

Example

Decorators `readonly` and `enumerable(false)` will be applied to the property `method` before it is installed on class `C`. This allows the decorator to change the implementation, and in this case, augment the descriptor to be `writable: false` and `enumerable: false`.

```
class C {
    @readonly
    @enumerable(false)
    method() { }
}

function readonly(target, key, descriptor) {
    descriptor.writable = false;
}

function enumerable(value) {
    return function (target, key, descriptor) {
        descriptor.enumerable = value;
    }
}
```

Computed properties

Initializing an object with dynamic properties can be a bit of a burden. Take the following example:

```
type NeighborMap = { [name: string]: Node };
type Node = { name: string; neighbors: NeighborMap };

function makeNode(name: string, initialNeighbor: Node): Node {
    var neighbors: NeighborMap = {};
    neighbors[name] = initialNeighbor;
    return { ...initialNeighbor, neighbors };
}
```

```

    neighbors[initialNeighbor.name] = initialNeighbor;
    return { name: name, neighbors: neighbors };
}

```

Here we need to create a variable to hold on to the neighbor-map so that we can initialize it. With TypeScript 1.5, we can let the compiler do the heavy lifting:

```

function makeNode(name: string, initialNeighbor: Node): Node {
    return {
        name: name,
        neighbors: {
            [initialNeighbor.name]: initialNeighbor
        }
    }
}

```

Support for UMD and System module output

In addition to `AMD` and `commonjs` module loaders, TypeScript now supports emitting modules `UMD` ([Universal Module Definition](#)) and `System` module formats.

Usage:

```
tsc --module umd
```

and

```
tsc --module system
```

Unicode codepoint escapes in strings

ES6 introduces escapes that allow users to represent a Unicode codepoint using just a single escape.

As an example, consider the need to escape a string that contains the character '𠮷'. In UTF-16/UCS2, '𠮷' is represented as a surrogate pair, meaning that it's encoded using a pair of 16-bit code units of values, specifically `0xD842` and `0xDFB7`. Previously this meant that you'd have to escape the codepoint as `"\uD842\uDFB7"`. This has the major downside that it's difficult to discern two independent characters from a surrogate pair.

With ES6's codepoint escapes, you can cleanly represent that exact character in strings and template strings with a single escape: `"\u{20bb7}"`. TypeScript will emit the string in ES3/ES5 as `"\uD842\uDFB7"`.

Tagged template strings in ES3/ES5

In TypeScript 1.4, we added support for template strings for all targets, and tagged templates for just ES6. Thanks to some considerable work done by [@ivogabe](#), we bridged the gap for tagged templates in ES3 and ES5.

When targeting ES3/ES5, the following code

```

function oddRawStrings(strs: TemplateStringsArray, n1, n2) {
    return strs.raw.filter((raw, index) => index % 2 === 1);
}

oddRawStrings `Hello \n${123} \t ${456}\n world`

```

will be emitted as

```
function oddRawStrings(strs, n1, n2) {
    return strs.raw.filter(function (raw, index) {
        return index % 2 === 1;
    });
}
_a = ["Hello \n", " \t ", "\n world"], _a.raw = ["Hello \\n", " \\t ", "\\n world"], oddRawStrings(_a, 123, 456
));
var _a;
```

AMD-dependency optional names

`/// <amd-dependency path="x" />` informs the compiler about a non-TS module dependency that needs to be injected in the resulting module's require call; however, there was no way to consume this module in the TS code.

The new `amd-dependency name` property allows passing an optional name for an amd-dependency:

```
/// <amd-dependency path="legacy/moduleA" name="moduleA"/>
declare var moduleA:MyType
moduleA.callStuff()
```

Generated JS code:

```
define(["require", "exports", "legacy/moduleA"], function (require, exports, moduleA) {
    moduleA.callStuff()
});
```

Project support through `tsconfig.json`

Adding a `tsconfig.json` file in a directory indicates that the directory is the root of a TypeScript project. The `tsconfig.json` file specifies the root files and the compiler options required to compile the project. A project is compiled in one of the following ways:

- By invoking `tsc` with no input files, in which case the compiler searches for the `tsconfig.json` file starting in the current directory and continuing up the parent directory chain.
- By invoking `tsc` with no input files and a `-project` (or just `-p`) command line option that specifies the path of a directory containing a `tsconfig.json` file.

Example

```
{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "sourceMap": true,
  }
}
```

See the [tsconfig.json wiki page](#) for more details.

--rootDir command line option

Option `--outDir` duplicates the input hierarchy in the output. The compiler computes the root of the input files as the longest common path of all input files; and then uses that to replicate all its substructure in the output.

Sometimes this is not desirable, for instance inputs `FolderA\FolderB\1.ts` and `FolderA\FolderB\2.ts` would result in output structure mirroring `FolderA\FolderB\`. Now if a new file `FolderA\3.ts` is added to the input, the output structure will pop out to mirror `FolderA\`.

`--rootDir` specifies the input directory to be mirrored in output instead of computing it.

--noEmitHelpers command line option

The TypeScript compiler emits a few helpers like `__extends` when needed. The helpers are emitted in every file they are referenced in. If you want to consolidate all helpers in one place, or override the default behavior, use `--noEmitHelpers` to instructs the compiler not to emit them.

--newLine command line option

By default the output new line character is `\r\n` on Windows based systems and `\n` on *nix based systems. `--newLine` command line flag allows overriding this behavior and specifying the new line character to be used in generated output files.

--inlineSourceMap and inlineSources command line options

`--inlineSourceMap` causes source map files to be written inline in the generated `.js` files instead of in a independent `.js.map` file. `--inlineSources` allows for additionally inlining the source `.ts` file into the `.js` file.

Union types

Overview

Union types are a powerful way to express a value that can be one of several types. For example, you might have an API for running a program that takes a commandline as either a `string`, a `string[]` or a function that returns a `string`. You can now write:

```
interface RunOptions {
    program: string;
    commandline: string[] | string | (() => string);
}
```

Assignment to union types works very intuitively -- anything you could assign to one of the union type's members is assignable to the union:

```
var opts: RunOptions = /* ... */;
opts.commandline = '-hello world'; // OK
opts.commandline = ['-hello', 'world']; // OK
opts.commandline = [42]; // Error, number is not string or string[]
```

When reading from a union type, you can see any properties that are shared by them:

```
if (opts.length === 0) { // OK, string and string[] both have 'length' property
    console.log("it's empty");
}
```

Using Type Guards, you can easily work with a variable of a union type:

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

Stricter Generics

With union types able to represent a wide range of type scenarios, we've decided to improve the strictness of certain generic calls. Previously, code like this would (surprisingly) compile without error:

```
function equal<T>(lhs: T, rhs: T): boolean {
    return lhs === rhs;
}

// Previously: No error
// New behavior: Error, no best common type between 'string' and 'number'
var e = equal(42, 'hello');
```

With union types, you can now specify the desired behavior at both the function declaration site and the call site:

```
// 'choose' function where types must match
```

```

function choose1<T>(a: T, b: T): T { return Math.random() > 0.5 ? a : b }
var a = choose1('hello', 42); // Error
var b = choose1<string|number>('hello', 42); // OK

// 'choose' function where types need not match
function choose2<T, U>(a: T, b: U): T|U { return Math.random() > 0.5 ? a : b }
var c = choose2('bar', 'foo'); // OK, c: string
var d = choose2('hello', 42); // OK, d: string|number

```

Better Type Inference

Union types also allow for better type inference in arrays and other places where you might have multiple kinds of values in a collection:

```

var x = [1, 'hello']; // x: Array<string|number>
x[0] = 'world'; // OK
x[0] = false; // Error, boolean is not string or number

```

let declarations

In JavaScript, `var` declarations are "hoisted" to the top of their enclosing scope. This can result in confusing bugs:

```

console.log(x); // meant to write 'y' here
/* later in the same block */
var x = 'hello';

```

The new ES6 keyword `let`, now supported in TypeScript, declares a variable with more intuitive "block" semantics. A `let` variable can only be referred to after its declaration, and is scoped to the syntactic block where it is defined:

```

if (foo) {
    console.log(x); // Error, cannot refer to x before its declaration
    let x = 'hello';
}
else {
    console.log(x); // Error, x is not declared in this block
}

```

`let` is only available when targeting ECMAScript 6 (`--target ES6`).

const declarations

The other new ES6 declaration type supported in TypeScript is `const`. A `const` variable may not be assigned to, and must be initialized where it is declared. This is useful for declarations where you don't want to change the value after its initialization:

```

const halfPi = Math.PI / 2;
halfPi = 2; // Error, can't assign to a `const`

```

`const` is only available when targeting ECMAScript 6 (`--target ES6`).

Template strings

TypeScript now supports ES6 template strings. These are an easy way to embed arbitrary expressions in strings:

```
var name = "TypeScript";
var greeting = `Hello, ${name}! Your name has ${name.length} characters`;
```

When compiling to pre-ES6 targets, the string is decomposed:

```
var name = "TypeScript!";
var greeting = "Hello, " + name + "! Your name has " + name.length + " characters";
```

Type Guards

A common pattern in JavaScript is to use `typeof` or `instanceof` to examine the type of an expression at runtime. TypeScript now understands these conditions and will change type inference accordingly when used in an `if` block.

Using `typeof` to test a variable:

```
var x: any = /* ... */;
if(typeof x === 'string') {
    console.log(x.substr(1)); // Error, 'substr' does not exist on 'string'
}
// x is still any here
x.unknown(); // OK
```

Using `typeof` with union types and `else`:

```
var x: string | HTMLElement = /* ... */;
if(typeof x === 'string') {
    // x is string here, as shown above
}
else {
    // x is HTMLElement here
    console.log(x.innerHTML);
}
```

Using `instanceof` with classes and union types:

```
class Dog { woof() {} }
class Cat { meow() {} }
var pet: Dog|Cat = /* ... */;
if (pet instanceof Dog) {
    pet.woof(); // OK
}
else {
    pet.woof(); // Error
}
```

Type Aliases

You can now define an *alias* for a type using the `type` keyword:

```
type PrimitiveArray = Array<string|number|boolean>;
type MyNumber = number;
type NgScope = ng.IScope;
type Callback = () => void;
```

Type aliases are exactly the same as their original types; they are simply alternative names.

const enum (completely inlined enums)

Enums are very useful, but some programs don't actually need the generated code and would benefit from simply inlining all instances of enum members with their numeric equivalents. The new `const enum` declaration works just like a regular `enum` for type safety, but erases completely at compile time.

```
const enum Suit { Clubs, Diamonds, Hearts, Spades }
var d = Suit.Diamonds;
```

Compiles to exactly:

```
var d = 1;
```

TypeScript will also now compute enum values when possible:

```
enum MyFlags {
    None = 0,
    Neat = 1,
    Cool = 2,
    Awesome = 4,
    Best = Neat | Cool | Awesome
}
var b = MyFlags.Best; // emits var b = 7;
```

-noEmitOnError commandline option

The default behavior for the TypeScript compiler is to still emit .js files if there were type errors (for example, an attempt to assign a `string` to a `number`). This can be undesirable on build servers or other scenarios where only output from a "clean" build is desired. The new flag `noEmitOnError` prevents the compiler from emitting .js code if there were any errors.

This is now the default for MSBuild projects; this allows MSBuild incremental build to work as expected, as outputs are only generated on clean builds.

AMD Module names

By default AMD modules are generated anonymous. This can lead to problems when other tools are used to process the resulting modules like a bundlers (e.g. `r.js`).

The new `amd-module name` tag allows passing an optional module name to the compiler:

```
//// [amdModule.ts]
///<amd-module name='NamedModule'>
export class C { }
```

Will result in assigning the name `NamedModule` to the module as part of calling the AMD `define`:

```
//// [amdModule.js]
```

```
define("NamedModule", ["require", "exports"], function (require, exports) {
    var C = (function () {
        function C() {
        }
        return C;
    })();
    exports.C = C;
});
```

Protected

The new `protected` modifier in classes works like it does in familiar languages like C++, C#, and Java. A `protected` member of a class is visible only inside subclasses of the class in which it is declared:

```
class Thing {
  protected doSomething() { /* ... */ }
}

class MyThing extends Thing {
  public myMethod() {
    // OK, can access protected member from subclass
    this.doSomething();
  }
}
var t = new MyThing();
t.doSomething(); // Error, cannot call protected member from outside class
```

Tuple types

Tuple types express an array where the type of certain elements is known, but need not be the same. For example, you may want to represent an array with a `string` at position 0 and a `number` at position 1:

```
// Declare a tuple type
var x: [string, number];
// Initialize it
x = ['hello', 10]; // OK
// Initialize it incorrectly
x = [10, 'hello']; // Error
```

When accessing an element with a known index, the correct type is retrieved:

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'
```

Note that in TypeScript 1.4, when accessing an element outside the set of known indices, a union type is used instead:

```
x[3] = 'world'; // OK
console.log(x[5].toString()); // OK, 'string' and 'number' both have toString
x[6] = true; // Error, boolean isn't number or string
```

Performance Improvements

The 1.1 compiler is typically around 4x faster than any previous release. See [this blog post](#) for some impressive charts.

Better Module Visibility Rules

TypeScript now only strictly enforces the visibility of types in modules if the `--declaration` flag is provided. This is very useful for Angular scenarios, for example:

```
module MyControllers {
    interface ZooScope extends ng.IScope {
        animals: Animal[];
    }
    export class ZooController {
        // Used to be an error (cannot expose ZooScope), but now is only
        // an error when trying to generate .d.ts files
        constructor(public $scope: ZooScope) { }
        /* more code */
    }
}
```