“

TypeScript’s Strict Compiler Options

Daniel Danielecki

TypeScript Compiler’s Rules

Compiler Options

Top Level	files, extends, include, exclude and references
"compilerOptions"	
Type Checking	allowUnreachableCode, allowUnusedLabels, alwaysStrict, exactOptionalPropertyTypes, noFallthroughCasesInSwitch, noImplicitAny, noImplicitOverride, noImplicitReturns, noImplicitThis, noPropertyAccessFromIndexSignature, noUncheckedIndexedAccess, noUnusedLocals, noUnusedParameters, strict, strictBindCallApply, strictFunctionTypes, strictNullChecks, strictPropertyInitialization and useUnknownInCatchVariables
Modules	allowUmdGlobalAccess, baseUrl, module, moduleResolution, noResolve, paths, resolveJsonModule, rootDir, rootDirs, typeRoots and types
Emit	declaration, declarationDir, declarationMap, downlevelIteration, emitBOM, emitDeclarationOnly, importHelpers, importsNotUsedAsValues, inlineSourceMap, inlineSources, mapRoot, newLine, noEmit, noEmitHelpers, noEmitOnError, outDir, outFile, preserveConstEnums, removeComments, sourceMap, sourceRoot and stripInternal
JavaScript Support	allowJs, checkJs and maxNodeModuleJsDepth
Editor Support	disableSizeLimit and plugins
Interop Constraints	allowSyntheticDefaultImports, esModuleInterop, forceConsistentCasingInFileNames, isolatedModules and preserveSymlinks
Backwards Compatibility	charset, keyofStringsOnly, noImplicitUseStrict, noStrictGenericChecks, out, suppressExcessPropertyErrors and suppressImplicitAnyIndexErrors

Compiler Options

Top Level	files, extends, include, exclude and references
"compilerOptions"	
Type Checking	allowUnreachableCode, allowUnusedLabels, alwaysStrict, exactOptionalPropertyTypes, noFallthroughCasesInSwitch, noImplicitAny, noImplicitOverride, noImplicitReturns, noImplicitThis, noPropertyAccessFromIndexSignature, noUncheckedIndexedAccess, noUnusedLocals, noUnusedParameters, strict, strictBindCallApply, strictFunctionTypes, strictNullChecks, strictPropertyInitialization and useUnknownInCatchVariables
Modules	allowUmdGlobalAccess, baseUrl, module, moduleResolution, noResolve, paths, resolveJsonModule, rootDir, rootDirs, typeRoots and types
Emit	declaration, declarationDir, declarationMap, downlevelIteration, emitBOM, emitDeclarationOnly, importHelpers, importsNotUsedAsValues, inlineSourceMap, inlineSources, mapRoot, newLine, noEmit, noEmitHelpers, noEmitOnError, outDir, outFile, preserveConstEnums, removeComments, sourceMap, sourceRoot and stripInternal
JavaScript Support	allowJs, checkJs and maxNodeModuleJsDepth
Editor Support	disableSizeLimit and plugins
Interop Constraints	allowSyntheticDefaultImports, esModuleInterop, forceConsistentCasingInFileNames, isolatedModules and preserveSymlinks
Backwards Compatibility	charset, keyofStringsOnly, noImplicitUseStrict, noStrictGenericChecks, out, suppressExcessPropertyErrors and suppressImplicitAnyIndexErrors

"compilerOptions"

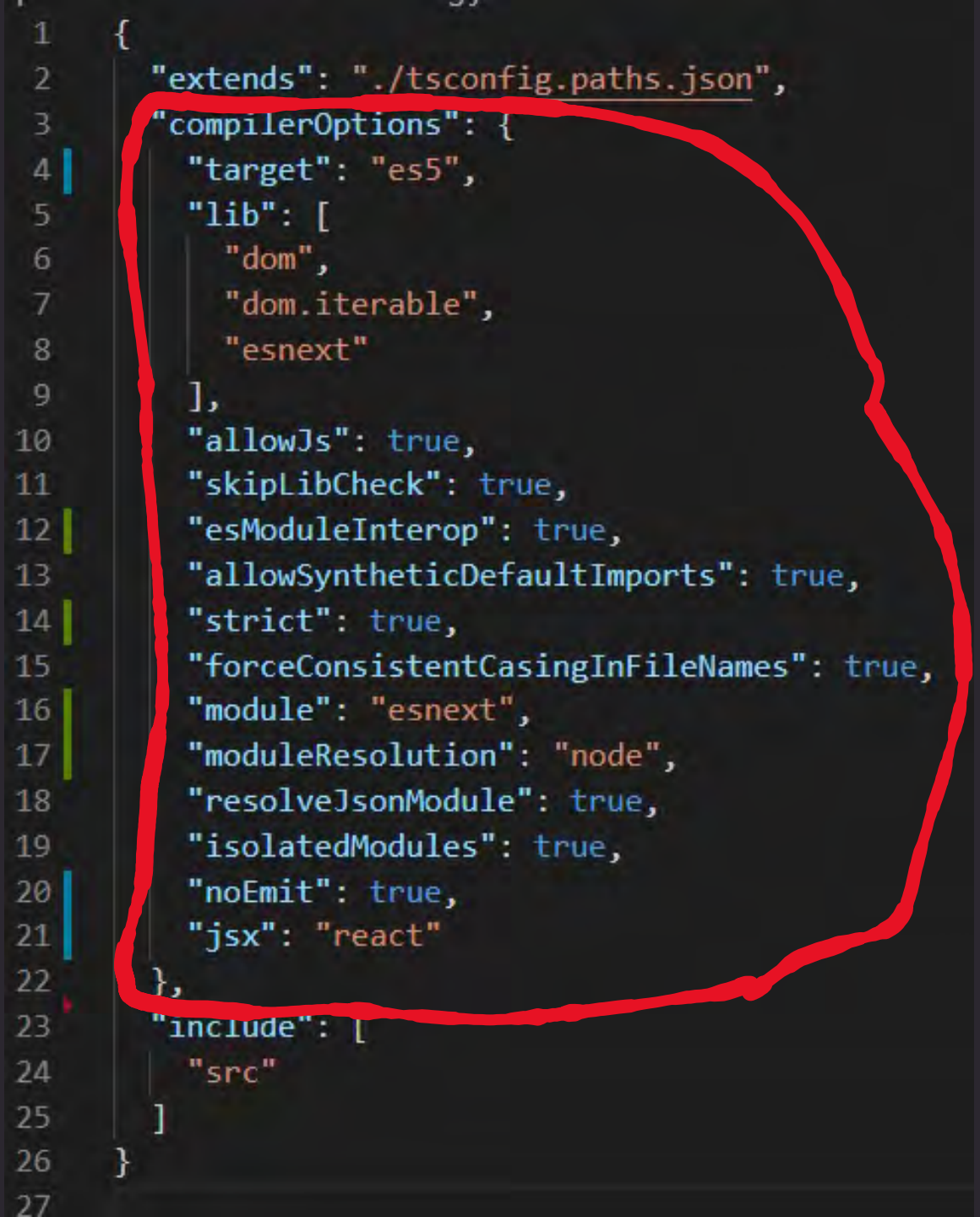
Project Options	allowJs, checkJs, composite, declaration, declarationMap, downlevelIteration, importHelpers, incremental, isolatedModules, jsx, lib, module, noEmit, outDir, outFile, plugins, removeComments, rootDir, sourceMap, target and tsBuildInfoFile
Strict Checks	alwaysStrict, noImplicitAny, noImplicitThis, strict, strictBindCallApply, strictFunctionTypes, strictNullChecks and strictPropertyInitialization
Module Resolution	allowSyntheticDefaultImports, allowUmdGlobalAccess, baseUrl, esModuleInterop, moduleResolution, paths, preserveSymlinks, rootDirs, typeRoots and types
Source Maps	inlineSourceMap, inlineSources, mapRoot and sourceRoot
Linter Checks	noFallthroughCasesInSwitch, noImplicitOverride, noImplicitReturns, noPropertyAccessFromIndexSignature, noUncheckedIndexedAccess, noUnusedLocals and noUnusedParameters
Experimental	emitDecoratorMetadata and experimentalDecorators
Advanced	allowUnreachableCode, allowUnusedLabels, assumeChangesOnlyAffectDirectDependencies, charset, declarationDir, diagnostics, disableReferencedProjectLoad, disableSizeLimit, disableSolutionSearching, disableSourceOfProjectReferenceRedirect, emitBOM, emitDeclarationOnly, explainFiles, extendedDiagnostics, forceConsistentCasingInFileNames, generateCpuProfile, importsNotUsedAsValues, jsxFactory, jsxFragmentFactory, jsxImportSource, keyofStringsOnly, listEmittedFiles, listFiles, maxNodeModuleJsDepth, newLine, noEmitHelpers, noEmitOnError, noErrorTruncation, noImplicitUseStrict, noLib, noResolve, noStrictGenericChecks, out, preserveConstEnums, reactNamespace, resolveJsonModule, skipDefaultLibCheck, skipLibCheck, stripInternal, suppressExcessPropertyErrors, suppressImplicitAnyIndexErrors, traceResolution and useDefineForClassFields

"compilerOptions"

Project Options	allowJs, checkJs, composite, declaration, declarationMap, downlevelIteration, importHelpers, incremental, isolatedModules, jsx, lib, module, noEmit, outDir, outFile, plugins, removeComments, rootDir, sourceMap, target and tsBuildInfoFile
Strict Checks	alwaysStrict, noImplicitAny, noImplicitThis, strict, strictBindCallApply, strictFunctionTypes, strictNullChecks and strictPropertyInitialization
Module Resolution	allowSyntheticDefaultImports, allowUmdGlobalAccess, baseUrl, esModuleInterop, moduleResolution, paths, preserveSymlinks, rootDirs, typeRoots and types
Source Maps	inlineSourceMap, inlineSources, mapRoot and sourceRoot
Linter Checks	noFallthroughCasesInSwitch, noImplicitOverride, noImplicitReturns, noPropertyAccessFromIndexSignature, noUncheckedIndexedAccess, noUnusedLocals and noUnusedParameters
Experimental	emitDecoratorMetadata and experimentalDecorators
Advanced	allowUnreachableCode, allowUnusedLabels, assumeChangesOnlyAffectDirectDependencies, charset, declarationDir, diagnostics, disableReferencedProjectLoad, disableSizeLimit, disableSolutionSearching, disableSourceOfProjectReferenceRedirect, emitBOM, emitDeclarationOnly, explainFiles, extendedDiagnostics, forceConsistentCasingInFileNames, generateCpuProfile, importsNotUsedAsValues, jsxFactory, jsxFragmentFactory, jsxImportSource, keyofStringsOnly, listEmittedFiles, listFiles, maxNodeModuleJsDepth, newLine, noEmitHelpers, noEmitOnError, noErrorTruncation, noImplicitUseStrict, noLib, noResolve, noStrictGenericChecks, out, preserveConstEnums, reactNamespace, resolveJsonModule, skipDefaultLibCheck, skipLibCheck, stripInternal, suppressExcessPropertyErrors, suppressImplicitAnyIndexErrors, traceResolution and useDefineForClassFields

tsconfig

```jsonc
{
  "extends": "./tsconfig.paths.json",
  "compilerOptions": {
    "target": "es5",
    "lib": [
      "dom",
      "dom.iterable",
      "esnext"
    ],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react"
  },
  "include": [
    "src"
  ]
}
```



```
{
  "extends": "./tsconfig.paths.json",
  "compilerOptions": {
    "target": "es5",
    "lib": [
      "dom",
      "dom.iterable",
      "esnext"
    ],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react"
  },
  "include": [
    "src"
  ]
}
```

noImplicitAny

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitAny": false
  }
}
```

```
function sampleMethod(sampleParameter) {
  console.log("hello: ", sampleParameter);
}

sampleMethod("test");
```

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitAny": true
  }
}
```

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitAny": true
  }
}
```

```
function sampleMethod(sampleParameter) {
```

01-noImplicitAny.ts 1 of 1 problem

Parameter 'sampleParameter' implicitly has an 'any' type. ts(7006)

```
  console.log("hello: ", sampleParameter);
}

sampleMethod("test");
```

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitAny": true
  }
}
```

```
function sampleMethod(sampleParameter: string) {
  console.log("hello: ", sampleParameter);
}

sampleMethod("test");
```

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitAny": true
  }
}
```

```
function sampleMethod(sampleParameter) {
```

01-noImplicitAny.ts 1 of 1 problem

Parameter 'sampleParameter' implicitly has an 'any' type. ts(7006)

```
  console.log("hello: ", sampleParameter);
}

sampleMethod("test");
```

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitAny": true
  }
}
```

```
function sampleMethod(sampleParameter: string) {
  console.log("hello: ", sampleParameter);
}

sampleMethod("test");
```

noImplicitThis

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitThis": false
  }
}
```

```
class TotalAmount {
  itemPrice: number;
  tax: number;

  constructor(itemPrice: number, tax: number) {
    this.itemPrice = itemPrice;
    this.tax = tax;
  }

  calculateTotalAmount() {
    return function () {
      return this.itemPrice * this.tax;
    };
  }
}
```

tsconfig.json

```
1   {
2       "compilerOptions": {
3           "noImplicitThis": true
4       }
5   }
```

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitThis": true
  }
}
```

```
class TotalAmount {
  itemPrice: number;
  tax: number;

  constructor(itemPrice: number, tax: number) {
    this.itemPrice = itemPrice;
    this.tax = tax;
  }

  calculateTotalAmount() {
    return function () {
      return this.itemPrice * this.tax;
    };
  }
}
```

02-noImplicitThis.ts 1 of 2 problems

'this' implicitly has type 'any' because it does not have a type annotation. ts(2683)

02-noImplicitThis.ts(11, 12): An outer value of 'this' is shadowed by this container.

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitThis": true
  }
}
```

```
class TotalAmount {
  itemPrice: number;
  tax: number;

  constructor(itemPrice: number, tax: number) {
    this.itemPrice = itemPrice;
    this.tax = tax;
  }

  calculateTotalAmount() {
    return () => {
      return this.itemPrice * this.tax;
    };
  }
}
```

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitThis": true
  }
}
```

```
class TotalAmount {
  itemPrice: number;
  tax: number;

  constructor(itemPrice: number, tax: number) {
    this.itemPrice = itemPrice;
    this.tax = tax;
  }

  calculateTotalAmount() {
    return function () {
      return this.itemPrice * this.tax;
    };
  }
}
```

02-noImplicitThis.ts 1 of 2 problems

'this' implicitly has type 'any' because it does not have a type annotation. ts(2683)

02-noImplicitThis.ts(11, 12): An outer value of 'this' is shadowed by this container.

tsconfig.json

```json
{
  "compilerOptions": {
    "noImplicitThis": true
  }
}
```

```ts
class TotalAmount {
  itemPrice: number;
  tax: number;

  constructor(itemPrice: number, tax: number) {
    this.itemPrice = itemPrice;
    this.tax = tax;
  }

  calculateTotalAmount() {
    return () => {
      return this.itemPrice * this.tax;
    };
  }
}

```

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitThis": true
  }
}
```

```
class TotalAmount {
  itemPrice: number;
  tax: number;

  constructor(itemPrice: number, tax: number) {
    this.itemPrice = itemPrice;
    this.tax = tax;
  }

  calculateTotalAmount() {
    return function (this: TotalAmount) {
      return this.itemPrice * this.tax;
    };
  }
}

```

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitThis": true
  }
}
```

```
class TotalAmount {
  itemPrice: number;
  tax: number;

  constructor(itemPrice: number, tax: number) {
    this.itemPrice = itemPrice;
    this.tax = tax;
  }

  calculateTotalAmount() {
    return function () {
      return this.itemPrice * this.tax;
    };
  }
}
```

02-noImplicitThis.ts 1 of 2 problems

'this' implicitly has type 'any' because it does not have a type annotation. ts(2683)

02-noImplicitThis.ts(11, 12): An outer value of 'this' is shadowed by this container.

tsconfig.json

```
{
  "compilerOptions": {
    "noImplicitThis": true
  }
}
```

```
class TotalAmount {
  itemPrice: number;
  tax: number;

  constructor(itemPrice: number, tax: number) {
    this.itemPrice = itemPrice;
    this.tax = tax;
  }

  calculateTotalAmount() {
    return function (this: TotalAmount) {
      return this.itemPrice * this.tax;
    };
  }
}

```

noImplicitReturns

noImplicitOverride

https://www.typescriptlang.org/tsconfig#noImplicitOverride

noUnusedLocals

noUnusedParameters

allowUnreachableCode

Type Checking

Allow Unreachable Code - `allowUnreachableCode`

When:

- `undefined` (default) provide suggestions as warnings to editors
- `true` unreachable code is ignored
- `false` raises compiler errors about unreachable code

These warnings are only about code which is provably unreachable due to the use of JavaScript syntax, for example:

Default: `undefined`

Released: 1.8

Type Checking

Allow Unreachable Code - `allowUnreachableCode`

When:

- `undefined` (default) provide suggestions as warnings to editors
- `true` unreachable code is ignored
- `false` raises compiler errors about unreachable code

These warnings are only about code which is provably unreachable due to the use of JavaScript syntax, for example:

Default: `undefined`

Released: 1.8

allowUnusedLabels

Allow Unused Labels - `allowUnusedLabels`

When:

- `undefined` (default) provide suggestions as warnings to editors
- `true` unused labels are ignored
- `false` raises compiler errors about unused labels

Labels are very rare in JavaScript and typically indicate an attempt to write an object literal:

```
function verifyAge(age: number) {
  // Forgot 'return' statement
  if (age > 18) {
    verified: true;
Unused label.
  }
}
```

Default:
`undefined`

Released:
1.8

Allow Unused Labels - `allowUnusedLabels`

When:

- `undefined` (default) provide suggestions as warnings to editors
- `true` unused labels are ignored
- `false` raises compiler errors about unused labels

Labels are very rare in JavaScript and typically indicate an attempt to write an object literal:

```
function verifyAge(age: number) {
  // Forgot 'return' statement
  if (age > 18) {
    verified: true;
Unused label.
  }
}
```

Default:
`undefined`

Released:
1.8

strictNullChecks

tsconfig.json

```
{
  "compilerOptions": {
    "strictNullChecks": false
  }
}
```

```
function sampleMethod(sampleParameter: string) {
  let greeting: string = "Hello";
  console.log(greeting + ": ", sampleParameter);
  greeting = null;
  console.log(greeting + ": ", sampleParameter);
}

sampleMethod("test");
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictNullChecks": true
  }
}
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictNullChecks": true
  }
}
```

```
function sampleMethod(sampleParameter: string) {
    let greeting: string = "Hello";
    console.log(greeting + ": ", sampleParameter);
    greeting = null;
```

03-strictNullChecks.ts 1 of 1 problem

```
Type 'null' is not assignable to type 'string'. ts(2322)
```

```
    console.log(greeting + ": ", sampleParameter);
}

sampleMethod("test");
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictNullChecks": true
  }
}
```

```
function sampleMethod(sampleParameter: string) {
  let greeting: string | null = "Hello";
  console.log(greeting + ": ", sampleParameter);
  greeting = null;
  console.log(greeting + ": ", sampleParameter);
}

sampleMethod("test");
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictNullChecks": true
  }
}
```

```
function sampleMethod(sampleParameter: string) {
    let greeting: string = "Hello";
    console.log(greeting + ": ", sampleParameter);
    greeting = null;
```

03-strictNullChecks.ts 1 of 1 problem

```
Type 'null' is not assignable to type 'string'. ts(2322)
```

```
    console.log(greeting + ": ", sampleParameter);
}

sampleMethod("test");
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictNullChecks": true
  }
}
```

```
function sampleMethod(sampleParameter: string) {
  let greeting: string | null = "Hello";
  console.log(greeting + ": ", sampleParameter);
  greeting = null;
  console.log(greeting + ": ", sampleParameter);
}

sampleMethod("test");
```

strictPropertyInitialization

tsconfig.json

```
{
  "compilerOptions": {
    "strictPropertyInitialization": false
  }
}
```

```
class Person {
    public name: string;
}
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictNullChecks": true,
    "strictPropertyInitialization": true
  }
}
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictNullChecks": true,
    "strictPropertyInitialization": true
  }
}
```

tsconfig.json

```jsonc
{
  "compilerOptions": {
    "strictNullChecks": true,
    "strictPropertyInitialization": true
  }
}
```

```
class Person {
    public name: string;
```

04-strictPropertyInitialization.ts 1 of 1 problem

```
Property 'name' has no initializer and is not definitely assigned in the constructor. ts(2564)
```

```
}
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictNullChecks": true,
    "strictPropertyInitialization": true
  }
}
```

```
class Person {
    public name: string = "";
}
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictNullChecks": true,
    "strictPropertyInitialization": true
  }
}
```

```
class Person {
    public name: string;
```

04-strictPropertyInitialization.ts 1 of 1 problem

```
Property 'name' has no initializer and is not definitely assigned in the constructor. ts(2564)
```

```
}
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictNullChecks": true,
    "strictPropertyInitialization": true
  }
}
```

```
class Person {
    public name: string = "";
}
```

strictFunctionTypes

tsconfig.json

```
{
  "compilerOptions": {
    "strictFunctionTypes": false
  }
}
```

```
function sampleMethod(sampleParameter: string) {
  console.log("Hello, " + sampleParameter.toLowerCase());
}

type StringOrNumber = (definedTypeParameter: string | number) => void;

let sampleMethod2: StringOrNumber = sampleMethod;

sampleMethod2(10);
```

tsconfig.json

```
1  {
2    "compilerOptions": {
3      "strictFunctionTypes": true
4    }
5  }
```

tsconfig.json

```json
{
  "compilerOptions": {
    "strictFunctionTypes": true
  }
}
```

```ts
function sampleMethod(sampleParameter: string) {
  console.log("Hello, " + sampleParameter.toLowerCase());
}

type StringOrNumber = (definedTypeParameter: string | number) => void;

let sampleMethod2: StringOrNumber = sampleMethod;
```

05-strictFunctionTypes.ts 1 of 1 problem

```
Type '(sampleParameter: string) => void' is not assignable to type 'StringOrNumber'.
  Types of parameters 'sampleParameter' and 'definedTypeParameter' are incompatible.
    Type 'string | number' is not assignable to type 'string'.
      Type 'number' is not assignable to type 'string'. ts(2322)
```

```ts

sampleMethod2(10);
```

tsconfig.json

```json
{
  "compilerOptions": {
    "strictFunctionTypes": true
  }
}
```

```ts
function sampleMethod(sampleParameter: string | number) {
  console.log("Hello, " + sampleParameter.toLowerCase());
}

type StringOrNumber = (definedTypeParameter: string | number) => void;

let sampleMethod2: StringOrNumber = sampleMethod;

sampleMethod2(10);
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictFunctionTypes": false
  }
}
```

```
function sampleMethod(sampleParameter: string) {
  console.log("Hello, " + sampleParameter.toLowerCase());
}

type StringOrNumber = (definedTypeParameter: string | number) => void;

let sampleMethod2: StringOrNumber = sampleMethod;

sampleMethod2(10);
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictFunctionTypes": true
  }
}
```

```
function sampleMethod(sampleParameter: string | number) {
  console.log("Hello, " + sampleParameter.toLowerCase());
}

type StringOrNumber = (definedTypeParameter: string | number) => void;

let sampleMethod2: StringOrNumber = sampleMethod;

sampleMethod2(10);
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictFunctionTypes": false
  }
}
```

```
function sampleMethod(sampleParameter: string) {
  console.log("Hello, " + sampleParameter.toLowerCase());
}

type StringOrNumber = (definedTypeParameter: string | number) => void;

let sampleMethod2: StringOrNumber = sampleMethod;

sampleMethod2(10);
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictFunctionTypes": true
  }
}
```

```
function sampleMethod(sampleParameter: string) {
  console.log("Hello, " + sampleParameter.toLowerCase());
}

type StringOrNumber = (definedTypeParameter: string | number) => void;

let sampleMethod2: StringOrNumber = sampleMethod;
```

05-strictFunctionTypes.ts 1 of 1 problem

```
Type '(sampleParameter: string) => void' is not assignable to type 'StringOrNumber'.
  Types of parameters 'sampleParameter' and 'definedTypeParameter' are incompatible.
    Type 'string | number' is not assignable to type 'string'.
      Type 'number' is not assignable to type 'string'. ts(2322)
```

```

sampleMethod2(10);
```

tsconfig.json

```json
{
  "compilerOptions": {
    "strictFunctionTypes": true
  }
}
```

```ts
function sampleMethod(sampleParameter: string | number) {
  console.log("Hello, " + sampleParameter.toLowerCase());
}

type StringOrNumber = (definedTypeParameter: string | number) => void;

let sampleMethod2: StringOrNumber = sampleMethod;

sampleMethod2(10);
```

strictBindCallApply

noFallthroughCasesInSwitch

tsconfig.json

```json
{
  "compilerOptions": {
    "noFallthroughCasesInSwitch": false
  }
}
```

```ts
const lotteryNumber: number = 0;

switch (lotteryNumber) {
  case 0:
    console.log("even");
  case 1:
    console.log("odd");
    break;
}
```

tsconfig.json

```
{
  "compilerOptions": {
    "noFallthroughCasesInSwitch": true
  }
}
```

tsconfig.json

```json
{
  "compilerOptions": {
    "noFallthroughCasesInSwitch": true
  }
}
```

```ts
const lotteryNumber: number = 0;

switch (lotteryNumber) {
  case 0:
```

⚠ 06-noFallthroughCasesInSwitch.ts 1 of 1 problem

Fallthrough case in switch. ts(7029)

```ts
    console.log("even");
  case 1:
    console.log("odd");
    break;
}
```

tsconfig.json

```json
{
  "compilerOptions": {
    "noFallthroughCasesInSwitch": true
  }
}
```

```ts
const lotteryNumber: number = 0;

switch (lotteryNumber) {
  case 0:
    console.log("even");
    break;
  case 1:
    console.log("odd");
    break;
}
```

tsconfig.json

```json
{
  "compilerOptions": {
    "noFallthroughCasesInSwitch": true
  }
}
```

```ts
const lotteryNumber: number = 0;

switch (lotteryNumber) {
  case 0:
```

⚠ 06-noFallthroughCasesInSwitch.ts 1 of 1 problem

Fallthrough case in switch. ts(7029)

```ts
    console.log("even");
  case 1:
    console.log("odd");
    break;
}
```

tsconfig.json

```json
{
  "compilerOptions": {
    "noFallthroughCasesInSwitch": true
  }
}
```

```ts
const lotteryNumber: number = 0;

switch (lotteryNumber) {
  case 0:
    console.log("even");
    break;
  case 1:
    console.log("odd");
    break;
}
```

exactOptionalPropertyTypes

tsconfig.json

```json
{
  "compilerOptions": {
    "exactOptionalPropertyTypes": false
  }
}
```

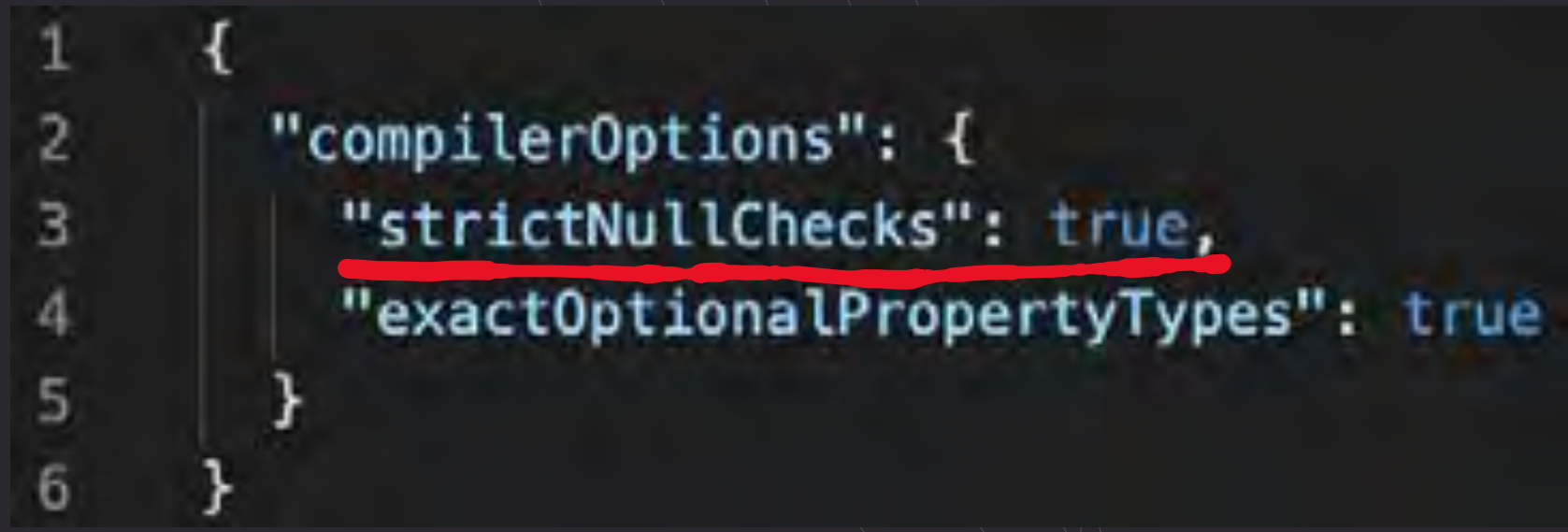
```ts
interface Person {
  name: string;
  age?: number; // Equivalent to "age?: number | undefined;"
}

const p: Person = {
  name: "Daniel",
  age: undefined,
};
```

tsconfig.json

```
{
  "compilerOptions": {
    "strictNullChecks": true,
    "exactOptionalPropertyTypes": true
  }
}
```

tsconfig.json



tsconfig.json

```json
{
  "compilerOptions": {
    "strictNullChecks": true,
    "exactOptionalPropertyTypes": true
  }
}
```

```ts
interface Person {
  name: string;
  age?: number; // Equivalent to "age?: number | undefined;"
}

const p: Person = {
  name: "Daniel",
  age: undefined,
};

```

07-exactOptionalPropertyTypes.ts 1 of 1 problem

Type 'undefined' is not assignable to type 'number'. ts(2322)

07-exactOptionalPropertyTypes.ts(3, 3): The expected type comes from property 'age' which is declared here on type 'Person'

tsconfig.json

```json
{
  "compilerOptions": {
    "strictNullChecks": true,
    "exactOptionalPropertyTypes": true
  }
}
```

```ts
interface Person {
  name: string;
  age?: number | undefined;
}

const p: Person = {
  name: "Daniel",
  age: undefined,
};
```

tsconfig.json

```json
{
  "compilerOptions": {
    "strictNullChecks": true,
    "exactOptionalPropertyTypes": true
  }
}
```

```ts
interface Person {
  name: string;
  age?: number; // Equivalent to "age?: number | undefined;"
}

const p: Person = {
  name: "Daniel",
  age: undefined,
};
```

07-exactOptionalPropertyTypes.ts 1 of 1 problem

Type 'undefined' is not assignable to type 'number'. ts(2322)

07-exactOptionalPropertyTypes.ts(3, 3): The expected type comes from property 'age' which is declared here on type 'Person'

tsconfig.json

```json
{
  "compilerOptions": {
    "strictNullChecks": true,
    "exactOptionalPropertyTypes": true
  }
}
```

```ts
interface Person {
  name: string;
  age?: number | undefined;
}

const p: Person = {
  name: "Daniel",
  age: undefined,
};
```

useUnknownInCatchVariables

tsconfig.json

```
{
  "compilerOptions": {
    "useUnknownInCatchVariables": false
  }
}
```

```
function main() {
  console.log("main");
  try {
    console.log("try");
  } catch (err) {
    console.log(err.message);
  }
}

main();
```

tsconfig.json

```
{
  "compilerOptions": {
    "useUnknownInCatchVariables": true
  }
}
```

tsconfig.json

```json
{
  "compilerOptions": {
    "useUnknownInCatchVariables": true
  }
}
```

```ts
function main() {
  console.log("main");
  try {
    console.log("try");
  } catch (err) {
    console.log("catch: ", err.message);
  }
}

main();
```

08-useUnknownInCatchVariables.ts 1 of 1 problem

Property 'message' does not exist on type 'unknown'. ts(2339)

tsconfig.json

```json
{
  "compilerOptions": {
    "useUnknownInCatchVariables": true
  }
}
```

```ts
function main() {
  console.log("main");
  try {
    console.log("try");
  } catch (err) {
    if (err instanceof Error) {
      console.log("catch: ", err.message);
    }
  }
}

main();
```

tsconfig.json

```json
{
  "compilerOptions": {
    "useUnknownInCatchVariables": true
  }
}
```

```ts
function main() {
  console.log("main");
  try {
    console.log("try");
  } catch (err) {
    console.log("catch: ", err.message);
  }
}

main();
```

08-useUnknownInCatchVariables.ts 1 of 1 problem

Property 'message' does not exist on type 'unknown'. ts(2339)

tsconfig.json

```json
{
  "compilerOptions": {
    "useUnknownInCatchVariables": true
  }
}
```

```ts
function main() {
  console.log("main");
  try {
    console.log("try");
  } catch (err) {
    if (err instanceof Error) {
      console.log("catch: ", err.message);
    }
  }
}

main();
```

alwaysStrict

Always Strict - `alwaysStrict`

Ensures that your files are parsed in the ECMAScript strict mode, and emit “use strict” for each source file.

ECMAScript strict mode was introduced in ES5 and provides behavior tweaks to the runtime of the JavaScript engine to improve performance, and makes a set of errors throw instead of silently ignoring them.

Recommended:
True

Default:
`false`, unless `strict` is set

Related:
`strict`

Released:
2.1

Always Strict - `alwaysStrict`

Ensures that your files are parsed in the ECMAScript strict mode, and emit "use strict" for each source file.

ECMAScript strict mode was introduced in ES5 and provides behavior tweaks to the runtime of the JavaScript engine to improve performance, and makes a set of errors throw instead of silently ignoring them.

Recommended:
True

Default:
`false`, unless `strict` is set

Related:
`strict`

Released:
2.1

Sloppy Mode vs Strict Mode

strict

Strict - `strict`

The `strict` flag enables a wide range of type checking behavior that results in stronger guarantees of program correctness. Turning this on is equivalent to enabling all of the *strict mode family* options, which are outlined below. You can then turn off individual strict mode family checks as needed.

Future versions of TypeScript may introduce additional stricter checking under this flag, so upgrades of TypeScript might result in new type errors in your program. When appropriate and possible, a corresponding flag will be added to disable that behavior.

Recommended:
True

Default:
`false`

Related:
`alwaysStrict`, `strictNullChecks`, `strictBindCallApply`, `strictFunctionTypes`, `strictPropertyInitiali` `noImplicitAny`, `noImplic` `useUnknownInCatchVaria`

Released:
2.3

Strict - `strict`

The `strict` flag enables a wide range of type checking behavior that results in stronger guarantees of program correctness. Turning this on is equivalent to enabling all of the *strict mode family* options, which are outlined below. You can then turn off individual strict mode family checks as needed.

Future versions of TypeScript may introduce additional stricter checking under this flag, so upgrades of TypeScript might result in new type errors in your program. When appropriate and possible, a corresponding flag will be added to disable that behavior.

Recommended:
True

Default:
`false`

Related:
`alwaysStrict`,
`strictNullChecks`,
`strictBindCallApply`,
`strictFunctionTypes`,
`strictPropertyInitiali`
`noImplicitAny`, `noImplic`
`useUnknownInCatchVaria`

Released:
2.3

Compiler Options

Top Level	files, extends, include, exclude and references
"compilerOptions"	
Type Checking	allowUnreachableCode, allowUnusedLabels, alwaysStrict, exactOptionalPropertyTypes, noFallthroughCasesInSwitch, noImplicitAny, noImplicitOverride, noImplicitReturns, noImplicitThis, noPropertyAccessFromIndexSignature, noUncheckedIndexedAccess, noUnusedLocals, noUnusedParameters, strict, strictBindCallApply, strictFunctionTypes, strictNullChecks, strictPropertyInitialization and useUnknownInCatchVariables
Modules	allowUmdGlobalAccess, baseUrl, module, moduleResolution, noResolve, paths, resolveJsonModule, rootDir, rootDirs, typeRoots and types
Emit	declaration, declarationDir, declarationMap, downlevelIteration, emitBOM, emitDeclarationOnly, importHelpers, importsNotUsedAsValues, inlineSourceMap, inlineSources, mapRoot, newLine, noEmit, noEmitHelpers, noEmitOnError, outDir, outFile, preserveConstEnums, removeComments, sourceMap, sourceRoot and stripInternal
JavaScript Support	allowJs, checkJs and maxNodeModuleJsDepth
Editor Support	disableSizeLimit and plugins
Interop Constraints	allowSyntheticDefaultImports, esModuleInterop, forceConsistentCasingInFileNames, isolatedModules and preserveSymlinks
Backwards Compatibility	charset, keyofStringsOnly, noImplicitUseStrict, noStrictGenericChecks, out, suppressExcessPropertyErrors and suppressImplicitAnyIndexErrors

importsNotUsedAsValues

Imports Not Used As Values - `importsNotUsedAsValues`

This flag controls how `import` works, there are 3 different options:

- `remove` : The default behavior of dropping `import` statements which only reference types.
- `preserve` : Preserves all `import` statements whose values or types are never used. This can cause imports/side-effects to be preserved.
- `error` : This preserves all imports (the same as the preserve option), but will error when a value import is only used as a type. This might be useful if you want to ensure no values are being accidentally imported, but still make side-effect imports explicit.

This flag works because you can use `import type` to explicitly create an `import` statement which should never be emitted into JavaScript.

Allowed: remove, preserve, error

Released: 3.8

Imports Not Used As Values - `importsNotUsedAsValues`

This flag controls how `import` works, there are 3 different options:

- `remove` : The default behavior of dropping `import` statements which only reference types.
- `preserve` : Preserves all `import` statements whose values or types are never used. This can cause imports/side-effects to be preserved.
- `error` : This preserves all imports (the same as the preserve option), but will error when a value import is only used as a type. This might be useful if you want to ensure no values are being accidentally imported, but still make side-effect imports explicit.

This flag works because you can use `import type` to explicitly create an `import` statement which should never be emitted into JavaScript.

Allowed:
remove,
preserve,
error

Released:
3.8

skipLibCheck
&
skipDefaultLibCheck

Completeness

Skip Default Lib Check - `skipDefaultLibCheck`

Use `skipLibCheck` instead. Skip type checking of default library declaration files.

Default:
`false`

Skip Lib Check - `skipLibCheck`

Skip type checking of declaration files.

This can save time during compilation at the expense of type-system accuracy. For example, two libraries could define two copies of the same `type` in an inconsistent way. Rather than doing a full check of all `d.ts` files, TypeScript will type check the code you specifically refer to in your app's source code.

A common case where you might think to use `skipLibCheck` is when there are two copies of a library's types in your `node_modules`. In these cases, you should consider using a feature like yarn's resolutions to ensure there is only one copy of that dependency in your tree or investigate how to ensure there is only one copy by understanding the dependency resolution to fix the issue without additional tooling.

Recommended:
True

Default:
`false`

Released:
2.0

Completeness

Skip Default Lib Check - `skipDefaultLibCheck`

Use `skipLibCheck` instead. Skip type checking of default library declaration files.

Default:
`false`

Skip Lib Check - `skipLibCheck`

Skip type checking of declaration files.

This can save time during compilation at the expense of type-system accuracy. For example, two libraries could define two copies of the same `type` in an inconsistent way. Rather than doing a full check of all `d.ts` files, TypeScript will type check the code you specifically refer to in your app's source code.

A common case where you might think to use `skipLibCheck` is when there are two copies of a library's types in your `node_modules`. In these cases, you should consider using a feature like yarn's resolutions to ensure there is only one copy of that dependency in your tree or investigate how to ensure there is only one copy by understanding the dependency resolution to fix the issue without additional tooling.

Recommended:
True

Default:
`false`

Released:
2.0

== npm overrides

forceConsistentCasingInFileNames

Force Consistent Casing In File Names - `forceConsistentCasingInFileNames`

TypeScript follows the case sensitivity rules of the file system it's running on. This can be problematic if some developers are working in a case-sensitive file system and others aren't. If a file attempts to import `fileManager.ts` by specifying `./FileManager.ts` the file will be found in a case-insensitive file system, but not on a case-sensitive file system.

When this option is set, TypeScript will issue an error if a program tries to include a file by a casing different from the casing on disk.

Recommended:
True

Default:
`false`

Force Consistent Casing In File Names - `forceConsistentCasingInFileNames`

TypeScript follows the case sensitivity rules of the file system it's running on. This can be problematic if some developers are working in a case-sensitive file system and others aren't. If a file attempts to import `fileManager.ts` by specifying `./FileManager.ts` the file will be found in a case-insensitive file system, but not on a case-sensitive file system.

When this option is set, TypeScript will issue an error if a program tries to include a file by a casing different from the casing on disk.

Recommended:
True

Default:
`false`

noImplicitUseStrict

No Implicit Use Strict - `noImplicitUseStrict`

You shouldn't need this. By default, when emitting a module file to a non-ES6 target, TypeScript emits a `"use strict";` prologue at the top of the file. This setting disables the prologue.

Default: `false`

```
define(["require", "exports"], function (require, exports) {
    exports.__esModule = true;
    exports.fn = void 0;
    function fn() { }
    exports.fn = fn;
});
```

```
define(["require", "exports"], function (require, exports) {
    "use strict";
    exports.__esModule = true;
    exports.fn = void 0;
    function fn() { }
    exports.fn = fn;
});
```

No Implicit Use Strict - `noImplicitUseStrict`

You shouldn't need this. By default, when emitting a module file to a non-ES6 target, TypeScript emits a `"use strict";` prologue at the top of the file. This setting disables the prologue.

Default: `false`

```
define(["require", "exports"], function (require, exports) {
    exports.__esModule = true;
    exports.fn = void 0;
    function fn() { }
    exports.fn = fn;
});
```

```
define(["require", "exports"], function (require, exports) {
    "use strict";
    exports.__esModule = true;
    exports.fn = void 0;
    function fn() { }
    exports.fn = fn;
});
```

noStrictGenericChecks

No Strict Generic Checks - `noStrictGenericChecks`

TypeScript will unify type parameters when comparing two generic functions.

```
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) {
  b = a; // Ok
  a = b; // Error
Type 'B' is not assignable to type 'A'.
  Types of parameters 'y' and 'y' are incompatible.
    Type 'U' is not assignable to type 'T'.
      'T' could be instantiated with an arbitrary type which could be
unrelated to 'U'.
}
```

This flag can be used to remove that check.

Default:
`false`

Released:
2.4

No Strict Generic Checks - `noStrictGenericChecks`

TypeScript will unify type parameters when comparing two generic functions.

```
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) {
  b = a; // Ok
  a = b; // Error
Type 'B' is not assignable to type 'A'.
  Types of parameters 'y' and 'y' are incompatible.
    Type 'U' is not assignable to type 'T'.
      'T' could be instantiated with an arbitrary type which could be
unrelated to 'U'.
}
```

This flag can be used to remove that check.

Default: `false`

Released: 2.4

suppressExcessPropertyErrors

Suppress Excess Property Errors - `suppressExcessPropertyErrors`

This disables reporting of excess property errors, such as the one shown in the following example:

Default: `false`

```
type Point = { x: number; y: number };
const p: Point = { x: 1, y: 3, m: 10 };
```

```
Type '{ x: number; y: number; m: number; }' is not assignable to type
'Point'.
  Object literal may only specify known properties, and 'm' does not exist
in type 'Point'.
```

This flag was added to help people migrate to the stricter checking of new object literals in TypeScript 1.6.

We don't recommend using this flag in a modern codebase, you can suppress one-off cases where you need it using `// @ts-ignore` .

Suppress Excess Property Errors - `suppressExcessPropertyErrors`

This disables reporting of excess property errors, such as the one shown in the following example:

Default: `false`

```
type Point = { x: number; y: number };
const p: Point = { x: 1, y: 3, m: 10 };
```

```
Type '{ x: number; y: number; m: number; }' is not assignable to type
'Point'.
  Object literal may only specify known properties, and 'm' does not exist
in type 'Point'.
```

This flag was added to help people migrate to the stricter checking of new object literals in TypeScript 1.6.

We don't recommend using this flag in a modern codebase, you can suppress one-off cases where you need it using `// @ts-ignore`.

suppressImplicitAnyIndexErrors

Suppress Implicit Any Index Errors - `suppressImplicitAnyIndexErrors`

Turning `suppressImplicitAnyIndexErrors` on suppresses reporting the error about implicit anys when indexing into objects, as shown in the following example:

```
const obj = { x: 10 };
console.log(obj["foo"]);
```

```
Element implicitly has an 'any' type because expression of type '"foo"'
can't be used to index type '{ x: number; }'.
  Property 'foo' does not exist on type '{ x: number; }'.
```

Using `suppressImplicitAnyIndexErrors` is quite a drastic approach. It is recommended to use a `@ts-ignore` comment instead:

```
const obj = { x: 10 };
// @ts-ignore
console.log(obj["foo"]);
```

Default:
`false`

Related:
`noImplicitAny`

Suppress Implicit Any Index Errors - `suppressImplicitAnyIndexErrors`

Turning `suppressImplicitAnyIndexErrors` on suppresses reporting the error about implicit anys when indexing into objects, as shown in the following example:

```
const obj = { x: 10 };
console.log(obj["foo"]);
```

```
Element implicitly has an 'any' type because expression of type '"foo"'
can't be used to index type '{ x: number; }'.
  Property 'foo' does not exist on type '{ x: number; }'.
```

Using `suppressImplicitAnyIndexErrors` is quite a drastic approach. It is recommended to use a `@ts-ignore` comment instead:

```
const obj = { x: 10 };
// @ts-ignore
console.log(obj["foo"]);
```

Default:
`false`

Related:
`noImplicitAny`

disableFilenameBasedTypeAcquisition

disableFilenameBasedTypeAcquisition - `disableFilenameBasedTypeAcquisition`

TypeScript's type acquisition can infer what types should be added based on filenames in a project. This means that having a file like `jquery.js` in your project would automatically download the types for JQuery from DefinitelyTyped.

You can disable this via `disableFilenameBasedTypeAcquisition`.

```
{
  "typeAcquisition": {
    "disableFilenameBasedTypeAcquisition": true
  }
}
```

Released:
4.1

disableFilenameBasedTypeAcquisition - `disableFilenameBasedTypeAcquisition`

TypeScript's type acquisition can infer what types should be added based on filenames in a project. This means that having a file like `jquery.js` in your project would automatically download the types for JQuery from DefinitelyTyped.

You can disable this via `disableFilenameBasedTypeAcquisition`.

```
{
  "typeAcquisition": {
    "disableFilenameBasedTypeAcquisition": true
  }
}
```

Released:
4.1

allowJs

default: false

checkJs

default: false

const checkJS = allowJs === true ? true : false

maxNodeModuleJsDepth

default: 0

const maxNodeModuleJsDepth = allowJs === true ? true : false

noPropertyAccessFromIndexSignature

https://www.typescriptlang.org/tsconfig#noPropertyAccessFromIndexSignature

noUncheckedIndexedAccess

https://www.typescriptlang.org/tsconfig#noUncheckedIndexedAccess

allowSyntheticDefaultImports

https://www.typescriptlang.org/tsconfig#allowSyntheticDefaultImports

disableSizeLimit

Editor Support

Disable Size Limit - `disableSizeLimit`

To avoid a possible memory bloat issues when working with very large JavaScript projects, there is an upper limit to the amount of memory TypeScript will allocate. Turning this flag on will remove the limit.

Default: `false`

Editor Support

Disable Size Limit - `disableSizeLimit`

To avoid a possible memory bloat issues when working with very large JavaScript projects, there is an upper limit to the amount of memory TypeScript will allocate. Turning this flag on will remove the limit.

Default: `false`

Official docs

https://www.typescriptlang.org/tsconfig

Official docs

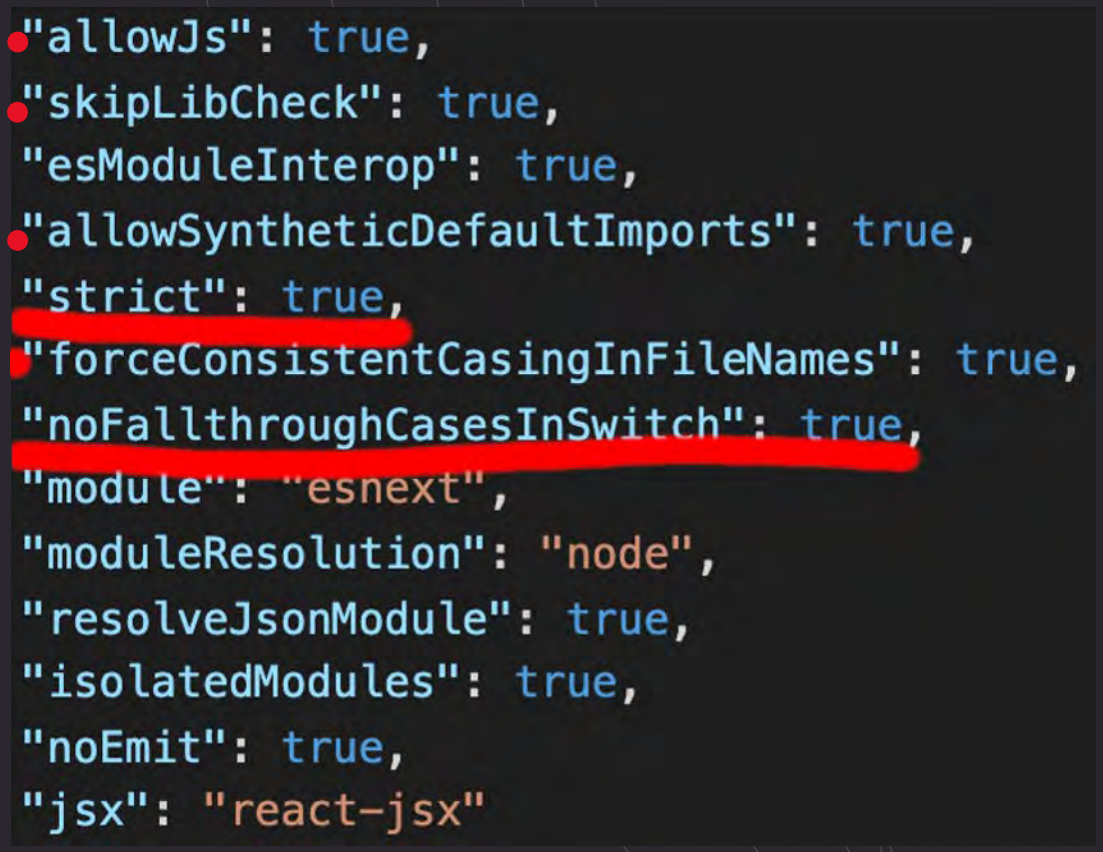
https://www.typescriptlang.org/docs/handbook/compiler-options.html

SPA Frameworks & Strict

```
"allowJs": true,
"skipLibCheck": true,
"esModuleInterop": true,
"allowSyntheticDefaultImports": true,
"strict": true,
"forceConsistentCasingInFileNames": true,
"noFallthroughCasesInSwitch": true,
"module": "esnext",
"moduleResolution": "node",
"resolveJsonModule": true,
"isolatedModules": true,
"noEmit": true,
"jsx": "react-jsx"
```

React

Rules by default (--strict doesn’t work)



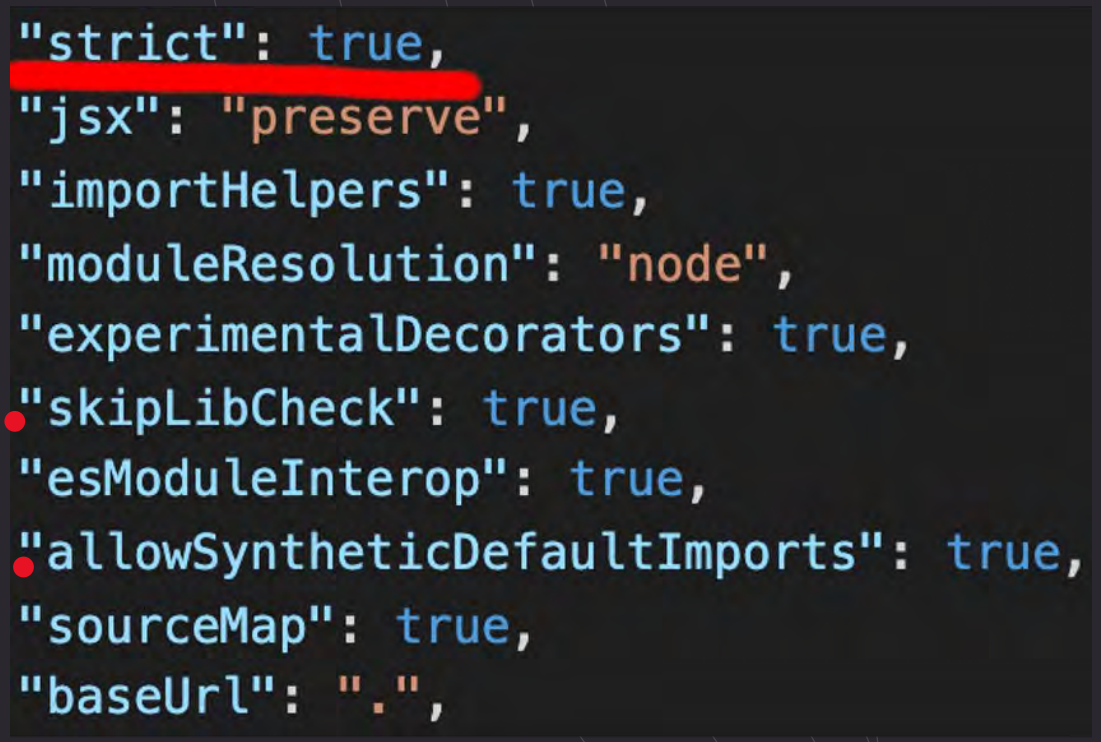
React

Rules by default (--strict doesn’t work)

```
"strict": true,
"jsx": "preserve",
"importHelpers": true,
"moduleResolution": "node",
"experimentalDecorators": true,
"skipLibCheck": true,
"esModuleInterop": true,
"allowSyntheticDefaultImports": true,
"sourceMap": true,
"baseUrl": ".",
```

Vue.js (2)

Rules by default (Vue.js 3 has extra “experimentalDecorators”)



Vue.js (2)

Rules by default (Vue.js 3 has extra “experimentalDecorators”)

```
"compilerOptions": {
  "baseUrl": "./",
  "outDir": "./dist/out-tsc",
  "sourceMap": true,
  "declaration": false,
  "downlevelIteration": true,
  "experimentalDecorators": true,
  "moduleResolution": "node",
  "importHelpers": true,
  "target": "es2015",
  "module": "es2020",
  "lib": [
    "es2018",
    "dom"
```

```
"compilerOptions": {
  "baseUrl": "./",
  "outDir": "./dist/out-tsc",
  "forceConsistentCasingInFileNames": true,
  "strict": true,
  "noImplicitReturns": true,
  "noFallthroughCasesInSwitch": true,
  "sourceMap": true,
  "declaration": false,
  "downlevelIteration": true,
  "experimentalDecorators": true,
  "moduleResolution": "node",
  "importHelpers": true,
  "target": "es2015",
```

Angular

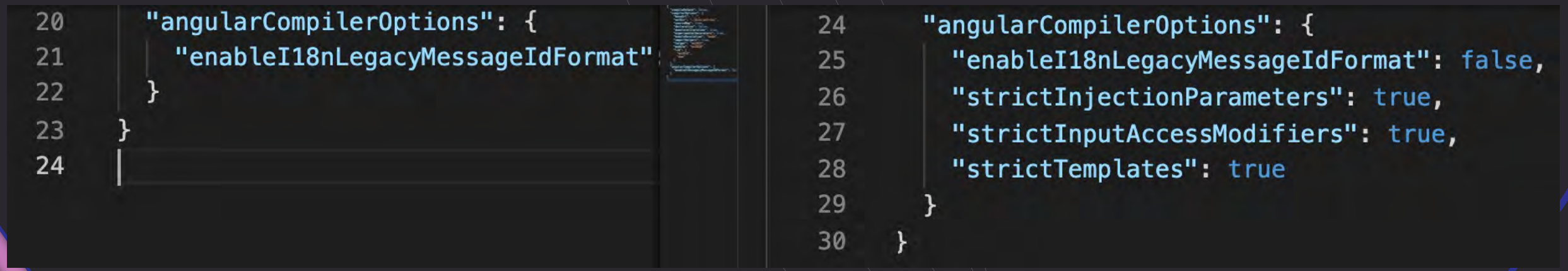
no strict vs --strict

```jsonc
"compilerOptions": {
  "baseUrl": "./",
  "outDir": "./dist/out-tsc",
  "sourceMap": true,
  "declaration": false,
  "downlevelIteration": true,
  "experimentalDecorators": true,
  "moduleResolution": "node",
  "importHelpers": true,
  "target": "es2015",
  "module": "es2020",
  "lib": [
    "es2018",
    "dom"
```

```jsonc
"compilerOptions": {
  "baseUrl": "./",
  "outDir": "./dist/out-tsc",
  "forceConsistentCasingInFileNames": true,
  "strict": true,
  "noImplicitReturns": true,
  "noFallthroughCasesInSwitch": true,
  "sourceMap": true,
  "declaration": false,
  "downlevelIteration": true,
  "experimentalDecorators": true,
  "moduleResolution": "node",
  "importHelpers": true,
  "target": "es2015",
```

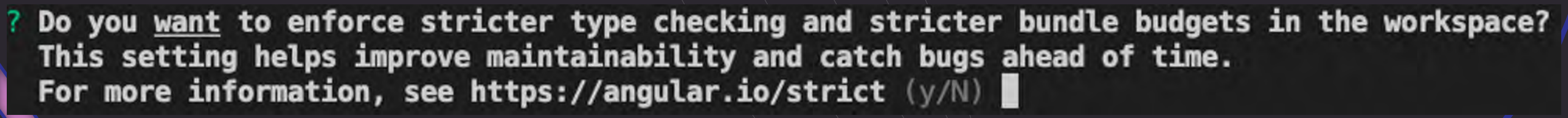
Angular

no strict vs --strict



Angular

no strict vs --strict



Angular

Forgot about –strict?

```
{
  "extends": "@tsconfig/svelte/tsconfig.json",

  "include": ["src/**/*"],
  "exclude": ["node_modules/*", "__sapper__/*", "public/*"]
}
```

```
"strict": false,
```

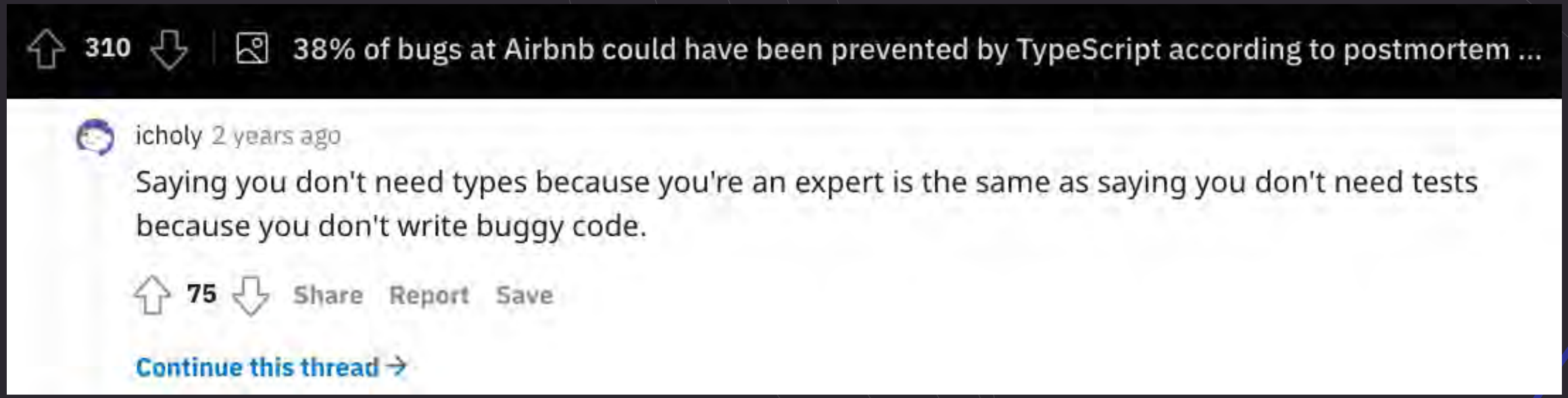
Svelte

No rules for TypeScript...

Default tsconfig	Angular	React	Svelte	Vue.js (2)
strict	ON / OFF	ON	OFF	ON
linter	2 out of 7	1 out of 7	OFF	OFF

Framework’s comparison

Angular is the winner



TypeScript helps preventing bugs

That’s not only what Reddit says…

To Type or Not to Type: Quantifying Detectable Bugs in JavaScript

Zheng Gao
University College London
London, UK
z.gao.12@ucl.ac.uk

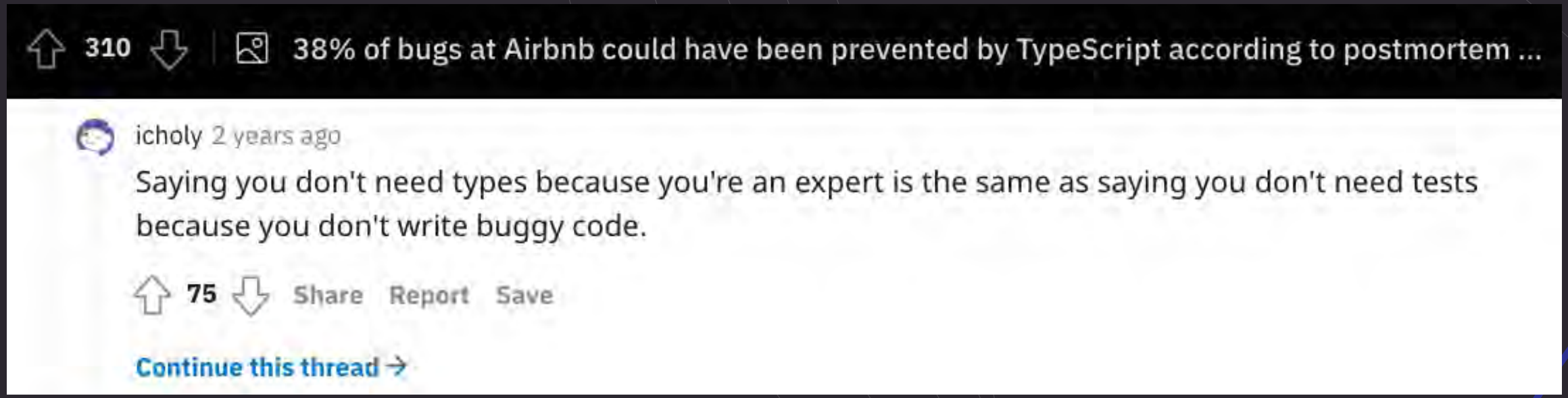
Christian Bird
Microsoft Research
Redmond, USA
cbird@microsoft.com

Earl T. Barr
University College London
London, UK
e.barr@ucl.ac.uk

search/completion and serving as documentation. Despite this uneven playing field, our central finding is that both static type systems find an important percentage of public bugs: both Flow 0.30 and TypeScript 2.0 successfully detect 15%!

TypeScript helps preventing bugs

It's a scientific fact



TypeScript helps preventing bugs

That’s not only what Reddit says…

Thanks