Skip to main content

Documentation

Type Checking JavaScript Files

TypeScript 2.3 and later support type-checking and reporting errors in `.js` files with `--checkJs`.

You can skip checking some files by adding `// @ts-nocheck` comment to them; conversely, you can choose to check only a few `.js` files by adding a `// @ts-check` comment to them without setting `--checkJs`. You can also ignore errors on specific lines by adding `// @ts-ignore` on the preceding line. Note that if you have a `tsconfig.json`, JS checking will respect strict flags like `noImplicitAny`, `strictNullChecks`, etc. However, because of the relative looseness of JS checking, combining strict flags with it may be surprising.

Here are some notable differences on how checking works in `.js` files compared to `.ts` files:

JSDoc types are used for type information

In a `.js` file, types can often be inferred just like in `.ts` files. Likewise, when types can't be inferred, they can be specified using JSDoc the same way that type annotations are used in a `.ts` file. Just like Typescript, `--noImplicitAny` will give you errors on the places that the compiler could not infer a type. (With the exception of open-ended object literals; see below for details.)

JSDoc annotations adorning a declaration will be used to set the type of that declaration. For example:

```
/** @type {number} */
var x;

x = 0;      // OK
x = false;  // Error: boolean is not assignable to number
```

You can find the full list of supported JSDoc patterns in the JSDoc support in JavaScript documentation (https://github.com/Microsoft/TypeScript/wiki/JSDoc-support-in-JavaScript).

Properties are inferred from assignments in class bodies

ES2015 does not have a means for declaring properties on classes. Properties are dynamically assigned, just like object literals.

In a `.js` file, the compiler infers properties from property assignments inside the class body. The type of properties is the type given in the constructor, unless it's not defined there, or the type in the constructor is undefined or null. In that case, the type is the union of the types of all the right-hand values in these assignments. Properties defined in the constructor are always assumed to exist, whereas ones defined just in methods, getters, or setters are considered optional.

```
class C {
    constructor() {
        this.constructorOnly = 0
        this.constructorUnknown = undefined
    }
    method() {
        this.constructorOnly = false // error, constructorOnly is a number
        this.constructorUnknown = "plunkbat" // ok, constructorUnknown is string | undefined
        this.methodOnly = 'ok'  // ok, but y could also be undefined
    }
    method2() {
        this.methodOnly = true  // also, ok, y's type is string | boolean | undefined
    }
}
```

If properties are never set in the class body, they are considered unknown. If your class has properties that are only read from, add and then annotate a declaration in the constructor with JSDoc to specify the type. You don't even have to give a value if it will be initialised later:

```
class C {
    constructor() {
        /** @type {number | undefined} */
        this.prop = undefined;
        /** @type {number | undefined} */
        this.count;
    }
}


let c = new C();
c.prop = 0;          // OK
c.count = "string";  // Error: string is not assignable to number|undefined
```

Constructor functions are equivalent to classes

Before ES2015, Javascript used constructor functions instead of classes. The compiler supports this pattern and understands constructor functions as equivalent to ES2015 classes. The property inference rules described above work exactly the same way.

```
function C() {
    this.constructorOnly = 0
    this.constructorUnknown = undefined
}
C.prototype.method = function() {
    this.constructorOnly = false // error
    this.constructorUnknown = "plunkbat" // OK, the type is string | undefined
}
```

CommonJS modules are supported

In a `.js` file, Typescript understands the CommonJS module format. Assignments to `exports` and `module.exports` are recognized as export declarations. Similarly, `require` function calls are recognized as module imports. For example:

```
// same as `import module "fs"`
const fs = require("fs");

// same as `export function readFile`
module.exports.readFile = function(f) {
    return fs.readFileSync(f);
}
```

The module support in Javascript is much more syntactically forgiving than Typescript's module support. Most combinations of assignments and declarations are supported.

Classes, functions, and object literals are namespaces

Classes are namespaces in `.js` files. This can be used to nest classes, for example:

```
class C {
}
C.D = class {
}
```

And, for pre-ES2015 code, it can be used to simulate static methods:

```
function Outer() {
  this.y = 2
}
Outer.Inner = function() {
  this.yy = 2
}
```

It can also be used to create simple namespaces:

```
var ns = {}
ns.C = class {
}
ns.func = function() {
}
```

Other variants are allowed as well:

```
// IIFE
var ns = (function (n) {
  return n || {};
})();
ns.CONST = 1

// defaulting to global
var assign = assign || function() {
  // code goes here
}
assign.extra = 1
```

Object literals are open-ended

In a `.ts` file, an object literal that initializes a variable declaration gives its type to the declaration. No new members can be added that were not specified in the original literal. This rule is relaxed in a `.js` file; object literals have an open-ended type (an index signature) that allows adding and looking up properties that were not defined originally. For instance:

```
var obj = { a: 1 };
obj.b = 2;  // Allowed
```

Object literals behave as if they have an index signature `[x:string]: any` that allows them to be treated as open maps instead of closed objects.

Like other special JS checking behaviors, this behavior can be changed by specifying a JSDoc type for the variable. For example:

```
/** @type {{a: number}} */
var obj = { a: 1 };
obj.b = 2;  // Error, type {a: number} does not have property b
```

null, undefined, and empty array initializers are of type any or any[]

Any variable, parameter or property that is initialized with null or undefined will have type any, even if strict null checks is turned on. Any variable, parameter or property that is initialized with [] will have type any[], even if strict null checks is turned on. The only exception is for properties that have multiple initializers as described above.

```
function Foo(i = null) {
    if (!i) i = 1;
    var j = undefined;
    j = 2;
    this.l = [];
}
var foo = new Foo();
foo.l.push(foo.i);
foo.l.push("end");
```

Function parameters are optional by default

Since there is no way to specify optionality on parameters in pre-ES2015 Javascript, all function parameters in `.js` file are considered optional. Calls with fewer arguments than the declared number of parameters are allowed.

It is important to note that it is an error to call a function with too many arguments.

For instance:

```
function bar(a, b) {
    console.log(a + " " + b);
}

bar(1);       // OK, second argument considered optional
bar(1, 2);
bar(1, 2, 3); // Error, too many arguments
```

JSDoc annotated functions are excluded from this rule. Use JSDoc optional parameter syntax to express optionality. e.g.:

```
/**
 * @param {string} [somebody] - Somebody's name.
 */
function sayHello(somebody) {
    if (!somebody) {
        somebody = 'John Doe';
    }
    console.log('Hello ' + somebody);
}

sayHello();
```

Var-args parameter declaration inferred from use of `arguments`

A function whose body has a reference to the `arguments` reference is implicitly considered to have a var-arg parameter (i.e. `(...arg: any[]) => any`). Use JSDoc var-arg syntax to specify the type of the arguments.

```
/** @param {...number} args */
function sum(/* numbers */) {
    var total = 0
    for (var i = 0; i < arguments.length; i++) {
      total += arguments[i]
    }
    return total
}
```

Unspecified type parameters default to `any`

Since there is no natural syntax for specifying generic type parameters in Javascript, an unspecified type parameter defaults to `any`.

In extends clause:

For instance, `React.Component` is defined to have two type parameters, `Props` and `State`. In a `.js` file, there is no legal way to specify these in the extends clause. By default the type arguments will be `any`:

```
import { Component } from "react";

class MyComponent extends Component {
    render() {
        this.props.b; // Allowed, since this.props is of type any
    }
}
```

Use JSDoc `@augments` to specify the types explicitly. for instance:

```
import { Component } from "react";

/**
 * @augments {Component<{a: number}, State>}
 */
class MyComponent extends Component {
    render() {
        this.props.b; // Error: b does not exist on {a:number}
    }
}
```

In JSDoc references

An unspecified type argument in JSDoc defaults to any:

```
/** @type{Array} */
var x = [];

x.push(1);        // OK
x.push("string"); // OK, x is of type Array<any>


/** @type{Array.<number>} */
var y = [];

y.push(1);        // OK
y.push("string"); // Error, string is not assignable to number
```

In function calls

A call to a generic function uses the arguments to infer the type parameters. Sometimes this process fails to infer any types, mainly because of lack of inference sources; in these cases, the type parameters will default to `any`. For example:

```
var p = new Promise((resolve, reject) => { reject() });

p; // Promise<any>;
```

Supported JSDoc

The list below outlines which constructs are currently supported when using JSDoc annotations to provide type information in JavaScript files.

Note any tags which are not explicitly listed below (such as `@async`) are not yet supported.

- `@type`
- `@param` (or `@arg` or `@argument`)
- `@returns` (or `@return`)
- `@typedef`
- `@callback`
- `@template`
- `@class` (or `@constructor`)
- `@this`
- `@extends` (or `@augments`)
- `@enum`

The meaning is usually the same, or a superset, of the meaning of the tag given at usejsdoc.org. The code below describes the differences and gives some example usage of each tag.

`@type`

You can use the "@type" tag and reference a type name (either primitive, defined in a TypeScript declaration, or in a JSDoc "@typedef" tag). You can use any Typescript type, and most JSDoc types.

```
/**
 * @type {string}
 */
var s;

/** @type {Window} */
var win;

/** @type {PromiseLike<string>} */
var promisedString;

// You can specify an HTML Element with DOM properties
/** @type {HTMLElement} */
var myElement = document.querySelector(selector);
element.dataset.myData = '';
```

`@type` can specify a union type — for example, something can be either a string or a boolean.

```
/**
 * @type {(string | boolean)}
 */
var sb;
```

Note that parentheses are optional for union types.

```
/**
 * @type {string | boolean}
 */
var sb;
```

You can specify array types using a variety of syntaxes:

```
/** @type {number[]} */
var ns;
/** @type {Array.<number>} */
var nds;
/** @type {Array<number>} */
var nas;
```

You can also specify object literal types. For example, an object with properties 'a' (string) and 'b' (number) uses the following syntax:

```
/** @type {{ a: string, b: number }} */
var var9;
```

You can specify map-like and array-like objects using string and number index signatures, using either standard JSDoc syntax or Typescript syntax.

```
/**
 * A map-like object that maps arbitrary `string` properties to `number`s.
 *
 * @type {Object.<string, number>}
 */
var stringToNumber;

/** @type {Object.<number, object>} */
var arrayLike;
```

The preceding two types are equivalent to the Typescript types `{ [x: string]: number }` and `{ [x: number]: any }`. The compiler understands both syntaxes.

You can specify function types using either Typescript or Closure syntax:

```
/** @type {function(string, boolean): number} Closure syntax */
var sbn;
/** @type {(s: string, b: boolean) => number} Typescript syntax */
var sbn2;
```

Or you can just use the unspecified `Function` type:

```
/** @type {Function} */
var fn7;
/** @type {function} */
var fn6;
```

Other types from Closure also work:

```
/**
 * @type {*} - can be 'any' type
 */
var star;
/**
 * @type {?} - unknown type (same as 'any')
 */
var question;
```

Casts

Typescript borrows cast syntax from Closure. This lets you cast types to other types by adding a `@type` tag before any parenthesized expression.

```
/**
 * @type {number | string}
 */
var numberOrString = Math.random() < 0.5 ? "hello" : 100;
var typeAssertedNumber = /** @type {number} */ (numberOrString)
```

Import types

You can also import declarations from other files using import types. This syntax is Typescript-specific and differs from the JSDoc standard:

```
/**
 * @param p { import("./a").Pet }
 */
function walk(p) {
    console.log(`Walking ${p.name}...`);
}
```

import types can also be used in type alias declarations:

```
/**
 * @typedef Pet { import("./a").Pet }
 */

/**
 * @type {Pet}
 */
var myPet;
myPet.name;
```

import types can be used to get the type of a value from a module if you don't know the type, or if it has a large type that is annoying to type:

```
/**
 * @type {typeof import("./a").x }
 */
var x = require("./a").x;
```

`@param` and `@returns`

`@param` uses the same type syntax as `@type`, but adds a parameter name. The parameter may also be declared optional by surrounding the name with square brackets:

```
// Parameters may be declared in a variety of syntactic forms
/**
 * @param {string}  p1 - A string param.
 * @param {string=} p2 - An optional param (Closure syntax)
 * @param {string} [p3] - Another optional param (JSDoc syntax).
 * @param {string} [p4="test"] - An optional param with a default value
 * @return {string} This is the result
 */
function stringsStringStrings(p1, p2, p3, p4){
  // TODO
}
```

Likewise, for the return type of a function:

```
/**
 * @return {PromiseLike<string>}
 */
function ps(){}

/**
 * @returns {{ a: string, b: number }} - May use '@returns' as well as '@return'
 */
function ab(){}
```

`@typedef`, `@callback`, and `@param`

`@typedef` may be used to define complex types. Similar syntax works with `@param`.

```
/**
 * @typedef {Object} SpecialType - creates a new type named 'SpecialType'
 * @property {string} prop1 - a string property of SpecialType
 * @property {number} prop2 - a number property of SpecialType
 * @property {number=} prop3 - an optional number property of SpecialType
 * @prop {number} [prop4] - an optional number property of SpecialType
 * @prop {number} [prop5=42] - an optional number property of SpecialType with default
 */
/** @type {SpecialType} */
var specialTypeObject;
```

You can use either `object` or `Object` on the first line.

```
/**
 * @typedef {object} SpecialType1 - creates a new type named 'SpecialType'
 * @property {string} prop1 - a string property of SpecialType
 * @property {number} prop2 - a number property of SpecialType
 * @property {number=} prop3 - an optional number property of SpecialType
 */
/** @type {SpecialType1} */
var specialTypeObject1;
```

`@param` allows a similar syntax for one-off type specifications. Note that the nested property names must be prefixed with the name of the parameter:

```
/**
 * @param {Object} options - The shape is the same as SpecialType above
 * @param {string} options.prop1
 * @param {number} options.prop2
 * @param {number=} options.prop3
 * @param {number} [options.prop4]
 * @param {number} [options.prop5=42]
 */
function special(options) {
  return (options.prop4 || 1001) + options.prop5;
}
```

`@callback` is similar to `@typedef`, but it specifies a function type instead of an object type:

```
/**
 * @callback Predicate
 * @param {string} data
 * @param {number} [index]
 * @returns {boolean}
 */
/** @type {Predicate} */
const ok = s => !(s.length % 2);
```

Of course, any of these types can be declared using Typescript syntax in a single-line `@typedef`:

```
/** @typedef {{ prop1: string, prop2: string, prop3?: number }} SpecialType */
/** @typedef {(data: string, index?: number) => boolean} Predicate */
```

`@template`

You can declare generic types with the `@template` tag:

```
/**
 * @template T
 * @param {T} p1 - A generic parameter that flows through to the return type
 * @return {T}
 */
function id(x){ return x }
```

Use comma or multiple tags to declare multiple type parameters:

```
/**
 * @template T,U,V
 * @template W,X
 */
```

You can also specify a type constraint before the type parameter name. Only the first type parameter in a list is constrained:

```
/**
 * @template {string} K - K must be a string or string literal
 * @template {{ serious(): string }} Seriousalizable - must have a serious method
 * @param {K} key
 * @param {Seriousalizable} object
 */
function seriousalize(key, object) {
  // ????
}
```

`@constructor`

The compiler infers constructor functions based on this-property assignments, but you can make checking stricter and suggestions better if you add a `@constructor` tag:

```
/**
 * @constructor
 * @param {number} data
 */
function C(data) {
  this.size = 0;
  this.initialize(data); // Should error, initializer expects a string
}
/**
 * @param {string} s
 */
C.prototype.initialize = function (s) {
  this.size = s.length
}

var c = new C(0);
var result = C(1); // C should only be called with new
```

With `@constructor`, `this` is checked inside the constructor function `C`, so you will get suggestions for the `initialize` method and an error if you pass it a number. You will also get an error if you call `C` instead of constructing it.

Unfortunately, this means that constructor functions that are also callable cannot use `@constructor`.

`@this`

The compiler can usually figure out the type of `this` when it has some context to work with. When it doesn't, you can explicitly specify the type of `this` with `@this`:

```
/**
 * @this {HTMLElement}
 * @param {*} e
 */
function callbackForLater(e) {
    this.clientHeight = parseInt(e) // should be fine!
}
```

`@extends`

When Javascript classes extend a generic base class, there is nowhere to specify what the type parameter should be. The `@extends` tag provides a place for that type parameter:

```
/**
 * @template T
 * @extends {Set<T>}
 */
class SortableSet extends Set {
  // ...
}
```

Note that `@extends` only works with classes. Currently, there is no way for a constructor function extend a class.

`@enum`

The `@enum` tag allows you to create an object literal whose members are all of a specified type. Unlike most object literals in Javascript, it does not allow other members.

```
/** @enum {number} */
const JSDocState = {
  BeginningOfLine: 0,
  SawAsterisk: 1,
  SavingComments: 2,
}
```

Note that `@enum` is quite different from, and much simpler than, Typescript's `enum`. However, unlike Typescript's enums, `@enum` can have any type:

```
/** @enum {function(number): number} */
const Math = {
  add1: n => n + 1,
  id: n => -n,
  sub1: n => n - 1,
}
```

More examples

```
var someObj = {
  /**
   * @param {string} param1 - Docs on property assignments work
   */
  x: function(param1){}
};

/**
 * As do docs on variable assignments
 * @return {Window}
 */
let someFunc = function(){};

/**
 * And class methods
 * @param {string} greeting The greeting to use
 */
Foo.prototype.sayHi = (greeting) => console.log("Hi!");

/**
 * And arrow functions expressions
 * @param {number} x - A multiplier
 */
let myArrow = x => x * x;

/**
 * Which means it works for stateless function components in JSX too
 * @param {{a: string, b: number}} test - Some param
 */
var sfc = (test) => <div>{test.a.charAt(0)}</div>;

/**
 * A parameter can be a class constructor, using Closure syntax.
 *
 * @param {{new(...args: any[]): object}} C - The class to register
 */
function registerClass(C) {}

/**
 * @param {...string} p1 - A 'rest' arg (array) of strings. (treated as 'any')
 */
function fn10(p1){}

/**
 * @param {...string} p1 - A 'rest' arg (array) of strings. (treated as 'any')
 */
function fn9(p1) {
  return p1.join();
}
```

Patterns that are known NOT to be supported

Referring to objects in the value space as types doesn't work unless the object also creates a type, like a constructor function.

```
function aNormalFunction() {

}
/**
 * @type {aNormalFunction}
 */
var wrong;
/**
 * Use 'typeof' instead:
 * @type {typeof aNormalFunction}
 */
var right;
```

Postfix equals on a property type in an object literal type doesn't specify an optional property:

```
/**
 * @type {{ a: string, b: number= }}
 */
var wrong;
/**
 * Use postfix question on the property name instead:
 * @type {{ a: string, b?: number }}
 */
var right;
```

Nullable types only have meaning if `strictNullChecks` is on:

```
/**
 * @type {?number}
 * With strictNullChecks: true -- number | null
 * With strictNullChecks: off  -- number
 */
var nullable;
```

Non-nullable types have no meaning and are treated just as their original type:

```
/**
 * @type {!number}
 * Just has type number
 */
var normal;
```

Unlike JSDoc's type system, Typescript only allows you to mark types as containing null or not. There is no explicit non-nullability – if strictNullChecks is on, then `number` is not nullable. If it is off, then `number` is nullable.