# FAQs

## Common "Bugs" That Aren't Bugs

> *I've found a long-overlooked bug in TypeScript!*

Here are some behaviors that may look like bugs, but aren't.

- These two empty classes can be used in place of each other
    - See the FAQ Entry on this page
- I can use a non-`void`-returning function where one returning `void` is expected
    - See the FAQ Entry on this page
    - Prior discussion at #4544
- I'm allowed to use a shorter parameter list where a longer one is expected
    - See the FAQ Entry on this page
    - Prior discussion at #370, #9300, #9765, #9825, #13043, #16871, #13529, #13977, #17868, #20274, #20541, #21868, #26324, #30876

- `private` class members are actually visible at runtime
  - See the FAQ Entry on this page for a commonly suggested "fix"
  - Prior discussion at #564, #1537, #2967, #3151, #6748, #8847, #9733, #11033
- This conditional type returns `never` when it should return the true branch.
  - See this issue for discussion about *distributive conditional types*.
- This mapped type returns a primitive type, not an object type.
  - Mapped types declared as `{ [ K in keyof T ]: U }` where T is a type parameter are known as *homomorphic mapped types*, which means that the mapped type is a structure preserving function of `T`. When type parameter `T` is instantiated with a primitive type the mapped type evaluates to the same primitive.
- A method and a function property of the same type behave differently.
  - Methods are always bivariant in their argument, while function properties are contravariant in their argument under `strictFunctionTypes`. More discussion here.

## Common Feature Requests

> *I want to request one of the following features...*

Here's a list of common feature requests and their corresponding issue. Please leave comments in these rather than logging new issues.

- Safe navigation operator, AKA CoffeeScript's null conditional/propagating/propagation operator, AKA C#'s' `?.` operator #16
- Minification #8
- Extension methods #9
- Partial classes #563
- Something to do with `this` #513
- Strong typing of `Function` members `call` / `bind` / `apply` #212
- Runtime function overloading #3442

## Type System Behavior

### What is structural typing?

TypeScript uses *structural typing*. This system is different than the type system employed by some other popular languages you may have used (e.g. Java, C#, etc.)

The idea behind structural typing is that two types are compatible if their *members* are compatible. For example, in C# or Java, two classes named `MyPoint` and `YourPoint`, both with public `int` properties `x` and `y`, are not interchangeable, even though they are identical. But in a structural type system, the fact that these types have different names is immaterial. Because they have the same members with the same types, they are identical.

This applies to subtype relationships as well. In C++, for example, you could only use a `Dog` in place of an `Animal` if `Animal` was explicitly in `Dog`'s class heritage. In TypeScript, this is not the case - a `Dog` with at least as many members (with appropriate types) as `Animal` is a subtype of `Animal` regardless of explicit heritage.

This can have some surprising consequences for programmers accustomed to working in a nominally-typed language. Many questions in this FAQ trace their roots to structural typing and its implications. Once you grasp the basics of it, however, it's very easy to reason about.

### What is type erasure?

TypeScript *removes* type annotations, interfaces, type aliases, and other type system constructs during compilation.

Input:

```
var x: SomeInterface;
```

Output:

```
var x;
```

This means that at run-time, there is no information present that says that some variable `x` was declared as being of type `SomeInterface`.

The lack of run-time type information can be surprising for programmers who are used to extensively using reflection or other metadata systems. Many questions in this FAQ boil down to "because types are erased".

### Why are getters without setters not considered read-only?

> *I wrote some code like this and expected an error:*

```
class Foo {
   get bar() {
     return 42;
   }
}
let x = new Foo();
// Expected error here
x.bar = 10;
```

This is now an error in TypeScript 2.0 and later. See #12 for the suggestion tracking this issue.

### Why are function parameters bivariant?

> *I wrote some code like this and expected an error:*

```
function trainDog(d: Dog) { ... }
function cloneAnimal(source: Animal, done: (result: Animal) => void): void { ...
}
let c = new Cat();

// Runtime error here occurs because we end up invoking 'trainDog' with a 'Cat'
cloneAnimal(c, trainDog);
```

This is an unsoundness resulting from the lack of explicit covariant/contravariant annotations in the type system. Because of this omission, TypeScript must be more permissive when asked whether `(x: Dog) => void` is assignable to `(x: Animal) => void`.

To understand why, consider two questions: Is `Dog[]` a subtype of `Animal[]`? *Should* `Dog[]` be a subtype of `Animal[]` in TypeScript?

The second question (*should* `Dog[]` be a subtype of `Animal[]`?) is easier to analyze. What if the answer was "no"?

```
function checkIfAnimalsAreAwake(arr: Animal[]) { ... }

let myPets: Dog[] = [spot, fido];

// Error? Can't substitute Dog[] for Animal[]?
checkIfAnimalsAreAwake(myPets);
```

This would be *incredibly annoying*. The code here is 100% correct provided that `checkIfAnimalsAreAwake` doesn't modify the array. There's not a good reason to reject this program on the basis that `Dog[]` can't be used in place of `Animal[]` - clearly a group of `Dog`s is a group of `Animal`s here.

Back to the first question. When the type system decides whether or not `Dog[]` is a subtype of `Animal[]`, it does the following computation (written here as if the compiler took no optimizations), among many others:

- Is `Dog[]` assignable to `Animal[]`?
- Is each member of `Dog[]` assignable to `Animal[]`?
    - Is `Dog[].push` assignable to `Animal[].push`?
        - Is the type `(x: Dog) => number` assignable to `(x: Animal) => number`?
            - Is the first parameter type in `(x: Dog) => number` assignable to or from first parameter type in `(x: Animal) => number`?
                - Is `Dog` assignable to or from `Animal`?
                    - Yes.

As you can see here, the type system must ask "Is the type `(x: Dog) => number` assignable to `(x: Animal) => number`?", which is the same question the type system needed to ask for the original question. If TypeScript forced contravariance on parameters (requiring `Animal` being assignable to `Dog`), then `Dog[]` would not be assignable to `Animal[]`.

In summary, in the TypeScript type system, the question of whether a more-specific-type-accepting function should be assignable to a function accepting a less-specific type provides a prerequisite answer to whether an *array* of that more specific type should be assignable to an array of a less specific type. Having the latter *not* be the case would not be an acceptable type system in the vast majority of cases, so we have to take a correctness trade-off for the specific case of function argument types.

### Why are functions with fewer parameters assignable to functions that take more parameters?

> *I wrote some code like this and expected an error:*
>
> ```
> function handler(arg: string) {
>     // ....
> }
>
> function doSomething(callback: (arg1: string, arg2: number) => void) {
> ```

```
    callback('hello', 42);
}

// Expected error because 'doSomething' wants a callback of
// 2 parameters, but 'handler' only accepts 1
doSomething(handler);
```

This is the expected and desired behavior. First, refer to the "substitutability" primer at the top of the FAQ -- `handler` is a valid argument for `callback` because it can safely ignore extra parameters.

Second, let's consider another case:

```
let items = [1, 2, 3];
items.forEach(arg => console.log(arg));
```

This is isomorphic to the example that "wanted" an error. At runtime, `forEach` invokes the given callback with three arguments (value, index, array), but most of the time the callback only uses one or two of the arguments. This is a very common JavaScript pattern and it would be burdensome to have to explicitly declare unused parameters.

> *But `forEach` should just mark its parameters as optional! e.g. `forEach(callback: (element?: T, index?: number, array?: T[]))`*

This is *not* what an optional callback parameter means. Function signatures are always read from the *caller's* perspective. If `forEach` declared that its callback parameters were optional, the meaning of that is "`forEach` **might call the callback with 0 arguments**".

The meaning of an optional callback parameter is *this*:

```
// Invoke the provided function with 0 or 1 argument
function maybeCallWithArg(callback: (x?: number) => void) {
    if (Math.random() > 0.5) {
        callback();
    } else {
        callback(42);
    }
}
```

`forEach` *always* provides all three arguments to its callback. You don't have to check for the `index` argument to be `undefined` - it's always there; it's not optional.

There is currently not a way in TypeScript to indicate that a callback parameter *must* be present. Note that this sort of enforcement wouldn't ever directly fix a bug. In other words, in a hypothetical world where `forEach` callbacks were required to accept a minimum of one argument, you'd have this code:

```
[1, 2, 3].forEach(() => console.log("just counting"));
             //   ~~ Error, not enough arguments?
```

which would be "fixed", but *not made any more correct*, by adding a parameter:

```
[1, 2, 3].forEach(x => console.log("just counting"));
                // OK, but doesn't do anything different at all
```

### Why are functions returning non-`void` assignable to function returning `void`?

> *I wrote some code like this and expected an error:*
>
> ```
> function doSomething(): number {
>     return 42;
> }
>
> function callMeMaybe(callback: () => void) {
>     callback();
> }
>
> // Expected an error because 'doSomething' returns number, but 'callMeMaybe'
> // expects void-returning function
> callMeMaybe(doSomething);
> ```

This is the expected and desired behavior. First, refer to the "substitutability" primer -- the fact that `doSomething` returns "more" information than `callMeMaybe` is a valid substitution.

Second, let's consider another case:

```
let items = [1, 2];
callMeMaybe(() => items.push(3));
```

This is isomorphic to the example that "wanted" an error. `Array#push` returns a number (the new length of the array), but it's a safe substitute to use for a `void`-returning function.

Another way to think of this is that a `void`-returning callback type says "I'm not going to look at your return value, if one exists".

### Why are all types assignable to empty interfaces?

> *I wrote some code like this and expected an error:*
>
> ```
> interface Thing { /* nothing here */ }
> function doSomething(a: Thing) {
>   // mysterious implementation here
> }
> // Expected some or all of these to be errors
> doSomething(window);
> doSomething(42);
> doSomething('huh?');
> ```

Types with no members can be substituted by *any* type. In this example, `window`, `42`, and `'huh?'` all have the required members of a `Thing` (there are none).

In general, you should *never* find yourself declaring an `interface` with no properties.

### Can I make a type alias nominal?

> *I wrote the following code and expected an error:*
>
> ```
> type SomeUrl = string;
> type FirstName = string;
> let x: SomeUrl = "http://www.typescriptlang.org/";
> let y: FirstName = "Bob";
> x = y; // Expected error
> ```

Type aliases are simply *aliases* -- they are indistinguishable from the types they refer to.

A workaround involving intersection types to make "branded primitives" is possible:

```
// Strings here are arbitrary, but must be distinct
type SomeUrl = string & {'this is a url': {}};
type FirstName = string & {'person name': {}};

// Add type assertions
let x = <SomeUrl>'';
let y = <FirstName>'bob';
x = y; // Error

// OK
let xs: string = x;
let ys: string = y;
xs = ys;
```

You'll need to add a type assertion wherever a value of this type is created. These can still be aliased by `string` and lose type safety.

### How do I prevent two types from being structurally compatible?

> *I would like the following code to produce an error:*
>
> ```
> interface ScreenCoordinate {
>   x: number;
>   y: number;
> }
> interface PrintCoordinate {
>   x: number;
>   y: number;
> }
> function sendToPrinter(pt: PrintCoordinate) {
>   // ...
> }
> function getCursorPos(): ScreenCoordinate {
>   // Not a real implementation
>   return { x: 0, y: 0 };
> ```

```
}
// This should be an error
sendToPrinter(getCursorPos());
```

A possible fix if you really want two types to be incompatible is to add a 'brand' member:

```
interface ScreenCoordinate {
  _screenCoordBrand: any;
  x: number;
  y: number;
}
interface PrintCoordinate {
  _printCoordBrand: any;
  x: number;
  y: number;
}

// Error
sendToPrinter(getCursorPos());
```

Note that this will require a type assertion wherever 'branded' objects are created:

```
function getCursorPos(): ScreenCoordinate {
  // Not a real implementation
  return <ScreenCoordinate>{ x: 0, y: 0 };
}
```

See also #202 for a suggestion tracking making this more intuitive.

### How do I check at run-time if an object implements some interface?

> *I want to write some code like this:*
>
> ```
> interface SomeInterface {
>   name: string;
>   length: number;
> }
> interface SomeOtherInterface {
>   questions: string[];
> }
>
> function f(x: SomeInterface|SomeOtherInterface) {
>   // Can't use instanceof on interface, help?
>   if (x instanceof SomeInterface) {
>     // ...
>   }
> }
> ```

TypeScript types are erased (https://en.wikipedia.org/wiki/Type_erasure) during compilation. This means there is no built-in mechanism for performing runtime type checks. It's up to you to decide how you want to distinguish objects.

A popular method is to check for properties on an object. You can use user-defined type guards to accomplish this:

```
function isSomeInterface(x: any): x is SomeInterface {
  return typeof x.name === 'string' && typeof x.length === 'number';

function f(x: SomeInterface|SomeOtherInterface) {
  if (isSomeInterface(x)) {
    console.log(x.name); // Cool!
  }
}
```

### Why doesn't this incorrect cast throw a runtime error?

> *I wrote some code like this:*
>
> ```
> let x: any = true;
> let y = <string>x; // Expected: runtime error (can't convert boolean to string)
> ```
>
> *or this*
>
> ```
> let a: any = 'hmm';
> let b = a as HTMLElement; // expected b === null
> ```

TypeScript has *type assertions*, not *type casts*. The intent of `<T>x` is to say "TypeScript, please treat `x` as a `T`", not to perform a type-safe run-time conversion. Because types are erased, there is no direct equivalent of C#'s `expr as type` or `(type)expr` syntax.

### Why don't I get type checking for `(number) => string` or `(T) => T`?

> *I wrote some code like this and expected an error:*
>
> ```
> let myFunc: (number) => string = (n) => 'The number in hex is ' + n.toString(16);
> // Expected error because boolean is not number
> console.log(myFunc(true));
> ```

Parameter names in function types are required. The code as written describes a function taking one parameter named `number` of type `any`. In other words, this declaration

```
let myFunc: (number) => string;
```

is equivalent to this one

```
let myFunc: (number: any) => string;
```

You should instead write:

```
let myFunc: (myArgName: number) => string;
```

To avoid this problem, turn on the `noImplicitAny` flag, which will issue a warning about the implicit `any` parameter type.

### Why am I getting an error about a missing index signature?

> *These three functions seem to do the same thing, but the last one is an error. Why is this the case?*

```
interface StringMap {
  [key: string]: string;
}

function a(): StringMap {
  return { a: "1" }; // OK
}

function b(): StringMap {
  var result: StringMap = { a: "1" };
  return result; // OK
}

function c(): StringMap {
  var result = { a: "1" };
  return result; // Error - result lacks index signature, why?
}
```

This isn't now an error in TypeScript 1.8 and later. As for earlier versions:

Contextual typing occurs when the context of an expression gives a hint about what its type might be. For example, in this initialization:

```
var x: number = y;
```

The expression `y` gets a contextual type of `number` because it's initializing a value of that type. In this case, nothing special happens, but in other cases more interesting things will occur.

One of the most useful cases is functions:

```
// Error: string does not contain a function called 'ToUpper'
var x: (n: string) => void = (s) => console.log(s.ToUpper());
```

How did the compiler know that `s` was a `string`? If you wrote that function expression by itself, `s` would be of type `any` and there wouldn't be any error issued. But because the function was contextually typed by the type of `x`, the parameter `s` acquired the type `string`. Very useful!

At the same time, an index signature specifies the type when an object is indexed by a `string` or a `number`. Naturally, these signatures are part of type checking:

```
var x: { [n: string]: Car; };
var y: { [n: string]: Animal; };
x = y; // Error: Cars are not Animals, this is invalid
```

The lack of an index signature is also important:

```
var x: { [n: string]: Car; };
var y: { name: Car; };
x = y; // Error: y doesn't have an index signature that returns a Car
```

The problem with assuming that objects don't have index signatures is that you then have no way to initialize an object with an index signature:

```
var c: Car;
// Error, or not?
var x: { [n: string]: Car } = { 'mine': c };
```

The solution is that when an object literal is contextually typed by a type with an index signature, that index signature is added to the type of the object literal if it matches. For example:

```
var c: Car;
var a: Animal;
// OK
var x: { [n: string]: Car } = { 'mine': c };
// Not OK: Animal is not Car
var y: { [n: string]: Car } = { 'mine': a };
```

Let's look at the original function:

```
function c(): StringMap {
  var result = { a: "1" };
  return result; // Error - result lacks index signature, why?
}
```

Because `result`'s type does not have an index signature, the compiler throws an error.

### Why am I getting `Supplied parameters do not match any signature` error?

A function or a method implementation signature is not part of the overloads.

```
function createLog(message:string): number;
function createLog(source:string, message?:string): number {
  return 0;
}

createLog("message"); // OK
createLog("source", "message"); // ERROR: Supplied parameters do not match any
signature
```

When having at least one overload signature declaration, only the overloads are visible. The last signature declaration, also known as the implementation signature, does not contribute to the shape of your signature. So to get the desired behavior you will need to add an additional overload:

```
function createLog(message:string): number;
function createLog(source:string, message:string): number
function createLog(source:string, message?:string): number {
  return 0;
}
```

The rationale here is that since JavaScript does not have function overloading, you will be doing parameter checking in your function, and this your function implementation might be more permissive than what you would want your users to call you through.

For instance you can require your users to call you using matching pairs of arguments, and implement this correctly without having to allow mixed argument types:

```
function compare(a: string, b: string): void;
function compare(a: number, b: number): void;
function compare(a: string|number, b: string|number): void {
  // Just an implementation and not visible to callers
}

compare(1,2) // OK
compare("s", "l") // OK
compare (1, "l") // Error.
```

---

## Functions

### Why can't I use `x` in the destructuring `function f({ x: number }) { /* ... */ }`?

> *I wrote some code like this and got an unexpected error:*
>
> ```
> function f({x: number}) {
>   // Error, x is not defined?
>   console.log(x);
> }
> ```

Destructuring syntax is counterintuitive for those accustomed to looking at TypeScript type literals. The syntax `f({x: number})` declares a destructuring *from the property* `x` *to the local* `number`.

Looking at the emitted code for this is instructive:

```
function f(_a) {
  // Not really what we were going for
  var number = _a.x;
}
```

To write this code correctly, you should write:

```
function f({x}: {x: number}) {
  // OK
```

```
  console.log(x);
}
```

If you can provide a default for all properties, it's preferable to write:

```
function f({x = 0}) {
  // x: number
  console.log(x);
}
```

---

## Classes

### Why do these empty classes behave strangely?

> *I wrote some code like this and expected an error:*
>
> ```
> class Empty { /* empty */ }
>
> var e2: Empty = window;
> ```

See the question "Why are all types assignable to empty interfaces?" in this FAQ. It's worth re-iterating the advice from that answer: in general, you should *never* declare a `class` with no properties. This is true even for subclasses:

```
class Base {
  important: number;
  properties: number;
}
class Alpha extends Base { }
class Bravo extends Base { }
```

`Alpha` and `Bravo` are structurally identical to each other, and to `Base`. This has a lot of surprising effects, so don't do it! If you want `Alpha` and `Bravo` to be different, add a private property to each.

### When and why are classes nominal?

What explains the difference between these two lines of code?

```
class Alpha { x: number }
class Bravo { x: number }
class Charlie { private x: number }
class Delta { private x: number }

let a = new Alpha(), b = new Bravo(), c = new Charlie(), d = new Delta();

a = b; // OK
c = d; // Error
```

In TypeScript, classes are compared structurally. The one exception to this is `private` and `protected` members. When a member is private or protected, it must *originate in the same declaration* to be considered the same as another private or protected member.

### Why does `this` get orphaned in my instance methods?

> *I wrote some code like this:*

```
class MyClass {
  x = 10;
  someCallback() {
    console.log(this.x); // Prints 'undefined', not 10
    this.someMethod(); // Throws error "this.method is not a function"
  }
  someMethod() {

  }
}

let obj = new MyClass();
window.setTimeout(obj.someCallback, 10);
```

Synonyms and alternate symptoms:

> - *Why are my class properties `undefined` in my callback?*
> - *Why does `this` point to `window` in my callback?*
> - *Why does `this` point to `undefined` in my callback?*
> - *Why am I getting an error `this.someMethod is not a function`?*
> - *Why am I getting an error `Cannot read property 'someMethod' of undefined`?*

In JavaScript, the value of `this` inside a function is determined as follows:

1. Was the function the result of calling `.bind`? If so, `this` is the first argument passed to `bind`
2. Was the function *directly* invoked via a property access expression `expr.method()`? If so, `this` is `expr`
3. Otherwise, `this` is `undefined` (in "strict" mode), or `window` in non-strict mode

The offending problem is this line of code:

```
window.setTimeout(obj.someCallback, 10);
```

Here, we provided a function reference to `obj.someCallback` to `setTimeout`. The function was then invoked on something that wasn't the result of `bind` and wasn't *directly* invoked as a method. Thus, `this` in the body of `someCallback` referred to `window` (or `undefined` in strict mode).

Solutions to this are outlined here: http://stackoverflow.com/a/20627988/1704166

### What's the difference between `Bar` and `typeof Bar` when `Bar` is a `class`?

> *I wrote some code like this and don't understand the error I'm getting:*

```
class MyClass {
  someMethod() { }
}
var x: MyClass;
// Cannot assign 'typeof MyClass' to MyClass? Huh?
x = MyClass;
```

It's important to remember that in JavaScript, classes are just functions. We refer to the class object itself -- the *value* `MyClass` -- as a *constructor function*. When a constructor function is invoked with `new`, we get back an object that is an *instance* of the class.

So when we define a class, we actually define two different *types*.

The first is the one referred to by the class' name; in this case, `MyClass`. This is the *instance* type of the class. It defines the properties and methods that an *instance* of the class has. It's the type returned by invoking the class' constructor.

The second type is anonymous. It is the type that the constructor function has. It contains a *construct signature* (the ability to be invoked with `new`) that returns an *instance* of the class. It also contains any `static` properties and methods the class might have. This type is typically referred to as the "static side" of the class because it contains those static members (as well as being the *constructor* for the class). We can refer to this type with the type query operator `typeof`.

The `typeof` operator (when used in a *type* position) expresses the *type* of an *expression*. Thus, `typeof MyClass` refers to the type of the expression `MyClass` - the *constructor function* that produces instances of `MyClass`.

### Why do my derived class property initializers overwrite values set in the base class constructor?

See #1617 for this and other initialization order questions

### What's the difference between `declare class` and `interface`?

TODO: Write up common symptoms of `declare class` / `interface` confusion.

See http://stackoverflow.com/a/14348084/1704166

### What does it mean for an interface to extend a class?

> *What does this code mean?*
>
> ```
> class Foo {
>   /* ... */
> }
> interface Bar extends Foo {
>
> }
> ```

This makes a type called `Bar` that has the same members as the instance shape of `Foo`. However, if `Foo` has private members, their corresponding properties in `Bar` must be implemented by a class which has `Foo` in its heritage. In general, this pattern is best avoided, especially if `Foo` has private members.

### Why am I getting "TypeError: [base class name] is not defined in `__extends`?

> *I wrote some code like this:*
>
> ```
> /** file1.ts **/
> class Alpha { /* ... */ }
>
> /** file2.ts **/
> class Bravo extends Alpha { /* ... */ }
> ```
>
> *I'm seeing a runtime error in `__extends`:*
>
> ```
> Uncaught TypeError: Alpha is not defined
> ```

The most common cause of this is that your HTML page includes a `<script>` tag for file2.ts, but not file1.ts. Add a script tag for the base class' output *before* the script tag for the derived class.

### Why am I getting "TypeError: Cannot read property 'prototype' of undefined" in `__extends`?

> *I wrote some code:*
>
> ```
> /** file1.ts **/
> class Alpha { /* ... */ }
>
> /** file2.ts **/
> class Bravo extends Alpha { /* ... */ }
> ```
>
> *I'm seeing a runtime error in `__extends`:*
>
> ```
> Uncaught TypeError: Cannot read property 'prototype' of undefined
> ```

This can happen for a few reasons.

The first is that, within a single file, you defined the derived class *before* the base class. Re-order the file so that base classes are declared before the derived classes.

If you're using `--out`, the compiler may be confused about what order you intended the files to be in. See the section "How do I control file ordering..." in the FAQ.

If you're not using `--out`, your script tags in the HTML file may be the wrong order. Re-order your script tags so that files defining base classes are included before the files defining the derived classes.

Finally, if you're using a third-party bundler of some sort, that bundler may be ordering files incorrectly. Refer to that tool's documentation to understand how to properly order the input files in the resulting output.

### Why doesn't extending built-ins like `Error`, `Array`, and `Map` work?

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

As a result, 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.

---

## Generics

### Why is `A<string>` assignable to `A<number>` for `interface A<T> { }`?

> *I wrote some code and expected an error:*

```
interface Something<T> {
  name: string;
}
let x: Something<number>;
let y: Something<string>;
// Expected error: Can't convert Something<number> to Something<string>!
x = y;
```

TypeScript uses a structural type system. When determining compatibility between `Something<number>` and `Something<string>`, we examine each *member* of each type. If all of the members are compatible, then the types themselves are compatible. But because `Something<T>` doesn't *use* `T` in any member, it doesn't matter what type `T` is - it has no bearing on whether the types are compatible.

In general, you should *never* have type parameters which are unused. The type will have unexpected compatibility (as shown here) and will also fail to have proper generic type inference in function calls.

### Why doesn't type inference work on this interface: `interface Foo<T> { }`?

> *I wrote some code like this:*

```
interface Named<T> {
  name: string;
}
class MyNamed<T> implements Named<T> {
  name: 'mine';
}
function findByName<T>(x: Named<T>): T {
  // TODO: Implement
  return undefined;
}

var x: MyNamed<string>;
var y = findByName(x); // expected y: string, got y: {}
```

TypeScript uses a structural type system. This structural-ness also applies during generic type inference. When inferring the type of `T` in the function call, we try to find *members* of type `T` on the `x` argument to figure out what `T` should be. Because there are no members which use `T`, there is nothing to infer from, so we return `{}`.

Note that if you use `T`, you get correct inference:

```
interface Named<T> {
  name: string;
  value: T; // <-- added
}
class MyNamed<T> implements Named<T> {
  name: 'mine';
```

```
  value: T; // <-- added
}
function findByName<T>(x: Named<T>): T {
  // TODO: Implement
  return undefined;
}

var x: MyNamed<string>;
var y = findByName(x); // got y: string;
```

Remember: You should *never* have unused type parameters! See the previous question for more reasons why this is bad.

### Why can't I write `typeof T`, `new T`, or `instanceof T` in my generic function?

> *I want to write some code like this:*
>
> ```
> function doSomething<T>(x: T) {
>   // Can't find name T?
>   let xType = typeof T;
>   let y = new xType();
>   // Same here?
>   if(someVar instanceof typeof T) {
>
>   }
>   // How do I instantiate?
>   let z = new T();
> }
> ```

Generics are erased during compilation. This means that there is no *value* `T` at runtime inside `doSomething`. The normal pattern that people try to express here is to use the constructor function for a class either as a factory or as a runtime type check. In both cases, using a *construct signature* and providing it as a parameter will do the right thing:

```
function create<T>(ctor: { new(): T }) {
  return new ctor();
}
var c = create(MyClass); // c: MyClass

function isReallyInstanceOf<T>(ctor: { new(...args: any[]): T }, obj: T) {
  return obj instanceof ctor;
}
```

---

## Modules

### Why are imports being elided in my emit?

> *I wrote some code like this*

> ```
> import someModule = require('./myMod');
>
> let x: someModule.SomeType = /* something */;
> ```
>
> *and the emit looked like this:*
>
> ```
> // Expected to see "var someModule = require('./myMod');" here!
>
> var x = /* something */;
> ```

TypeScript assumes that module imports do not have side effects, so it removes module imports that aren't used in any *expression*.

Use `import "mod"` syntax to force the module to be loaded.

```
import "./myMod"; // For side effects
```

You can also simply reference the module. This is the most universal workaround. A single use will do:

```
import someModule = require('./myMod');
someModule; // Used for side effects
```

### Why don't namespaces merge across different module files?

TODO: Port content from http://stackoverflow.com/questions/30357634/how-do-i-use-namespaces-with-typescript-external-modules

---

## Enums

### What's the difference between `enum` and `const enum`s?

TODO: Write up common symptoms of `enum` / `const enum` confusion.

See http://stackoverflow.com/questions/28818849/how-do-the-different-enum-variants-work-in-typescript

---

## Type Guards

### Why doesn't `x instanceof Foo` narrow `x` to `Foo`?

It depends what `x` is. If the type of `x` was originally not even compatible with `Foo`, then it wouldn't make much sense to narrow the type, so we don't.

More likely, you'll find yourself in this situation when `x` had the type `any`. The motivating example for this is something like the following:

```
function doIt(x) {
    if (x instanceof Object) {
        // Assume 'x' is a well-known object which
        // we know how to handle specifically
```

```
    }

    // Treat 'x' as a primitive
}
```

You'll see this type of code in TypeScript code that predates union types, or TypeScript code that's been ported over from JavaScript. If we narrowed from `any` to `Object` then there's not much you could really do with `x`. Using any properties that aren't in `Object` will lead to an error. This is not just true of `Object`, it's true of any other type with a defined set of properties.

### Why doesn't `isFoo(x)` narrow `x` to `Foo` when `isFoo` is a type guard?

TODO, but it is strongly related to the above section.

---

## Decorators

### Decorators on function declarations

TODO: Answer. Also, what did we mean here?

### What's the difference between `@dec` and `@dec()`? Shouldn't they be equivalent?

TODO: Answer

---

## JSX and React

### I wrote `declare var MyComponent: React.Component;`, why can't I write `<MyComponent />`?

> *I wrote some code like this. Why is there an error?*
>
> ```
> class Display extends React.Component<any, any> {
>     render() { ... }
> }
>
> let SomeThing: Display = /* ... */;
> // Error here, isn't this OK?
> let jsx = <SomeThing />;
> ```

This is a confusion between the *instance* and *static* side of a class. When React instantiates a component, it's invoking a *constructor function*. So when TypeScript sees a JSX `<TagName />`, it is validating that the result of *constructing* `TagName` produces a valid component.

But by declaring `let SomeThing: Display`, the code is indicating that `SomeThing` is the class *instance*, not the class *constructor*. Indeed, it would be a run-time error to write:

```
let SomeThing = new Display();
let jsx = <SomeThing />; // Not gonna work
```

The easiest fix is to use the `typeof` type operator.

```
let SomeThing: typeof Display = /* ... */;
```

---

## Things That Don't Work

### You should emit classes like this so they have real private members

> *If I write code like this:*
>
> ```
> class Foo {
>     private x = 0;
>     increment(): number {
>         this.x++;
>         return x;
>     }
> }
> ```
>
> *You should emit code like this so that 'x' is truly private:*
>
> ```
> var Foo = (function () {
>     var x = 0;
>
>     function Foo() {
>     }
>     Foo.prototype.increment = function () {
>         x++;
>         return x;
>     };
>     return Foo;
> })();
> ```

This code doesn't work. It creates a *single* private field that all classes share:

```
var a = new Foo();
a.increment(); // Prints 1
a.increment(); // Prints 2
var b = new Foo(); // increments on b should be independent of a
b.increment(); // Supposed to print 1, prints 3
a.increment(); // Should print 3, prints 4
```

### You should emit classes like this so they don't lose `this` in callbacks

> *If I write code like this:*
>
> ```
> class MyClass {
>     method() {
> ```

```
    }
}
```

*You should emit code like this so that I can't mess up `this` in callbacks:*

```
var MyClass = (function () {
    function MyClass() {
        this.method = function() {

        }
    }
    return MyClass;
})();
```

Two problems here.

First, the proposed behavior change is not in line with the ECMAScript specification. There isn't really anything else to be said on that front -- TypeScript must have the same runtime behavior as JavaScript.

Second, the runtime characteristics of this class are very surprising. Instead of allocating one closure per method, this allocates one closure per method *per instance*. This is expensive in terms of class initialization cost, memory pressure, and GC performance.

### You should have some class initialization which is impossible to emit code for

TODO: Port content from #1617

---

## External Tools

### How do I write unit tests with TypeScript?

- Typescript Deep Dive

---

## Commandline Behavior

### Why did adding an `import` or `export` modifier break my program?

*I wrote a program:*

```
/* myApp.ts */
function doSomething() {
    console.log('Hello, world!');
}
doSomething();
```

*I compiled it with `tsc --module commonjs myApp.ts --out app.js` and ran `node app.js` and got the expected output.*

*Then I added an `import` to it:*

```
import fs = require('fs');
function doSomething() {
    console.log('Hello, world!');
}
doSomething();
```

*Or added an `export` to it:*

```
export function doSomething() {
    console.log('Hello, world!');
}
doSomething();
```

*And now nothing happens when I run `app.js` !*

Modules -- those files containing top-level `export` or `import` statements -- are always compiled 1:1 with their corresponding js files. The `--out` option only controls where *script* (non-module) code is emitted. In this case, you should be running `node myApp.js`, because the *module* `myApp.ts` is always emitted to the file `myApp.js`.

This behavior has been fixed as of TypeScript 1.8; combining `--out` and `--module` is now an error for CommonJS module output.

### How do I control file ordering in combined output ( `--out` )?

The order of the generated files in the output follows that of the input files after the pre-processing pass.

The compiler performs a pre-processing pass on input files to resolve all *triple-slash reference directives* and *module import statements*. During this process, additional files can be added to the compilation.

The process starts with a set of root files; these are the file names specified on the command-line or in the `"files"` list in the `tsconfig.json` file. These root files are pre-processed in the *same order* they are specified. Before a file is added to the list, all triple-slash references and import statements in it are processed, and their targets included. Triple-slash references and import statements are resolved in a *depth-first manner*, in the order they appear in the file.

See more information about resolving triple-slash reference directives at triple-slash directives documentation and module import statements resolution at module resolution documentation.

### What does the error "Exported variable [name] has or is using private name [name]" mean?

This error occurs when you use the `--declaration` flag because the compiler is trying to produce a declaration file that *exactly* matches the module you defined.

Let's say you have this code:

```
/// MyFile.ts
class Test {
    // ... other members ....
    constructor(public parent: Test){}
}

export let t = new Test("some thing");
```

To produce a declaration file, the compiler has to write out a type for `t`:

```
/// MyFile.d.ts, auto-generated
export let t: ___fill in the blank___;
```

The member `t` has the type `Test`. The type `Test` is not visible because it's not exported, so we can't write `t: Test`.

In the *very simplest* cases, we could rewrite `Test`'s shape as an object type literal. But for the vast majority of cases, this doesn't work. As written, `Test`'s shape is self-referential and can't be written as an anonymous type. This also doesn't work if `Test` has any private or protected members. So rather than let you get 65% of the way through writing a realistic class and then start erroring then, we just issue the error (you're almost certainly going to hit later anyway) right away and save you the trouble.

To avoid this error:

1. Export the declarations used in the type in question
2. Specify an explicit type annotation for the compiler to use when writing declarations.

### Why does `--outDir` moves output after adding a new file?

`--outDir` specifies the "root" directory of the output. The compiler needs a "root" directory in the source to mirror into the output directory. If `--rootDir` is not specified, the compiler will compute one; this is based on a common path calculation, which is the longest common prefix of all your input files. Obviously this changes with adding a new file to the compilation that has a shorter path prefix.

To ensure the output does not change with adding new files, specify `--rootDir` on the command-line or in your `tsconfig.json`.

## `tsconfig.json` Behavior

### Why is a file in the `exclude` list still picked up by the compiler?

`tsconfig.json` turns a folder into a "project". Without specifying any `"exclude"` or `"files"` entries, all files in the folder containing the `tsconfig.json` and all its sub-directories are included in your compilation.

If you want to exclude some of the files, use `"exclude"`. If you would rather specify all the files instead of letting the compiler look them up, use `"files"`.

That was `tsconfig.json` automatic inclusion. There is a different issue, which is module resolution. By module resolution, I mean the compiler trying to understand what `ns` means in an import statement like: `import * ns from "mod"`. To do so, the compiler needs the definition of a module, this could be a .ts file for your own code, or a .d.ts for an imported definition file. If the file was found, it will be included regardless of whether it was excluded in the previous steps or not.

So to exclude a file from the compilation, you need to exclude both the file itself and **all** files that have an `import` or `/// <reference path="..." />` directive to it.

Use `tsc --listFiles` to list what files are included in your compilation, and `tsc --traceResolution` to see why they were included.

### How can I specify an `include`?

There is no way now to indicate an `"include"` to a file outside the current folder in the `tsconfig.json` (tracked by #1927). You can achieve the same result by either:

1. Using a `"files"` list, or ;
2. Adding a `/// <reference path="..." />` directive in one of the files in your directory.

### Why am I getting the `error TS5055: Cannot write file 'xxx.js' because it would overwrite input file.` when using JavaScript files?

For a TypeScript file, the TypeScript compiler by default emits the generated JavaScript files in the same directory with the same base file name. Because the TypeScript files and emitted JavaScript files always have different file extensions, it is safe to do so. However, if you have set the `allowJs` compiler option to `true` and didn't set any emit output options ( `outFile` and `outDir` ), the compiler will try to emit JavaScript source files by the same rule, which will result in the emitted JavaScript file having the same file name with the source file. To avoid accidentally overwriting your source file, the compiler will issue this warning and skip writing the output files.

There are multiple ways to solve this issue, though all of them involve configuring compiler options, therefore it is recommended that you have a `tsconfig.json` file in the project root to enable this. If you don't want JavaScript files included in your project at all, simply set the `allowJs` option to `false` ; If you do want to include and compile these JavaScript files, set the `outDir` option or `outFile` option to direct the emitted files elsewhere, so they won't conflict with your source files; If you just want to include the JavaScript files for editing and don't need to compile, set the `noEmit` compiler option to `true` to skip the emitting check.

## Comments

### Why some comments are not preserved in emitted JavaScript even when `--removeComments` is not specified?

TypeScript compiler uses a position of a node in the abstract syntax tree to retrieve its comments during emit. Because the compiler does not store all tokens into the tree, some comments may be missed in an output JavaScript file. For example, we do not store following tokens into the tree `,` , `{` , `}` , `(` , `)` . Therefore, trailing comments or leading comments of such tokens cannot be retrieved during emit. At the moment, there is not an easy method to preserve such comments without storing those tokens. Doing so, however, can significantly increase the tree size and potentially have performance impact.

Some cases where TypeScript compiler will not be able to preserve your comments:

```
/* comment */
<div>
    {/* comment will not be emitted */}
</div>

var x = {
    prop1: 1, // won't get emitted because we can't retrieve this comment
    prop2: 2  // will be emitted
}

function foo() /* this comment can't be preserved */ { }
```

### Why Copyright comments are removed when `--removeComments` is true?

TypeScript compiler will preserve copyright comment regardless of `--removeComments`. For a comment to be considered a copyright comment, it must have the following characteristics:

- a top-of-file comment following by empty line, separating it from the first statement.
- begin with `/*!`

---

# Glossary and Terms in this FAQ

### Dogs, Cats, and Animals, Oh My

For some code examples, we'll use a hypothetical type hierarchy:

```
      Animal
     /      \
   Dog      Cat
```

That is, all `Dog`s are `Animal`s, all `Cat`s are `Animal`s, but e.g. a function expecting a `Dog` cannot accept an argument of type `Cat`. If you want to try these examples in the TypeScript Playground, start with this template:

```ts
interface Animal {
  move(): void;
}
interface Dog extends Animal {
  woof: string;
}
interface Cat extends Animal {
  meow: string;
}
```

Other examples will use DOM types like `HTMLElement` and `HTMLDivElement` to highlight concrete real-world implications of certain behaviors.

### "Substitutability"

Many answers relating to the type system make reference to Substitutability. This is a principle that says that if an object `X` can be used in place of some object `Y`, then `X` is a *subtype* of `Y`. We also commonly say that `X` is *assignable to* `Y` (these terms have slightly different meanings in TypeScript, but the difference is not important here).

In other words, if I ask for a `fork`, a `spork` is an acceptable *substitute* because it has the same functions and properties of a `fork` (three prongs and a handle).

### Trailing, leading, and detached comments

TypeScript classifies comments into three different types:

- Leading comment : a comment before a node followed by newline.
- Trailing comment : a comment after a node and in the same line as the node.

- Detached comment : a comment that is not part of any node such as copyright comment.

```
/*! Top-of-file copyright comment is a detached comment */

/* Leading comments of the function AST node */
function foo /* trailing comments of the function name, "foo", AST node */ () {
  /* Detached comment */

  let x = 10;
}
```

---

# GitHub Process Questions

### What do the labels on these issues mean?

> *What are all the labels people keep putting on my issues?*

- **help wanted**: We are accepting pull requests to implement this feature or fix this bug. PRs must adhere to the rules specified in `CONTRIBUTING.md`
- **Breaking Change**: Fixing this bug or implementing this feature will break code that someone could have plausibly written (i.e. we do not consider new errors in nonsense code like `undefined.throwSomething()` to be breaking changes)
- **By Design**: This is an intentional behavior of TypeScript
- **Canonical**: This issue contains a lengthy explanation of a common question or misconception
- **Committed**: Someone from the TypeScript team will fix this bug or implement this feature
- **Declined**: For reasons explained in the issue, we are not going to accept this suggestion (note: See "I disagree with the outcome..." section)
- **Discussion**: This issue is a discussion with no defined outcome. The TypeScript team may weigh in on these issues, but they are not regularly reviewed
- **Duplicate**: This issue is the same, or has the same root cause, as another issue
- **Effort**: Easy/Moderate/Difficult: For issues marked as 'help wanted', these are an approximation of how difficult we think fixing the bug or implementing the feature will be. As a rough guide, fixing typos or modifying lib.d.s are generally Easy; work that requires understanding the basics of the code base is Moderate; things marked Difficult will require an understanding that is rare outside the core TypeScript team
- **good first issue**: These are 'Effort: easy' issues, good for your first contribution
- **ES6 / ES7 / ES Next**: Refers to issues related to features found in these specific ECMAScript versions
- **External**: Catch-all bucket when an issue reported is not an issue with TypeScript, but rather an external tool, library, website, person, or situation
- **Fixed**: This bug has been fixed. Generally, you will see these bugs fixed in the nightly version( `npm install typescript@next` ) within 24-48 hours
- **High Priority**: Issues affecting runtime behavior or high-occurrence crashes
- **Infrastructure**: Technical debt associated with the TypeScript project
- **In Discussion**: The suggestion is ready to be discussed at a Design Meeting or Suggestion Backlog Slog
- **Needs More Info**: The team needs more information about this suggestion or bug in order to understand what's going on. Generally, Suggestions will start out as Needs More Info, graduate to Needs Proposal, then finally go to In Discussion
- **Needs Proposal**: A suggestion that has a well-understood use case and a plausible outline of a solution, but lacks a formal definition of how exactly the problem will be solved

- **Out of Scope**: A suggestion that is outside the design parameters of TypeScript, either because it is a poor fit (e.g. make TypeScript look exactly like C#), is outside the constraints of the language (e.g. asm.js compilation), or better belongs to another tool or process (e.g. a built-in Collections library, or a runtime language feature that should start in the ECMAScript committee)
- **Question**: The issue is (intentionally or otherwise) simply asking a question about TypeScript. Answers to Questions, if provided, will generally be to-the-point because we do not have time to be a support community for all TypeScript users; please use Stack Overflow for TypeScript questions.
- **Revisit**: A suggestion or bug that can't be adequately addressed today, but will probably be able to be addressed in the future (e.g. we need to wait for the ECMAScript committee to make up its mind)
- **Suggestion**: Any suggestion
- **Too Complex**: Relative to the complexity required to implement or understand it, the suggestion does not provide enough value. This is a subjective measure, see "I disagree with the outcome...")
- **Won't Fix**: While the behavior described is agreed to be incorrect, the cost (in time, complexity, performance, etc.) is too high to justify taking a fix relative to the cost of simply living with the bug

### I disagree with the outcome of this suggestion

> *I don't think this suggestion should have been closed! What can I do next?*

To date, we've received over 1,000 suggestions on the TypeScript GitHub repo. We do our best to read, understand, prioritize, formalize and implement these suggestions. User feedback has been critical in shaping the success of the project. That said, sometimes we'll make decisions that you don't agree with, and sometimes we'll make the wrong call. What should you do if you think we should reconsider something? Let's walk through the five stages of grief.

*Denial*: It's healthy to believe that a suggestion might come back later. Do keep leaving feedback! We look at all comments on all issues - closed or otherwise. If you encounter a problem that would have been addressed by a suggestion, leave a comment explaining what you were doing and how you could have had a better experience. Having a record of these use cases helps us reprioritize.

*Anger*: Don't be angry. Specifically, remember that the TypeScript team does not have the resources to continuously relitigate closed suggestions.

*Bargaining*: Ask yourself: is there a smaller thing that would work? Many suggestions are simply too large of a hammer or too small of a nail. Think about the problem you're experiencing for a while and see if you can come up with a simpler solution that accomplishes the same goal.

*Depression*: Try not to be depressed about declined suggestions. The features that make it into the language instead might solve your problem in an even better way than you could have imagined.

*Acceptance*: Repeat this mantra: *I will accept the features I cannot have, have courage to submit pull requests for those I can, and the wisdom to know the difference by looking at the GitHub labels.*