Getting set up

First you'll need to install TypeScript 1.4 from `npm`.

Once that's done, you'll need to link it from wherever your project resides. If you don't link from within a Node project, it will just link globally.

```
npm install -g typescript
npm link typescript
```

Once that's done, just grab our definitions file either

- By using tsd with the command `tsd query typescript --action install`.
- Going directly to the source on our repository.

> For a overview of the general TypeScript compiler architecture and layering, see [[Architectural Overview]]

That's it, you're ready to go. Now you can try out some of the following examples.

A minimal compiler

Let's try to write a barebones compiler that can compile a TypeScript string to its corresponding JavaScript. We will need to create a `Program`. This is as simple as calling `createProgram`. `createProgram` abstracts any interaction with the underlying system in the `CompilerHost` interface. The `CompilerHost` allows the compiler to read and write files, get the current directory, ensure that files and directories exist, and query some of the underlying system properties such as case sensitivity and new line characters. For convenience, we expose a function to create a default host using `createCompilerHost`.

```
/// <reference path="typings/node/node.d.ts" />
/// <reference path="typings/typescript/typescript.d.ts" />

import ts = require("typescript");

export function compile(filenames: string[], options: ts.CompilerOptions): void {
    var host = ts.createCompilerHost(options);
    var program = ts.createProgram(filenames, options, host);
```

```
    var checker = ts.createTypeChecker(program, /*produceDiagnostics*/ true);
    var result = checker.emitFiles();

    var allDiagnostics = program.getDiagnostics()
        .concat(checker.getDiagnostics())
        .concat(result.diagnostics);

    allDiagnostics.forEach(diagnostic => {
        var lineChar = diagnostic.file.getLineAndCharacterFromPosition(diagnostic.start);
        console.log(`${diagnostic.file.filename} (${lineChar.line},${lineChar.character}): $
    });

    console.log(`Process exiting with code '${result.emitResultStatus}'.`);
    process.exit(result.emitResultStatus);
}

compile(process.argv.slice(2), { noEmitOnError: true, noImplicitAny: true,
                                 target: ts.ScriptTarget.ES5, module: ts.ModuleKind.CommonJS
```

A simple transform function

Creating a compiler is simple enough, but you may not want to actually do traditional reads and writes from the file system; for instance, you may have a text buffer for your TypeScript input, and you may want to send/store the resulting JavaScript as JSON. What's more, you may want to use/modify the resulting JavaScript in some way. In such a case, you will need to provide your own `CompilerHost`.

```
/// <reference path="typings/node/node.d.ts" />
/// <reference path="typings/typescript/typescript.d.ts" />

import ts = require("typescript");
import fs = require("fs");
import path = require("path");

function transform(contents: string, libSource: string, compilerOptions: ts.CompilerOptions
    // Generated outputs
    var outputs = [];
    // Create a compilerHost object to allow the compiler to read and write files
    var compilerHost = {
        getSourceFile: function (filename, languageVersion) {
            if (filename === "file.ts")
                return ts.createSourceFile(filename, contents, compilerOptions.target, "0");
```

```
            if (filename === "lib.d.ts")
                return ts.createSourceFile(filename, libSource, compilerOptions.target, "0")
            return undefined;
        },
        writeFile: function (name, text, writeByteOrderMark) {
            outputs.push({ name: name, text: text, writeByteOrderMark: writeByteOrderMark })
        },
        getDefaultLibFilename: function () { return "lib.d.ts"; },
        useCaseSensitiveFileNames: function () { return false; },
        getCanonicalFileName: function (filename) { return filename; },
        getCurrentDirectory: function () { return ""; },
        getNewLine: function () { return "\n"; }
    };
    // Create a program from inputs
    var program = ts.createProgram(["file.ts"], compilerOptions, compilerHost);
    // Query for early errors
    var errors = program.getDiagnostics();
    // Do not generate code in the presence of early errors
    if (!errors.length) {
        // Type check and get semantic errors
        var checker = program.getTypeChecker(true);
        errors = checker.getDiagnostics();
        // Generate output
        checker.emitFiles();
    }
    return {
        outputs: outputs,
        errors: errors.map(function (e) { return e.file.filename + "(" + e.file.getLineAndCh
    };
}

// Calling our transform function using a simple TypeScript variable declarations,
// and loading the default library like:
var source = "var x: number  = 'string'";
var libSource = fs.readFileSync(path.join(path.dirname(require.resolve('typescript')), 'lib.
var result = transform(source, libSource);

console.log(JSON.stringify(result));
```

will generate the following output:

```
{
    "outputs": [
        {
            "name": "file.js",
```

```
            "text": "var x = 'string';\n"
        }
    ],
    "errors": [
        "file.ts(1): Type 'string' is not assignable to type 'number'."
    ]
}
```

Traversing the AST with a little linter

As mentioned above, the `Node` interface is the root of our AST. Generally, we use the `forEachChild` function in a recursive manner to traverse. This subsumes the visitor pattern and often gives more flexibility.

As an example of how one could traverse the AST, consider a minimal linter that does the following:

- Checks that all looping construct bodies are enclosed by curly braces.
- Checks that all if/else bodies are enclosed by curly braces.
- The "stricter" equality operators (`===`/`!==`) are used instead of the "loose" ones (`==`/`!=`).

```
/// <reference path="typings/node/node.d.ts" />
/// <reference path="typings/typescript/typescript.d.ts" />

import ts = require("typescript");

export function delint(sourceFile: ts.SourceFile) {
    delintNode(sourceFile);

    function delintNode(node: ts.Node) {
        switch (node.kind) {
            case ts.SyntaxKind.ForStatement:
            case ts.SyntaxKind.ForInStatement:
            case ts.SyntaxKind.WhileStatement:
            case ts.SyntaxKind.DoStatement:
                if ((<ts.IterationStatement>node).statement.kind !== ts.SyntaxKind.Block) {
                    report(node, "A looping statement's contents should be wrapped in a bloc
                }
                break;
            case ts.SyntaxKind.IfStatement:
                var ifStatement = (<ts.IfStatement>node);
                if (ifStatement.thenStatement.kind !== ts.SyntaxKind.Block) {
```

```
                    report(ifStatement.thenStatement, "An if statement's contents should be
                }
                if (ifStatement.elseStatement &&
                    ifStatement.elseStatement.kind !== ts.SyntaxKind.Block && ifStatement.el
                    report(ifStatement.elseStatement, "An else statement's contents should b
                }
                break;

            case ts.SyntaxKind.BinaryExpression:
                var op = (<ts.BinaryExpression>node).operator;

                if (op === ts.SyntaxKind.EqualsEqualsToken || op === ts.SyntaxKind.Exclamati
                    report(node, "Use '===' and '!=='.")
                }
                break;
        }

        ts.forEachChild(node, delintNode);
    }

    function report(node: ts.Node, message: string) {
        var lineChar = sourceFile.getLineAndCharacterFromPosition(node.getStart());
        console.log(`${sourceFile.filename} (${lineChar.line},${lineChar.character}): ${mess
    }
}

var fileNames = process.argv.slice(2);
var options: ts.CompilerOptions = { target: ts.ScriptTarget.ES6, module: ts.ModuleKind.AMD }
var host = ts.createCompilerHost(options);
var program = ts.createProgram(fileNames, options, host);

program.getSourceFiles().forEach(delint);
```

In this example, we did not need to create a type checker because all we wanted to do was traverse each `SourceFile`.

Incremental build support using the language services

> Please refer to the [[Using the Language Service API]] page for more details.

The services layer provide a set of additional set of utilities that can help simplify some complex scenarios. In the snippet below, we will try to build an incremental build server that watches a set of files and update the only the outputs of the file that changed. We will achieve this through creating a LanguageService object.

Similar to the program in the previous example, we need a LanguageServiceHost. The LanguageServiceHost augments the concept of a file with a version, isOpen flag, and a ScriptSnapshot. Version, allows the language service to track changes to files. isOpen, tells the language service to keep AST in memory as the file is in use. ScriptSnapshot is an abstraction over text that allows the language service to query for changes.

```
/// <reference path="typings/node/node.d.ts" />
/// <reference path="typings/typescript/typescript.d.ts" />

import fs = require("fs");
import ts = require("typescript");
import path = require("path");


function watch(filenames: string[], options: ts.CompilerOptions) {
    var files: ts.Map<{ version: number; text: string; }> = {};

    // Add the default library file
    filenames.unshift(path.join(path.dirname(require.resolve('typescript')), 'lib.d.ts'));

    // initialize the list of files
    filenames.forEach(filename => {
        files[filename] = { version: 0, text: fs.readFileSync(filename).toString() };
    });

    // Create the language service host to allow the LS to communicate with the host
    var servicesHost: ts.LanguageServiceHost = {
        getScriptFileNames: () => filenames,
        getScriptVersion: (filename) => files[filename] && files[filename].version.toString(
        getScriptSnapshot: (filename) => {
            var file = files[filename];
            return {
                getText: (start, end) => file.text.substring(start, end),
                getLength: () => file.text.length,
                getLineStartPositions: () => [],
                getChangeRange: (oldSnapshot) => undefined
            };
        },
        getCurrentDirectory: () => process.cwd(),
        getScriptIsOpen: () => true,
        getCompilationSettings: () => options,
        getDefaultLibFilename:(options) => 'lib.d.ts',
        log: (message) => console.log(message)
    };
```

```
// Create the language service files
var services = ts.createLanguageService(servicesHost, ts.createDocumentRegistry())

// Now let's watch the files
filenames.forEach(filename => {
    // First time around, emit all files
    emitFile(filename);

    // Add a watch on the file to handle next change
    fs.watchFile(filename,
        { persistent: true, interval: 250 },
        (curr, prev) => {
            // Check timestamp
            if (+curr.mtime <= +prev.mtime) {
                return;
            }

            var file = files[filename];

            // Update the version to signal a change in the file
            file.version++;

            // Clear the text to force a new read
            file.text = fs.readFileSync(filename).toString();

            // write the changes to disk
            emitFile(filename);
        });
});


function emitFile(filename: string) {
    var output = services.getEmitOutput(filename);

    if (output.emitOutputStatus === ts.EmitReturnStatus.Succeeded) {
        console.log(`Emitting ${filename}`);
    }
    else {
        console.log(`Emitting ${filename} failed`);
        var allDiagnostics = services.getCompilerOptionsDiagnostics()
            .concat(services.getSyntacticDiagnostics(filename))
            .concat(services.getSemanticDiagnostics(filename));

        allDiagnostics.forEach(diagnostic => {
            var lineChar = diagnostic.file.getLineAndCharacterFromPosition(diagnostic.st
```

```
                console.log(`  ${diagnostic.file && diagnostic.file.filename} (${lineChar.li
            });
        }

        output.outputFiles.forEach(o => {
            fs.writeFileSync(o.name, o.text, "utf8");
        });
    }
}

// Initialize files constituting the program as all .ts files in the current directory
var currentDirectoryFiles = fs.readdirSync(process.cwd()).
    filter(filename=> filename.length >= 3 && filename.substr(filename.length - 3, 3) === ".
    //map(filename => path.join(process.cwd(), filename));

// Start the watcher
watch(currentDirectoryFiles, { target: ts.ScriptTarget.ES5, module: ts.ModuleKind.CommonJS }
```

Pretty printer using the LS Formatter

> The formatting interfaces used here are part of the typescript 1.4 package but is not currently exposed in the public typescript.d.ts. The typings should be exposed in the next release.

```
/// <reference path="typings/node/node.d.ts" />
/// <reference path="typings/typescript/typescript.d.ts" />

import ts = require("typescript");

// Note: this uses ts.formatting which is part of the typescript 1.4 package but is not curr
//       exposed in the public typescript.d.ts. The typings should be exposed in the next re
function format(text: string) {
    var options = getDefaultOptions();

    // Parse the source text
    var sourceFile = ts.createSourceFile("file.ts", text, ts.ScriptTarget.Latest, "0");
    fixupParentReferences(sourceFile);

    // Get the formatting edits on the input sources
    var edits = (<any>ts).formatting.formatDocument(sourceFile, getRuleProvider(options), op

    // Apply the edits on the input code
    return applyEdits(text, edits);
```

```
function getRuleProvider(options: ts.FormatCodeOptions) {
    // Share this between multiple formatters using the same options.
    // This represents the bulk of the space the formatter uses.
    var ruleProvider = new (<any>ts).formatting.RulesProvider();
    ruleProvider.ensureUpToDate(options);
    return ruleProvider;
}

function applyEdits(text: string, edits: ts.TextChange[]): string {
    // Apply edits in reverse on the existing text
    var result = text;
    for (var i = edits.length - 1; i >= 0; i--) {
        var change = edits[i];
        var head = result.slice(0, change.span.start());
        var tail = result.slice(change.span.start() + change.span.length())
        result = head + change.newText + tail;
    }
    return result;
}

function getDefaultOptions(): ts.FormatCodeOptions {
    return {
        IndentSize: 4,
        TabSize: 4,
        NewLineCharacter: '\r\n',
        ConvertTabsToSpaces: true,
        InsertSpaceAfterCommaDelimiter: true,
        InsertSpaceAfterSemicolonInForStatements: true,
        InsertSpaceBeforeAndAfterBinaryOperators: true,
        InsertSpaceAfterKeywordsInControlFlowStatements: true,
        InsertSpaceAfterFunctionKeywordForAnonymousFunctions: false,
        InsertSpaceAfterOpeningAndBeforeClosingNonemptyParenthesis: false,
        PlaceOpenBraceOnNewLineForFunctions: false,
        PlaceOpenBraceOnNewLineForControlBlocks: false,
    };
}

function fixupParentReferences(sourceFile: ts.SourceFile) {
    var parent: ts.Node = sourceFile;
    function walk(n: ts.Node): void {
        n.parent = parent;

        var saveParent = parent;
        parent = n;
        ts.forEachChild(n, walk);
```

```
            parent = saveParent;
        }
        ts.forEachChild(sourceFile, walk);
    }
}


var code = "var a=function(v:number){return 0+1+2+3;\n}";
var result = format(code);
console.log(result);
```