> ### *Disclaimer*
>
> *Keep in mind that this is not yet a stable API - we're releasing this as version 0.5, and things will be changing over time. As a first iteration, there will be a few rough edges. We encourage any and all feedback from the community to improve the API. To allow users to transition between future releases, we will be documenting any [[API Breaking Changes]] per new release.*

Getting set up

First you'll need to install TypeScript >=1.6 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
```

You will also need the Node.js declaration files for some of these samples. To acquire the declaration files, run:

```
npm install -D @types/node
```

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

The compiler API has a few main components:

- A `Program` which is the TypeScript terminology for your whole application
- A `CompilerHost` which represents the users' system, with an API for reading files, checking directories and case sensitivity etc.
- Many `SourceFile` s which represent each source file in the application, hosting both the text and TypeScript AST

A minimal compiler

This example is a barebones compiler which takes a list of TypeScript files and compiles them to their corresponding JavaScript.

We will need to create a `Program`, via `createProgram` - this will create a default `CompilerHost` which uses the file system to get files.

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

function compile(fileNames: string[], options: ts.CompilerOptions): void {
  let program = ts.createProgram(fileNames, options);
  let emitResult = program.emit();

  let allDiagnostics = ts
    .getPreEmitDiagnostics(program)
    .concat(emitResult.diagnostics);

  allDiagnostics.forEach(diagnostic => {
    if (diagnostic.file) {
      let { line, character } = ts.getLineAndCharacterOfPosition(diagnostic.file,
diagnostic.start!);
      let message = ts.flattenDiagnosticMessageText(diagnostic.messageText, "\n");
      console.log(`${diagnostic.file.fileName} (${line + 1},${character + 1}):
${message}`);
    } else {
      console.log(ts.flattenDiagnosticMessageText(diagnostic.messageText, "\n"));
    }
  });

  let exitCode = emitResult.emitSkipped ? 1 : 0;
  console.log(`Process exiting with code '${exitCode}'.`);
  process.exit(exitCode);
}

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 not too many lines of code, but you may want to just get the corresponding JavaScript output given TypeScript sources. For this you can use `ts.transpileModule` to get a string => string transformation in two lines.

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

const source = "let x: string  = 'string'";

let result = ts.transpileModule(source, { compilerOptions: { module:
ts.ModuleKind.CommonJS }});

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

Getting the DTS from a JavaScript file

This will only work in TypeScript 3.7 and above. This example shows how you can take a list of JavaScript files and will show their generated d.ts files in the terminal.

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

function compile(fileNames: string[], options: ts.CompilerOptions): void {
  // Create a Program with an in-memory emit
  const createdFiles = {}
  const host = ts.createCompilerHost(options);
  host.writeFile = (fileName: string, contents: string) => createdFiles[fileName] =
contents
```

```
  // Prepare and emit the d.ts files
  const program = ts.createProgram(fileNames, options, host);
  program.emit();

  // Loop through all the input files
  fileNames.forEach(file => {
    console.log("### JavaScript\n")
    console.log(host.readFile(file))

    console.log("### Type Definition\n")
    const dts = file.replace(".js", ".d.ts")
    console.log(createdFiles[dts])
  })
}

// Run the compiler
compile(process.argv.slice(2), {
  allowJs: true,
  declaration: true,
  emitDeclarationOnly: true,
});
```

Re-printing Sections of a TypeScript File

This example will log out sub-sections of a TypeScript or JavaScript source file, this pattern is useful when you want the code for your app to be the source of truth. For example showcasing exports via their JSDoc comments.

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

/**
 * Prints out particular nodes from a source file
 *
 * @param file a path to a file
 * @param identifiers top level identifiers available
 */
function extract(file: string, identifiers: string[]): void {
  // Create a Program to represent the project, then pull out the
  // source file to parse its AST.
  let program = ts.createProgram([file], { allowJs: true });
  const sourceFile = program.getSourceFile(file);

  // To print the AST, we'll use TypeScript's printer
  const printer = ts.createPrinter({ newLine: ts.NewLineKind.LineFeed });

  // To give constructive error messages, keep track of found and un-found
identifiers
  const unfoundNodes = [], foundNodes = [];

  // Loop through the root AST nodes of the file
  ts.forEachChild(sourceFile, node => {
    let name = "";

    // This is an incomplete set of AST nodes which could have a top level
identifier
    // it's left to you to expand this list, which you can do by using
    // https://ts-ast-viewer.com/ to see the AST of a file then use the same
patterns
    // as below
    if (ts.isFunctionDeclaration(node)) {
      name = node.name.text;
      // Hide the method body when printing
      node.body = undefined;
    } else if (ts.isVariableStatement(node)) {
      name = node.declarationList.declarations[0].name.getText(sourceFile);
    } else if (ts.isInterfaceDeclaration(node)){
      name = node.name.text
    }

    const container = identifiers.includes(name) ? foundNodes : unfoundNodes;
    container.push([name, node]);
  });

  // Either print the found nodes, or offer a list of what identifiers were found
  if (!foundNodes.length) {
    console.log(`Could not find any of ${identifiers.join(", ")} in ${file}, found:
${unfoundNodes.filter(f => f[0]).map(f => f[0]).join(", ")}.`);
    process.exitCode = 1;
  } else {
    foundNodes.map(f => {
      const [name, node] = f;
      console.log("### " + name + "\n");
      console.log(printer.printNode(ts.EmitHint.Unspecified, node, sourceFile)) +
"\n";
    });
  }
}

// Run the extract function with the script's arguments
extract(process.argv[2], process.argv.slice(3));
```

Traversing the AST with a little linter

The `Node` interface is the root interface for the TypeScript AST. Generally, we use the `forEachChild` function in a recursive manner to iterate through the tree. This subsumes the visitor pattern and often gives more flexibility.

As an example of how one could traverse a file's 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 (`==` / `!=`).

```
import { readFileSync } from "fs";
import * as ts from "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 ((node as ts.IterationStatement).statement.kind !== ts.SyntaxKind.Block)
{
          report(
            node,
            'A looping statement\'s contents should be wrapped in a block body.'
          );
        }
        break;

      case ts.SyntaxKind.IfStatement:
        const ifStatement = node as ts.IfStatement;
        if (ifStatement.thenStatement.kind !== ts.SyntaxKind.Block) {
          report(ifStatement.thenStatement, 'An if statement\'s contents should be
wrapped in a block body.');
        }
        if (
          ifStatement.elseStatement &&
          ifStatement.elseStatement.kind !== ts.SyntaxKind.Block &&
          ifStatement.elseStatement.kind !== ts.SyntaxKind.IfStatement
        ) {
          report(
            ifStatement.elseStatement,
            'An else statement\'s contents should be wrapped in a block body.'
          );
        }
        break;

      case ts.SyntaxKind.BinaryExpression:
        const op = (node as ts.BinaryExpression).operatorToken.kind;
        if (op === ts.SyntaxKind.EqualsEqualsToken || op ===
ts.SyntaxKind.ExclamationEqualsToken) {
          report(node, 'Use \'===\' and \'!==\'.');
        }
        break;
    }

    ts.forEachChild(node, delintNode);
  }
  function report(node: ts.Node, message: string) {
    const { line, character } =
sourceFile.getLineAndCharacterOfPosition(node.getStart());
    console.log(`${sourceFile.fileName} (${line + 1},${character + 1}):
${message}`);
  }
}

const fileNames = process.argv.slice(2);
fileNames.forEach(fileName => {
  // Parse a file
  const sourceFile = ts.createSourceFile(
    fileName,
    readFileSync(fileName).toString(),
    ts.ScriptTarget.ES2015,
    /*setParentNodes */ true
  );

  // delint it
  delint(sourceFile);
});
```

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

All possible `ts.SyntaxKind` can be found under enum here.

Writing an incremental program watcher

TypeScript 2.7 introduces two new APIs: one for creating "watcher" programs that provide set of APIs to trigger rebuilds, and a "builder" API that watchers can take advantage of. `BuilderProgram`s are `Program` instances that are smart enough to cache errors and emit on modules from previous compilations if they or their dependencies haven't been updated in a cascading manner. A watcher can leverage builder program instances to only update results (like errors, and emit) of affected files in a compilation. This can speed up large projects with many files.

This API is used internally in the compiler to implement its `--watch` mode, but can also be leveraged by other tools as follows:

```
import ts = require("typescript");

const formatHost: ts.FormatDiagnosticsHost = {
  getCanonicalFileName: path => path,
  getCurrentDirectory: ts.sys.getCurrentDirectory,
  getNewLine: () => ts.sys.newLine
};

function watchMain() {
  const configPath = ts.findConfigFile(
    /*searchPath*/ "./",
    ts.sys.fileExists,
    "tsconfig.json"
  );
  if (!configPath) {
    throw new Error("Could not find a valid 'tsconfig.json'.");
  }

  // TypeScript can use several different program creation "strategies":
  //  * ts.createEmitAndSemanticDiagnosticsBuilderProgram,
  //  * ts.createSemanticDiagnosticsBuilderProgram
  //  * ts.createAbstractBuilder
  // The first two produce "builder programs". These use an incremental strategy
  // to only re-check and emit files whose contents may have changed, or whose
  // dependencies may have changes which may impact change the result of prior
  // type-check and emit.
  // The last uses an ordinary program which does a full type check after every
  // change.
  // Between `createEmitAndSemanticDiagnosticsBuilderProgram` and
  // `createSemanticDiagnosticsBuilderProgram`, the only difference is emit.
  // For pure type-checking scenarios, or when another tool/process handles emit,
  // using `createSemanticDiagnosticsBuilderProgram` may be more desirable.
  const createProgram = ts.createSemanticDiagnosticsBuilderProgram;

  // Note that there is another overload for `createWatchCompilerHost` that takes
  // a set of root files.
  const host = ts.createWatchCompilerHost(
    configPath,
    {},
    ts.sys,
    createProgram,
    reportDiagnostic,
    reportWatchStatusChanged
  );

  // You can technically override any given hook on the host, though you probably
  // don't need to.
  // Note that we're assuming `origCreateProgram` and `origPostProgramCreate`
  // doesn't use `this` at all.
  const origCreateProgram = host.createProgram;
  host.createProgram = (rootNames: ReadonlyArray<string>, options, host, oldProgram)
=> {
    console.log("** We're about to create the program! **");
    return origCreateProgram(rootNames, options, host, oldProgram);
  };
  const origPostProgramCreate = host.afterProgramCreate;

  host.afterProgramCreate = program => {
    console.log("** We finished making the program! **");
    origPostProgramCreate!(program);
  };

  // `createWatchProgram` creates an initial program, watches files, and updates
  // the program over time.
  ts.createWatchProgram(host);
```

```
}

function reportDiagnostic(diagnostic: ts.Diagnostic) {
  console.error("Error", diagnostic.code, ":", ts.flattenDiagnosticMessageText(
diagnostic.messageText, formatHost.getNewLine()));
}

/**
 * Prints a diagnostic every time the watch status changes.
 * This is mainly for messages like "Starting compilation" or "Compilation
completed".
 */
function reportWatchStatusChanged(diagnostic: ts.Diagnostic) {
  console.info(ts.formatDiagnostic(diagnostic, formatHost));
}

watchMain();
```

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 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 updates only the outputs of the files 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`, an `isOpen` flag, and a `ScriptSnapshot`. The `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.

If you are simply trying to implement watch-style functionality, we encourage you to explore the above watcher API.

```
import * as fs from "fs";
import * as ts from "typescript";

function watch(rootFileNames: string[], options: ts.CompilerOptions) {
  const files: ts.MapLike<{ version: number }> = {};

  // initialize the list of files
  rootFileNames.forEach(fileName => {
    files[fileName] = { version: 0 };
  });

  // Create the language service host to allow the LS to communicate with the host
  const servicesHost: ts.LanguageServiceHost = {
    getScriptFileNames: () => rootFileNames,
    getScriptVersion: fileName =>
      files[fileName] && files[fileName].version.toString(),
    getScriptSnapshot: fileName => {
      if (!fs.existsSync(fileName)) {
        return undefined;
      }

      return ts.ScriptSnapshot.fromString(fs.readFileSync(fileName).toString());
    },
    getCurrentDirectory: () => process.cwd(),
    getCompilationSettings: () => options,
    getDefaultLibFileName: options => ts.getDefaultLibFilePath(options),
    fileExists: ts.sys.fileExists,
    readFile: ts.sys.readFile,
    readDirectory: ts.sys.readDirectory,
    directoryExists: ts.sys.directoryExists,
    getDirectories: ts.sys.getDirectories,
  };

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

  // Now let's watch the files
  rootFileNames.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;
      }

      // Update the version to signal a change in the file
      files[fileName].version++;

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

  function emitFile(fileName: string) {
    let output = services.getEmitOutput(fileName);

    if (!output.emitSkipped) {
      console.log(`Emitting ${fileName}`);
    } else {
      console.log(`Emitting ${fileName} failed`);
      logErrors(fileName);
    }

    output.outputFiles.forEach(o => {
      fs.writeFileSync(o.name, o.text, "utf8");
    });
  }
  function logErrors(fileName: string) {
    let allDiagnostics = services
      .getCompilerOptionsDiagnostics()
      .concat(services.getSyntacticDiagnostics(fileName))
      .concat(services.getSemanticDiagnostics(fileName));

    allDiagnostics.forEach(diagnostic => {
      let message = ts.flattenDiagnosticMessageText(diagnostic.messageText, "\n");
      if (diagnostic.file) {
        let { line, character } = diagnostic.file.getLineAndCharacterOfPosition(
          diagnostic.start!
        );
        console.log(`  Error ${diagnostic.file.fileName} (${line + 1},${character
+1}): ${message}`);
      } else {
        console.log(`  Error: ${message}`);
      }
    });
  }
}

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

// Start the watcher
watch(currentDirectoryFiles, { module: ts.ModuleKind.CommonJS });
```

Customizing module resolution

You can override the standard way the compiler resolves modules by implementing optional method: `CompilerHost.resolveModuleNames`:

> `CompilerHost.resolveModuleNames(moduleNames: string[], containingFile: string): string[]`.

The method is given a list of module names in a file, and is expected to return an array of size `moduleNames.length`, each element of the array stores either:

- an instance of `ResolvedModule` with non-empty property `resolvedFileName` - resolution for corresponding name from `moduleNames` array or
- `undefined` if module name cannot be resolved.

You can invoke the standard module resolution process via calling `resolveModuleName`:

> `resolveModuleName(moduleName: string, containingFile: string, options: CompilerOptions, moduleResolutionHost: ModuleResolutionHost): ResolvedModuleNameWithFallbackLocations`.

This function returns an object that stores result of module resolution (value of `resolvedModule` property) as well as list of file names that were considered candidates before making current decision.

```
import * as ts from "typescript";
import * as path from "path";

function createCompilerHost(options: ts.CompilerOptions, moduleSearchLocations:
string[]): ts.CompilerHost {
  return {
    getSourceFile,
    getDefaultLibFileName: () => "lib.d.ts",
    writeFile: (fileName, content) => ts.sys.writeFile(fileName, content),
    getCurrentDirectory: () => ts.sys.getCurrentDirectory(),
    getDirectories: path => ts.sys.getDirectories(path),
    getCanonicalFileName: fileName =>
      ts.sys.useCaseSensitiveFileNames ? fileName : fileName.toLowerCase(),
    getNewLine: () => ts.sys.newLine,
    useCaseSensitiveFileNames: () => ts.sys.useCaseSensitiveFileNames,
    fileExists,
    readFile,
    resolveModuleNames
  };

  function fileExists(fileName: string): boolean {
    return ts.sys.fileExists(fileName);
  }

  function readFile(fileName: string): string | undefined {
    return ts.sys.readFile(fileName);
  }

  function getSourceFile(fileName: string, languageVersion: ts.ScriptTarget,
onError?: (message: string) => void) {
    const sourceText = ts.sys.readFile(fileName);
    return sourceText !== undefined
      ? ts.createSourceFile(fileName, sourceText, languageVersion)
      : undefined;
  }

  function resolveModuleNames(
    moduleNames: string[],
    containingFile: string
  ): ts.ResolvedModule[] {
    const resolvedModules: ts.ResolvedModule[] = [];
    for (const moduleName of moduleNames) {
      // try to use standard resolution
      let result = ts.resolveModuleName(moduleName, containingFile, options, {
        fileExists,
        readFile
      });
      if (result.resolvedModule) {
        resolvedModules.push(result.resolvedModule);
      } else {
        // check fallback locations, for simplicity assume that module at location
        // should be represented by '.d.ts' file
        for (const location of moduleSearchLocations) {
          const modulePath = path.join(location, moduleName + ".d.ts");
          if (fileExists(modulePath)) {
            resolvedModules.push({ resolvedFileName: modulePath });
          }
        }
      }
    }
    return resolvedModules;
  }
}

function compile(sourceFiles: string[], moduleSearchLocations: string[]): void {
  const options: ts.CompilerOptions = {
    module: ts.ModuleKind.AMD,
    target: ts.ScriptTarget.ES5
  };
  const host = createCompilerHost(options, moduleSearchLocations);
  const program = ts.createProgram(sourceFiles, options, host);

  /// do something with program...
}
```

Creating and Printing a TypeScript AST

TypeScript has factory functions and a printer API that you can use in conjunction.

- The factory allows you to generate new tree nodes in TypeScript's AST format.
- The printer can take an existing tree (either one produced by `createSourceFile` or by factory functions), and produce an output string.

Here is an example that utilizes both to produce a factorial function:

```
import ts = require("typescript");

function makeFactorialFunction() {
  const functionName = ts.factory.createIdentifier("factorial");
  const paramName = ts.factory.createIdentifier("n");
  const parameter = ts.factory.createParameterDeclaration(
    /*decorators*/ undefined,
    /*modifiers*/ undefined,
    /*dotDotDotToken*/ undefined,
    paramName
  );

  const condition = ts.factory.createBinaryExpression(paramName,
ts.SyntaxKind.LessThanEqualsToken, ts.factory.createNumericLiteral(1));
  const ifBody =
ts.factory.createBlock([ts.factory.createReturnStatement(ts.factory.createNumericLitera
 /*multiline*/ true);

  const decrementedArg = ts.factory.createBinaryExpression(paramName,
ts.SyntaxKind.MinusToken, ts.factory.createNumericLiteral(1));
  const recurse = ts.factory.createBinaryExpression(paramName,
ts.SyntaxKind.AsteriskToken, ts.factory.createCallExpression(functionName,
/*typeArgs*/ undefined, [decrementedArg]));
  const statements = [ts.factory.createIfStatement(condition, ifBody),
ts.factory.createReturnStatement(recurse)];

  return ts.factory.createFunctionDeclaration(
    /*decorators*/ undefined,
    /*modifiers*/ [ts.factory.createToken(ts.SyntaxKind.ExportKeyword)],
    /*asteriskToken*/ undefined,
    functionName,
    /*typeParameters*/ undefined,
    [parameter],
    /*returnType*/ ts.factory.createKeywordTypeNode(ts.SyntaxKind.NumberKeyword),
    ts.factory.createBlock(statements, /*multiline*/ true)
  );
}

const resultFile = ts.createSourceFile("someFileName.ts", "",
ts.ScriptTarget.Latest, /*setParentNodes*/ false, ts.ScriptKind.TS);
const printer = ts.createPrinter({ newLine: ts.NewLineKind.LineFeed });

const result = printer.printNode(ts.EmitHint.Unspecified, makeFactorialFunction(),
resultFile);
console.log(result);
```

Using the Type Checker

In this example we will walk the AST and use the checker to serialize class information. We'll use the type checker to get symbol and type information, while grabbing JSDoc comments for exported classes, their constructors, and respective constructor parameters.

```
import * as ts from "typescript";
import * as fs from "fs";

interface DocEntry {
  name?: string;
  fileName?: string;
  documentation?: string;
  type?: string;
  constructors?: DocEntry[];
  parameters?: DocEntry[];
  returnType?: string;
}

/** Generate documentation for all classes in a set of .ts files */
```

```
function generateDocumentation(
  fileNames: string[],
  options: ts.CompilerOptions
): void {
  // Build a program using the set of root file names in fileNames
  let program = ts.createProgram(fileNames, options);

  // Get the checker, we will use it to find more about classes
  let checker = program.getTypeChecker();
  let output: DocEntry[] = [];

  // Visit every sourceFile in the program
  for (const sourceFile of program.getSourceFiles()) {
    if (!sourceFile.isDeclarationFile) {
      // Walk the tree to search for classes
      ts.forEachChild(sourceFile, visit);
    }
  }

  // print out the doc
  fs.writeFileSync("classes.json", JSON.stringify(output, undefined, 4));

  return;

  /** visit nodes finding exported classes */
  function visit(node: ts.Node) {
    // Only consider exported nodes
    if (!isNodeExported(node)) {
      return;
    }

    if (ts.isClassDeclaration(node) && node.name) {
      // This is a top level class, get its symbol
      let symbol = checker.getSymbolAtLocation(node.name);
      if (symbol) {
        output.push(serializeClass(symbol));
      }
      // No need to walk any further, class expressions/inner declarations
      // cannot be exported
    } else if (ts.isModuleDeclaration(node)) {
      // This is a namespace, visit its children
      ts.forEachChild(node, visit);
    }
  }

  /** Serialize a symbol into a json object */
  function serializeSymbol(symbol: ts.Symbol): DocEntry {
    return {
      name: symbol.getName(),
      documentation:
ts.displayPartsToString(symbol.getDocumentationComment(checker)),
      type: checker.typeToString(
        checker.getTypeOfSymbolAtLocation(symbol, symbol.valueDeclaration!)
      )
    };
  }

  /** Serialize a class symbol information */
  function serializeClass(symbol: ts.Symbol) {
    let details = serializeSymbol(symbol);

    // Get the construct signatures
    let constructorType = checker.getTypeOfSymbolAtLocation(
      symbol,
      symbol.valueDeclaration!
    );
    details.constructors = constructorType
      .getConstructSignatures()
      .map(serializeSignature);
    return details;
  }

  /** Serialize a signature (call or construct) */
  function serializeSignature(signature: ts.Signature) {
    return {
      parameters: signature.parameters.map(serializeSymbol),
      returnType: checker.typeToString(signature.getReturnType()),
      documentation:
ts.displayPartsToString(signature.getDocumentationComment(checker))
    };
  }

  /** True if this is visible outside this file, false otherwise */
  function isNodeExported(node: ts.Node): boolean {
    return (
      (ts.getCombinedModifierFlags(node as ts.Declaration) &
ts.ModifierFlags.Export) !== 0 ||
      (!!node.parent && node.parent.kind === ts.SyntaxKind.SourceFile)
    );
  }
}

generateDocumentation(process.argv.slice(2), {
  target: ts.ScriptTarget.ES5,
  module: ts.ModuleKind.CommonJS
});
```

to try this:

```
tsc docGenerator.ts --m commonjs
node docGenerator.js test.ts
```

Passing an input like:

```
/**
 * Documentation for C
 */
class C {
    /**
     * constructor documentation
     * @param a my parameter documentation
     * @param b another parameter documentation
     */
    constructor(a: string, b: C) { }
}
```

We should get output like:

```
[
    {
        "name": "C",
        "documentation": "Documentation for C ",
        "type": "typeof C",
        "constructors": [
            {
                "parameters": [
                    {
                        "name": "a",
                        "documentation": "my parameter documentation",
                        "type": "string"
                    },
                    {
                        "name": "b",
                        "documentation": "another parameter documentation",
                        "type": "C"
                    }
                ],
                "returnType": "C",
                "documentation": "constructor documentation"
            }
        ]
    }
]
```