# Checker

Ok, yeah, so it's a 30k LOC file. Why 30k lines in one file? Well there's a few main arguments:

- All of the checker is in one place.
- Save memory by making it a global, to quote a comment in the parser:

  ```
  // Implement the parser as a singleton module. We do this for perf reasons becaus
  // can actually be expensive enough to impact us on projects with many source fil
  ```

- Lots of these functions need to know a lot about each other, the top of the function `createTypeChecker` has a set of variables which are global within all of these functions and are liberally accessed.

  Switching to different files means probably making [god objects][god], and the checker needs to be extremely fast. We want to avoid additional calls for ambient context. There are architectural patterns for this, but it's better to assume good faith that they've been explored already (8 years down the line now.)

Anyway, better to get started somewhere. I asked online about how people would try to study a file like this and I think one of the best paths is by following a particular story as a file gets checked.

## An entry-point

The likely entry point is via a Program. The program has a memoized type-checker created in `getDiagnosticsProducingTypeChecker` which creates a type checker.

The initial start of type checking starts with `getDiagnosticsWorker`, worker in this case isn't a threading term I believe ( at least I can't find anything like that in the code ) - it is set up to listen for diagnostic results (e.g. warns/fails) and then triggers `checkSourceFileWorker`.

This function starts at the root `Node` of any TS/JS file node tree: `SourceFile`. It will then have to recurse through all of the AST nodes in it's tree.

It doesn't start with a single recursive function though, it starts by looking through the SourceFile's `statements` and through each one of those to get all the nodes. For example:

```
// Statement 1
const hi = () => "Hello";

// Statement 2
console.log(hi());
```

Which looks a bit like:

```
SourceFile
 statements:

  - VariableStatement
    - declarationList: VariableDeclarationList # (because any const cna have many declaratio
      - variables: VariableStatement
        - etc

  - ExpressionStatement
    - expression: CallExpression # outer console.log
      - expression: PropertyAccessExpression
        - etc
      - arguments: CallExpression
        - etc
```

See AST Explorer

Each node has a different variable to work with (so you can't just say `if node.children { node.children.foreach(lookAtNode) }` ) but instead you need to examine each node individually.

## Checking a Statement

Initially the meat of the work starts in `checkSourceElementWorker` which has a by `switch` statement that contains all legitimate nodes which can start a statement. Each node in the tree then does it's checking.

Let's try get a really early error, with this bad code:

```
// A return statement shouldn't exist here in strict mode (or any mode?)
return;
~~~~~~
```

It goes into `checkReturnStatement` which uses `getContainingFunction` to determine if the `return` lives inside a `function-like` node ( e.g. `MethodSignature`, `CallSignature`, `JSDocSignature`, `ConstructSignature`, `IndexSignature`, `FunctionType`, `JSDocFunctionType`, `ConstructorType`).

Because the parent of the `return` statement is the root (`parent: SourceFileObject`) then it's going to fail. This triggers `grammarErrorOnFirstToken` which will raise the error: `A 'return' statement can only be used within a function body.ts(1108)` and declare the error underline to the first token inside that node.

## Checking Type Equality

```
const myString = "Hello World";
const myInt = 123;
// Error: This condition will always return 'false' since the types '123' and '"Hello World
```

```
if (myInt === myString) {
  // Do something
}
```

To get to this error message:

- `checkSourceElementWorker` loops through the 3 statements in the `SourceFile`
- In the third, it goes through:
  - `checkIfStatement`
  - `checkTruthinessExpression`
  - `checkExpression`
  - `checkBinaryLikeExpression` where it fails.
- The fail comes from inside `isTypeRelatedTo` which has a set of heuristics for whether the types are the same (`isSimpleTypeRelatedTo` uses cached data in NodeLinks, and `checkTypeRelatedTo` dives deeper) and if not then. It will raise.

### Caching Data While Checking

Note there are two ways in which TypeScript is used, as a server and as a one-off compiler. In a server, we want to re-use as much as possible between API requests, and so the Node tree is treated as immutable data until there is a new AST.

This gets tricky inside the Type Checker, which for speed reasons needs to cache data somewhere. The solution to this is the `NodeLinks` property on a Node. The Type Checker fills this up during the run and re-uses it, then it is discarded next time you re-run the checker.

### Type Flags

Because TypeScript is a structural type system, every type can reasonably be compared with every other type. One of the main ways in which TypeScript keeps track of the underlying data-model is via the `enum TypeFlags`. Accessed via `.flags` on any node, it is a value which is used via bitmasking to let you know what data it represents.

If you wanted to check for whether the type is a union:

```
if (target.flags & TypeFlags.Union && source.flags & TypeFlags.Object) {
  // is a union object
}
```

When running the compiler in debug mode, you can see a string version of this via `target.__debugFlags`.

### Type Comparison

The entry point for comparing two types happens in `checkTypeRelatedTo`. This function is mostly about handling the diagnostic results from any checking though and doesn't do the work. The honour of that goes to [`isRelatedTo`][18] which:

- Figures out what the source and target types should be (based on freshness (a literal which was created in an expression), whether it is substituted () or simplifiable (a type which depends first on resolving another type))
- First, check if they're identical via `isIdenticalTo`. The check for most objects occurs in `recursiveTypeRelatedTo`, unions and intersections have a check that compares each value in `eachTypeRelatedToSomeType` which eventually calls `recursiveTypeRelatedTo` for each item in the type.
- The heavy lifting of the comparison depending on what flags are set on the node is done in `structuredTypeRelatedTo`. Where it picks off one by one different possible combinations for matching and returns early as soon as possible.

A lot of the functions related to type checking return a [`Ternary`][24], which an enum with three states, true, false and maybe. This gives the checker the chance to admit that it probably can't figure out whether a match is true currently (maybe it hit the 100 depth limit for example) and potentially could be figured out coming in from a different resolution.

TODO: what are substituted types?

## Debugging Advice

- If you want to find a diagnostic in the codebase, search for `diag(WXZY` e.g ‘`, you'll find`src/compiler/diagnosticInformationMap.generated.ts‘ has it being referenced by a key. Search for that key.
- If you're working from an error and want to see the path it took to get there, you can add a breakpoint in `createFileDiagnostic` which should get called for all diagnostic errors.