These changes list where implementation differs between versions as the spec and compiler are simplified and inconsistencies are corrected.

> *For breaking changes to the compiler/services API, please check the [[API Breaking Changes]] page.*

TypeScript 4.5

`lib.d.ts` Changes for TypeScript 4.5

TypeScript 4.5 contains changes to its built-in declaration files which may affect your compilation; however, [these changes were fairly minimal](), and we expect most code will be unaffected.

Inference Changes from `Awaited`

Because `Awaited` is now used in `lib.d.ts` and as a result of `await`, you may see certain generic types change that might cause incompatibilities. This may cause issues when providing explicit type arguments to functions like `Promise.all`, `Promise.allSettled`, etc.

Often, you can make a fix by removing type arguments altogether.

```diff
- Promise.all<boolean, boolean>(...)
+ Promise.all(...)
```

More involved cases will require you to replace a list of type arguments with a single type argument of a tuple-like type.

```diff
- Promise.all<boolean, boolean>(...)
+ Promise.all<[boolean, boolean]>(...)
```

However, there will be occasions when a fix will be a little bit more involved, and replacing the types with a tuple of the original type arguments won't be enough. [One example where this occasionally comes up]() is when an element is possibly a `Promise` or non-`Promise`. In those cases, it's no longer okay to unwrap the underlying element type.

```diff
- Promise.all<boolean | undefined, boolean | undefined>(...)
+ Promise.all<[Promise<boolean> | undefined, Promise<boolean> | undefined]>(...)
```

Template Strings Use `.concat()`

Template strings in TypeScript previously just used the `+` operator when targeting ES3 or ES5; however, this leads to some divergences between the use of `.valueOf()` and `.toString()` which ends up being less spec-compliant. This is usually not noticeable, but is particularly important when using upcoming standard library additions like [Temporal]().

TypeScript now uses calls to `.concat()` on `strings`. This gives code the same behavior regardless of whether it targets ES3 and ES5, or ES2015 and later. Most code should be unaffected, but you might now see different results on values that define separate `valueOf()` and `toString()` methods.

```
import moment = require("moment");

// Before: "Moment: Wed Nov 17 2021 16:23:57 GMT-0800"
//  After: "Moment: 1637195037348"
console.log(`Moment: ${moment()}`);
```

More more information, see the original issue.

Compiler Options Checking at the Root of `tsconfig.json`

It's an easy mistake to accidentally forget about the `compilerOptions` section in a `tsconfig.json`. To help catch this mistake, in TypeScript 4.5, it is an error to add a top-level field which matches any of the available options in `compilerOptions` *without* having also defined `compilerOptions` in that `tsconfig.json`.

Restrictions on Assignability to Conditional Types

TypeScript no longer allows types to be assignable to conditional types that use `infer`, or that are distributive. Doing so previously often ended up causing major performance issues. For more information, see the specific change on GitHub.

TypeScript 4.4

`lib.d.ts` Changes for TypeScript 4.4

As with every TypeScript version, declarations for `lib.d.ts` (especially the declarations generated for web contexts), have changed. You can consult our list of known `lib.dom.d.ts` changes to understand what is impacted.

More-Compliant Indirect Calls for Imported Functions

In earlier versions of TypeScript, calling an import from CommonJS, AMD, and other non-ES module systems would set the `this` value of the called function. Specifically, in the following example, when calling `fooModule.foo()`, the `foo()` method will have `fooModule` set as the value of `this`.

```
// Imagine this is our imported module, and it has an export named 'foo'.
let fooModule = {
    foo() {
        console.log(this);
    }
};

fooModule.foo();
```

This is not the way exported functions in ECMAScript are supposed to work when we call them. That's why TypeScript 4.4 intentionally discards the `this` value when calling imported functions, by using the following emit.

```
// Imagine this is our imported module, and it has an export named 'foo'.
let fooModule = {
    foo() {
        console.log(this);
    }
};

// Notice we're actually calling '(0, fooModule.foo)' now, which is subtly
different.
(0, fooModule.foo)();
```

For more information, you can read up more here.

Using `unknown` in Catch Variables

Users running with the `--strict` flag may see new errors around `catch` variables being `unknown` due to the new `--useUnknownForCatchVariables` flag, especially if the existing code assumes only `Error` values have been caught. This often results in error messages such as:

```
Property 'message' does not exist on type 'unknown'.
Property 'name' does not exist on type 'unknown'.
Property 'stack' does not exist on type 'unknown'.
Object is of type 'unknown'.
```

To get around this, you can specifically add runtime checks to ensure that the thrown type matches your expected type. Otherwise, you can just use a type assertion, add an explicit `: any` to your catch variable, or turn off `--useUnknownInCatchVariables`.

Broader Always-Truthy Promise Checks

In prior versions, TypeScript introduced "Always Truthy Promise checks" to catch code where an `await` may have been forgotten; however, the checks only applied to named declarations. That meant that while this code would correctly receive an error...

```
async function foo(): Promise<boolean> {
    return false;
}

async function bar(): Promise<string> {
    const fooResult = foo();
    if (fooResult) {        // <- error! :D
        return "true";
    }
    return "false";
}
```

...the following code would not.

```ts
async function foo(): Promise<boolean> {
    return false;
}

async function bar(): Promise<string> {
    if (foo()) {             // <- no error :(
        return "true";
    }
    return "false";
}
```

TypeScript 4.4 now flags both. For more information, [read up on the original change](#).

Abstract Properties Do Not Allow Initializers

The following code is now an error because abstract properties may not have initializers:

```ts
abstract class C {
    abstract prop = 1;
    //       ~~~~
    // Property 'prop' cannot have an initializer because it is marked abstract.
}
```

Instead, you may only specify a type for the property:

```ts
abstract class C {
    abstract prop: number;
}
```

TypeScript 4.3

Union Enums Cannot Be Compared to Arbitrary Numbers

Certain `enum` s are considered *union* `enum` *s* when their members are either automatically filled in, or trivially written. In those cases, an enum can recall each value that it potentially represents.

In TypeScript 4.3, if a value with a union `enum` type is compared with a numeric literal that it could never be equal to, then the type-checker will isue an error.

```ts
enum E {
  A = 0,
  B = 1,
}

function doSomething(x: E) {
  // Error! This condition will always return 'false' since the types 'E' and '-1'
have no overlap.
  if (x === -1) {
```

```
    // ...
  }
}
```

As a workaround, you can re-write an annotation to include the appropriate literal type.

```
enum E {
  A = 0,
  B = 1,
}

// Include -1 in the type, if we're really certain that -1 can come through.
function doSomething(x: E | -1) {
  if (x === -1) {
    // ...
  }
}
```

You can also use a type-assertion on the value.

```
enum E {
  A = 0,
  B = 1,
}

function doSomething(x: E) {
  // Use a type asertion on 'x' because we know we're not actually just dealing with
values from 'E'.
  if ((x as number) === -1) {
    // ...
  }
}
```

Alternatively, you can re-declare your enum to have a non-trivial initializer so that any number is both assignable and comparable to that enum. This may be useful if the intent is for the enum to specify a few well-known values.

```
enum E {
  // the leading + on 0 opts TypeScript out of inferring a union enum.
  A = +0,
  B = 1,
}
```

For more details, see the original change

TypeScript 4.2

`noImplicitAny` Errors Apply to Loose `yield` Expressions

When a `yield` expression is captured, but isn't contextually typed (i.e. TypeScript can't figure out what the type is), TypeScript will now issue an implicit `any` error.

```
function* g1() {
  const value = yield 1; // report implicit any error
}

function* g2() {
  yield 1; // result is unused, no error
}

function* g3() {
  const value: string = yield 1; // result is contextually typed by type annotation
of `value`, no error.
}

function* g3(): Generator<number, void, string> {
  const value = yield 1; // result is contextually typed by return-type annotation
of `g3`, no error.
}
```

See more details in [the corresponding changes](#).

Type Arguments in JavaScript Are Not Parsed as Type Arguments

Type arguments were already not allowed in JavaScript, but in TypeScript 4.2, the parser will parse them in a more spec-compliant way. So when writing the following code in a JavaScript file:

```
f<T>(100)
```

TypeScript will parse it as the following JavaScript:

```
(f < T) > (100)
```

This may impact you if you were leveraging TypeScript's API to parse type constructs in JavaScript files, which may have occurred when trying to parse Flow files.

The `in` Operator No Longer Allows Primitive Types on the Right Side

In JavaScript, it is a runtime error to use a non-object type on the right side of the `in` operator. TypeScript 4.2 ensures this can be caught at design-time.

```
"foo" in 42
//       ~~
// error! The right-hand side of an 'in' expression must not be a primitive.
```

This check is fairly conservative for the most part, so if you have received an error about this, it is likely an issue in the code.

TypeScript 4.1

`abstract` Members Can't Be Marked `async`

Members marked as `abstract` can no longer be marked as `async`. The fix here is to remove the `async` keyword, since callers are only concerned with the return type.

`resolve`'s Parameters Are No Longer Optional in `Promise`s

When writing code like the following

```
new Promise(resolve => {
    doSomethingAsync(() => {
        doSomething();
        resolve();
    })
})
```

You may get an error like the following:

```
  resolve()
  ~~~~~~~~~
error TS2554: Expected 1 arguments, but got 0.
  An argument for 'value' was not provided.
```

This is because `resolve` no longer has an optional parameter, so by default, it must now be passed a value. Often this catches legitimate bugs with using `Promise`s. The typical fix is to pass it the correct argument, and sometimes to add an explicit type argument.

```
new Promise<number>(resolve => {
    //     ^^^^^^^^
    doSomethingAsync(value => {
        doSomething();
        resolve(value);
        //      ^^^^^
    })
})
```

However, sometimes `resolve()` really does need to be called without an argument. In these cases, we can give `Promise` an explicit `void` generic type argument (i.e. write it out as `Promise<void>`). This leverages new functionality in TypeScript 4.1 where a potentially-`void` trailing parameter can become optional.

```
new Promise<void>(resolve => {
    //     ^^^^^^
    doSomethingAsync(() => {
        doSomething();
        resolve();
```

```
    })
})
```

TypeScript 4.1 ships with a quick fix to help fix this break.

`any` and `unknown` are considered possibly falsy in `&&` expressions

Note: *This change, and the description of the previous behavior, apply only under* `--strictNullChecks`.

Previously, when an `any` or `unknown` appeared on the left-hand side of an `&&`, it was assumed to be definitely truthy, which made the type of the expression the type of the right-hand side:

```
// Before:

function before(x: any, y: unknown) {
    const definitelyThree = x && 3; // 3
    const definitelyFour = y && 4;  // 4
}

// Passing any falsy values here demonstrates that `definitelyThree` and
`definitelyFour`
// are not, in fact, definitely 3 and 4 at runtime.
before(false, 0);
```

In TypeScript 4.1, under `--strictNullChecks`, when `any` or `unknown` appears on the left-hand side of an `&&`, the type of the expression is `any` or `unknown`, respectively:

```
// After:

function after(x: any, y: unknown) {
    const maybeThree = x && 3; // any
    const maybeFour = y && 4;  // unknown
}
```

This change introduces new errors most frequently where TypeScript previously failed to notice that an `unknown` in an `&&` expression may not produce a `boolean`:

```
function isThing(x: unknown): boolean {
    return x && typeof x === "object" && x.hasOwnProperty("thing");
//  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  error!
//  Type 'unknown' is not assignable to type 'boolean'.
}
```

If `x` is a falsy value other than `false`, the function will return it, in conflict with the `boolean` return type annotation. The error can be resolved by replacing the first `x` in the return expression with `!!x`.

See more details on the [implementing pull request](https://github.com/microsoft/TypeScript/pull/39529).

Conditional Spreads Create Optional Properties

In JavaScript, object spreads (like `{ ...foo }`) don't operate over falsy values. So in code like `{ ...foo }`, `foo` will be skipped over if it's `null` or `undefined`.

Many users take advantage of this to spread in properties "conditionally".

```
interface Person {
    name: string;
    age: number;
    location: string;
}

interface Animal {
    name: string;
    owner: Person;
}

function copyOwner(pet?: Animal) {
    return {
        ...(pet && pet.owner),
        otherStuff: 123
    }
}

// We could also use optional chaining here:

function copyOwner(pet?: Animal) {
    return {
        ...(pet?.owner),
        otherStuff: 123
    }
}
```

Here, if `pet` is defined, the properties of `pet.owner` will be spread in - otherwise, no properties will be spread into the returned object.

The return type of `copyOwner` was previously a union type based on each spread:

```
{ x: number } | { x: number, name: string, age: number, location: string }
```

This modeled exactly how the operation would occur: if `pet` was defined, all the properties from `Person` would be present; otherwise, none of them would be defined on the result. It was an all-or-nothing operation.

However, we've seen this pattern taken to the extreme, with hundreds of spreads in a single object, each spread potentially adding in hundreds or thousands of properties. It turns out that for various reasons, this ends up being extremely expensive, and usually for not much benefit.

In TypeScript 4.1, the returned type instead uses all-optional properties.

```
{
    x: number;
```

```
    name?: string;
    age?: number;
    location?: string;
}
```

This ends up performing better and generally displaying better too.

For more details, see the original change.

Unmatched parameters are no longer related

TypeScript would previously relate parameters that didn't correspond to each other by relating them to the type `any`. With changes in TypeScript 4.1, the language now skips this process entirely. This means that some cases of assignability will now fail, but it also means that some cases of overload resolution can fail as well. For example, the overloads of `util.promisify` in Node.js may select a different overload in TypeScript 4.1, sometimes causing different errors downstream.

As a workaround, you may be best using a type assertion to squelch errors.

TypeScript 4.0

Properties Overridding Accessors (and vice versa) is an Error

Previously, it was only an error for properties to override accessors, or accessors to override properties, when using `useDefineForClassFields`; however, TypeScript now always issues an error when declaring a property in a derived class that would override a getter or setter in the base class.

```
class Base {
    get foo() {
        return 100;
    }
    set foo() {
        // ...
    }
}

class Derived extends Base {
    foo = 10;
//  ~~~
// error!
// 'foo' is defined as an accessor in class 'Base',
// but is overridden here in 'Derived' as an instance property.
}
```

```
class Base {
    prop = 10;
}

class Derived extends Base {
    get prop() {
```

```
    //   ~~~~
    // error!
    // 'prop' is defined as a property in class 'Base', but is overridden here in
'Derived' as an accessor.
        return 100;
    }
}
```

Operands for `delete` must be optional.

When using the `delete` operator in `strictNullChecks`, the operand must now be `any`, `unknown`, `never`, or be optional (in that it contains `undefined` in the type). Otherwise, use of the `delete` operator is an error.

```
interface Thing {
    prop: string;
}

function f(x: Thing) {
    delete x.prop;
    //     ~~~~~~
    // error! The operand of a 'delete' operator must be optional.
}
```

See more details on [the implementing pull request](#).

See more details on [the implementing pull request](#).

TypeScript 3.9

Parsing Differences in Optional Chaining and Non-Null Assertions

TypeScript recently implemented the optional chaining operator, but we've received user feedback that the behavior of optional chaining (`?.`) with the non-null assertion operator (`!`) is extremely counter-intuitive.

Specifically, in previous versions, the code

```
foo?.bar!.baz
```

was interpreted to be equivalent to the following JavaScript.

```
(foo?.bar).baz
```

In the above code the parentheses stop the "short-circuiting" behavior of optional chaining, so if `foo` is `undefined`, accessing `baz` will cause a runtime error.

The Babel team who pointed this behavior out, and most users who provided feedback to us, believe that this behavior is wrong. We do too! The thing we heard the most was that the `!` operator should just "disappear" since the intent was to remove `null` and `undefined` from the type of `bar`.

In other words, most people felt that the original snippet should be interpreted as

```
foo?.bar.baz
```

which just evaluates to `undefined` when `foo` is `undefined`.

This is a breaking change, but we believe most code was written with the new interpretation in mind. Users who want to revert to the old behavior can add explicit parentheses around the left side of the `!` operator.

```
(foo?.bar)!.baz
```

For more information, [see the corresponding pull request](#).

`}` and `>` are Now Invalid JSX Text Characters

The JSX Specification forbids the use of the `}` and `>` characters in text positions. TypeScript and Babel have both decided to enforce this rule to be more comformant. The new way to insert these characters is to use an HTML escape code (e.g. `<span> 2 &gt 1 </div>`) or insert an expression with a string literal (e.g. `<span> 2 {">"} 1 </div>`).

In the presence of code like this, you'll get an error message along the lines of

```
Unexpected token. Did you mean `{'>'}` or `&gt;`?
Unexpected token. Did you mean `{'}'}` or `&rbrace;`?
```

For example:

```
let directions = <span>Navigate to: Menu Bar > Tools > Options</div>
//                                           ~       ~
// Unexpected token. Did you mean `{'>'}` or `&gt;`?
```

For more information, see the corresponding [pull request](#).

Stricter Checks on Intersections and Optional Properties

Generally, an intersection type like `A & B` is assignable to `C` if either `A` or `B` is assignable to `C`; however, sometimes that has problems with optional properties. For example, take the following:

```
interface A {
    a: number; // notice this is 'number'
}

interface B {
    b: string;
}

interface C {
    a?: boolean; // notice this is 'boolean'
```

```
    b: string;
}

declare let x: A & B;
declare let y: C;

y = x;
```

In previous versions of TypeScript, this was allowed because while `A` was totally incompatible with `C`, `B` *was* compatible with `C`.

In TypeScript 3.9, so long as every type in an intersection is a concrete object type, the type system will consider all of the properties at once. As a result, TypeScript will see that the `a` property of `A & B` is incompatible with that of `C`:

```
Type 'A & B' is not assignable to type 'C'.
  Types of property 'a' are incompatible.
    Type 'number' is not assignable to type 'boolean | undefined'.
```

For more information on this change, [see the corresponding pull request](https://github.com/microsoft/TypeScript/pull/37195).

Intersections Reduced By Discriminant Properties

There are a few cases where you might end up with types that describe values that just don't exist. For example

```
declare function smushObjects<T, U>(x: T, y: U): T & U;

interface Circle {
    kind: "circle";
    radius: number;
}

interface Square {
    kind: "square";
    sideLength: number;
}

declare let x: Circle;
declare let y: Square;

let z = smushObjects(x, y);
console.log(z.kind);
```

This code is slightly weird because there's really no way to create an intersection of a `Circle` and a `Square` - they have two incompatible `kind` fields. In previous versions of TypeScript, this code was allowed and the type of `kind` itself was `never` because `"circle" & "square"` described a set of values that could `never` exist.

In TypeScript 3.9, the type system is more aggressive here - it notices that it's impossible to intersect `Circle` and `Square` because of their `kind` properties. So instead of collapsing the type of `z.kind` to `never`, it collapses the type of `z` itself (`Circle & Square`) to `never`. That means the above code now errors with:

```
Property 'kind' does not exist on type 'never'.
```

Most of the breaks we observed seem to correspond with slightly incorrect type declarations. For more details, see the original pull request.

Getters/Setters are No Longer Enumerable

In older versions of TypeScript, `get` and `set` accessors in classes were emitted in a way that made them enumerable; however, this wasn't compliant with the ECMAScript specification which states that they must be non-enumerable. As a result, TypeScript code that targeted ES5 and ES2015 could differ in behavior.

With recent changes, TypeScript 3.9 now conforms more closely with ECMAScript in this regard.

Type Parameters That Extend `any` No Longer Act as `any`

In previous versions of TypeScript, a type parameter constrained to `any` could be treated as `any`.

```
function foo<T extends any>(arg: T) {
    arg.spfjgerijghoied; // no error!
}
```

This was an oversight, so TypeScript 3.9 takes a more conservative approach and issues an error on these questionable operations.

```
function foo<T extends any>(arg: T) {
    arg.spfjgerijghoied;
    //  ~~~~~~~~~~~~~~~
    // Property 'spfjgerijghoied' does not exist on type 'T'.
}
```

See the original pull request for more details.

`export *` is Always Retained

In previous TypeScript versions, declarations like `export * from "foo"` would be dropped in our JavaScript output if `foo` didn't export any values. This sort of emit is problematic because it's type-directed and can't be emulated by Babel. TypeScript 3.9 will always emit these `export *` declarations. In practice, we don't expect this to break much existing code, but bundlers may have a harder time tree-shaking the code.

You can see the specific changes in the original pull request.

Exports Now Use Getters for Live Bindings

When targeting module systems like CommonJS in ES5 and above, TypeScript will use get accessors to emulate live bindings so that changes to a variable in one module are witnessed in any exporting modules. This change is meant to make TypeScript's emit more compliant with ECMAScript modules.

For more details, see the PR that applies this change.

Exports are Hoisted and Initially Assigned

TypeScript now hoists exported declarations to the top of the file when targeting module systems like CommonJS in ES5 and above. This change is meant to make TypeScript's emit more compliant with ECMAScript modules. For example, code like

```
export * from "mod";
export const nameFromMod = 0;
```

previously had output like

```
__exportStar(exports, require("mod"));
exports.nameFromMod = 0;
```

However, because exports now use `get`-accessors, this assignment would throw because `__exportStar` now makes get-accesors which can't be overridden with a simple assignment. Instead, TypeScript 3.9 emits the following:

```
exports.nameFromMod = void 0;
__exportStar(exports, require("mod"));
exports.nameFromMod = 0;
```

See [the original pull request]() for more information.

TypeScript 3.8

Stricter Assignability Checks to Unions with Index Signatures

Previously, excess properties were unchecked when assigning to unions where *any* type had an index signature - even if that excess property could *never* satisfy that index signature. In TypeScript 3.8, the type-checker is stricter, and only "exempts" properties from excess property checks if that property could plausibly satisfy an index signature.

```
const obj1: { [x: string]: number } | { a: number };

obj1 = { a: 5, c: 'abc' }
//             ~
// Error!
// The type '{ [x: string]: number }' no longer exempts 'c'
// from excess property checks on '{ a: number }'.

let obj2: { [x: string]: number } | { [x: number]: number };

obj2 = { a: 'abc' };
//       ~
// Error!
// The types '{ [x: string]: number }' and '{ [x: number]: number }' no longer
exempts 'a'
// from excess property checks against '{ [x: number]: number }',
```

```
// and it *is* sort of an excess property because 'a' isn't a numeric property name.
// This one is more subtle.
```

Optional Arguments with no Inferences are Correctly Marked as Implicitly `any`

In the following code, `param` is now marked with an error under `noImplicitAny`.

```
function foo(f: () => void) {
    // ...
}

foo((param?) => {
    // ...
});
```

This is because there is no corresponding parameter for the type of `f` in `foo`. This seems unlikely to be intentional, but it can be worked around by providing an explicit type for `param`.

`object` in JSDoc is No Longer `any` Under `noImplicitAny`

Historically, TypeScript's support for checking JavaScript has been lax in certain ways in order to provide an approachable experience.

For example, users often used `Object` in JSDoc to mean, "some object, I dunno what", we've treated it as `any`.

```
// @ts-check

/**
 * @param thing {Object} some object, i dunno what
 */
function doSomething(thing) {
    let x = thing.x;
    let y = thing.y;
    thing();
}
```

This is because treating it as TypeScript's `Object` type would end up in code reporting uninteresting errors, since the `Object` type is an extremely vague type with few capabilities other than methods like `toString` and `valueOf`.

However, TypeScript *does* have a more useful type named `object` (notice that lowercase `o`). The `object` type is more restrictive than `Object`, in that it rejects all primitive types like `string`, `boolean`, and `number`. Unfortunately, both `Object` and `object` were treated as `any` in JSDoc.

Because `object` can come in handy and is used significantly less than `Object` in JSDoc, we've removed the special-case behavior in JavaScript files when using `noImplicitAny` so that in JSDoc, the `object` type really refers to the non-primitive `object` type.

TypeScript 3.6

Class Members Named `"constructor"` Are Now Constructors

As per the ECMAScript specification, class declarations with methods named `constructor` are now constructor functions, regardless of whether they are declared using identifier names, or string names.

```
class C {
    "constructor"() {
        console.log("I am the constructor now.");
    }
}
```

A notable exception, and the workaround to this break, is using a computed property whose name evaluates to `"constructor"`.

```
class D {
    ["constructor"]() {
        console.log("I'm not a constructor - just a plain method!");
    }
}
```

DOM Updates

Many declarations have been removed or changed within `lib.dom.d.ts`. This includes (but isn't limited to) the following:

- The global `window` is no longer defined as type `Window` - instead, it is defined as type `Window & typeof globalThis`. In some cases, it may be better to refer to its type as `typeof window`.
- `GlobalFetch` is gone. Instead, use `WindowOrWorkerGlobalScope`
- Certain non-standard properties on `Navigator` are gone.
- The `experimental-webgl` context is gone. Instead, use `webgl` or `webgl2`.

JSDoc Comments No Longer Merge

In JavaScript files, TypeScript will only consult immediately preceding JSDoc comments to figure out declared types.

```
/**
 * @param {string} arg
 */
/**
 * oh, hi, were you trying to type something?
 */
function whoWritesFunctionsLikeThis(arg) {
    // 'arg' has type 'any'
}
```

Keywords Cannot Contain Escape Sequences

Previously keywords were not allowed to contain escape sequences. TypeScript 3.6 disallows them.

```
while (true) {
    \u0063ontinue;
//  ~~~~~~~~~~~~~
//  error! Keywords cannot contain escape characters.
}
```

TypeScript 3.5

Generic type parameters are implicitly constrained to `unknown`

In TypeScript 3.5, [generic type parameters without an explicit constraint are now implicitly constrained to `unknown`], whereas previously the implicit constraint of type parameters was the empty object type `{}`.

In practice, `{}` and `unknown` are pretty similar, but there are a few key differences:

- `{}` can be indexed with a string (`k["foo"]`), though this is an implicit `any` error under `--noImplicitAny`.
- `{}` is assumed to not be `null` or `undefined`, whereas `unknown` is possibly one of those values.
- `{}` is assignable to `object`, but `unknown` is not.

On the caller side, this typically means that assignment to `object` will fail, and methods on `Object` like `toString`, `toLocaleString`, `valueOf`, `hasOwnProperty`, `isPrototypeOf`, and `propertyIsEnumerable` will no longer be available.

```
function foo<T>(x: T): [T, string] {
    return [x, x.toString()]
    //           ~~~~~~~~ error! Property 'toString' does not exist on type 'T'.
}
```

As a workaround, you can add an explicit constraint of `{}` to a type parameter to get the old behavior.

```
//             vvvvvvvvvv
function foo<T extends {}>(x: T): [T, string] {
    return [x, x.toString()]
}
```

From the caller side, failed inferences for generic type arguments will result in `unknown` instead of `{}`.

```
function parse<T>(x: string): T {
    return JSON.parse(x);
}

// k has type 'unknown' - previously, it was '{}'.
const k = parse("...");
```

As a workaround, you can provide an explicit type argument:

```
// 'k' now has type '{}'
const k = parse<{}>("...");
```

`{ [k: string]: unknown }` is no longer a wildcard assignment target

The index signature `{ [s: string]: any }` in TypeScript behaves specially: it's a valid assignment target for any object type. This is a special rule, since types with index signatures don't normally produce this behavior.

Since its introduction, the type `unknown` in an index signature behaved the same way:

```
let dict: { [s: string]: unknown };
// Was OK
dict = () => {};
```

In general this rule makes sense; the implied constraint of "all its properties are some subtype of `unknown`" is trivially true of any object type. However, in TypeScript 3.5, this special rule is removed for `{ [s: string]: unknown }`.

This was a necessary change because of the change from `{}` to `unknown` when generic inference has no candidates. Consider this code:

```
declare function someFunc(): void;
declare function fn<T>(arg: { [k: string]: T }): void;
fn(someFunc);
```

In TypeScript 3.4, the following sequence occurred:

- No candidates were found for `T`
- `T` is selected to be `{}`
- `someFunc` isn't assignable to `arg` because there are no special rules allowing arbitrary assignment to `{ [k: string]: {} }`
- The call is correctly rejected

Due to changes around unconstrained type parameters falling back to `unknown` (see above), `arg` would have had the type `{ [k: string]: unknown }`, which anything is assignable to, so the call would have incorrectly been allowed. That's why TypeScript 3.5 removes the specialized assignability rule to permit assignment to `{ [k: string]: unknown }`.

Note that fresh object literals are still exempt from this check.

```
const obj = { m: 10 };
// OK
const dict: { [s: string]: unknown } = obj;
```

Depending on the intended behavior of `{ [s: string]: unknown }`, several alternatives are available:

- `{ [s: string]: any }`

- `{ [s: string]: {} }`
- `object`
- `unknown`
- `any`

We recommend sketching out your desired use cases and seeing which one is the best option for your particular use case.

Improved excess property checks in union types

Background

TypeScript has a feature called *excess property checking* in object literals. This feature is meant to detect typos for when a type isn't expecting a specific property.

```
type Style = {
    alignment: string,
    color?: string
};

const s: Style = {
    alignment: "center",
    colour: "grey"
//  ^^^^^^ error!
};
```

Rationale and Change

In TypeScript 3.4 and earlier, certain excess properties were allowed in situations where they really shouldn't have been.

Consider this code:

```
type Point = {
    x: number;
    y: number;
};

type Label = {
    name: string;
};

const pl: Point | Label = {
    x: 0,
    y: 0,
    name: true // <- danger!
};
```

Excess property checking was previously only capable of detecting properties which weren't present in *any* member of a target union type.

In TypeScript 3.5, these excess properties are now correctly detected, and the sample above correctly issues an error.

Note that it's still legal to be assignable to multiple parts of a union:

```
const pl: Point | Label = {
    x: 0,
    y: 0,
    name: "origin" // OK
};
```

Workarounds

We have not witnessed examples where this checking hasn't caught legitimate issues, but in a pinch, any of the workarounds to disable excess property checking will apply:

- Add a type assertion onto the object (e.g. `{ myProp: SomeType } as ExpectedType`)
- Add an index signature to the expected type to signal that unspecified properties are expected (e.g. `interface ExpectedType { myProp: SomeType; [prop: string]: unknown }`)

Fixes to Unsound Writes to Indexed Access Types

Background

TypeScript allows you to represent the abstract operation of accessing a property of an object via the name of that property:

```
type A = {
    s: string;
    n: number;
};

function read<K extends keyof A>(arg: A, key: K): A[K] {
    return arg[key];
}

const a: A = { s: "", n: 0 };
const x = read(a, "s"); // x: string
```

While commonly used for reading values from an object, you can also use this for writes:

```
function write<K extends keyof A>(arg: A, key: K, value: A[K]): void {
    arg[key] = value;
}
```

Change and Rationale

In TypeScript 3.4, the logic used to validate a *write* was much too permissive:

```
function write<K extends keyof A>(arg: A, key: K, value: A[K]): void {
    // ???
    arg[key] = "hello, world";
```

```
}
// Breaks the object by putting a string where a number should be
write(a, "n");
```

In TypeScript 3.5, this logic is fixed and the above sample correctly issues an error.

Workarounds

Most instances of this error represent potential errors in the relevant code.

One example we found looked like this:

```
type T = {
    a: string,
    x: number,
    y: number
};
function write<K extends keyof T>(obj: T, k: K) {
    // Trouble waiting
    obj[k] = 1;
}
const someObj: T = { a: "", x: 0, y: 0 };
// Note: write(someObj, "a") never occurs, so the code is technically bug-free (?)
write(someObj, "x");
write(someObj, "y");
```

This function can be fixed to only accept keys which map to numeric properties:

```
// Generic helper type that produces the keys of an object
// type which map to properties of some other specific type
type KeysOfType<TObj, TProp, K extends keyof TObj = keyof TObj> = K extends K ?
TObj[K] extends TProp ? K : never : never;

function write(obj: SomeObj, k: KeysOfType<SomeObj, number>) {
    // OK
    obj[k] = 1;
}

const someObj: SomeObj = { a: "", x: 0, y: 0 };
write(someObj, "x");
write(someObj, "y");
// Correctly an error
write(someObj, "a");
```

`lib.d.ts` includes the `Omit` helper type

TypeScript 3.5 includes a new `Omit` helper type. As a result, any global declarations of `Omit` included in your project will result in the following error message:

```
Duplicate identifier 'Omit'.
```

Two workarounds may be used here:

1. Delete the duplicate declaration and use the one provided in `lib.d.ts`.
2. Export the existing declaration from a module file or a namespace to avoid a global collision. Existing usages can use an `import` or explicit reference to your project's old `Omit` type.

`Object.keys` rejects primitives in ES5

Background

In ECMAScript 5 environments, `Object.keys` throws an exception if passed any non-`object` argument:

```
// Throws if run in an ES5 runtime
Object.keys(10);
```

In ECMAScript 2015, `Object.keys` returns `[]` if its argument is a primitive:

```
// [] in ES6 runtime
Object.keys(10);
```

Rationale and Change

This is a potential source of error that wasn't previously identified.

In TypeScript 3.5, if `target` (or equivalently `lib`) is `ES5`, calls to `Object.keys` must pass a valid `object`.

Workarounds

In general, errors here represent possible exceptions in your application and should be treated as such. If you happen to know through other means that a value is an `object`, a type assertion is appropriate:

```
function fn(arg: object | number, isArgActuallyObject: boolean) {
    if (isArgActuallyObject) {
        const k = Object.keys(arg as object);
    }
}
```

Note that this change interacts with the change in generic inference from `{}` to `unknown`, because `{}` is a valid `object` whereas `unknown` isn't:

```
declare function fn<T>(): T;

// Was OK in TypeScript 3.4, errors in 3.5 under --target ES5
Object.keys(fn());
```

TypeScript 3.4

Top-level `this` is now typed

The type of top-level `this` is now typed as `typeof globalThis` instead of `any`. As a consequence, you may receive errors for accessing unknown values on `this` under `noImplicitAny`.

```
// previously okay in noImplicitAny, now an error
this.whargarbl = 10;
```

Note that code compiled under `noImplicitThis` will not experience any changes here.

Propagated generic type arguments

In certain cases, TypeScript 3.4's improved inference might produce functions that are generic, rather than ones that take and return their constraints (usually `{}`).

```
declare function compose<T, U, V>(f: (arg: T) => U, g: (arg: U) => V): (arg: T) =>
V;

function list<T>(x: T) { return [x]; }
function box<T>(value: T) { return { value }; }

let f = compose(list, box);
let x = f(100)

// In TypeScript 3.4, 'x.value' has the type
//
//   number[]
//
// but it previously had the type
//
//   {}[]
//
// So it's now an error to push in a string.
x.value.push("hello");
```

An explicit type annotation on `x` can get rid of the error.

Contextual return types flow in as contextual argument types

TypeScript now uses types that flow into function calls (like `then` in the below example) to contextually type function arguments (like the arrow function in the below example).

```
function isEven(prom: Promise<number>): Promise<{ success: boolean }> {
    return prom.then((x) => {
        return x % 2 === 0 ?
            { success: true } :
            Promise.resolve({ success: false });
    });
}
```

This is generally an improvement, but in the above example it causes `true` and `false` to acquire literal types which is undesirable.

```
Argument of type '(x: number) => Promise<{ success: false; }> | { success: true; }' is
not assignable to parameter of type '(value: number) => { success: false; } |
PromiseLike<{ success: false; }>'.
  Type 'Promise<{ success: false; }> | { success: true; }' is not assignable to type
'{ success: false; } | PromiseLike<{ success: false; }>'.
    Type '{ success: true; }' is not assignable to type '{ success: false; } |
PromiseLike<{ success: false; }>'.
      Type '{ success: true; }' is not assignable to type '{ success: false; }'.
        Types of property 'success' are incompatible.
```

The appropriate workaround is to add type arguments to the appropriate call - the `then` method call in this example.

```
function isEven(prom: Promise<number>): Promise<{ success: boolean }> {
    //               vvvvvvvvvvvvvvvvvv
    return prom.then<{success: boolean}>((x) => {
        return x % 2 === 0 ?
            { success: true } :
            Promise.resolve({ success: false });
    });
}
```

Consistent inference priorities outside of `strictFunctionTypes`

In TypeScript 3.3 with `--strictFunctionTypes` off, generic types declared with `interface` were assumed to always be covariant with respect to their type parameter. For function types, this behavior was generally not observable. However, for generic `interface` types that used their type parameters with `keyof` positions - a contravariant use - these types behaved incorrectly.

In TypeScript 3.4, variance of types declared with `interface` is now correctly measured in all cases. This causes an observable breaking change for interfaces that used a type parameter only in `keyof` (including places like `Record<K, T>` which is an alias for a type involving `keyof K`). The example above is one such possible break.

```
interface HasX { x: any }
interface HasY { y: any }

declare const source: HasX | HasY;
declare const properties: KeyContainer<HasX>;

interface KeyContainer<T> {
    key: keyof T;
}

function readKey<T>(source: T, prop: KeyContainer<T>) {
    console.log(source[prop.key])
}
```

```
// This call should have been rejected, because we might
// incorrectly be reading 'x' from 'HasY'. It now appropriately errors.
readKey(source, properties);
```

This error is likely indicative of an issue with the original code.

TypeScript 3.2

`lib.d.ts` updates

`wheelDelta` and friends have been removed.

`wheelDeltaX`, `wheelDelta`, and `wheelDeltaZ` have all been removed as they is a deprecated properties on `WheelEvent`s.

Solution: Use `deltaX`, `deltaY`, and `deltaZ` instead.

More specific types

Certain parameters no longer accept `null`, or now accept more specific types as per the corresponding specifications that describe the DOM.

TypeScript 3.1

Some vendor-specific types are removed from `lib.d.ts`

TypeScript's built-in `.d.ts` library (`lib.d.ts` and family) is now partially generated from Web IDL files from the DOM specification. As a result some vendor-specific types have been removed.

▶ Click here to read the full list of removed types:

Recommendations:

If your run-time guarantees that some of these names are available at run-time (e.g. for an IE-only app), add the declarations locally in your project, e.g.:

For `Element.msMatchesSelector`, add the following to a local `dom.ie.d.ts`

```
interface Element {
    msMatchesSelector(selectors: string): boolean;
}
```

Similarly, to add `clearImmediate` and `setImmediate`, you can add a declaration for `Window` in your local `dom.ie.d.ts`:

```
interface Window {
    clearImmediate(handle: number): void;
    setImmediate(handler: (...args: any[]) => void): number;
    setImmediate(handler: any, ...args: any[]): number;
}
```

Narrowing functions now intersects `{}`, `Object`, and unconstrained generic type parameters.

The following code will now complain about `x` no longer being callable:

```
function foo<T>(x: T | (() => string)) {
    if (typeof x === "function") {
        x();
//      ~~~
// Cannot invoke an expression whose type lacks a call signature. Type '(() =>
string) | (T & Function)' has no compatible call signatures.
    }
}
```

This is because, unlike previously where `T` would be narrowed away, it is now *expanded* into `T & Function`. However, because this type has no call signatures declared, the type system won't find any common call signature between `T & Function` and `() => string`.

Instead, consider using a more specific type than `{}` or `Object`, and consider adding additional constraints to what you expect `T` might be.

TypeScript 3.0

The `unknown` keyword is reserved

`unknown` is now a reserved type name, as it is now a built-in type. Depending on your intended use of `unknown`, you may want to remove the declaration entirely (favoring the newly introduced `unknown` type), or rename it to something else.

Intersecting with `null` / `undefined` reduces to `null` / `undefined` outside of `strictNullChecks`

In the following example, `A` has the type `null` and `B` has the type `undefined` when `strictNullChecks` is turned off:

```
type A = { a: number } & null;      // null
type B = { a: number } & undefined; // undefined
```

This is because TypeScript 3.0 is better at reducing subtypes and supertypes in intersection and union types respectively; however, because `null` and `undefined` are both considered subtypes of every other type when `strictNullChecks` is off, an intersection with some object type and either will always reduce to `null` or `undefined`.

Recommendation

If you were relying on `null` and `undefined` to be "identity" elements under intersections, you should look for a way to use `unknown` instead of `null` or `undefined` wherever they appeared

TypeScript 2.9

`keyof` now includes `string`, `number` and `symbol` keys

TypeScript 2.9 generalizes index types to include `number` and `symbol` named properties. Previously, the `keyof` operator and mapped types only supported `string` named properties.

```
function useKey<T, K extends keyof T>(o: T, k: K) {
    var name: string = k;  // Error: keyof T is not assignable to string
}
```

Recommendations:

- If your functions are only able to handle string named property keys, use `Extract<keyof T, string>` in the declaration:

  ```
  function useKey<T, K extends Extract<keyof T, string>>(o: T, k: K) {
    var name: string = k;  // OK
  }
  ```

- If your functions are open to handling all property keys, then the changes should be done down-stream:

  ```
  function useKey<T, K extends keyof T>(o: T, k: K) {
    var name: string | number | symbol = k;
  }
  ```

- Otherwise use `--keyofStringsOnly` compiler option to disable the new behavior.

Trailing commas not allowed on rest parameters

The following code is a compiler error as of #22262:

```
function f(
    a: number,
    ...b: number[], // Illegal trailing comma
) {}
```

Trailing commas on rest parameters are not valid JavaScript, and the syntax is now an error in TypeScript too.

In `strictNullChecks`, an unconstrained type parameter is no longer assignable to `object`

The following code is a compiler error under `strictNullChecks` as of #24013:

```
function f<T>(x: T) {
    const y: object | null | undefined = x;
}
```

It may be fulfilled with any type (eg, `string` or `number`), so it was incorrect to allow. If you encounter this issue, either constrain your type parameter to `object` to only allow object types for it, or compare against `{}` instead of `object` (if the intent was to allow any type).

TypeScript 2.8

Unused type parameters are checked under `--noUnusedParameters`

As per [#20568](), unused type parameters were previously reported under `--noUnusedLocals`, but are now instead reported under `--noUnusedParameters`.

Some MS-specific types are removed from `lib.d.ts`

Some MS-specific types are removed from the DOM definition to better align with the standard. Types removed include:

- `MSApp`
- `MSAppAsyncOperation`
- `MSAppAsyncOperationEventMap`
- `MSBaseReader`
- `MSBaseReaderEventMap`
- `MSExecAtPriorityFunctionCallback`
- `MSHTMLWebViewElement`
- `MSManipulationEvent`
- `MSRangeCollection`
- `MSSiteModeEvent`
- `MSUnsafeFunctionCallback`
- `MSWebViewAsyncOperation`
- `MSWebViewAsyncOperationEventMap`
- `MSWebViewSettings`

`HTMLObjectElement` no longer has an `alt` attribute

As per [#21386](), the DOM libraries have been updated to reflect the WHATWG standard.

If you need to continue using the `alt` attribute, consider reopening `HTMLObjectElement` via interface merging in the global scope:

```
// Must be in a global .ts file or a 'declare global' block.
interface HTMLObjectElement {
    alt: string;
}
```

TypeScript 2.7

For a full list of breaking changes see the [breaking change issues](#).

Tuples now have a fixed length property

The following code used to have no compile errors:

```
var pair: [number, number] = [1, 2];
var triple: [number, number, number] = [1, 2, 3];
pair = triple;
```

However, this *was* an error:

```
triple = pair;
```

Now both assignments are an error. This is because tuples now have a length property whose type is their length. So `pair.length: 2`, but `triple.length: 3`.

Note that certain non-tuple patterns were allowed previously, but are no longer allowed:

```
const struct: [string, number] = ['key'];
for (const n of numbers) {
  struct.push(n);
}
```

The best fix for this is to make your own type that extends Array:

```
interface Struct extends Array<string | number> {
  '0': string;
  '1'?: number;
}
const struct: Struct = ['key'];
for (const n of numbers) {
  struct.push(n);
}
```

Under `allowSyntheticDefaultImports`, types for default imports are synthesized less often for TS and JS files

In the past, we'd synthesize a default import in the typesystem for a TS or JS file written like so:

```
export const foo = 12;
```

meaning the module would have the type `{foo: number, default: {foo: number}}`. This would be wrong, because the file would be emitted with an `__esModule` marker, so no popular module loader would ever create a synthetic default for it when loading the file, and the `default` member that the typesystem inferred was there would never exist at runtime. Now that we emulate this synthetic default behavior in our emit under the `ESModuleInterop` flag, we've tightened the typechecker behavior to match the shape you'd expect to see at runtime. Without the intervention of other tools at runtime, this change should only point out mistakenly incorrect import default usages which should be changed to namespace imports.

Stricter checking for indexed access generic type constraints

Previously the constraint of an indexed access type was only computed if the type had an index signature, otherwise it was `any`. That allowed invalid assignments to go unchecked. In TS 2.7.1, the compiler is a bit smarter here, and will compute the constraint to be the union of all possible properties here.

```
interface O {
    foo?: string;
}

function fails<K extends keyof O>(o: O, k: K) {
    var s: string = o[k]; // Previously allowed, now an error
                          // string | undefined is not assignable to a string
}
```

`in` expressions are treated as type guards

For a `n in x` expression, where `n` is a string literal or string literal type and `x` is a union type, the "true" branch narrows to types which have an optional or required property `n`, and the "false" branch narrows to types which have an optional or missing property `n`. This may result in cases where the type of a variable is narrowed to `never` in the false branch if the type is declared to always have the the property `n`.

```
var x: { foo: number };

if ("foo" in x) {
    x; // { foo: number }
}
else {
    x; // never
}
```

Structurally-equivalent classes are not reduced in conditional operator

Previously classes that were structurally equivalent were reduced to their best common type in a conditional or `||` operator. Now these classes are maintained in a union type to allow for more accurate checking for `instanceof` operators.

```
class Animal {

}

class Dog {
```

```
    park() { }
}

var a = Math.random() ? new Animal() : new Dog();
// typeof a now Animal | Dog, previously Animal
```

`CustomEvent` is now a generic type

`CustomEvent` now has a type parameter for the type of the `details` property. If you are extending from it, you will need to specify an additional type parameter.

```
class MyCustomEvent extends CustomEvent {
}
```

should become

```
class MyCustomEvent extends CustomEvent<any> {
}
```

TypeScript 2.6

For full list of breaking changes see the breaking change issues.

Write-only references are unused

The following code used to have no compile errors:

```
function f(n: number) {
    n = 0;
}

class C {
    private m: number;
    constructor() {
        this.m = 0;
    }
}
```

Now when the `--noUnusedLocals` and `--noUnusedParameters` compiler options are enabled, both `n` and `m` will be marked as unused, because their values are never *read*. Previously TypeScript would only check whether their values were *referenced*.

Also recursive functions that are only called within their own bodies are considered unused.

```
function f() {
    f(); // Error: 'f' is declared but its value is never read
}
```

Arbitrary expressions are forbidden in export assignments in ambient contexts

Previously, constructs like

```
declare module "foo" {
    export default "some" + "string";
}
```

was not flagged as an error in ambient contexts. Expressions are generally forbidden in declaration files and ambient modules, as things like `typeof` have unclear intent, so this was inconsistent with our handling of executable code elsewhere in these contexts. Now, anything which is not an identifier or qualified name is flagged as an error. The correct way to make a DTS for a module with the value shape described above would be like so:

```
declare module "foo" {
    const _default: string;
    export default _default;
}
```

The compiler already generated definitions like this, so this should only be an issue for definitions which were written by hand.

TypeScript 2.4

For full list of breaking changes see the [breaking change issues](#).

Weak Type Detection

TypeScript 2.4 introduces the concept of "weak types". Any type that contains nothing but a set of all-optional properties is considered to be *weak*. For example, this `Options` type is a weak type:

```
interface Options {
    data?: string,
    timeout?: number,
    maxRetries?: number,
}
```

In TypeScript 2.4, it's now an error to assign anything to a weak type when there's no overlap in properties. For example:

```
function sendMessage(options: Options) {
    // ...
}

const opts = {
    payload: "hello world!",
    retryOnFail: true,
}
```

```
// Error!
sendMessage(opts);
// No overlap between the type of 'opts' and 'Options' itself.
// Maybe we meant to use 'data'/'maxRetries' instead of 'payload'/'retryOnFail'.
```

Recommendation

1. Declare the properties if they really do exist.
2. Add an index signature to the weak type (i.e. `[propName: string]: {}`).
3. Use a type assertion (i.e. `opts as Options`).

Return types as inference targets

TypeScript can now make inferences from contextual types to the return type of a call. This means that some code may now appropriately error. As an example of a new errors you might spot as a result:

```
let x: Promise<string> = new Promise(resolve => {
    resolve(10);
    //      ~~ Error! Type 'number' is not assignable to 'string'.
});
```

Stricter variance in callback parameters

TypeScript's checking of callback parameters is now covariant with respect to immediate signature checks. Previously it was bivariant, which could sometimes let incorrect types through. Basically, this means that callback parameters and classes that contain callbacks are checked more carefully, so Typescript will require stricter types in this release. This is particularly true of Promises and Observables due to the way in which their APIs are specified.

Promises

Here is an example of improved Promise checking:

```
let p = new Promise((c, e) => { c(12) });
let u: Promise<number> = p;
    ~
Type 'Promise<{}>' is not assignable to 'Promise<number>'
```

The reason this occurs is that TypeScript is not able to infer the type argument `T` when you call `new Promise`. As a result, it just infers `Promise<{}>`. Unfortunately, this allows you to write `c(12)` and `c('foo')`, even though the declaration of `p` explicitly says that it must be `Promise<number>`.

Under the new rules, `Promise<{}>` is not assignable to `Promise<number>` because it breaks the callbacks to Promise. TypeScript still isn't able to infer the type argument, so to fix this you have to provide the type argument yourself:

```
let p: Promise<number> = new Promise<number>((c, e) => { c(12) });
//                                  ^^^^^^^^ explicit type arguments here
```

This requirement helps find errors in the body of the promise code. Now if you mistakenly call `c('foo')`, you get the following error:

```
let p: Promise<number> = new Promise<number>((c, e) => { c('foo') });
//                                                            ~~~~~
//  Argument of type '"foo"' is not assignable to 'number'
```

(Nested) Callbacks

Other callbacks are affected by the improved callback checking as well, primarily nested callbacks. Here's an example with a function that takes a callback, which takes a nested callback. The nested callback is now checked co-variantly.

```
declare function f(
  callback: (nested: (error: number, result: any) => void, index: number) => void
): void;

f((nested: (error: number) => void) => { log(error) });
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'(error: number) => void' is not assignable to (error: number, result: any) => void'
```

The fix is easy in this case. Just add the missing parameter to the nested callback:

```
f((nested: (error: number, result: any) => void) => { });
```

Stricter checking for generic functions

TypeScript now tries to unify type parameters when comparing two single-signature types. As a result, you'll get stricter checks when relating two generic signatures, and may catch some bugs.

```
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) {
    a = b;  // Error
    b = a;  // Ok
}
```

Recommendation

Either correct the definition or use `--noStrictGenericChecks`.

Type parameter inference from contextual types

Prior to TypeScript 2.4, in the following example

```
let f: <T>(x: T) => T = y => y;
```

`y` would have the type `any`. This meant the program would type-check, but you could technically do anything with `y`, such as the following:

```
let f: <T>(x: T) => T = y => y() + y.foo.bar;
```

Recommendation: Appropriately re-evaluate whether your generics have the correct constraint, or are even necessary. As a last resort, annotate your parameters with the `any` type.

TypeScript 2.3

For full list of breaking changes see the [breaking change issues]().

Empty generic parameter lists are flagged as error

Example

```
class X<> {}  // Error: Type parameter list cannot be empty.
function f<>() {}  // Error: Type parameter list cannot be empty.
const x: X<> = new X<>();  // Error: Type parameter list cannot be empty.
```

TypeScript 2.2

For full list of breaking changes see the [breaking change issues]().

Changes to DOM API's in the standard library

- Standard library now has declarations for `Window.fetch`; dependencies to `@types\whatwg-fetch` will cause conflicting declaration errors and will need to be removed.
- Standard library now has declarations for `ServiceWorker`; dependencies on `@types\service_worker_api` will cause conflicting declaration errors and will need to be removed.

TypeScript 2.1

For full list of breaking changes see the [breaking change issues]().

Generated constructor code substitutes the return value of `super(...)` calls as `this`

In ES2015, constructors which return an object implicitly substitute the value of `this` for any callers of `super(...)`. As a result, it is necessary to capture any potential return value of `super(...)` and replace it with `this`.

Example

A class `C` as:

```
class C extends B {
    public a: number;
    constructor() {
        super();
        this.a = 0;
    }
}
```

Will generate code as:

```
var C = (function (_super) {
    __extends(C, _super);
    function C() {
        var _this = _super.call(this) || this;
        _this.a = 0;
        return _this;
    }
    return C;
}(B));
```

Notice:

- `_super.call(this)` is captured into a local variable `_this`
- All uses of `this` in the constructor body has been replaced by the result of the `super` call (i.e. `_this`)
- Each constructor now returns explicitly its `this` , to enable for correct inheritance

It is worth noting that the use of `this` before `super(...)` is already an error as of [TypeScript 1.8](#)

Extending built-ins like `Error` , `Array` , and `Map` may no longer work

As part of substituting the value of `this` with the value returned by a `super(...)` call, subclassing `Error` , `Array` , and others may no longer work as expected. This is due to the fact that constructor functions for `Error` , `Array` , and the like use ECMAScript 6's `new.target` to adjust the prototype chain; however, there is no way to ensure a value for `new.target` when invoking a constructor in ECMAScript 5. Other downlevel compilers generally have the same limitation by default.

Example

For a subclass like the following:

```
class FooError extends Error {
    constructor(m: string) {
        super(m);
    }
    sayHello() {
        return "hello " + this.message;
    }
}
```

you may find that:

- methods may be `undefined` on objects returned by constructing these subclasses, so calling `sayHello` will result in an error.
- `instanceof` will be broken between instances of the subclass and their instances, so `(new FooError()) instanceof FooError` will return `false`.

Recommendation

As a recommendation, you can manually adjust the prototype immediately after any `super(...)` calls.

```
class FooError extends Error {
    constructor(m: string) {
        super(m);

        // Set the prototype explicitly.
        Object.setPrototypeOf(this, FooError.prototype);
    }

    sayHello() {
        return "hello " + this.message;
    }
}
```

However, any subclass of `FooError` will have to manually set the prototype as well. For runtimes that don't support [`Object.setPrototypeOf`](), you may instead be able to use [`__proto__`]().

Unfortunately, [these workarounds will not work on Internet Explorer 10 and prior](). One can manually copy methods from the prototype onto the instance itself (i.e. `FooError.prototype` onto `this`), but the prototype chain itself cannot be fixed.

Literal types are inferred by default for `const` variables and `readonly` properties

String, numeric, boolean and enum literal types are not inferred by default for `const` declarations and `readonly` properties. This means your variables/properties an have more narrowed type than before. This could manifest in using comparison operators such as `===` and `!==`.

Example

```
const DEBUG = true; // Now has type `true`, previously had type `boolean`

if (DEBUG === false) { /// Error: operator '===' can not be applied to 'true' and
'false'
    ...
}
```

Recommendation

For types intentionally needed to be wider, cast to the base type:

```
const DEBUG = <boolean>true; // type is `boolean`
```

No type narrowing for captured variables in functions and class expressions

String, numeric and boolean literal types will be inferred if the generic type parameter has a constraint of `string`, `number` or `boolean` respectively. Moreover the rule of failing if no best common super-type for inferences in the case of literal types if they have the same base type (e.g. `string`).

Example

```
declare function push<T extends string>(...args: T[]): T;

var x = push("A", "B", "C"); // inferred as "A" | "B" | "C" in TS 2.1, was string in TS 2.0
```

Recommendation

Specify the type argument explicitly at call site:

```
var x = push<string>("A", "B", "C"); // x is string
```

Implicit-any error raised for un-annotated callback arguments with no matching overload arguments

Previously the compiler silently gave the argument of the callback (`c` below) a type `any`. The reason is how the compiler resolves function expressions while doing overload resolution.Starting with TypeScript 2.1 an error will be reported under `--noImplicitAny`.

Example

```
declare function func(callback: () => void): any;
declare function func(callback: (arg: number) => void): any;

func(c => { });
```

Recommendation

Remove the first overload, since it is rather meaningless; the function above can still be called with a call back with 1 or 0 required arguments, as it is safe for functions to ignore additional arguments.

```
declare function func(callback: (arg: number) => void): any;

func(c => { });
func(() => { });
```

Alternatively, you can either specify an explicit type annotation on the callback argument:

```
func((c:number) => { });
```

Comma operators on side-effect-free expressions is now flagged as an error

Mostly, this should catch errors that were previously allowed as valid comma expressions.

Example

```
let x = Math.pow((3, 5)); // x = NaN, was meant to be `Math.pow(3, 5)`

// This code does not do what it appears to!
let arr = [];
switch(arr.length) {
  case 0, 1:
    return 'zero or one';
  default:
    return 'more than one';
}
```

Recommendation

`--allowUnreachableCode` will disable the warning for the whole compilation. Alternatively, you can use the `void` operator to suppress the error for specific comma expressions:

```
let a = 0;
let y = (void a, 1); // no warning for `a`
```

Changes to DOM API's in the standard library

- **Node.firstChild**, **Node.lastChild**, **Node.nextSibling**, **Node.previousSibling**, **Node.parentElement** and **Node.parentNode** are now `Node | null` instead of `Node`.

See #11113 for more details.

Recommendation is to explicitly check for `null` or use the `!` assertion operator (e.g. `node.lastChild!`).

TypeScript 2.0

For full list of breaking changes see the breaking change issues.

No type narrowing for captured variables in functions and class expressions

Type narrowing does not cross function and class expressions, as well as lambda expressions.

Example

```
var x: number | string;

if (typeof x === "number") {
```

```
    function inner(): number {
        return x; // Error, type of x is not narrowed, c is number | string
    }
    var y: number = x; // OK, x is number
}
```

In the previous pattern the compiler can not tell when the callback will execute. Consider:

```
var x: number | string = "a";
if (typeof x === "string") {
    setTimeout(() => console.log(x.charAt(0)), 0);
}
x = 5;
```

It is wrong to assume `x` is a `string` when `x.charAt()` is called, as indeed it isn't.

Recommendation

Use constants instead:

```
const x: number | string = "a";
if (typeof x === "string") {
    setTimeout(() => console.log(x.charAt(0)), 0);
}
```

Generic type parameters are now narrowed

Example

```
function g<T>(obj: T) {
    var t: T;
    if (obj instanceof RegExp) {
         t = obj; // RegExp is not assignable to T
    }
}
```

Recommendation Either declare your locals to be a specific type and not the generic type parameter, or use a type assertion.

Getters with no setters are automatically inferred to be `readonly` properties

Example

```
class C {
  get x() { return 0; }
}
```

```
var c = new C();
c.x = 1; // Error Left-hand side is a readonly property
```

Recommendation

Define a setter or do not write to the property.

Function declarations not allowed in blocks in strict mode

This is already a run-time error under strict mode. Starting with TypeScript 2.0, it will be flagged as a compile-time error as well.

Example

```
if( true ) {
    function foo() {}
}

export = foo;
```

Recommendation

Use function expressions instead:

```
if( true ) {
    const foo = function() {}
}
```

`TemplateStringsArray` is now immutable

ES2015 tagged templates always pass their tag an immutable array-like object that has a property called `raw` (which is also immutable). TypeScript names this object the `TemplateStringsArray`.

Conveniently, `TemplateStringsArray` was assignable to an `Array<string>`, so it's possible users took advantage of this to use a shorter type for their tag parameters:

```
function myTemplateTag(strs: string[]) {
    // ...
}
```

However, in TypeScript 2.0, the language now supports the `readonly` modifier and can express that these objects are immutable. As a result, `TemplateStringsArray` has also been made immutable, and is no longer assignable to `string[]`.

Recommendation

Use `TemplateStringsArray` explicitly (or use `ReadonlyArray<string>`).

TypeScript 1.8

For full list of breaking changes see the [breaking change issues](#).

Modules are now emitted with a `"use strict";` prologue

Modules were always parsed in strict mode as per ES6, but for non-ES6 targets this was not respected in the generated code. Starting with TypeScript 1.8, emitted modules are always in strict mode. This shouldn't have any visible changes in most code as TS considers most strict mode errors as errors at compile time, but it means that some things which used to silently fail at runtime in your TS code, like assigning to `NaN`, will now loudly fail. You can reference the [MDN Article](#) on strict mode for a detailed list of the differences between strict mode and non-strict mode.

To disable this behavior, pass `--noImplicitUseStrict` on the command line or set it in your tsconfig.json file.

Exporting non-local names from a module

In accordance with the ES6/ES2015 spec, it is an error to export a non-local name from a module.

Example

```
export { Promise }; // Error
```

Recommendation

Use a local variable declaration to capture the global name before exporting it.

```
const localPromise = Promise;
export { localPromise as Promise };
```

Reachability checks are enabled by default

In TypeScript 1.8 we've added a set of [reachability checks](#) to prevent certain categories of errors. Specifically

1. check if code is reachable (enabled by default, can be disabled via `allowUnreachableCode` compiler option)

    ```
    function test1() {
        return 1;
        return 2; // error here
    }

    function test2(x) {
        if (x) {
            return 1;
        }
        else {
            throw new Error("NYI")
        }
        var y = 1; // error here
    }
    ```

2. check if label is unused (enabled by default, can be disabled via `allowUnusedLabels` compiler option)

```
l: // error will be reported - label `l` is unused
while (true) {
}

(x) => { x:x } // error will be reported - label `x` is unused
```

3. check if all code paths in function with return type annotation return some value (disabled by default, can be enabled via `noImplicitReturns` compiler option)

```
// error will be reported since function does not return anything explicitly
when `x` is falsy.
function test(x): number {
    if (x) return 10;
}
```

4. check if control flow falls through cases in switch statement (disabled by default, can be enabled via `noFallthroughCasesInSwitch` compiler option). Note that cases without statements are not reported.

```
switch(x) {
    // OK
    case 1:
    case 2:
        return 1;
}
switch(x) {
    case 1:
        if (y) return 1;
    case 2:
        return 2;
}
```

If these errors are showing up in your code and you still think that scenario when they appear is legitimate you can suppress errors with compiler options.

`--module` is not allowed alongside `--outFile` unless `--module` is specified as one of `amd` or `system`.

Previously specifying both while using modules would result in an empty `out` file and no error.

Changes to DOM API's in the standard library

- **ImageData.data** is now of type `Uint8ClampedArray` instead of `number[]`. See #949 for more details.
- **HTMLSelectElement .options** is now of type `HTMLCollection` instead of `HTMLSelectElement`. See #1558 for more details.
- **HTMLTableElement.createCaption**, **HTMLTableElement.createTBody**, **HTMLTableElement.createTFoot**, **HTMLTableElement.createTHead**, **HTMLTableElement.insertRow**,

HTMLTableSectionElement.insertRow, and **HTMLTableElement.insertRow** now return `HTMLTableRowElement` instead of `HTMLElement`. See [#3583](#) for more details.
- **HTMLTableRowElement.insertCell** now return `HTMLTableCellElement` instead of `HTMLElement`. See [#3583](#) for more details.
- **IDBObjectStore.createIndex** and **IDBDatabase.createIndex** second argument is now of type `IDBObjectStoreParameters` instead of `any`. See [#5932](#) for more details.
- **DataTransferItemList.Item** returns type now is `DataTransferItem` instead of `File`. See [#6106](#) for more details.
- **Window.open** return type now is `Window` instead of `any`. See [#6418](#) for more details.
- **WeakMap.clear** as removed. See [#6500](#) for more details.

Disallow `this` accessing before super-call

ES6 disallows accessing `this` in a constructor declaration.

For example:

```
class B {
    constructor(that?: any) {}
}

class C extends B {
    constructor() {
        super(this);  // error;
    }
}

class D extends B {
    private _prop1: number;
    constructor() {
        this._prop1 = 10;  // error
        super();
    }
}
```

TypeScript 1.7

For full list of breaking changes see the [breaking change issues](#).

Changes in inferring the type from `this`

In a class, the type of the value `this` will be inferred to the `this` type. This means subsequent assignments from values the original type can fail.

Example:

```
class Fighter {
    /** @returns the winner of the fight. */
    fight(opponent: Fighter) {
```

```ts
        let theVeryBest = this;
        if (Math.rand() < 0.5) {
            theVeryBest = opponent; // error
        }
        return theVeryBest
    }
}
```

Recommendations:

Add a type annotation:

```ts
class Fighter {
    /** @returns the winner of the fight. */
    fight(opponent: Fighter) {
        let theVeryBest: Fighter = this;
        if (Math.rand() < 0.5) {
            theVeryBest = opponent; // no error
        }
        return theVeryBest
    }
}
```

Automatic semicolon insertion after class member modifiers

The keywords `abstract, public, protected` and `private` are *FutureReservedWords* in ECMAScript 3 and are subject to automatic semicolon insertion. Previously, TypeScript did not insert semicolons when these keywords were on their own line. Now that this is fixed, `abstract class D` no longer correctly extends `C` in the following example, and instead declares a concrete method `m` and an additional property named `abstract` .

Note that `async` and `declare` already correctly did ASI.

Example:

```ts
abstract class C {
    abstract m(): number;
}
abstract class D extends C {
    abstract
    m(): number;
}
```

Recommendations:

Remove line breaks after keywords when defining class members. In general, avoid relying on automatic semicolon insertion.

TypeScript 1.6

For full list of breaking changes see the [breaking change issues](#).

Strict object literal assignment checking

It is an error to specify properties in an object literal that were not specified on the target type, when assigned to a variable or passed for a parameter of a non-empty target type.

This new strictness can be disabled with the --suppressExcessPropertyErrors compiler option.

Example:

```
var x: { foo: number };
x = { foo: 1, baz: 2 };  // Error, excess property `baz`

var y: { foo: number, bar?: number };
y = { foo: 1, baz: 2 };  // Error, excess or misspelled property `baz`
```

Recommendations:

To avoid the error, there are few remedies based on the situation you are looking into:

If the target type accepts additional properties, add an indexer:

```
var x: { foo: number, [x: string]: any };
x = { foo: 1, baz: 2 };  // OK, `baz` matched by index signature
```

If the source types are a set of related types, explicitly specify them using union types instead of just specifying the base type.

```
let animalList: (Dog | Cat | Turkey)[] = [    // use union type instead of Animal
    {name: "Milo", meow: true },
    {name: "Pepper", bark: true},
    {name: "koko", gobble: true}
];
```

Otherwise, explicitly cast to the target type to avoid the warning message:

```
interface Foo {
    foo: number;
}
interface FooBar {
    foo: number;
    bar: number;
}
var y: Foo;
y = <FooBar>{ foo: 1, bar: 2 };
```

CommonJS module resolution no longer assumes paths are relative

Previously, for the files `one.ts` and `two.ts`, an import of `"one"` in `two.ts` would resolve to `one.ts` if they resided in the same directory.

In TypeScript 1.6, `"one"` is no longer equivalent to "./one" when compiling with CommonJS. Instead, it is searched as relative to an appropriate `node_modules` folder as would be resolved by runtimes such as Node.js. For details, see [the issue that describes the resolution algorithm](#).

Example:

`./one.ts`

```
export function f() {
    return 10;
}
```

`./two.ts`

```
import { f as g } from "one";
```

Recommendations:

Fix any non-relative import names that were unintended (strongly suggested).

`./one.ts`

```
export function f() {
    return 10;
}
```

`./two.ts`

```
import { f as g } from "./one";
```

Set the `--moduleResolution` compiler option to `classic`.

Function and class default export declarations can no longer merge with entities intersecting in their meaning

Declaring an entity with the same name and in the same space as a default export declaration is now an error; for example,

```
export default function foo() {
}

namespace foo {
    var x = 100;
}
```

and

```
export default class Foo {
    a: number;
```

```
}

interface Foo {
    b: string;
}
```

both cause an error.

However, in the following example, merging is allowed because the namespace does does not have a meaning in the value space:

```
export default class Foo {
}

namespace Foo {
}
```

Recommendations:

Declare a local for your default export and use a separate `export default` statement as so:

```
class Foo {
    a: number;
}

interface foo {
    b: string;
}

export default Foo;
```

For more details see the originating issue.

Module bodies are parsed in strict mode

In accordance with the ES6 spec, module bodies are now parsed in strict mode. module bodies will behave as if `"use strict"` was defined at the top of their scope; this includes flagging the use of `arguments` and `eval` as variable or parameter names, use of future reserved words as variables or parameters, use of octal numeric literals, etc..

Changes to DOM API's in the standard library

- **MessageEvent** and **ProgressEvent** constructors now expect arguments; see issue #4295 for more details.
- **ImageData** constructor now expects arguments; see issue #4220 for more details.
- **File** constructor now expects arguments; see issue #3999 for more details.

System module output uses bulk exports

The compiler uses the new bulk-export variation of the `_export` function in the System module format that takes any object containing key value pairs (optionally an entire module object for export *) as arguments instead of key, value.

The module loader needs to be updated to v0.17.1 or higher.

.js content of npm package is moved from 'bin' to 'lib' folder

Entry point of TypeScript npm package was moved from `bin` to `lib` to unblock scenarios when 'node_modules/typescript/bin/typescript.js' is served from IIS (by default `bin` is in the list of hidden segments so IIS will block access to this folder).

TypeScript npm package does not install globally by default

TypeScript 1.6 removes the `preferGlobal` flag from package.json. If you rely on this behaviour please use `npm install -g typescript`.

Decorators are checked as call expressions

Starting with 1.6, decorators type checking is more accurate; the compiler will checks a decorator expression as a call expression with the decorated entity as a parameter. This can cause error to be reported that were not in previous releases.

TypeScript 1.5

For full list of breaking changes see the breaking change issues.

Referencing `arguments` in arrow functions is not allowed

This is an alignment with the ES6 semantics of arrow functions. Previously arguments within an arrow function would bind to the arrow function arguments. As per ES6 spec draft 9.2.12, arrow functions do not have an arguments objects. In TypeScript 1.5, the use of arguments object in arrow functions will be flagged as an error to ensure your code ports to ES6 with no change in semantics.

Example:

```
function f() {
    return () => arguments; // Error: The 'arguments' object cannot be referenced in an arrow function.
}
```

Recommendations:

```
// 1. Use named rest args
function f() {
    return (...args) => { args; }
}

// 2. Use function expressions instead
function f() {
    return function(){ arguments; }
}
```

Enum reference in-lining changes

For regular enums, pre 1.5, the compiler *only* inline constant members, and a member was only constant if its initializer was a literal. That resulted in inconsistent behavior depending on whether the enum value is initialized with a literal or an expression. Starting with Typescript 1.5 all non-const enum members are not inlined.

Example:

```
var x = E.a;  // previously inlined as "var x = 1; /*E.a*/"

enum E {
   a = 1
}
```

Recommendation: Add the `const` modifier to the enum declaration to ensure it is consistently inlined at all consumption sites.

For more details see issue [#2183](#2183).

Contextual type flows through `super` and parenthesized expressions

Prior to this release, contextual types did not flow through parenthesized expressions. This has forced explicit type casts, especially in cases where parentheses are *required* to make an expression parse.

In the examples below, `m` will have a contextual type, where previously it did not.

```
var x: SomeType = (n) => ((m) => q);
var y: SomeType = t ? (m => m.length) : undefined;

class C extends CBase<string> {
    constructor() {
        super({
            method(m) { return m.length; }
        });
    }
}
```

See issues [#1425](#1425) and [#920](#920) for more details.

DOM interface changes

TypeScript 1.5 refreshes the DOM types in lib.d.ts. This is the first major refresh since TypeScript 1.0; many IE-specific definitions have been removed in favor of the standard DOM definitions, as well as adding missing types like Web Audio and touch events.

Workaround:

You can keep using older versions of the library with newer version of the compiler. You will need to include a local copy of a previous version in your project. Here is the [last released version before this change (TypeScript 1.5-alpha)](#).

Here is a list of changes:

- Property `selection` is removed from type `Document`
- Property `clipboardData` is removed from type `Window`
- Removed interface `MSEventAttachmentTarget`
- Properties `onresize`, `disabled`, `uniqueID`, `removeNode`, `fireEvent`, `currentStyle`, `runtimeStyle` are removed from type `HTMLElement`
- Property `url` is removed from type `Event`
- Properties `execScript`, `navigate`, `item` are removed from type `Window`
- Properties `documentMode`, `parentWindow`, `createEventObject` are removed from type `Document`
- Property `parentWindow` is removed from type `HTMLDocument`
- Property `setCapture` does not exist anywhere now
- Property `releaseCapture` does not exist anywhere now
- Properties `setAttribute`, `styleFloat`, `pixelLeft` are removed from type `CSSStyleDeclaration`
- Property `selectorText` is removed from type `CSSRule`
- `CSSStyleSheet.rules` is of type `CSSRuleList` instead of `MSCSSRuleList`
- `documentElement` is of type `Element` instead of `HTMLElement`
- `Event` has a new required property `returnValue`
- `Node` has a new required property `baseURI`
- `Element` has a new required property `classList`
- `Location` has a new required property `origin`
- Properties `MSPOINTER_TYPE_MOUSE`, `MSPOINTER_TYPE_TOUCH` are removed from type `MSPointerEvent`
- `CSSStyleRule` has a new required property `readonly`
- Property `execUnsafeLocalFunction` is removed from type `MSApp`
- Global method `toStaticHTML` is removed
- `HTMLCanvasElement.getContext` now returns `CanvasRenderingContext2D | WebGLRenderingContex`
- Removed extension types `Dataview`, `Weakmap`, `Map`, `Set`
- `XMLHttpRequest.send` has two overloads `send(data?: Document): void;` and `send(data?: String): void;`
- `window.orientation` is of type `string` instead of `number`
- IE-specific `attachEvent` and `detachEvent` are removed from `Window`

Here is a list of libraries that are partly or entirely replaced by the added DOM types:

- `DefinitelyTyped/auth0/auth0.d.ts`
- `DefinitelyTyped/gamepad/gamepad.d.ts`
- `DefinitelyTyped/interactjs/interact.d.ts`
- `DefinitelyTyped/webaudioapi/waa.d.ts`
- `DefinitelyTyped/webcrypto/WebCrypto.d.ts`

For more details, please see the full change.

Class bodies are parsed in strict mode

In accordance with the ES6 spec, class bodies are now parsed in strict mode. Class bodies will behave as if `"use strict"` was defined at the top of their scope; this includes flagging the use of `arguments` and `eval` as variable or parameter names, use of future reserved words as variables or parameters, use of octal numeric literals, etc..

TypeScript 1.4

For full list of breaking changes see the [breaking change issues](#).

See [issue #868](#) for more details about breaking changes related to Union Types

Multiple Best Common Type Candidates

Given multiple viable candidates from a Best Common Type computation we now choose an item (depending on the compiler's implementation) rather than the first item.

```
var a: { x: number; y?: number };
var b: { x: number; z?: number };

// was { x: number; z?: number; }[]
// now { x: number; y?: number; }[]
var bs = [b, a];
```

This can happen in a variety of circumstances. A shared set of required properties and a disjoint set of other properties (optional or otherwise), empty types, compatible signature types (including generic and non-generic signatures when type parameters are stamped out with `any`).

Recommendation Provide a type annotation if you need a specific type to be chosen

```
var bs: { x: number; y?: number; z?: number }[] = [b, a];
```

Generic Type Inference

Using different types for multiple arguments of type T is now an error, even with constraints involved:

```
declare function foo<T>(x: T, y:T): T;
var r = foo(1, ""); // r used to be {}, now this is an error
```

With constraints:

```
interface Animal { x }
interface Giraffe extends Animal { y }
interface Elephant extends Animal { z }
function f<T extends Animal>(x: T, y: T): T { return undefined; }
var g: Giraffe;
var e: Elephant;
f(g, e);
```

See https://github.com/Microsoft/TypeScript/pull/824#discussion_r18665727 for explanation.

Recommendations Specify an explicit type parameter if the mismatch was intentional:

```
var r = foo<{}>(1, ""); // Emulates 1.0 behavior
var r = foo<string|number>(1, ""); // Most useful
var r = foo<any>(1, ""); // Easiest
f<Animal>(g, e);
```

or rewrite the function definition to specify that mismatches are OK:

```
declare function foo<T,U>(x: T, y:U): T|U;
function f<T extends Animal, U extends Animal>(x: T, y: U): T|U { return undefined;
}
```

Generic Rest Parameters

You cannot use heterogeneous argument types anymore:

```
function makeArray<T>(...items: T[]): T[] { return items; }
var r = makeArray(1, ""); // used to return {}[], now an error
```

Likewise for `new Array(...)`

Recommendations Declare a back-compatible signature if the 1.0 behavior was desired:

```
function makeArray<T>(...items: T[]): T[];
function makeArray(...items: {}[]): {}[];
function makeArray<T>(...items: T[]): T[] { return items; }
```

Overload Resolution with Type Argument Inference

```
var f10: <T>(x: T, b: () => (a: T) => void, y: T) => T;
var r9 = f10('', () => (a => a.foo), 1); // r9 was any, now this is an error
```

Recommendations Manually specify a type parameter

```
var r9 = f10<any>('', () => (a => a.foo), 1);
```

Strict Mode Parsing for Class Declarations and Class Expressions

ECMAScript 2015 Language Specification (ECMA-262 6th Edition) specifies that *ClassDeclaration* and *ClassExpression* are strict mode productions. Thus, additional restrictions will be applied when parsing a class declaration or class expression.

Examples:

```
class implements {}  // Invalid: implements is a reserved word in strict mode
class C {
    foo(arguments: any) {   // Invalid: "arguments" is not allow as a function
```

```
argument
        var eval = 10;       // Invalid: "eval" is not allowed as the left-hand-side
expression
        arguments = [];      // Invalid: arguments object is immutable
    }
}
```

For complete list of strict mode restrictions, please see Annex C - The Strict Mode of ECMAScript of ECMA-262 6th Edition.

TypeScript 1.1

For full list of breaking changes see the [breaking change issues](#).

Working with null and undefined in ways that are observably incorrect is now an error

Examples:

```
var ResultIsNumber17 = +(null + undefined);
// Operator '+' cannot be applied to types 'undefined' and 'undefined'.

var ResultIsNumber18 = +(null + null);
// Operator '+' cannot be applied to types 'null' and 'null'.

var ResultIsNumber19 = +(undefined + undefined);
// Operator '+' cannot be applied to types 'undefined' and 'undefined'.
```

Similarly, using null and undefined directly as objects that have methods now is an error

Examples:

```
null.toBAZ();

undefined.toBAZ();
```