01_Basic Types.md

Introduction

For programs to be useful, we need to be able to work with some of the simplest units of data: numbers, strings, structures, boolean values, and the like. In TypeScript, we support much the same types as you would expect in JavaScript, with a convenient enumeration type thrown in to help things along.

Boolean

The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a `boolean` value.

```
let isDone: boolean = false;
```

Number

As in JavaScript, all numbers in TypeScript are floating point values. These floating point numbers get the type `number`. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals introduced in ECMAScript 2015.

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

String

Another fundamental part of creating programs in JavaScript for webpages and servers alike is working with textual data. As in other languages, we use the type `string` to refer to these textual datatypes. Just like JavaScript, TypeScript also uses double quotes (`"`) or single quotes (`'`) to surround string data.

```
let color: string = "blue";
color = 'red';
```

You can also use *template strings*, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (`` ` ``) character, and embedded expressions are of the form `${ expr }`.

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`;
```

This is equivalent to declaring `sentence` like so:

```
let sentence: string = "Hello, my name is " + fullName + ".\n\n" +
    "I'll be " + (age + 1) + " years old next month.";
```

Array

TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by `[]` to denote an array of that element type:

```
let list: number[] = [1, 2, 3];
```

The second way uses a generic array type, `Array<elemType>`:

```
let list: Array<number> = [1, 2, 3];
```

Tuple

Tuple types allow you to express an array where the type of a fixed number of elements is known, but need not be the same. For example, you may want to represent a value as a pair of a `string` and a `number`:

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

When accessing an element with a known index, the correct type is retrieved:

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'
```

When accessing an element outside the set of known indices, a union type is used instead:

```
x[3] = "world"; // OK, 'string' can be assigned to 'string | number'

console.log(x[5].toString()); // OK, 'string' and 'number' both have 'toString'

x[6] = true; // Error, 'boolean' isn't 'string | number'
```

Union types are an advanced topic that we'll cover in a later chapter.

Enum

A helpful addition to the standard set of datatypes from JavaScript is the `enum`. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

By default, enums begin numbering their members starting at `0`. You can change this by manually setting the value of one of its members. For example, we can start the previous example at `1` instead of `0`:

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green;
```

Or, even manually set all the values in the enum:

```
enum Color {Red = 1, Green = 2, Blue = 4}
let c: Color = Color.Green;
```

A handy feature of enums is that you can also go from a numeric value to the name of that value in the enum. For example, if we had the value `2` but weren't sure what that mapped to in the `Color` enum above, we could look up the corresponding name:

```
enum Color {Red = 1, Green, Blue}
let colorName: string = Color[2];

console.log(colorName); // Displays 'Green' as its value is 2 above
```

Any

We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content, e.g. from the user or a 3rd party library. In these cases, we want to opt-out of type-checking and let the values pass through compile-time checks. To do so, we label these with the `any` type:

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

The `any` type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation. You might expect `Object` to play a similar role, as it does in other languages. But variables of type `Object` only allow you to assign any value to them - you can't call arbitrary methods on them, even ones that actually exist:

```
let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.
```

The `any` type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:

```
let list: any[] = [1, true, "free"];

list[1] = 100;
```

Void

`void` is a little like the opposite of `any`: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

```
function warnUser(): void {
    console.log("This is my warning message");
}
```

Declaring variables of type `void` is not useful because you can only assign `undefined` or `null` to them:

```
let unusable: void = undefined;
```

Null and Undefined

In TypeScript, both `undefined` and `null` actually have their own types named `undefined` and `null` respectively. Much like `void`, they're not extremely useful on their own:

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

By default `null` and `undefined` are subtypes of all other types. That means you can assign `null` and `undefined` to something like `number`.

However, when using the `--strictNullChecks` flag, `null` and `undefined` are only assignable to `void` and their respective types. This helps avoid *many* common errors. In cases where you want to pass in either a `string` or `null` or `undefined`, you can use the union type `string | null | undefined`. Once again, more on union types later on.

> As a note: we encourage the use of `--strictNullChecks` when possible, but for the purposes of this handbook, we will assume it is turned off.

Never

The `never` type represents the type of values that never occur. For instance, `never` is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns; Variables also acquire the type `never` when narrowed by any type guards that can never be true.

The `never` type is a subtype of, and assignable to, every type; however, *no* type is a subtype of, or assignable to, `never` (except `never` itself). Even `any` isn't assignable to `never`.

Some examples of functions returning `never`:

```
// Function returning never must have unreachable end point
function error(message: string): never {
    throw new Error(message);
}

// Inferred return type is never
function fail() {
    return error("Something failed");
}

// Function returning never must have unreachable end point
function infiniteLoop(): never {
    while (true) {
    }
}
```

Object

`object` is a type that represents the non-primitive type, i.e. any thing that is not `number`, `string`, `boolean`, `symbol`, `null`, or `undefined`.

With `object` type, APIs like `Object.create` can be better represented. For example:

```
declare function create(o: object | null): void;

create({ prop: 0 }); // OK
create(null); // OK

create(42); // Error
create("string"); // Error
create(false); // Error
create(undefined); // Error
```

Type assertions

Sometimes you'll end up in a situation where you'll know more about a value than TypeScript does. Usually this will happen when you know the type of some entity could be more specific than its current type.

Type assertions are a way to tell the compiler "trust me, I know what I'm doing." A type assertion is like a type cast in other languages, but performs no special checking or restructuring of data. It has no runtime impact, and is used purely by the compiler. TypeScript assumes that you, the programmer, have performed any special checks that you need.

Type assertions have two forms. One is the "angle-bracket" syntax:

```
let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;
```

And the other is the `as`-syntax:

```
let someValue: any = "this is a string";

let strLength: number = (someValue as string).length;
```

The two samples are equivalent. Using one over the other is mostly a choice of preference; however, when using TypeScript with JSX, only `as`-style assertions are allowed.

A note about `let`

You may've noticed that so far, we've been using the `let` keyword instead of JavaScript's `var` keyword which you might be more familiar with. The `let` keyword is actually a newer JavaScript construct that TypeScript makes available. We'll discuss the details later, but many common problems in JavaScript are alleviated by using `let`, so you should use it instead of `var` whenever possible.

02_Variable Declarations.md

Variable Declarations

`let` and `const` are two relatively new types of variable declarations in JavaScript. As we mentioned earlier, `let` is similar to `var` in some respects, but allows users to avoid some of the common "gotchas" that users run into in JavaScript. `const` is an augmentation of `let` in that it prevents re-assignment to a variable.

With TypeScript being a superset of JavaScript, the language naturally supports `let` and `const`. Here we'll elaborate more on these new declarations and why they're preferable to `var`.

If you've used JavaScript offhandedly, the next section might be a good way to refresh your memory. If you're intimately familiar with all the quirks of `var` declarations in JavaScript, you might find it easier to skip ahead.

`var` declarations

Declaring a variable in JavaScript has always traditionally been done with the `var` keyword.

```
var a = 10;
```

As you might've figured out, we just declared a variable named `a` with the value `10`.

We can also declare a variable inside of a function:

```
function f() {
    var message = "Hello, world!";

    return message;
}
```

and we can also access those same variables within other functions:

```
function f() {
    var a = 10;
    return function g() {
            var b = a + 1;
            return b;
        }
    }

    var g = f();
    g(); // returns '11'
```

In this above example, `g` captured the variable `a` declared in `f`. At any point that `g` gets called, the value of `a` will be tied to the value of `a` in `f`. Even if `g` is called once `f` is done running, it will be able to access and modify `a`.

```
function f() {
    var a = 1;

    a = 2;
    var b = g();
    a = 3;

    return b;

    function g() {
        return a;
    }
}

f(); // returns '2'
```

Scoping rules

`var` declarations have some odd scoping rules for those used to other languages. Take the following example:

```
function f(shouldInitialize: boolean) {
    if (shouldInitialize) {
        var x = 10;
    }

    return x;
}

f(true);  // returns '10'
f(false); // returns 'undefined'
```

Some readers might do a double-take at this example. The variable `x` was declared *within the `if` block*, and yet we were able to access it from outside that block. That's because `var` declarations are accessible anywhere within their containing function, module, namespace, or global scope - all which we'll go over later on -

regardless of the containing block. Some people call this *`var`-scoping* or *function-scoping*. Parameters are also function scoped.

These scoping rules can cause several types of mistakes. One problem they exacerbate is the fact that it is not an error to declare the same variable multiple times:

```
function sumMatrix(matrix: number[][]) {
    var sum = 0;
    for (var i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (var i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }

    return sum;
}
```

Maybe it was easy to spot out for some, but the inner `for`-loop will accidentally overwrite the variable `i` because `i` refers to the same function-scoped variable. As experienced developers know by now, similar sorts of bugs slip through code reviews and can be an endless source of frustration.

Variable capturing quirks

Take a quick second to guess what the output of the following snippet is:

```
for (var i = 0; i < 10; i++) {
    setTimeout(function() { console.log(i); }, 100 * i);
}
```

For those unfamiliar, `setTimeout` will try to execute a function after a certain number of milliseconds (though waiting for anything else to stop running).

Ready? Take a look:

```
10
10
10
10
10
10
10
10
10
10
```

Many JavaScript developers are intimately familiar with this behavior, but if you're surprised, you're certainly not alone. Most people expect the output to be

```
0
1
2
3
4
5
6
7
8
9
```

Remember what we mentioned earlier about variable capturing? Every function expression we pass to `setTimeout` actually refers to the same `i` from the same scope.

Let's take a minute to consider what that means. `setTimeout` will run a function after some number of milliseconds, *but only* after the `for` loop has stopped executing; By the time the `for` loop has stopped executing, the value of `i` is `10`. So each time the given function gets called, it will print out `10`!

A common work around is to use an IIFE - an Immediately Invoked Function Expression - to capture `i` at each iteration:

```
for (var i = 0; i < 10; i++) {
    // capture the current state of 'i'
    // by invoking a function with its current value
    (function(i) {
        setTimeout(function() { console.log(i); }, 100 * i);
    })(i);
}
```

This odd-looking pattern is actually pretty common. The `i` in the parameter list actually shadows the `i` declared in the `for` loop, but since we named them the same, we didn't have to modify the loop body too much.

`let` declarations

By now you've figured out that `var` has some problems, which is precisely why `let` statements were introduced. Apart from the keyword used, `let` statements are written the same way `var` statements are.

```
let hello = "Hello!";
```

The key difference is not in the syntax, but in the semantics, which we'll now dive into.

Block-scoping

When a variable is declared using `let`, it uses what some call *lexical-scoping* or *block-scoping*. Unlike variables declared with `var` whose scopes leak out to their containing function, block-scoped variables are not visible outside of their nearest containing block or `for`-loop.

```
function f(input: boolean) {
    let a = 100;

    if (input) {
        // Still okay to reference 'a'
        let b = a + 1;
        return b;
    }

    // Error: 'b' doesn't exist here
    return b;
}
```

Here, we have two local variables `a` and `b`. `a`'s scope is limited to the body of `f` while `b`'s scope is limited to the containing `if` statement's block.

Variables declared in a `catch` clause also have similar scoping rules.

```
try {
    throw "oh no!";
}
catch (e) {
    console.log("Oh well.");
}

// Error: 'e' doesn't exist here
console.log(e);
```

Another property of block-scoped variables is that they can't be read or written to before they're actually declared. While these variables are "present" throughout their scope, all points up until their declaration are part of their *temporal dead zone*. This is just a sophisticated way of saying you can't access them before the `let` statement, and luckily TypeScript will let you know that.

```
a++; // illegal to use 'a' before it's declared;
let a;
```

Something to note is that you can still *capture* a block-scoped variable before it's declared. The only catch is that it's illegal to call that function before the declaration. If targeting ES2015, a modern runtime will throw an error; however, right now TypeScript is permissive and won't report this as an error.

```
function foo() {
    // okay to capture 'a'
    return a;
}

// illegal call 'foo' before 'a' is declared
// runtimes should throw an error here
foo();

let a;
```

For more information on temporal dead zones, see relevant content on the Mozilla Developer Network.

Re-declarations and Shadowing

With `var` declarations, we mentioned that it didn't matter how many times you declared your variables; you just got one.

```
function f(x) {
    var x;
    var x;

    if (true) {
        var x;
    }
}
```

In the above example, all declarations of `x` actually refer to the *same* `x`, and this is perfectly valid. This often ends up being a source of bugs. Thankfully, `let` declarations are not as forgiving.

```
let x = 10;
let x = 20; // error: can't re-declare 'x' in the same scope
```

The variables don't necessarily need to both be block-scoped for TypeScript to tell us that there's a problem.

```
function f(x) {
    let x = 100; // error: interferes with parameter declaration
}

function g() {
    let x = 100;
    var x = 100; // error: can't have both declarations of 'x'
}
```

That's not to say that block-scoped variable can never be declared with a function-scoped variable. The block-scoped variable just needs to be declared within a distinctly different block.

```
function f(condition, x) {
    if (condition) {
        let x = 100;
        return x;
    }

    return x;
}

f(false, 0); // returns '0'
f(true, 0);  // returns '100'
```

The act of introducing a new name in a more nested scope is called *shadowing*. It is a bit of a double-edged sword in that it can introduce certain bugs on its own in the event of accidental shadowing, while also preventing certain bugs. For instance, imagine we had written our earlier `sumMatrix` function using `let` variables.

```
function sumMatrix(matrix: number[][]) {
    let sum = 0;
    for (let i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (let i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }

    return sum;
}
```

This version of the loop will actually perform the summation correctly because the inner loop's `i` shadows `i` from the outer loop.

Shadowing should *usually* be avoided in the interest of writing clearer code. While there are some scenarios where it may be fitting to take advantage of it, you should use your best judgement.

Block-scoped variable capturing

When we first touched on the idea of variable capturing with `var` declaration, we briefly went into how variables act once captured. To give a better intuition of this, each time a scope is run, it creates an "environment" of variables. That environment and its captured variables can exist even after everything within its scope has finished executing.

```
function theCityThatAlwaysSleeps() {
    let getCity;
    if (true) {
        let city = "Seattle";
        getCity = function() {
            return city;
        }
    }

    return getCity();
}
```

Because we've captured `city` from within its environment, we're still able to access it despite the fact that the `if` block finished executing.

Recall that with our earlier `setTimeout` example, we ended up needing to use an IIFE to capture the state of a variable for every iteration of the `for` loop. In effect, what we were doing was creating a new variable environment for our captured variables. That was a bit of a pain, but luckily, you'll never have to do that again in TypeScript.

`let` declarations have drastically different behavior when declared as part of a loop. Rather than just introducing a new environment to the loop itself, these declarations sort of create a new scope *per iteration*. Since this is what we were doing anyway with our IIFE, we can change our old `setTimeout` example to just use a `let` declaration.

```
for (let i = 0; i < 10 ; i++) {
    setTimeout(function() { console.log(i); }, 100 * i);
}
```

and as expected, this will print out

```
0
1
2
3
4
5
6
7
8
9
```

`const` declarations

`const` declarations are another way of declaring variables.

```
const numLivesForCat = 9;
```

They are like `let` declarations but, as their name implies, their value cannot be changed once they are bound. In other words, they have the same scoping rules as `let`, but you can't re-assign to them.

This should not be confused with the idea that the values they refer to are *immutable*.

```
const numLivesForCat = 9;
const kitty = {
    name: "Aurora",
    numLives: numLivesForCat,
}

// Error
kitty = {
    name: "Danielle",
    numLives: numLivesForCat
};

// all "okay"
kitty.name = "Rory";
kitty.name = "Kitty";
kitty.name = "Cat";
kitty.numLives--;
```

Unless you take specific measures to avoid it, the internal state of a `const` variable is still modifiable. Fortunately, TypeScript allows you to specify that members of an object are `readonly`. The chapter on Interfaces has the details.

`let` vs. `const`

Given that we have two types of declarations with similar scoping semantics, it's natural to find ourselves asking which one to use. Like most broad questions, the answer is: it depends.

Applying the principle of least privilege, all declarations other than those you plan to modify should use `const`. The rationale is that if a variable didn't need to get written to, others working on the same codebase shouldn't automatically be able to write to the object, and will need to consider whether they really need to reassign to the variable. Using `const` also makes code more predictable when reasoning about flow of data.

Use your best judgement, and if applicable, consult the matter with the rest of your team.

The majority of this handbook uses `let` declarations.

Destructuring

Another ECMAScript 2015 feature that TypeScript has is destructuring. For a complete reference, see the article on the Mozilla Developer Network. In this section, we'll give a short overview.

Array destructuring

The simplest form of destructuring is array destructuring assignment:

```
let input = [1, 2];
let [first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2
```

This creates two new variables named `first` and `second`. This is equivalent to using indexing, but is much more convenient:

```
first = input[0];
second = input[1];
```

Destructuring works with already-declared variables as well:

```
// swap variables
[first, second] = [second, first];
```

And with parameters to a function:

```
function f([first, second]: [number, number]) {
    console.log(first);
    console.log(second);
}
f([1, 2]);
```

You can create a variable for the remaining items in a list using the syntax `...`:

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(first); // outputs 1
console.log(rest); // outputs [ 2, 3, 4 ]
```

Of course, since this is JavaScript, you can just ignore trailing elements you don't care about:

```
let [first] = [1, 2, 3, 4];
console.log(first); // outputs 1
```

Or other elements:

```
let [, second, , fourth] = [1, 2, 3, 4];
```

Object destructuring

You can also destructure objects:

```
let o = {
    a: "foo",
    b: 12,
    c: "bar"
};
let { a, b } = o;
```

This creates new variables `a` and `b` from `o.a` and `o.b`. Notice that you can skip `c` if you don't need it.

Like array destructuring, you can have assignment without declaration:

```
({ a, b } = { a: "baz", b: 101 });
```

Notice that we had to surround this statement with parentheses. JavaScript normally parses a `{` as the start of block.

You can create a variable for the remaining items in an object using the syntax `...`:

```
let { a, ...passthrough } = o;
let total = passthrough.b + passthrough.c.length;
```

Property renaming

You can also give different names to properties:

```
let { a: newName1, b: newName2 } = o;
```

Here the syntax starts to get confusing. You can read `a: newName1` as "`a` as `newName1`". The direction is left-to-right, as if you had written:

```
let newName1 = o.a;
let newName2 = o.b;
```

Confusingly, the colon here does *not* indicate the type. The type, if you specify it, still needs to be written after the entire destructuring:

```
let { a, b }: { a: string, b: number } = o;
```

Default values

Default values let you specify a default value in case a property is undefined:

```
function keepWholeObject(wholeObject: { a: string, b?: number }) {
    let { a, b = 1001 } = wholeObject;
}
```

`keepWholeObject` now has a variable for `wholeObject` as well as the properties `a` and `b`, even if `b` is undefined.

Function declarations

Destructuring also works in function declarations. For simple cases this is straightforward:

```
type C = { a: string, b?: number }
function f({ a, b }: C): void {
    // ...
}
```

But specifying defaults is more common for parameters, and getting defaults right with destructuring can be tricky. First of all, you need to remember to put the pattern before the default value.

```
function f({ a="", b=0 } = {}): void {
    // ...
}
f();
```

> The snippet above is an example of type inference, explained later in the handbook.

Then, you need to remember to give a default for optional properties on the destructured property instead of the main initializer. Remember that `C` was defined with `b` optional:

```
function f({ a, b = 0 } = { a: "" }): void {
    // ...
}
f({ a: "yes" }); // ok, default b = 0
f(); // ok, default to { a: "" }, which then defaults b = 0
f({}); // error, 'a' is required if you supply an argument
```

Use destructuring with care. As the previous example demonstrates, anything but the simplest destructuring expression is confusing. This is especially true with deeply nested destructuring, which gets *really* hard to understand even without piling on renaming, default values, and type annotations. Try to keep destructuring expressions small and simple. You can always write the assignments that destructuring would generate yourself.

Spread

The spread operator is the opposite of destructuring. It allows you to spread an array into another array, or an object into another object. For example:

```
let first = [1, 2];
let second = [3, 4];
let bothPlus = [0, ...first, ...second, 5];
```

This gives bothPlus the value `[0, 1, 2, 3, 4, 5]`. Spreading creates a shallow copy of `first` and `second`. They are not changed by the spread.

You can also spread objects:

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };
let search = { ...defaults, food: "rich" };
```

Now `search` is `{ food: "rich", price: "$$", ambiance: "noisy" }`. Object spreading is more complex than array spreading. Like array spreading, it proceeds from left-to-right, but the result is still an object. This means that properties that come later in the spread object overwrite properties that come earlier. So if we modify the previous example to spread at the end:

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };
let search = { food: "rich", ...defaults };
```

Then the `food` property in `defaults` overwrites `food: "rich"`, which is not what we want in this case.

Object spread also has a couple of other surprising limits. First, it only includes an objects' own, enumerable properties. Basically, that means you lose methods when you spread instances of an object:

```
class C {
  p = 12;
  m() {
  }
}
let c = new C();
let clone = { ...c };
clone.p; // ok
clone.m(); // error!
```

Second, the Typescript compiler doesn't allow spreads of type parameters from generic functions. That feature is expected in future versions of the language.

03_Interfaces.md

Introduction

One of TypeScript's core principles is that type-checking focuses on the *shape* that values have. This is sometimes called "duck typing" or "structural subtyping". In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.

Our First Interface

The easiest way to see how interfaces work is to start with a simple example:

```
function printLabel(labelledObj: { label: string }) {
    console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

The type-checker checks the call to `printLabel`. The `printLabel` function has a single parameter that requires that the object passed in has a property called `label` of type string. Notice that our object actually has more properties than this, but the compiler only checks that *at least* the ones required are present and match the types required. There are some cases where TypeScript isn't as lenient, which we'll cover in a bit.

We can write the same example again, this time using an interface to describe the requirement of having the `label` property that is a string:

```
interface LabelledValue {
    label: string;
}

function printLabel(labelledObj: LabelledValue) {
    console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

The interface `LabelledValue` is a name we can now use to describe the requirement in the previous example. It still represents having a single property called `label` that is of type string. Notice we didn't have to explicitly say that the object we pass to `printLabel` implements this interface like we might have to in other languages. Here, it's only the shape that matters. If the object we pass to the function meets the requirements listed, then it's allowed.

It's worth pointing out that the type-checker does not require that these properties come in any sort of order, only that the properties the interface requires are present and have the required type.

Optional Properties

Not all properties of an interface may be required. Some exist under certain conditions or may not be there at all. These optional properties are popular when creating patterns like "option bags" where you pass an object to a function that only has a couple of properties filled in.

Here's an example of this pattern:

```
interface SquareConfig {
    color?: string;
    width?: number;
}

function createSquare(config: SquareConfig): {color: string; area: number} {
    let newSquare = {color: "white", area: 100};
    if (config.color) {
        newSquare.color = config.color;
    }
    if (config.width) {
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}

let mySquare = createSquare({color: "black"});
```

Interfaces with optional properties are written similar to other interfaces, with each optional property denoted by a `?` at the end of the property name in the declaration.

The advantage of optional properties is that you can describe these possibly available properties while still also preventing use of properties that are not part of the interface. For example, had we mistyped the name of the `color` property in `createSquare`, we would get an error message letting us know:

```
interface SquareConfig {
    color?: string;
    width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
    let newSquare = {color: "white", area: 100};
    if (config.clor) {
        // Error: Property 'clor' does not exist on type 'SquareConfig'
        newSquare.color = config.clor;
    }
    if (config.width) {
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}

let mySquare = createSquare({color: "black"});
```

Readonly properties

Some properties should only be modifiable when an object is first created. You can specify this by putting `readonly` before the name of the property:

```
interface Point {
    readonly x: number;
    readonly y: number;
}
```

You can construct a `Point` by assigning an object literal. After the assignment, `x` and `y` can't be changed.

```
let p1: Point = { x: 10, y: 20 };
p1.x = 5; // error!
```

TypeScript comes with a `ReadonlyArray<T>` type that is the same as `Array<T>` with all mutating methods removed, so you can make sure you don't change your arrays after creation:

```
let a: number[] = [1, 2, 3, 4];
let ro: ReadonlyArray<number> = a;
ro[0] = 12; // error!
```

```
ro.push(5); // error!
ro.length = 100; // error!
a = ro; // error!
```

On the last line of the snippet you can see that even assigning the entire `ReadonlyArray` back to a normal array is illegal. You can still override it with a type assertion, though:

```
a = ro as number[];
```

`readonly` vs `const`

The easiest way to remember whether to use readonly or const is to ask whether you're using it on a variable or a property. Variables use `const` whereas properties use `readonly`.

Excess Property Checks

In our first example using interfaces, TypeScript lets us pass `{ size: number; label: string; }` to something that only expected a `{ label: string; }`. We also just learned about optional properties, and how they're useful when describing so-called "option bags".

However, combining the two naively would let you to shoot yourself in the foot the same way you might in JavaScript. For example, taking our last example using `createSquare`:

```
interface SquareConfig {
    color?: string;
    width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
    // ...
}

let mySquare = createSquare({ colour: "red", width: 100 });
```

Notice the given argument to `createSquare` is spelled *`colour`* instead of `color`. In plain JavaScript, this sort of thing fails silently.

You could argue that this program is correctly typed, since the `width` properties are compatible, there's no `color` property present, and the extra `colour` property is insignificant.

However, TypeScript takes the stance that there's probably a bug in this code. Object literals get special treatment and undergo *excess property checking* when assigning them to other variables, or passing them as arguments. If an object literal has any properties that the "target type" doesn't have, you'll get an error.

```
// error: 'colour' not expected in type 'SquareConfig'
let mySquare = createSquare({ colour: "red", width: 100 });
```

Getting around these checks is actually really simple. The easiest method is to just use a type assertion:

```
let mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig);
```

However, a better approach might be to add a string index signature if you're sure that the object can have some extra properties that are used in some special way. If `SquareConfig` can have `color` and `width` properties with the above types, but could *also* have any number of other properties, then we could define it like so:

```
interface SquareConfig {
    color?: string;
    width?: number;
    [propName: string]: any;
}
```

We'll discuss index signatures in a bit, but here we're saying a `SquareConfig` can have any number of properties, and as long as they aren't `color` or `width`, their types don't matter.

One final way to get around these checks, which might be a bit surprising, is to assign the object to another variable: Since `squareOptions` won't undergo excess property checks, the compiler won't give you an error.

```
let squareOptions = { colour: "red", width: 100 };
let mySquare = createSquare(squareOptions);
```

Keep in mind that for simple code like above, you probably shouldn't be trying to "get around" these checks. For more complex object literals that have methods and hold state, you might need to keep these techniques in mind, but a majority of excess property errors are actually bugs. That means if you're running into excess property checking problems for something like option bags, you might need to revise some of your type declarations. In this instance, if it's okay to pass an object with both a `color` or `colour` property to `createSquare`, you should fix up the definition of `SquareConfig` to reflect that.

Function Types

Interfaces are capable of describing the wide range of shapes that JavaScript objects can take. In addition to describing an object with properties, interfaces are also capable of describing function types.

To describe a function type with an interface, we give the interface a call signature. This is like a function declaration with only the parameter list and return type given. Each parameter in the parameter list requires both name and type.

```
interface SearchFunc {
    (source: string, subString: string): boolean;
}
```

Once defined, we can use this function type interface like we would other interfaces. Here, we show how you can create a variable of a function type and assign it a function value of the same type.

```
let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
    let result = source.search(subString);
    return result > -1;
}
```

For function types to correctly type-check, the names of the parameters do not need to match. We could have, for example, written the above example like this:

```
let mySearch: SearchFunc;
mySearch = function(src: string, sub: string): boolean {
    let result = src.search(sub);
    return result > -1;
}
```

Function parameters are checked one at a time, with the type in each corresponding parameter position checked against each other. If you do not want to specify types at all, TypeScript's contextual typing can infer the argument types since the function value is assigned directly to a variable of type `SearchFunc`. Here, also, the return type of our function expression is implied by the values it returns (here `false` and `true`). Had the function expression returned numbers or strings, the type-checker would have warned us that return type doesn't match the return type described in the `SearchFunc` interface.

```
let mySearch: SearchFunc;
mySearch = function(src, sub) {
    let result = src.search(sub);
    return result > -1;
}
```

Indexable Types

Similarly to how we can use interfaces to describe function types, we can also describe types that we can "index into" like `a[10]`, or `ageMap["daniel"]`. Indexable types have an *index signature* that describes the types we can use to index into the object, along with the corresponding return types when indexing. Let's take an example:

```
interface StringArray {
    [index: number]: string;
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];
```

Above, we have a `StringArray` interface that has an index signature. This index signature states that when a `StringArray` is indexed with a `number`, it will return a `string`.

There are two types of supported index signatures: string and number. It is possible to support both types of indexers, but the type returned from a numeric indexer must be a subtype of the type returned from the string indexer. This is because when indexing with a `number`, JavaScript will actually convert that to a `string` before indexing into an object. That means that indexing with `100` (a `number`) is the same thing as indexing with `"100"` (a `string`), so the two need to be consistent.

```
class Animal {
    name: string;
}
class Dog extends Animal {
    breed: string;
}

// Error: indexing with a numeric string might get you a completely separate type of Animal!
interface NotOkay {
    [x: number]: Animal;
    [x: string]: Dog;
}
```

While string index signatures are a powerful way to describe the "dictionary" pattern, they also enforce that all properties match their return type. This is because a string index declares that `obj.property` is also available as `obj["property"]`. In the following example, `name`'s type does not match the string index's type, and the type-checker gives an error:

```
interface NumberDictionary {
    [index: string]: number;
    length: number;    // ok, length is a number
    name: string;      // error, the type of 'name' is not a subtype of the indexer
}
```

Finally, you can make index signatures readonly in order to prevent assignment to their indices:

```
interface ReadonlyStringArray {
    readonly [index: number]: string;
}
let myArray: ReadonlyStringArray = ["Alice", "Bob"];
myArray[2] = "Mallory"; // error!
```

You can't set `myArray[2]` because the index signature is readonly.

Class Types

Implementing an interface

One of the most common uses of interfaces in languages like C# and Java, that of explicitly enforcing that a class meets a particular contract, is also possible in TypeScript.

```
interface ClockInterface {
    currentTime: Date;
}

class Clock implements ClockInterface {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

You can also describe methods in an interface that are implemented in the class, as we do with `setTime` in the below example:

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}

class Clock implements ClockInterface {
    currentTime: Date;
    setTime(d: Date) {
        this.currentTime = d;
    }
    constructor(h: number, m: number) { }
}
```

Interfaces describe the public side of the class, rather than both the public and private side. This prohibits you from using them to check that a class also has particular types for the private side of the class instance.

Difference between the static and instance sides of classes

When working with classes and interfaces, it helps to keep in mind that a class has *two* types: the type of the static side and the type of the instance side. You may notice that if you create an interface with a construct signature and try to create a class that implements this interface you get an error:

```
interface ClockConstructor {
    new (hour: number, minute: number);
}

class Clock implements ClockConstructor {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

This is because when a class implements an interface, only the instance side of the class is checked. Since the constructor sits in the static side, it is not included in this check.

Instead, you would need to work with the static side of the class directly. In this example, we define two interfaces, `ClockConstructor` for the constructor and `ClockInterface` for the instance methods. Then for convenience we define a constructor function `createClock` that creates instances of the type that is passed to it.

```
interface ClockConstructor {
    new (hour: number, minute: number): ClockInterface;
}
interface ClockInterface {
    tick();
}

function createClock(ctor: ClockConstructor, hour: number, minute: number):
ClockInterface {
    return new ctor(hour, minute);
}

class DigitalClock implements ClockInterface {
    constructor(h: number, m: number) { }
    tick() {
        console.log("beep beep");
    }
}
class AnalogClock implements ClockInterface {
    constructor(h: number, m: number) { }
    tick() {
        console.log("tick tock");
    }
}

let digital = createClock(DigitalClock, 12, 17);
let analog = createClock(AnalogClock, 7, 32);
```

Because `createClock`'s first parameter is of type `ClockConstructor`, in `createClock(AnalogClock, 7, 32)`, it checks that `AnalogClock` has the correct constructor signature.

Extending Interfaces

Like classes, interfaces can extend each other. This allows you to copy the members of one interface into another, which gives you more flexibility in how you separate your interfaces into reusable components.

```
interface Shape {
    color: string;
}

interface Square extends Shape {
    sideLength: number;
}

let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
```

An interface can extend multiple interfaces, creating a combination of all of the interfaces.

```
interface Shape {
    color: string;
}

interface PenStroke {
    penWidth: number;
}

interface Square extends Shape, PenStroke {
    sideLength: number;
}

let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```

Hybrid Types

As we mentioned earlier, interfaces can describe the rich types present in real world JavaScript. Because of JavaScript's dynamic and flexible nature, you may occasionally encounter an object that works as a combination of some of the types described above.

One such example is an object that acts as both a function and an object, with additional properties:

```
interface Counter {
    (start: number): string;
    interval: number;
    reset(): void;
}

function getCounter(): Counter {
    let counter = <Counter>function (start: number) { };
    counter.interval = 123;
    counter.reset = function () { };
    return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;
```

When interacting with 3rd-party JavaScript, you may need to use patterns like the above to fully describe the shape of the type.

Interfaces Extending Classes

When an interface type extends a class type it inherits the members of the class but not their implementations. It is as if the interface had declared all of the members of the class without providing an implementation. Interfaces inherit even the private and protected members of a base class. This means that when you create an interface that extends a class with private or protected members, that interface type can only be implemented by that class or a subclass of it.

This is useful when you have a large inheritance hierarchy, but want to specify that your code works with only subclasses that have certain properties. The subclasses don't have to be related besides inheriting from the base class. For example:

```
class Control {
    private state: any;
}

interface SelectableControl extends Control {
    select(): void;
}

class Button extends Control implements SelectableControl {
    select() { }
}

class TextBox extends Control {
    select() { }
}
```

```
// Error: Property 'state' is missing in type 'Image'.
class Image implements SelectableControl {
    select() { }
}

class Location {

}
```

In the above example, `SelectableControl` contains all of the members of `Control`, including the private `state` property. Since `state` is a private member it is only possible for descendants of `Control` to implement `SelectableControl`. This is because only descendants of `Control` will have a `state` private member that originates in the same declaration, which is a requirement for private members to be compatible.

Within the `Control` class it is possible to access the `state` private member through an instance of `SelectableControl`. Effectively, a `SelectableControl` acts like a `Control` that is known to have a `select` method. The `Button` and `TextBox` classes are subtypes of `SelectableControl` (because they both inherit from `Control` and have a `select` method), but the `Image` and `Location` classes are not.

04_Classes.md

Introduction

Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components, but this may feel a bit awkward to programmers more comfortable with an object-oriented approach, where classes inherit functionality and objects are built from these classes. Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers will be able to build their applications using this object-oriented class-based approach. In TypeScript, we allow developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

Classes

Let's take a look at a simple class-based example:

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
```

```
let greeter = new Greeter("world");
```

The syntax should look familiar if you've used C# or Java before. We declare a new class `Greeter`. This class has three members: a property called `greeting`, a constructor, and a method `greet`.

You'll notice that in the class when we refer to one of the members of the class we prepend `this.`. This denotes that it's a member access.

In the last line we construct an instance of the `Greeter` class using `new`. This calls into the constructor we defined earlier, creating a new object with the `Greeter` shape, and running the constructor to initialize it.

Inheritance

In TypeScript, we can use common object-oriented patterns. One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance.

Let's take a look at an example:

```
class Animal {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}m.`);
    }
}

class Dog extends Animal {
    bark() {
        console.log('Woof! Woof!');
    }
}

const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

This example shows the most basic inheritance feature: classes inherit properties and methods from base classes. Here, `Dog` is a *derived* class that derives from the `Animal` *base* class using the `extends` keyword. Derived classes are often called *subclasses*, and base classes are often called *superclasses*.

Because `Dog` extends the functionality from `Animal`, we were able to create an instance of `Dog` that could both `bark()` and `move()`.

Let's now look at a more complex example.

```
class Animal {
    name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}

class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 45) {
        console.log("Galloping...");
        super.move(distanceInMeters);
    }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

This example covers a few other features we didn't previously mention. Again, we see the `extends` keywords used to create two new subclasses of `Animal`: `Horse` and `Snake`.

One difference from the prior example is that each derived class that contains a constructor function *must* call `super()` which will execute the constructor of the base class. What's more, before we *ever* access a property on `this` in a constructor body, we *have* to call `super()`. This is an important rule that TypeScript will enforce.

The example also shows how to override methods in the base class with methods that are specialized for the subclass. Here both `Snake` and `Horse` create a `move` method that overrides the `move` from `Animal`, giving it functionality specific to each class. Note that even though `tom` is declared as an `Animal`, since its value is a `Horse`, calling `tom.move(34)` will call the overriding method in `Horse`:

```
Slithering...
Sammy the Python moved 5m.
Galloping...
Tommy the Palomino moved 34m.
```

Public, private, and protected modifiers

Public by default

In our examples, we've been able to freely access the members that we declared throughout our programs. If you're familiar with classes in other languages, you may have noticed in the above examples we haven't had to use the word `public` to accomplish this; for instance, C# requires that each member be explicitly labeled `public` to be visible. In TypeScript, each member is `public` by default.

You may still mark a member `public` explicitly. We could have written the `Animal` class from the previous section in the following way:

```
class Animal {
    public name: string;
    public constructor(theName: string) { this.name = theName; }
    public move(distanceInMeters: number) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}
```

Understanding `private`

When a member is marked `private`, it cannot be accessed from outside of its containing class. For example:

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

new Animal("Cat").name; // Error: 'name' is private;
```

TypeScript is a structural type system. When we compare two different types, regardless of where they came from, if the types of all members are compatible, then we say the types themselves are compatible.

However, when comparing types that have `private` and `protected` members, we treat these types differently. For two types to be considered compatible, if one of them has a `private` member, then the other must have a `private` member that originated in the same declaration. The same applies to `protected` members.

Let's look at an example to better see how this plays out in practice:

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

class Rhino extends Animal {
    constructor() { super("Rhino"); }
}

class Employee {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

let animal = new Animal("Goat");
let rhino = new Rhino();
let employee = new Employee("Bob");

animal = rhino;
animal = employee; // Error: 'Animal' and 'Employee' are not compatible
```

In this example, we have an `Animal` and a `Rhino`, with `Rhino` being a subclass of `Animal`. We also have a new class `Employee` that looks identical to `Animal` in terms of shape. We create some instances of these classes and then try to assign them to each other to see what will happen. Because `Animal` and `Rhino` share the `private` side of their shape from the same declaration of `private name: string` in `Animal`, they are compatible. However, this is not the case for `Employee`. When we try to assign from an `Employee` to `Animal` we get an error that these types are not compatible. Even though `Employee` also has a `private` member called `name`, it's not the one we declared in `Animal`.

Understanding `protected`

The `protected` modifier acts much like the `private` modifier with the exception that members declared `protected` can also be accessed within deriving classes. For example,

```
class Person {
    protected name: string;
    constructor(name: string) { this.name = name; }
}

class Employee extends Person {
    private department: string;

    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name); // error
```

Notice that while we can't use `name` from outside of `Person`, we can still use it from within an instance method of `Employee` because `Employee` derives from `Person`.

A constructor may also be marked `protected`. This means that the class cannot be instantiated outside of its containing class, but can be extended. For example,

```
class Person {
    protected name: string;
    protected constructor(theName: string) { this.name = theName; }
}

// Employee can extend Person
class Employee extends Person {
    private department: string;

    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
let john = new Person("John"); // Error: The 'Person' constructor is protected
```

Readonly modifier

You can make properties readonly by using the `readonly` keyword. Readonly properties must be initialized at their declaration or in the constructor.

```
class Octopus {
    readonly name: string;
    readonly numberOfLegs: number = 8;
    constructor (theName: string) {
        this.name = theName;
    }
}
let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit"; // error! name is readonly.
```

Parameter properties

In our last example, we had to declare a readonly member `name` and a constructor parameter `theName` in the `Octopus` class, and we then immediately set `name` to `theName`. This turns out to be a very common practice. *Parameter properties* let you create and initialize a member in one place. Here's a further revision of the previous `Octopus` class using a parameter property:

```
class Octopus {
    readonly numberOfLegs: number = 8;
    constructor(readonly name: string) {
    }
}
```

Notice how we dropped `theName` altogether and just use the shortened `readonly name: string` parameter on the constructor to create and initialize the `name` member. We've consolidated the declarations and assignment into one location.

Parameter properties are declared by prefixing a constructor parameter with an accessibility modifier or `readonly`, or both. Using `private` for a parameter property declares and initializes a private member; likewise, the same is done for `public`, `protected`, and `readonly`.

Accessors

TypeScript supports getters/setters as a way of intercepting accesses to a member of an object. This gives you a way of having finer-grained control over how a member is accessed on each object.

Let's convert a simple class to use `get` and `set`. First, let's start with an example without getters and setters.

```
class Employee {
    fullName: string;
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    console.log(employee.fullName);
}
```

While allowing people to randomly set `fullName` directly is pretty handy, this might get us in trouble if people can change names on a whim.

In this version, we check to make sure the user has a secret passcode available before we allow them to modify the employee. We do this by replacing the direct access to `fullName` with a `set` that will check the passcode. We add a corresponding `get` to allow the previous example to continue to work seamlessly.

```
let passcode = "secret passcode";

class Employee {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }
    set fullName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._fullName = newName;
        }
        else {
            console.log("Error: Unauthorized update of employee!");
        }
    }
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    console.log(employee.fullName);
}
```

To prove to ourselves that our accessor is now checking the passcode, we can modify the passcode and see that when it doesn't match we instead get the message warning us we don't have access to update the employee.

A couple of things to note about accessors:

First, accessors require you to set the compiler to output ECMAScript 5 or higher. Downlevelling to ECMAScript 3 is not supported. Second, accessors with a `get` and no `set` are automatically inferred to be `readonly`. This is helpful when generating a `.d.ts` file from your code, because users of your property can see that they can't change it.

Static Properties

Up to this point, we've only talked about the *instance* members of the class, those that show up on the object when it's instantiated. We can also create *static* members of a class, those that are visible on the class itself rather than on the instances. In this example, we use `static` on the origin, as it's a general value for all grids. Each instance accesses this value through prepending the name of the class. Similarly to prepending `this.` in front of instance accesses, here we prepend `Grid.` in front of static accesses.

```
class Grid {
    static origin = {x: 0, y: 0};
    calculateDistanceFromOrigin(point: {x: number; y: number;}) {
        let xDist = (point.x - Grid.origin.x);
        let yDist = (point.y - Grid.origin.y);
        return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
    }
    constructor (public scale: number) { }
}

let grid1 = new Grid(1.0);  // 1x scale
let grid2 = new Grid(5.0);  // 5x scale
```

```
console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
```

Abstract Classes

Abstract classes are base classes from which other classes may be derived. They may not be instantiated directly. Unlike an interface, an abstract class may contain implementation details for its members. The `abstract` keyword is used to define abstract classes as well as abstract methods within an abstract class.

```
abstract class Animal {
    abstract makeSound(): void;
    move(): void {
        console.log("roaming the earth...");
    }
}
```

Methods within an abstract class that are marked as abstract do not contain an implementation and must be implemented in derived classes. Abstract methods share a similar syntax to interface methods. Both define the signature of a method without including a method body. However, abstract methods must include the `abstract` keyword and may optionally include access modifiers.

```
abstract class Department {

    constructor(public name: string) {
    }

    printName(): void {
        console.log("Department name: " + this.name);
    }

    abstract printMeeting(): void; // must be implemented in derived classes
}

class AccountingDepartment extends Department {

    constructor() {
        super("Accounting and Auditing"); // constructors in derived classes must
call super()
    }

    printMeeting(): void {
        console.log("The Accounting Department meets each Monday at 10am.");
    }

    generateReports(): void {
        console.log("Generating accounting reports...");
    }
}

let department: Department; // ok to create a reference to an abstract type
department = new Department(); // error: cannot create an instance of an abstract class
department = new AccountingDepartment(); // ok to create and assign a non-abstract subclass
department.printName();
department.printMeeting();
department.generateReports(); // error: method doesn't exist on declared abstract type
```

Advanced Techniques

Constructor functions

When you declare a class in TypeScript, you are actually creating multiple declarations at the same time. The first is the type of the *instance* of the class.

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter: Greeter;
greeter = new Greeter("world");
console.log(greeter.greet());
```

Here, when we say `let greeter: Greeter`, we're using `Greeter` as the type of instances of the class `Greeter`. This is almost second nature to programmers from other object-oriented languages.

We're also creating another value that we call the *constructor function*. This is the function that is called when we `new` up instances of the class. To see what this looks like in practice, let's take a look at the JavaScript created by the above example:

```
let Greeter = (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
})();

let greeter;
greeter = new Greeter("world");
console.log(greeter.greet());
```

Here, `let Greeter` is going to be assigned the constructor function. When we call `new` and run this function, we get an instance of the class. The constructor function also contains all of the static members of the class. Another way to think of each class is that there is an *instance* side and a *static* side.

Let's modify the example a bit to show this difference:

```
class Greeter {
    static standardGreeting = "Hello, there";
    greeting: string;
    greet() {
        if (this.greeting) {
            return "Hello, " + this.greeting;
        }
        else {
            return Greeter.standardGreeting;
        }
    }
}

let greeter1: Greeter;
greeter1 = new Greeter();
console.log(greeter1.greet());

let greeterMaker: typeof Greeter = Greeter;
greeterMaker.standardGreeting = "Hey there!";

let greeter2: Greeter = new greeterMaker();
console.log(greeter2.greet());
```

In this example, `greeter1` works similarly to before. We instantiate the `Greeter` class, and use this object. This we have seen before.

Next, we then use the class directly. Here we create a new variable called `greeterMaker`. This variable will hold the class itself, or said another way its constructor function. Here we use `typeof Greeter`, that is "give me the type of the `Greeter` class itself" rather than the instance type. Or, more precisely, "give me the type of the symbol called `Greeter`," which is the type of the constructor function. This type will contain all of the static members of Greeter along with the constructor that creates instances of the `Greeter` class. We show this by using `new` on `greeterMaker`, creating new instances of `Greeter` and invoking them as before.

Using a class as an interface

As we said in the previous section, a class declaration creates two things: a type representing instances of the class and a constructor function. Because classes create types, you can use them in the same places you would be able to use interfaces.

```
class Point {
    x: number;
    y: number;
}

interface Point3d extends Point {
    z: number;
}

let point3d: Point3d = {x: 1, y: 2, z: 3};
```

05_Functions.md

Introduction

Functions are the fundamental building block of any applications in JavaScript. They're how you build up layers of abstraction, mimicking classes, information hiding, and modules. In TypeScript, while there are classes, namespaces, and modules, functions still play the key role in describing how to *do* things. TypeScript also adds some new capabilities to the standard JavaScript functions to make them easier to work with.

Functions

To begin, just as in JavaScript, TypeScript functions can be created both as a named function or as an anonymous function. This allows you to choose the most appropriate approach for your application, whether you're building a list of functions in an API or a one-off function to hand off to another function.

To quickly recap what these two approaches look like in JavaScript:

```
// Named function
function add(x, y) {
    return x + y;
}

// Anonymous function
let myAdd = function(x, y) { return x + y; };
```

Just as in JavaScript, functions can refer to variables outside of the function body. When they do so, they're said to `capture` these variables. While understanding how this works, and the trade-offs when using this technique, are outside of the scope of this article, having a firm understanding how this mechanic is an important piece of working with JavaScript and TypeScript.

```
let z = 100;

function addToZ(x, y) {
    return x + y + z;
}
```

Function Types

Typing the function

Let's add types to our simple examples from earlier:

```
function add(x: number, y: number): number {
    return x + y;
}

let myAdd = function(x: number, y: number): number { return x + y; };
```

We can add types to each of the parameters and then to the function itself to add a return type. TypeScript can figure the return type out by looking at the return statements, so we can also optionally leave this off in many cases.

Writing the function type

Now that we've typed the function, let's write the full type of the function out by looking at each piece of the function type.

```
let myAdd: (x: number, y: number) => number =
    function(x: number, y: number): number { return x + y; };
```

A function's type has the same two parts: the type of the arguments and the return type. When writing out the whole function type, both parts are required. We write out the parameter types just like a parameter list, giving each parameter a name and a type. This name is just to help with readability. We could have instead written:

```
let myAdd: (baseValue: number, increment: number) => number =
    function(x: number, y: number): number { return x + y; };
```

As long as the parameter types line up, it's considered a valid type for the function, regardless of the names you give the parameters in the function type.

The second part is the return type. We make it clear which is the return type by using a fat arrow (`=>`) between the parameters and the return type. As mentioned before, this is a required part of the function type, so if the function doesn't return a value, you would use `void` instead of leaving it off.

Of note, only the parameters and the return type make up the function type. Captured variables are not reflected in the type. In effect, captured variables are part of the "hidden state" of any function and do not make up its API.

Inferring the types

In playing with the example, you may notice that the TypeScript compiler can figure out the type if you have types on one side of the equation but not the other:

```
// myAdd has the full function type
let myAdd = function(x: number, y: number): number { return  x + y; };

// The parameters 'x' and 'y' have the type number
let myAdd: (baseValue: number, increment: number) => number =
    function(x, y) { return x + y; };
```

This is called "contextual typing", a form of type inference. This helps cut down on the amount of effort to keep your program typed.

Optional and Default Parameters

In TypeScript, every parameter is assumed to be required by the function. This doesn't mean that it can't be given `null` or `undefined`, but rather, when the function is called the compiler will check that the user has provided a value for each parameter. The compiler also assumes that these parameters are the only parameters that will be passed to the function. In short, the number of arguments given to a function has to match the number of parameters the function expects.

```
function buildName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");                  // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too many parameters
let result3 = buildName("Bob", "Adams");         // ah, just right
```

In JavaScript, every parameter is optional, and users may leave them off as they see fit. When they do, their value is `undefined`. We can get this functionality in TypeScript by adding a `?` to the end of parameters we want to be optional. For example, let's say we want the last name parameter from above to be optional:

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
```

```
        return firstName + " " + lastName;
    else
        return firstName;
}

let result1 = buildName("Bob");                   // works correctly now
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too many parameters
let result3 = buildName("Bob", "Adams");          // ah, just right
```

Any optional parameters must follow required parameters. Had we wanted to make the first name optional rather than the last name, we would need to change the order of parameters in the function, putting the first name last in the list.

In TypeScript, we can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes `undefined` in its place. These are called default-initialized parameters. Let's take the previous example and default the last name to `"Smith"`.

```
function buildName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");                   // works correctly now, returns "Bob Smith"
let result2 = buildName("Bob", undefined);        // still works, also returns "Bob Smith"
let result3 = buildName("Bob", "Adams", "Sr.");  // error, too many parameters
let result4 = buildName("Bob", "Adams");          // ah, just right
```

Default-initialized parameters that come after all required parameters are treated as optional, and just like optional parameters, can be omitted when calling their respective function. This means optional parameters and trailing default parameters will share commonality in their types, so both

```
function buildName(firstName: string, lastName?: string) {
    // ...
}
```

and

```
function buildName(firstName: string, lastName = "Smith") {
    // ...
}
```

share the same type `(firstName: string, lastName?: string) => string`. The default value of `lastName` disappears in the type, only leaving behind the fact that the parameter is optional.

Unlike plain optional parameters, default-initialized parameters don't *need* to occur after required parameters. If a default-initialized parameter comes before a required parameter, users need to explicitly pass `undefined` to get the default initialized value. For example, we could write our last example with only a default initializer on `firstName`:

```
function buildName(firstName = "Will", lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");                  // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too many parameters
let result3 = buildName("Bob", "Adams");         // okay and returns "Bob Adams"
let result4 = buildName(undefined, "Adams");     // okay and returns "Will Adams"
```

Rest Parameters

Required, optional, and default parameters all have one thing in common: they talk about one parameter at a time. Sometimes, you want to work with multiple parameters as a group, or you may not know how many parameters a function will ultimately take. In JavaScript, you can work with the arguments directly using the `arguments` variable that is visible inside every function body.

In TypeScript, you can gather these arguments together into a variable:

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

Rest parameters are treated as a boundless number of optional parameters. When passing arguments for a rest parameter, you can use as many as you want; you can even pass none. The compiler will build an array of the arguments passed in with the name given after the ellipsis (`...`), allowing you to use it in your function.

The ellipsis is also used in the type of the function with rest parameters:

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

let buildNameFun: (fname: string, ...rest: string[]) => string = buildName;
```

`this`

Learning how to use `this` in JavaScript is something of a rite of passage. Since TypeScript is a superset of JavaScript, TypeScript developers also need to learn how to use `this` and how to spot when it's not being used correctly. Fortunately, TypeScript lets you catch incorrect uses of `this` with a couple of techniques. If you need to learn how `this` works in JavaScript, though, first read Yehuda Katz's Understanding JavaScript Function Invocation and "this". Yehuda's article explains the inner workings of `this` very well, so we'll just cover the basics here.

`this` and arrow functions

In JavaScript, `this` is a variable that's set when a function is called. This makes it a very powerful and flexible feature, but it comes at the cost of always having to know about the context that a function is executing in. This is notoriously confusing, especially when returning a function or passing a function as an argument.

Let's look at an example:

```
let deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        return function() {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Notice that `createCardPicker` is a function that itself returns a function. If we tried to run the example, we would get an error instead of the expected alert box. This is because the `this` being used in the function created by `createCardPicker` will be set to `window` instead of our `deck` object. That's because we call `cardPicker()` on its own. A top-level non-method syntax call like this will use `window` for `this`. (Note: under strict mode, `this` will be `undefined` rather than `window`).

We can fix this by making sure the function is bound to the correct `this` before we return the function to be used later. This way, regardless of how it's later used, it will still be able to see the original `deck` object. To do this, we change the function expression to use the ECMAScript 6 arrow syntax. Arrow functions capture the `this` where the function is created rather than where it is invoked:

```
let deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        // NOTE: the line below is now an arrow function, allowing us to capture
'this' right here
        return () => {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Even better, TypeScript will warn you when you make this mistake if you pass the `--noImplicitThis` flag to the compiler. It will point out that `this` in `this.suits[pickedSuit]` is of type `any`.

`this` parameters

Unfortunately, the type of `this.suits[pickedSuit]` is still `any`. That's because `this` comes from the function expression inside the object literal. To fix this, you can provide an explicit `this` parameter. `this` parameters are fake parameters that come first in the parameter list of a function:

```
function f(this: void) {
    // make sure `this` is unusable in this standalone function
}
```

Let's add a couple of interfaces to our example above, `Card` and `Deck`, to make the types clearer and easier to reuse:

```
interface Card {
    suit: string;
    card: number;
}
interface Deck {
    suits: string[];
    cards: number[];
    createCardPicker(this: Deck): () => Card;
}
let deck: Deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    // NOTE: The function now explicitly specifies that its callee must be of type
Deck
    createCardPicker: function(this: Deck) {
        return () => {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Now TypeScript knows that `createCardPicker` expects to be called on a `Deck` object. That means that `this` is of type `Deck` now, not `any`, so `--noImplicitThis` will not cause any errors.

`this` parameters in callbacks

You can also run into errors with `this` in callbacks, when you pass functions to a library that will later call them. Because the library that calls your callback will call it like a normal function, `this` will be `undefined`. With some work you can use `this` parameters to prevent errors with callbacks too. First, the library author needs to annotate the callback type with `this`:

```
interface UIElement {
    addClickListener(onclick: (this: void, e: Event) => void): void;
}
```

`this: void` means that `addClickListener` expects `onclick` to be a function that does not require a `this` type. Second, annotate your calling code with `this`:

```
class Handler {
    info: string;
    onClickBad(this: Handler, e: Event) {
        // oops, used this here. using this callback would crash at runtime
        this.info = e.message;
    }
}
let h = new Handler();
uiElement.addClickListener(h.onClickBad); // error!
```

With `this` annotated, you make it explicit that `onClickBad` must be called on an instance of `Handler`. Then TypeScript will detect that `addClickListener` requires a function that has `this: void`. To fix the error, change the type of `this`:

```
class Handler {
    info: string;
    onClickGood(this: void, e: Event) {
        // can't use this here because it's of type void!
        console.log('clicked!');
    }
}
let h = new Handler();
uiElement.addClickListener(h.onClickGood);
```

Because `onClickGood` specifies its `this` type as `void`, it is legal to pass to `addClickListener`. Of course, this also means that it can't use `this.info`. If you want both then you'll have to use an arrow function:

```
class Handler {
    info: string;
    onClickGood = (e: Event) => { this.info = e.message }
}
```

This works because arrow functions don't capture `this`, so you can always pass them to something that expects `this: void`. The downside is that one arrow function is created per object of type Handler. Methods, on the other hand, are only created once and attached to Handler's prototype. They are shared between all objects of type Handler.

Overloads

JavaScript is inherently a very dynamic language. It's not uncommon for a single JavaScript function to return different types of objects based on the shape of the arguments passed in.

```
let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x): any {
    // Check to see if we're working with an object/array
    // if so, they gave us the deck and we'll pick the card
    if (typeof x == "object") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { suit:
"hearts", card: 4 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);
```

```
let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

Here the `pickCard` function will return two different things based on what the user has passed in. If the users passes in an object that represents the deck, the function will pick the card. If the user picks the card, we tell them which card they've picked. But how do we describe this to the type system?

The answer is to supply multiple function types for the same function as a list of overloads. This list is what the compiler will use to resolve function calls. Let's create a list of overloads that describe what our `pickCard` accepts and what it returns.

```
let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
    // Check to see if we're working with an object/array
    // if so, they gave us the deck and we'll pick the card
    if (typeof x == "object") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { suit: "hearts", card: 4 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

With this change, the overloads now give us type-checked calls to the `pickCard` function.

In order for the compiler to pick the correct typecheck, it follows a similar process to the underlying JavaScript. It looks at the overload list, and proceeding with the first overload attempts to call the function with the provided parameters. If it finds a match, it picks this overload as the correct overload. For this reason, it's customary to order overloads from most specific to least specific.

Note that the `function pickCard(x): any` piece is not part of the overload list, so it only has two overloads: one that takes an object and one that takes a number. Calling `pickCard` with any other parameter types would cause an error.

06_Generics.md

Introduction

A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable. Components that are capable of working on the data of today as well as the data of tomorrow will give you the most flexible capabilities for building up large software systems.

In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is *generics*, that is, being able to create a component that can work over a variety of types rather than a single one. This allows users to consume these components and use their own types.

Hello World of Generics

To start off, let's do the "hello world" of generics: the identity function. The identity function is a function that will return back whatever is passed in. You can think of this in a similar way to the `echo` command.

Without generics, we would either have to give the identity function a specific type:

```
function identity(arg: number): number {
    return arg;
}
```

Or, we could describe the identity function using the `any` type:

```
function identity(arg: any): any {
    return arg;
}
```

While using `any` is certainly generic in that it will cause the function to accept any and all types for the type of `arg`, we actually are losing the information about what that type was when the function returns. If we passed in a number, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned. Here, we will use a *type variable*, a special kind of variable that works on types rather than values.

```
function identity<T>(arg: T): T {
    return arg;
}
```

We've now added a type variable `T` to the identity function. This `T` allows us to capture the type the user provides (e.g. `number`), so that we can use that information later. Here, we use `T` again as the return type. On inspection, we can now see the same type is used for the argument and the return type. This allows us to traffic that type information in one side of the function and out the other.

We say that this version of the `identity` function is generic, as it works over a range of types. Unlike using `any`, it's also just as precise (ie, it doesn't lose any information) as the first `identity` function that used numbers for the argument and return type.

Once we've written the generic identity function, we can call it in one of two ways. The first way is to pass all of the arguments, including the type argument, to the function:

```
let output = identity<string>("myString");  // type of output will be 'string'
```

Here we explicitly set `T` to be `string` as one of the arguments to the function call, denoted using the `<>` around the arguments rather than `()`.

The second way is also perhaps the most common. Here we use *type argument inference* -- that is, we want the compiler to set the value of `T` for us automatically based on the type of the argument we pass in:

```
let output = identity("myString");  // type of output will be 'string'
```

Notice that we didn't have to explicitly pass the type in the angle brackets (`<>`); the compiler just looked at the value `"myString"`, and set `T` to its type. While type argument inference can be a helpful tool to keep code shorter and more readable, you may need to explicitly pass in the type arguments as we did in the previous example when the compiler fails to infer the type, as may happen in more complex examples.

Working with Generic Type Variables

When you begin to use generics, you'll notice that when you create generic functions like `identity`, the compiler will enforce that you use any generically typed parameters in the body of the function correctly. That is, that you actually treat these parameters as if they could be any and all types.

Let's take our `identity` function from earlier:

```
function identity<T>(arg: T): T {
    return arg;
}
```

What if we want to also log the length of the argument `arg` to the console with each call? We might be tempted to write this:

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
```

When we do, the compiler will give us an error that we're using the `.length` member of `arg`, but nowhere have we said that `arg` has this member. Remember, we said earlier that these type variables stand in for any and all types, so someone using this function could have passed in a `number` instead, which does not have a `.length` member.

Let's say that we've actually intended this function to work on arrays of `T` rather than `T` directly. Since we're working with arrays, the `.length` member should be available. We can describe this just like we would create arrays of other types:

```
function loggingIdentity<T>(arg: T[]): T[] {
    console.log(arg.length);  // Array has a .length, so no more error
    return arg;
}
```

You can read the type of `loggingIdentity` as "the generic function `loggingIdentity` takes a type parameter `T`, and an argument `arg` which is an array of `T`s, and returns an array of `T`s." If we passed in an array of numbers, we'd get an array of numbers back out, as `T` would bind to `number`. This allows us to use our generic type variable `T` as part of the types we're working with, rather than the whole type, giving us greater flexibility.

We can alternatively write the sample example this way:

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {
    console.log(arg.length);  // Array has a .length, so no more error
    return arg;
}
```

You may already be familiar with this style of type from other languages. In the next section, we'll cover how you can create your own generic types like `Array<T>`.

Generic Types

In previous sections, we created generic identity functions that worked over a range of types. In this section, we'll explore the type of the functions themselves and how to create generic interfaces.

The type of generic functions is just like those of non-generic functions, with the type parameters listed first, similarly to function declarations:

```
function identity<T>(arg: T): T {
    return arg;
}
```

```
let myIdentity: <T>(arg: T) => T = identity;
```

We could also have used a different name for the generic type parameter in the type, so long as the number of type variables and how the type variables are used line up.

```
function identity<T>(arg: T): T {
    return arg;
}

let myIdentity: <U>(arg: U) => U = identity;
```

We can also write the generic type as a call signature of an object literal type:

```
function identity<T>(arg: T): T {
    return arg;
}

let myIdentity: {<T>(arg: T): T} = identity;
```

Which leads us to writing our first generic interface. Let's take the object literal from the previous example and move it to an interface:

```
interface GenericIdentityFn {
    <T>(arg: T): T;
}

function identity<T>(arg: T): T {
    return arg;
}

let myIdentity: GenericIdentityFn = identity;
```

In a similar example, we may want to move the generic parameter to be a parameter of the whole interface. This lets us see what type(s) we're generic over (e.g. `Dictionary<string>` rather than just `Dictionary`). This makes the type parameter visible to all the other members of the interface.

```
interface GenericIdentityFn<T> {
    (arg: T): T;
}

function identity<T>(arg: T): T {
    return arg;
}
```

```
let myIdentity: GenericIdentityFn<number> = identity;
```

Notice that our example has changed to be something slightly different. Instead of describing a generic function, we now have a non-generic function signature that is a part of a generic type. When we use `GenericIdentityFn`, we now will also need to specify the corresponding type argument (here: `number`), effectively locking in what the underlying call signature will use. Understanding when to put the type parameter directly on the call signature and when to put it on the interface itself will be helpful in describing what aspects of a type are generic.

In addition to generic interfaces, we can also create generic classes. Note that it is not possible to create generic enums and namespaces.

Generic Classes

A generic class has a similar shape to a generic interface. Generic classes have a generic type parameter list in angle brackets (`<>`) following the name of the class.

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };
```

This is a pretty literal use of the `GenericNumber` class, but you may have noticed that nothing is restricting it to only use the `number` type. We could have instead used `string` or even more complex objects.

```
let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };

console.log(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

Just as with interface, putting the type parameter on the class itself lets us make sure all of the properties of the class are working with the same type.

As we covered in our section on classes, a class has two sides to its type: the static side and the instance side. Generic classes are only generic over their instance side rather than their static side, so when working with classes, static members can not use the class's type parameter.

Generic Constraints

If you remember from an earlier example, you may sometimes want to write a generic function that works on a set of types where you have some knowledge about what capabilities that set of types will have. In our `loggingIdentity` example, we wanted to be able to access the `.length` property of `arg`, but the compiler could not prove that every type had a `.length` property, so it warns us that we can't make this assumption.

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
```

Instead of working with any and all types, we'd like to constrain this function to work with any and all types that also have the `.length` property. As long as the type has this member, we'll allow it, but it's required to have at least this member. To do so, we must list our requirement as a constraint on what T can be.

To do so, we'll create an interface that describes our constraint. Here, we'll create an interface that has a single `.length` property and then we'll use this interface and the `extends` keyword to denote our constraint:

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);  // Now we know it has a .length property, so no more
error
    return arg;
}
```

Because the generic function is now constrained, it will no longer work over any and all types:

```
loggingIdentity(3);  // Error, number doesn't have a .length property
```

Instead, we need to pass in values whose type has all the required properties:

```
loggingIdentity({length: 10, value: 3});
```

Using Type Parameters in Generic Constraints

You can declare a type parameter that is constrained by another type parameter. For example, here we'd like to get a property from an object given its name. We'd like to ensure that we're not accidentally grabbing a property that does not exist on the `obj`, so we'll place a constraint between the two types:

```
function getProperty<T, K extends keyof T>(obj: T, key: K) {
    return obj[key];
}

let x = { a: 1, b: 2, c: 3, d: 4 };

getProperty(x, "a"); // okay
getProperty(x, "m"); // error: Argument of type 'm' isn't assignable to 'a' | 'b'
| 'c' | 'd'.
```

Using Class Types in Generics

When creating factories in TypeScript using generics, it is necessary to refer to class types by their constructor functions. For example,

```
function create<T>(c: {new(): T; }): T {
    return new c();
}
```

A more advanced example uses the prototype property to infer and constrain relationships between the constructor function and the instance side of class types.

```
class BeeKeeper {
    hasMask: boolean;
}

class ZooKeeper {
    nametag: string;
}

class Animal {
    numLegs: number;
}

class Bee extends Animal {
    keeper: BeeKeeper;
}

class Lion extends Animal {
    keeper: ZooKeeper;
}

function createInstance<A extends Animal>(c: new () => A): A {
    return new c();
}

createInstance(Lion).keeper.nametag;  // typechecks!
createInstance(Bee).keeper.hasMask;   // typechecks!
```

07_Enums.md

Enums

Enums allow us to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases. TypeScript provides both numeric and string-based enums.

Numeric enums

We'll first start off with numeric enums, which are probably more familiar if you're coming from other languages. An enum can be defined using the `enum` keyword.

```
enum Direction {
    Up = 1,
    Down,
    Left,
    Right,
}
```

Above, we have a numeric enum where `Up` is initialized with `1`. All of the following members are auto-incremented from that point on. In other words, `Direction.Up` has the value `1`, `Down` has `2`, `Left` has `3`, and `Right` has `4`.

If we wanted, we could leave off the initializers entirely:

```
enum Direction {
    Up,
    Down,
    Left,
    Right,
}
```

Here, `Up` would have the value `0`, `Down` would have `1`, etc. This auto-incrementing behavior is useful for cases where we might not care about the member values themselves, but do care that each value is distinct from other values in the same enum.

Using an enum is simple: just access any member as a property off of the enum itself, and declare types using the name of the enum:

```
enum Response {
    No = 0,
    Yes = 1,
}
```

```
function respond(recipient: string, message: Response): void {
    // ...
}

respond("Princess Caroline", Response.Yes)
```

Numeric enums can be mixed in computed and constant members (see below). The short story is, enums without initializers either need to be first, or have to come after numeric enums initialized with numeric constants or other constant enum members. In other words, the following isn't allowed:

```
enum E {
    A = getSomeValue(),
    B, // error! 'A' is not constant-initialized, so 'B' needs an initializer
}
```

String enums

String enums are a similar concept, but have some subtle runtime differences as documented below. In a string enum, each member has to be constant-initialized with a string literal, or with another string enum member.

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
}
```

While string enums don't have auto-incrementing behavior, string enums have the benefit that they "serialize" well. In other words, if you were debugging and had to read the runtime value of a numeric enum, the value is often opaque - it doesn't convey any useful meaning on its own (though reverse mapping can often help), string enums allow you to give a meaningful and readable value when your code runs, independent of the name of the enum member itself.

Heterogeneous enums

Technically enums can be mixed with string and numeric members, but it's not clear why you would ever want to do so:

```
enum BooleanLikeHeterogeneousEnum {
    No = 0,
    Yes = "YES",
}
```

Unless you're really trying to take advantage of JavaScript's runtime behavior in a clever way, it's advised that you don't do this.

Computed and constant members

Each enum member has a value associated with it which can be either *constant* or *computed*. An enum member is considered constant if:

- It is the first member in the enum and it has no initializer, in which case it's assigned the value `0`:

  ```
  // E.X is constant:
  enum E { X }
  ```

- It does not have an initializer and the preceding enum member was a *numeric* constant. In this case the value of the current enum member will be the value of the preceding enum member plus one.

  ```
  // All enum members in 'E1' and 'E2' are constant.

  enum E1 { X, Y, Z }

  enum E2 {
      A = 1, B, C
  }
  ```

- The enum member is initialized with a constant enum expression. A constant enum expression is a subset of TypeScript expressions that can be fully evaluated at compile time. An expression is a constant enum expression if it is:

  1. a literal enum expression (basically a string literal or a numeric literal)
  2. a reference to previously defined constant enum member (which can originate from a different enum).
  3. a parenthesized constant enum expression
  4. one of the `+`, `-`, `~` unary operators applied to constant enum expression
  5. `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `>>>`, `&`, `|`, `^` binary operators with constant enum expressions as operands It is a compile time error for constant enum expressions to be evaluated to `NaN` or `Infinity`.

In all other cases enum member is considered computed.

```
enum FileAccess {
    // constant members
    None,
    Read    = 1 << 1,
    Write   = 1 << 2,
    ReadWrite  = Read | Write,
    // computed member
    G = "123".length
}
```

Union enums and enum member types

There is a special subset of constant enum members that aren't calculated: literal enum members. A literal enum member is a constant enum member with no initialized value, or with values that are initialized to

- any string literal (e.g. `"foo"`, `"bar`, `"baz"`)
- any numeric literal (e.g. `1`, `100`)
- a unary minus applied to any numeric literal (e.g. `-1`, `-100`)

When all members in an enum have literal enum values, some special semantics come to play.

The first is that enum members also become types as well! For example, we can say that certain members can *only* have the value of an enum member:

```
enum ShapeKind {
    Circle,
    Square,
}

interface Circle {
    kind: ShapeKind.Circle;
    radius: number;
}

interface Square {
    kind: ShapeKind.Square;
    sideLength: number;
}

let c: Circle = {
    kind: ShapeKind.Square,
    //    ~~~~~~~~~~~~~~~~ Error!
    radius: 100,
}
```

The other change is that enum types themselves effectively become a *union* of each enum member. While we haven't discussed [union types](./Advanced Types.md#union-types) yet, all that you need to know is that with union enums, the type system is able to leverage the fact that it knows the exact set of values that exist in the enum itself. Because of that, TypeScript can catch silly bugs where we might be comparing values incorrectly. For example:

```
enum E {
    Foo,
    Bar,
}

function f(x: E) {
```

```
    if (x !== E.Foo || x !== E.Bar) {
        //             ~~~~~~~~~~~
        // Error! Operator '!==' cannot be applied to types 'E.Foo' and 'E.Bar'.
    }
}
```

In that example, we first checked whether `x` was *not* `E.Foo`. If that check succeeds, then our `||` will short-circuit, and the body of the 'if' will get run. However, if the check didn't succeed, then `x` can *only* be `E.Foo`, so it doesn't make sense to see whether it's equal to `E.Bar`.

Enums at runtime

Enums are real objects that exist at runtime. For example, the following enum

```
enum E {
    X, Y, Z
}
```

can actually be passed around to functions

```
function f(obj: { X: number }) {
    return obj.X;
}

// Works, since 'E' has a property named 'X' which is a number.
f(E);
```

Reverse mappings

In addition to creating an object with property names for members, numeric enums members also get a *reverse mapping* from enum values to enum names. For example, in this example:

```
enum Enum {
    A
}
let a = Enum.A;
let nameOfA = Enum[a]; // "A"
```

TypeScript might compile this down to something like the the following JavaScript:

```
var Enum;
(function (Enum) {
    Enum[Enum["A"] = 0] = "A";
})(Enum || (Enum = {}));
```

```
var a = Enum.A;
var nameOfA = Enum[a]; // "A"
```

In this generated code, an enum is compiled into an object that stores both forward (`name` -> `value`) and reverse (`value` -> `name`) mappings. References to other enum members are always emitted as property accesses and never inlined.

Keep in mind that string enum members *do not* get a reverse mapping generated at all.

`const` enums

In most cases, enums are a perfectly valid solution. However sometimes requirements are tighter. To avoid paying the cost of extra generated code and additional indirection when accessing enum values, it's possible to use `const` enums. Const enums are defined using the `const` modifier on our enums:

```
const enum Enum {
    A = 1,
    B = A * 2
}
```

Const enums can only use constant enum expressions and unlike regular enums they are completely removed during compilation. Const enum members are inlined at use sites. This is possible since const enums cannot have computed members.

```
const enum Directions {
    Up,
    Down,
    Left,
    Right
}

let directions = [Directions.Up, Directions.Down, Directions.Left,
Directions.Right]
```

in generated code will become

```
var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* Right */];
```

Ambient enums

Ambient enums are used to describe the shape of already existing enum types.

```
declare enum Enum {
    A = 1,
    B,
    C = 2
}
```

One important difference between ambient and non-ambient enums is that, in regular enums, members that don't have an initializer will be considered constant if its preceding enum member is considered constant. In contrast, an ambient (and non-const) enum member that does not have initializer is *always* considered computed.

08_Type Inference.md

Introduction

In this section, we will cover type inference in TypeScript. Namely, we'll discuss where and how types are inferred.

Basics

In TypeScript, there are several places where type inference is used to provide type information when there is no explicit type annotation. For example, in this code

```
let x = 3;
```

The type of the `x` variable is inferred to be `number`. This kind of inference takes place when initializing variables and members, setting parameter default values, and determining function return types.

In most cases, type inference is straightforward. In the following sections, we'll explore some of the nuances in how types are inferred.

Best common type

When a type inference is made from several expressions, the types of those expressions are used to calculate a "best common type". For example,

```
let x = [0, 1, null];
```

To infer the type of `x` in the example above, we must consider the type of each array element. Here we are given two choices for the type of the array: `number` and `null`. The best common type algorithm considers each candidate type, and picks the type that is compatible with all the other candidates.

Because the best common type has to be chosen from the provided candidate types, there are some cases where types share a common structure, but no one type is the super type of all candidate types. For example:

```
let zoo = [new Rhino(), new Elephant(), new Snake()];
```

Ideally, we may want `zoo` to be inferred as an `Animal[]`, but because there is no object that is strictly of type `Animal` in the array, we make no inference about the array element type. To correct this, instead explicitly provide the type when no one type is a super type of all other candidates:

```
let zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
```

When no best common type is found, the resulting inference is the union array type, `(Rhino | Elephant | Snake)[]`.

Contextual Type

Type inference also works in "the other direction" in some cases in TypeScript. This is known as "contextual typing". Contextual typing occurs when the type of an expression is implied by its location. For example:

```
window.onmousedown = function(mouseEvent) {
    console.log(mouseEvent.clickTime);  //<- Error
};
```

For the code above to give the type error, the TypeScript type checker used the type of the `Window.onmousedown` function to infer the type of the function expression on the right hand side of the assignment. When it did so, it was able to infer the type of the `mouseEvent` parameter. If this function expression were not in a contextually typed position, the `mouseEvent` parameter would have type `any`, and no error would have been issued.

If the contextually typed expression contains explicit type information, the contextual type is ignored. Had we written the above example:

```
window.onmousedown = function(mouseEvent: any) {
    console.log(mouseEvent.clickTime);  //<- Now, no error is given
};
```

The function expression with an explicit type annotation on the parameter will override the contextual type. Once it does so, no error is given as no contextual type applies.

Contextual typing applies in many cases. Common cases include arguments to function calls, right hand sides of assignments, type assertions, members of object and array literals, and return statements. The contextual type also acts as a candidate type in best common type. For example:

```
function createZoo(): Animal[] {
    return [new Rhino(), new Elephant(), new Snake()];
}
```

In this example, best common type has a set of four candidates: `Animal`, `Rhino`, `Elephant`, and `Snake`. Of these, `Animal` can be chosen by the best common type algorithm.

09_Type Compatibility.md

Introduction

Type compatibility in TypeScript is based on structural subtyping. Structural typing is a way of relating types based solely on their members. This is in contrast with nominal typing. Consider the following code:

```
interface Named {
    name: string;
}

class Person {
    name: string;
}

let p: Named;
// OK, because of structural typing
p = new Person();
```

In nominally-typed languages like C# or Java, the equivalent code would be an error because the `Person` class does not explicitly describe itself as being an implementer of the `Named` interface.

TypeScript's structural type system was designed based on how JavaScript code is typically written. Because JavaScript widely uses anonymous objects like function expressions and object literals, it's much more natural to represent the kinds of relationships found in JavaScript libraries with a structural type system instead of a nominal one.

A Note on Soundness

TypeScript's type system allows certain operations that can't be known at compile-time to be safe. When a type system has this property, it is said to not be "sound". The places where TypeScript allows unsound behavior were carefully considered, and throughout this document we'll explain where these happen and the motivating scenarios behind them.

Starting out

The basic rule for TypeScript's structural type system is that `x` is compatible with `y` if `y` has at least the same members as `x`. For example:

```
interface Named {
    name: string;
}

let x: Named;
// y's inferred type is { name: string; location: string; }
let y = { name: "Alice", location: "Seattle" };
x = y;
```

To check whether `y` can be assigned to `x`, the compiler checks each property of `x` to find a corresponding compatible property in `y`. In this case, `y` must have a member called `name` that is a string. It does, so the assignment is allowed.

The same rule for assignment is used when checking function call arguments:

```
function greet(n: Named) {
    console.log("Hello, " + n.name);
}
greet(y); // OK
```

Note that `y` has an extra `location` property, but this does not create an error. Only members of the target type (`Named` in this case) are considered when checking for compatibility.

This comparison process proceeds recursively, exploring the type of each member and sub-member.

Comparing two functions

While comparing primitive types and object types is relatively straightforward, the question of what kinds of functions should be considered compatible is a bit more involved. Let's start with a basic example of two functions that differ only in their parameter lists:

```
let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error
```

To check if `x` is assignable to `y`, we first look at the parameter list. Each parameter in `x` must have a corresponding parameter in `y` with a compatible type. Note that the names of the parameters are not considered, only their types. In this case, every parameter of `x` has a corresponding compatible parameter in `y`, so the assignment is allowed.

The second assignment is an error, because `y` has a required second parameter that `x` does not have, so the assignment is disallowed.

You may be wondering why we allow 'discarding' parameters like in the example `y = x`. The reason for this assignment to be allowed is that ignoring extra function parameters is actually quite common in JavaScript. For example, `Array#forEach` provides three parameters to the callback function: the array element, its index, and the containing array. Nevertheless, it's very useful to provide a callback that only uses the first parameter:

```
let items = [1, 2, 3];

// Don't force these extra parameters
items.forEach((item, index, array) => console.log(item));

// Should be OK!
items.forEach(item => console.log(item));
```

Now let's look at how return types are treated, using two functions that differ only by their return type:

```
let x = () => ({name: "Alice"});
let y = () => ({name: "Alice", location: "Seattle"});

x = y; // OK
y = x; // Error, because x() lacks a location property
```

The type system enforces that the source function's return type be a subtype of the target type's return type.

Function Parameter Bivariance

When comparing the types of function parameters, assignment succeeds if either the source parameter is assignable to the target parameter, or vice versa. This is unsound because a caller might end up being given a function that takes a more specialized type, but invokes the function with a less specialized type. In practice, this sort of error is rare, and allowing this enables many common JavaScript patterns. A brief example:

```
enum EventType { Mouse, Keyboard }

interface Event { timestamp: number; }
interface MouseEvent extends Event { x: number; y: number }
interface KeyEvent extends Event { keyCode: number }

function listenEvent(eventType: EventType, handler: (n: Event) => void) {
    /* ... */
}

// Unsound, but useful and common
listenEvent(EventType.Mouse, (e: MouseEvent) => console.log(e.x + "," + e.y));

// Undesirable alternatives in presence of soundness
```

```
listenEvent(EventType.Mouse, (e: Event) => console.log((<MouseEvent>e).x + "," +
(<MouseEvent>e).y));
listenEvent(EventType.Mouse, <(e: Event) => void>((e: MouseEvent) =>
console.log(e.x + "," + e.y)));

// Still disallowed (clear error). Type safety enforced for wholly incompatible
types
listenEvent(EventType.Mouse, (e: number) => console.log(e));
```

Optional Parameters and Rest Parameters

When comparing functions for compatibility, optional and required parameters are interchangeable. Extra optional parameters of the source type are not an error, and optional parameters of the target type without corresponding parameters in the source type are not an error.

When a function has a rest parameter, it is treated as if it were an infinite series of optional parameters.

This is unsound from a type system perspective, but from a runtime point of view the idea of an optional parameter is generally not well-enforced since passing `undefined` in that position is equivalent for most functions.

The motivating example is the common pattern of a function that takes a callback and invokes it with some predictable (to the programmer) but unknown (to the type system) number of arguments:

```
function invokeLater(args: any[], callback: (...args: any[]) => void) {
    /* ... Invoke callback with 'args' ... */
}

// Unsound - invokeLater "might" provide any number of arguments
invokeLater([1, 2], (x, y) => console.log(x + ", " + y));

// Confusing (x and y are actually required) and undiscoverable
invokeLater([1, 2], (x?, y?) => console.log(x + ", " + y));
```

Functions with overloads

When a function has overloads, each overload in the source type must be matched by a compatible signature on the target type. This ensures that the target function can be called in all the same situations as the source function.

Enums

Enums are compatible with numbers, and numbers are compatible with enums. Enum values from different enum types are considered incompatible. For example,

```
enum Status { Ready, Waiting };
enum Color { Red, Blue, Green };
```

```
let status = Status.Ready;
status = Color.Green;  // Error
```

Classes

Classes work similarly to object literal types and interfaces with one exception: they have both a static and an instance type. When comparing two objects of a class type, only members of the instance are compared. Static members and constructors do not affect compatibility.

```
class Animal {
    feet: number;
    constructor(name: string, numFeet: number) { }
}

class Size {
    feet: number;
    constructor(numFeet: number) { }
}

let a: Animal;
let s: Size;

a = s;  // OK
s = a;  // OK
```

Private and protected members in classes

Private and protected members in a class affect their compatibility. When an instance of a class is checked for compatibility, if the target type contains a private member, then the source type must also contain a private member that originated from the same class. Likewise, the same applies for an instance with a protected member. This allows a class to be assignment compatible with its super class, but *not* with classes from a different inheritance hierarchy which otherwise have the same shape.

Generics

Because TypeScript is a structural type system, type parameters only affect the resulting type when consumed as part of the type of a member. For example,

```
interface Empty<T> {
}
let x: Empty<number>;
let y: Empty<string>;

x = y;  // OK, because y matches structure of x
```

In the above, `x` and `y` are compatible because their structures do not use the type argument in a differentiating way. Changing this example by adding a member to `Empty<T>` shows how this works:

```
interface NotEmpty<T> {
    data: T;
}
let x: NotEmpty<number>;
let y: NotEmpty<string>;

x = y;  // Error, because x and y are not compatible
```

In this way, a generic type that has its type arguments specified acts just like a non-generic type.

For generic types that do not have their type arguments specified, compatibility is checked by specifying `any` in place of all unspecified type arguments. The resulting types are then checked for compatibility, just as in the non-generic case.

For example,

```
let identity = function<T>(x: T): T {
    // ...
}

let reverse = function<U>(y: U): U {
    // ...
}

identity = reverse;  // OK, because (x: any) => any matches (y: any) => any
```

Advanced Topics

Subtype vs Assignment

So far, we've used "compatible", which is not a term defined in the language spec. In TypeScript, there are two kinds of compatibility: subtype and assignment. These differ only in that assignment extends subtype compatibility with rules to allow assignment to and from `any`, and to and from `enum` with corresponding numeric values.

Different places in the language use one of the two compatibility mechanisms, depending on the situation. For practical purposes, type compatibility is dictated by assignment compatibility, even in the cases of the `implements` and `extends` clauses.

For more information, see the TypeScript spec.

10_Advanced Types.md

Intersection Types

An intersection type combines multiple types into one. This allows you to add together existing types to get a single type that has all the features you need. For example, `Person & Serializable & Loggable` is a `Person` *and* `Serializable` *and* `Loggable`. That means an object of this type will have all members of all three types.

You will mostly see intersection types used for mixins and other concepts that don't fit in the classic object-oriented mold. (There are a lot of these in JavaScript!) Here's a simple example that shows how to create a mixin:

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U>{};
    for (let id in first) {
        (<any>result)[id] = (<any>first)[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id];
        }
    }
    return result;
}

class Person {
    constructor(public name: string) { }
}
interface Loggable {
    log(): void;
}
class ConsoleLogger implements Loggable {
    log() {
        // ...
    }
}
var jim = extend(new Person("Jim"), new ConsoleLogger());
var n = jim.name;
jim.log();
```

Union Types

Union types are closely related to intersection types, but they are used very differently. Occasionally, you'll run into a library that expects a parameter to be either a `number` or a `string`. For instance, take the following function:

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: any) {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}

padLeft("Hello world", 4); // returns "    Hello world"
```

The problem with `padLeft` is that its `padding` parameter is typed as `any`. That means that we can call it with an argument that's neither a `number` nor a `string`, but TypeScript will be okay with it.

```
let indentedString = padLeft("Hello world", true); // passes at compile time,
fails at runtime.
```

In traditional object-oriented code, we might abstract over the two types by creating a hierarchy of types. While this is much more explicit, it's also a little bit overkill. One of the nice things about the original version of `padLeft` was that we were able to just pass in primitives. That meant that usage was simple and concise. This new approach also wouldn't help if we were just trying to use a function that already exists elsewhere.

Instead of `any`, we can use a *union type* for the `padding` parameter:

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: string | number) {
    // ...
}

let indentedString = padLeft("Hello world", true); // errors during compilation
```

A union type describes a value that can be one of several types. We use the vertical bar (`|`) to separate each type, so `number | string | boolean` is the type of a value that can be a `number`, a `string`, or a `boolean`.

If we have a value that has a union type, we can only access members that are common to all types in the union.

```
interface Bird {
    fly();
    layEggs();
}

interface Fish {
    swim();
    layEggs();
}

function getSmallPet(): Fish | Bird {
    // ...
}

let pet = getSmallPet();
pet.layEggs(); // okay
pet.swim();    // errors
```

Union types can be a bit tricky here, but it just takes a bit of intuition to get used to. If a value has the type `A | B`, we only know for *certain* that it has members that both `A` *and* `B` have. In this example, `Bird` has a member named `fly`. We can't be sure whether a variable typed as `Bird | Fish` has a `fly` method. If the variable is really a `Fish` at runtime, then calling `pet.fly()` will fail.

Type Guards and Differentiating Types

Union types are useful for modeling situations when values can overlap in the types they can take on. What happens when we need to know specifically whether we have a `Fish`? A common idiom in JavaScript to differentiate between two possible values is to check for the presence of a member. As we mentioned, you can only access members that are guaranteed to be in all the constituents of a union type.

```
let pet = getSmallPet();

// Each of these property accesses will cause an error
if (pet.swim) {
    pet.swim();
}
else if (pet.fly) {
    pet.fly();
}
```

To get the same code working, we'll need to use a type assertion:

```
let pet = getSmallPet();

if ((<Fish>pet).swim) {
    (<Fish>pet).swim();
}
else {
    (<Bird>pet).fly();
}
```

User-Defined Type Guards

Notice that we had to use type assertions several times. It would be much better if once we performed the check, we could know the type of `pet` within each branch.

It just so happens that TypeScript has something called a *type guard*. A type guard is some expression that performs a runtime check that guarantees the type in some scope. To define a type guard, we simply need to define a function whose return type is a *type predicate*:

```
function isFish(pet: Fish | Bird): pet is Fish {
    return (<Fish>pet).swim !== undefined;
}
```

`pet is Fish` is our type predicate in this example. A predicate takes the form `parameterName is Type`, where `parameterName` must be the name of a parameter from the current function signature.

Any time `isFish` is called with some variable, TypeScript will *narrow* that variable to that specific type if the original type is compatible.

```
// Both calls to 'swim' and 'fly' are now okay.

if (isFish(pet)) {
    pet.swim();
}
else {
    pet.fly();
}
```

Notice that TypeScript not only knows that `pet` is a `Fish` in the `if` branch; it also knows that in the `else` branch, you *don't* have a `Fish`, so you must have a `Bird`.

`typeof` type guards

Let's go back and write the code for the version of `padLeft` that uses union types. We could write it with type predicates as follows:

```
function isNumber(x: any): x is number {
    return typeof x === "number";
}

function isString(x: any): x is string {
    return typeof x === "string";
}

function padLeft(value: string, padding: string | number) {
    if (isNumber(padding)) {
        return Array(padding + 1).join(" ") + value;
    }
    if (isString(padding)) {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}
```

However, having to define a function to figure out if a type is a primitive is kind of a pain. Luckily, you don't need to abstract `typeof x === "number"` into its own function because TypeScript will recognize it as a type guard on its own. That means we could just write these checks inline.

```
function padLeft(value: string, padding: string | number) {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}
```

These *`typeof` type guards* are recognized in two different forms: `typeof v === "typename"` and `typeof v !== "typename"`, where `"typename"` must be `"number"`, `"string"`, `"boolean"`, or `"symbol"`. While TypeScript won't stop you from comparing to other strings, the language won't recognize those expressions as type guards.

`instanceof` type guards

If you've read about `typeof` type guards and are familiar with the `instanceof` operator in JavaScript, you probably have some idea of what this section is about.

`instanceof` type guards are a way of narrowing types using their constructor function. For instance, let's borrow our industrial string-padder example from earlier:

```
interface Padder {
    getPaddingString(): string
}

class SpaceRepeatingPadder implements Padder {
    constructor(private numSpaces: number) { }
    getPaddingString() {
        return Array(this.numSpaces + 1).join(" ");
    }
}

class StringPadder implements Padder {
    constructor(private value: string) { }
    getPaddingString() {
        return this.value;
    }
}

function getRandomPadder() {
    return Math.random() < 0.5 ?
        new SpaceRepeatingPadder(4) :
        new StringPadder("  ");
}

// Type is 'SpaceRepeatingPadder | StringPadder'
let padder: Padder = getRandomPadder();

if (padder instanceof SpaceRepeatingPadder) {
    padder; // type narrowed to 'SpaceRepeatingPadder'
}
if (padder instanceof StringPadder) {
    padder; // type narrowed to 'StringPadder'
}
```

The right side of the `instanceof` needs to be a constructor function, and TypeScript will narrow down to:

1. the type of the function's `prototype` property if its type is not `any`
2. the union of types returned by that type's construct signatures

in that order.

Nullable types

TypeScript has two special types, `null` and `undefined`, that have the values null and undefined respectively. We mentioned these briefly in [the Basic Types section](./Basic Types.md). By default, the type checker considers `null` and `undefined` assignable to anything. Effectively, `null` and `undefined` are valid values of every type. That means it's not possible to *stop* them from being assigned to any type, even when you would like to prevent it. The inventor of `null`, Tony Hoare, calls this his "billion dollar mistake".

The `--strictNullChecks` flag fixes this: when you declare a variable, it doesn't automatically include `null` or `undefined`. You can include them explicitly using a union type:

```
let s = "foo";
s = null; // error, 'null' is not assignable to 'string'
let sn: string | null = "bar";
sn = null; // ok

sn = undefined; // error, 'undefined' is not assignable to 'string | null'
```

Note that TypeScript treats `null` and `undefined` differently in order to match JavaScript semantics. `string | null` is a different type than `string | undefined` and `string | undefined | null`.

Optional parameters and properties

With `--strictNullChecks`, an optional parameter automatically adds `| undefined`:

```
function f(x: number, y?: number) {
    return x + (y || 0);
}
f(1, 2);
f(1);
f(1, undefined);
f(1, null); // error, 'null' is not assignable to 'number | undefined'
```

The same is true for optional properties:

```
class C {
    a: number;
    b?: number;
}
let c = new C();
c.a = 12;
c.a = undefined; // error, 'undefined' is not assignable to 'number'
c.b = 13;
c.b = undefined; // ok
c.b = null; // error, 'null' is not assignable to 'number | undefined'
```

Type guards and type assertions

Since nullable types are implemented with a union, you need to use a type guard to get rid of the `null`. Fortunately, this is the same code you'd write in JavaScript:

```
function f(sn: string | null): string {
    if (sn == null) {
        return "default";
    }
    else {
        return sn;
    }
}
```

The `null` elimination is pretty obvious here, but you can use terser operators too:

```
function f(sn: string | null): string {
    return sn || "default";
}
```

In cases where the compiler can't eliminate `null` or `undefined`, you can use the type assertion operator to manually remove them. The syntax is postfix `!`: `identifier!` removes `null` and `undefined` from the type of `identifier`:

```
function broken(name: string | null): string {
  function postfix(epithet: string) {
    return name.charAt(0) + '.  the ' + epithet; // error, 'name' is possibly null
  }
  name = name || "Bob";
  return postfix("great");
}

function fixed(name: string | null): string {
  function postfix(epithet: string) {
    return name!.charAt(0) + '.  the ' + epithet; // ok
  }
  name = name || "Bob";
  return postfix("great");
}
```

The example uses a nested function here because the compiler can't eliminate nulls inside a nested function (except immediately-invoked function expressions). That's because it can't track all calls to the nested function, especially if you return it from the outer function. Without knowing where the function is called, it can't know what the type of `name` will be at the time the body executes.

Type Aliases

Type aliases create a new name for a type. Type aliases are sometimes similar to interfaces, but can name primitives, unions, tuples, and any other types that you'd otherwise have to write by hand.

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
    if (typeof n === "string") {
        return n;
    }
    else {
        return n();
    }
}
```

Aliasing doesn't actually create a new type - it creates a new *name* to refer to that type. Aliasing a primitive is not terribly useful, though it can be used as a form of documentation.

Just like interfaces, type aliases can also be generic - we can just add type parameters and use them on the right side of the alias declaration:

```
type Container<T> = { value: T };
```

We can also have a type alias refer to itself in a property:

```
type Tree<T> = {
    value: T;
    left: Tree<T>;
    right: Tree<T>;
}
```

Together with intersection types, we can make some pretty mind-bending types:

```
type LinkedList<T> = T & { next: LinkedList<T> };

interface Person {
    name: string;
}

var people: LinkedList<Person>;
var s = people.name;
var s = people.next.name;
var s = people.next.next.name;
var s = people.next.next.next.name;
```

However, it's not possible for a type alias to appear anywhere else on the right side of the declaration:

```
type Yikes = Array<Yikes>; // error
```

Interfaces vs. Type Aliases

As we mentioned, type aliases can act sort of like interfaces; however, there are some subtle differences.

One difference is that interfaces create a new name that is used everywhere. Type aliases don't create a new name — for instance, error messages won't use the alias name. In the code below, hovering over `interfaced` in an editor will show that it returns an `Interface`, but will show that `aliased` returns object literal type.

```
type Alias = { num: number }
interface Interface {
    num: number;
}
declare function aliased(arg: Alias): Alias;
declare function interfaced(arg: Interface): Interface;
```

A second more important difference is that type aliases cannot be extended or implemented from (nor can they extend/implement other types). Because an ideal property of software is being open to extension, you should always use an interface over a type alias if possible.

On the other hand, if you can't express some shape with an interface and you need to use a union or tuple type, type aliases are usually the way to go.

String Literal Types

String literal types allow you to specify the exact value a string must have. In practice string literal types combine nicely with union types, type guards, and type aliases. You can use these features together to get enum-like behavior with strings.

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
class UIElement {
    animate(dx: number, dy: number, easing: Easing) {
        if (easing === "ease-in") {
            // ...
        }
        else if (easing === "ease-out") {
        }
        else if (easing === "ease-in-out") {
        }
        else {
            // error! should not pass null or undefined.
        }
    }
}

let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy"); // error: "uneasy" is not allowed here
```

You can pass any of the three allowed strings, but any other string will give the error

```
Argument of type '"uneasy"' is not assignable to parameter of type '"ease-in" |
"ease-out" | "ease-in-out"'
```

String literal types can be used in the same way to distinguish overloads:

```
function createElement(tagName: "img"): HTMLImageElement;
function createElement(tagName: "input"): HTMLInputElement;
// ... more overloads ...
function createElement(tagName: string): Element {
    // ... code goes here ...
}
```

Numeric Literal Types

TypeScript also has numeric literal types.

```
function rollDie(): 1 | 2 | 3 | 4 | 5 | 6 {
    // ...
}
```

These are seldom written explicitly, they can be useful when narrowing can catch bugs:

```
function foo(x: number) {
    if (x !== 1 || x !== 2) {
        //         ~~~~~~~
        // Operator '!==' cannot be applied to types '1' and '2'.
    }
}
```

In other words, `x` must be `1` when it gets compared to `2`, meaning that the above check is making an invalid comparison.

Enum Member Types

As mentioned in our section on enums, enum members have types when every member is literal-initialized.

Much of the time when we talk about "singleton types", we're referring to both enum member types as well as numeric/string literal types, though many users will use "singleton types" and "literal types" interchangeably.

Discriminated Unions

You can combine singleton types, union types, type guards, and type aliases to build an advanced pattern called *discriminated unions*, also known as *tagged unions* or *algebraic data types*. Discriminated unions are useful in functional programming. Some languages automatically discriminate unions for you; TypeScript instead builds on JavaScript patterns as they exist today. There are three ingredients:

1. Types that have a common, singleton type property — the *discriminant*.
2. A type alias that takes the union of those types — the *union*.
3. Type guards on the common property.

```
interface Square {
    kind: "square";
    size: number;
}
interface Rectangle {
    kind: "rectangle";
    width: number;
    height: number;
}
interface Circle {
    kind: "circle";
    radius: number;
}
```

First we declare the interfaces we will union. Each interface has a `kind` property with a different string literal type. The `kind` property is called the *discriminant* or *tag*. The other properties are specific to each interface. Notice that the interfaces are currently unrelated. Let's put them into a union:

```
type Shape = Square | Rectangle | Circle;
```

Now let's use the discriminated union:

```
function area(s: Shape) {
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
    }
}
```

Exhaustiveness checking

We would like the compiler to tell us when we don't cover all variants of the discriminated union. For example, if we add `Triangle` to `Shape`, we need to update `area` as well:

```
type Shape = Square | Rectangle | Circle | Triangle;
function area(s: Shape) {
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
    }
    // should error here - we didn't handle case "triangle"
}
```

There are two ways to do this. The first is to turn on `--strictNullChecks` and specify a return type:

```
function area(s: Shape): number { // error: returns number | undefined
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
    }
}
```

Because the `switch` is no longer exhaustive, TypeScript is aware that the function could sometimes return `undefined`. If you have an explicit return type `number`, then you will get an error that the return type is actually `number | undefined`. However, this method is quite subtle and, besides, `--strictNullChecks` does not always work with old code.

The second method uses the `never` type that the compiler uses to check for exhaustiveness:

```
function assertNever(x: never): never {
    throw new Error("Unexpected object: " + x);
}
function area(s: Shape) {
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
        default: return assertNever(s); // error here if there are missing cases
    }
}
```

Here, `assertNever` checks that `s` is of type `never` — the type that's left after all other cases have been removed. If you forget a case, then `s` will have a real type and you will get a type error. This method requires you to define an extra function, but it's much more obvious when you forget it.

Polymorphic `this` types

A polymorphic `this` type represents a type that is the *subtype* of the containing class or interface. This is called *F*-bounded polymorphism. This makes hierarchical fluent interfaces much easier to express, for example. Take a simple calculator that returns `this` after each operation:

```
class BasicCalculator {
    public constructor(protected value: number = 0) { }
    public currentValue(): number {
        return this.value;
    }
    public add(operand: number): this {
        this.value += operand;
        return this;
    }
    public multiply(operand: number): this {
        this.value *= operand;
        return this;
    }
    // ... other operations go here ...
}

let v = new BasicCalculator(2)
            .multiply(5)
            .add(1)
            .currentValue();
```

Since the class uses `this` types, you can extend it and the new class can use the old methods with no changes.

```
class ScientificCalculator extends BasicCalculator {
    public constructor(value = 0) {
        super(value);
    }
    public sin() {
        this.value = Math.sin(this.value);
        return this;
    }
    // ... other operations go here ...
}

let v = new ScientificCalculator(2)
        .multiply(5)
        .sin()
        .add(1)
        .currentValue();
```

Without `this` types, `ScientificCalculator` would not have been able to extend `BasicCalculator` and keep the fluent interface. `multiply` would have returned `BasicCalculator`, which doesn't have the `sin` method. However, with `this` types, `multiply` returns `this`, which is `ScientificCalculator` here.

Index types

With index types, you can get the compiler to check code that uses dynamic property names. For example, a common Javascript pattern is to pick a subset of properties from an object:

```
function pluck(o, names) {
    return names.map(n => o[n]);
}
```

Here's how you would write and use this function in TypeScript, using the **index type query** and **indexed access** operators:

```
function pluck<T, K extends keyof T>(o: T, names: K[]): T[K][] {
  return names.map(n => o[n]);
}

interface Person {
    name: string;
    age: number;
}
let person: Person = {
    name: 'Jarid',
    age: 35
};
let strings: string[] = pluck(person, ['name']); // ok, string[]
```

The compiler checks that `name` is actually a property on `Person`. The example introduces a couple of new type operators. First is `keyof T`, the **index type query operator**. For any type `T`, `keyof T` is the union of known, public property names of `T`. For example:

```
let personProps: keyof Person; // 'name' | 'age'
```

`keyof Person` is completely interchangeable with `'name' | 'age'`. The difference is that if you add another property to `Person`, say `address: string`, then `keyof Person` will automatically update to be `'name' | 'age' | 'address'`. And you can use `keyof` in generic contexts like `pluck`, where you can't possibly know the property names ahead of time. That means the compiler will check that you pass the right set of property names to `pluck`:

```
pluck(person, ['age', 'unknown']); // error, 'unknown' is not in 'name' | 'age'
```

The second operator is `T[K]`, the **indexed access operator**. Here, the type syntax reflects the expression syntax. That means that `person['name']` has the type `Person['name']` — which in our example is just `string`. However, just like index type queries, you can use `T[K]` in a generic context, which is where its real power comes to life. You just have to make sure that the type variable `K extends keyof T`. Here's another example with a function named `getProperty`.

```
function getProperty<T, K extends keyof T>(o: T, name: K): T[K] {
    return o[name]; // o[name] is of type T[K]
}
```

In `getProperty`, `o: T` and `name: K`, so that means `o[name]: T[K]`. Once you return the `T[K]` result, the compiler will instantiate the actual type of the key, so the return type of `getProperty` will vary according to which property you request.

```
let name: string = getProperty(person, 'name');
let age: number = getProperty(person, 'age');
let unknown = getProperty(person, 'unknown'); // error, 'unknown' is not in 'name'
| 'age'
```

Index types and string index signatures

`keyof` and `T[K]` interact with string index signatures. If you have a type with a string index signature, `keyof T` will just be `string`. And `T[string]` is just the type of the index signature:

```
interface Map<T> {
    [key: string]: T;
}
let keys: keyof Map<number>; // string
let value: Map<number>['foo']; // number
```

Mapped types

A common task is to take an existing type and make each of its properties optional:

```
interface PersonPartial {
    name?: string;
    age?: number;
}
```

Or we might want a readonly version:

```
interface PersonReadonly {
    readonly name: string;
    readonly age: number;
}
```

This happens often enough in Javascript that TypeScript provides a way to create new types based on old types — **mapped types**. In a mapped type, the new type transforms each property in the old type in the same way. For example, you can make all properties of a type `readonly` or optional. Here are a couple of examples:

```
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
}
type Partial<T> = {
    [P in keyof T]?: T[P];
}
```

And to use it:

```
type PersonPartial = Partial<Person>;
type ReadonlyPerson = Readonly<Person>;
```

Let's take a look at the simplest mapped type and its parts:

```
type Keys = 'option1' | 'option2';
type Flags = { [K in Keys]: boolean };
```

The syntax resembles the syntax for index signatures with a `for .. in` inside. There are three parts:

1. The type variable `K`, which gets bound to each property in turn.
2. The string literal union `Keys`, which contains the names of properties to iterate over.
3. The resulting type of the property.

In this simple example, `Keys` is a hard-coded list of property names and the property type is always `boolean`, so this mapped type is equivalent to writing:

```
type Flags = {
    option1: boolean;
    option2: boolean;
}
```

Real applications, however, look like `Readonly` or `Partial` above. They're based on some existing type, and they transform the properties in some way. That's where `keyof` and indexed access types come in:

```
type NullablePerson = { [P in keyof Person]: Person[P] | null }
type PartialPerson = { [P in keyof Person]?: Person[P] }
```

But it's more useful to have a general version.

```
type Nullable<T> = { [P in keyof T]: T[P] | null }
type Partial<T> = { [P in keyof T]?: T[P] }
```

In these examples, the properties list is `keyof T` and the resulting type is some variant of `T[P]`. This is a good template for any general use of mapped types. That's because this kind of transformation is homomorphic, which means that the mapping applies only to properties of `T` and no others. The compiler knows that it can copy all the existing property modifiers before adding any new ones. For example, if `Person.name` was readonly, `Partial<Person>.name` would be readonly and optional.

Here's one more example, in which `T[P]` is wrapped in a `Proxy<T>` class:

```
type Proxy<T> = {
    get(): T;
    set(value: T): void;
}
type Proxify<T> = {
    [P in keyof T]: Proxy<T[P]>;
}
function proxify<T>(o: T): Proxify<T> {
   // ... wrap proxies ...
}
let proxyProps = proxify(props);
```

Note that `Readonly<T>` and `Partial<T>` are so useful, they are included in TypeScript's standard library along with `Pick` and `Record`:

```
type Pick<T, K extends keyof T> = {
    [P in K]: T[P];
}
type Record<K extends string, T> = {
    [P in K]: T;
}
```

`Readonly`, `Partial` and `Pick` are homomorphic whereas `Record` is not. One clue that `Record` is not homomorphic is that it doesn't take an input type to copy properties from:

```
type ThreeStringProps = Record<'prop1' | 'prop2' | 'prop3', string>
```

Non-homomorphic types are essentially creating new properties, so they can't copy property modifiers from anywhere.

Inference from mapped types

Now that you know how to wrap the properties of a type, the next thing you'll want to do is unwrap them. Fortunately, that's pretty easy:

```
function unproxify<T>(t: Proxify<T>): T {
    let result = {} as T;
    for (const k in t) {
        result[k] = t[k].get();
    }
    return result;
}

let originalProps = unproxify(proxyProps);
```

Note that this unwrapping inference only works on homomorphic mapped types. If the mapped type is not homomorphic you'll have to give an explicit type parameter to your unwrapping function.

Conditional Types

TypeScript 2.8 introduces *conditional types* which add the ability to express non-uniform type mappings. A conditional type selects one of two possible types based on a condition expressed as a type relationship test:

```
T extends U ? X : Y
```

The type above means when `T` is assignable to `U` the type is `X`, otherwise the type is `Y`.

A conditional type `T extends U ? X : Y` is either *resolved* to `X` or `Y`, or *deferred* because the condition depends on one or more type variables. When `T` or `U` contains type variables, whether to resolve to `X` or `Y`, or to defer, is determined by whether or not a the type system has enough information to conclude that `T` is always assignable to `U`.

As an example of some types that are immediately resolved, we can take a look at the following example:

```
declare function f<T extends boolean>(x: T): T extends true ? string : number;

// Type is 'string | number
let x = f(Math.random() < 0.5)
```

Another example would be the `TypeName` type alias, which uses nested conditional types:

```
type TypeName<T> =
    T extends string ? "string" :
    T extends number ? "number" :
    T extends boolean ? "boolean" :
```

```
    T extends undefined ? "undefined" :
    T extends Function ? "function" :
    "object";

type T0 = TypeName<string>;  // "string"
type T1 = TypeName<"a">;  // "string"
type T2 = TypeName<true>;  // "boolean"
type T3 = TypeName<() => void>;  // "function"
type T4 = TypeName<string[]>;  // "object"
```

But as an example of a place where conditonal types are deferred - where they stick around instead of picking a branch - would be in the following:

```
interface Foo {
    propA: boolean;
    propB: boolean;
}

declare function f<T>(x: T): T extends Foo ? string : number;

function foo<U>(x: U) {
    // Has type 'U extends Foo ? string : number'
    let a = f(x);

    // This assignment is allowed though!
    let b: string | number = a;
}
```

In the above, the variable `a` has a conditional type that hasn't yet chosen a branch. When another piece of code ends up calling `foo`, it will substitute in `U` with some other type, and TypeScript will re-evaluate the conditional type, deciding whether it can actually pick a branch.

In the meantime, we can assign a conditional type to any other target type as long as each branch of the conditional is assignable to that target. So in our example above we were able to assign `U extends Foo ? string : number` to `string | number` since no matter what the conditional evaluates to, it's known to be either `string` or `number`.

Distributive conditional types

Conditional types in which the checked type is a naked type parameter are called *distributive conditional types*. Distributive conditional types are automatically distributed over union types during instantiation. For example, an instantiation of `T extends U ? X : Y` with the type argument `A | B | C` for `T` is resolved as `(A extends U ? X : Y) | (B extends U ? X : Y) | (C extends U ? X : Y)`.

Example

```
type T10 = TypeName<string | (() => void)>;  // "string" | "function"
type T12 = TypeName<string | string[] | undefined>;  // "string" | "object" |
"undefined"
type T11 = TypeName<string[] | number[]>;  // "object"
```

In instantiations of a distributive conditional type `T extends U ? X : Y`, references to `T` within the conditional type are resolved to individual constituents of the union type (i.e. `T` refers to the individual constituents *after* the conditional type is distributed over the union type). Furthermore, references to `T` within `X` have an additional type parameter constraint `U` (i.e. `T` is considered assignable to `U` within `X`).

Example

```
type BoxedValue<T> = { value: T };
type BoxedArray<T> = { array: T[] };
type Boxed<T> = T extends any[] ? BoxedArray<T[number]> : BoxedValue<T>;

type T20 = Boxed<string>;  // BoxedValue<string>;
type T21 = Boxed<number[]>;  // BoxedArray<number>;
type T22 = Boxed<string | number[]>;  // BoxedValue<string> | BoxedArray<number>;
```

Notice that `T` has the additional constraint `any[]` within the true branch of `Boxed<T>` and it is therefore possible to refer to the element type of the array as `T[number]`. Also, notice how the conditional type is distributed over the union type in the last example.

The distributive property of conditional types can conveniently be used to *filter* union types:

```
type Diff<T, U> = T extends U ? never : T;  // Remove types from T that are
assignable to U
type Filter<T, U> = T extends U ? T : never;  // Remove types from T that are not
assignable to U

type T30 = Diff<"a" | "b" | "c" | "d", "a" | "c" | "f">;  // "b" | "d"
type T31 = Filter<"a" | "b" | "c" | "d", "a" | "c" | "f">;  // "a" | "c"
type T32 = Diff<string | number | (() => void), Function>;  // string | number
type T33 = Filter<string | number | (() => void), Function>;  // () => void

type NonNullable<T> = Diff<T, null | undefined>;  // Remove null and undefined
from T

type T34 = NonNullable<string | number | undefined>;  // string | number
type T35 = NonNullable<string | string[] | null | undefined>;  // string |
string[]

function f1<T>(x: T, y: NonNullable<T>) {
    x = y;  // Ok
    y = x;  // Error
}
```

```
function f2<T extends string | undefined>(x: T, y: NonNullable<T>) {
    x = y;  // Ok
    y = x;  // Error
    let s1: string = x;  // Error
    let s2: string = y;  // Ok
}
```

Conditional types are particularly useful when combined with mapped types:

```
type FunctionPropertyNames<T> = { [K in keyof T]: T[K] extends Function ? K :
never }[keyof T];
type FunctionProperties<T> = Pick<T, FunctionPropertyNames<T>>;

type NonFunctionPropertyNames<T> = { [K in keyof T]: T[K] extends Function ? never
: K }[keyof T];
type NonFunctionProperties<T> = Pick<T, NonFunctionPropertyNames<T>>;

interface Part {
    id: number;
    name: string;
    subparts: Part[];
    updatePart(newName: string): void;
}

type T40 = FunctionPropertyNames<Part>;  // "updatePart"
type T41 = NonFunctionPropertyNames<Part>;  // "id" | "name" | "subparts"
type T42 = FunctionProperties<Part>;  // { updatePart(newName: string): void }
type T43 = NonFunctionProperties<Part>;  // { id: number, name: string, subparts:
Part[] }
```

Similar to union and intersection types, conditional types are not permitted to reference themselves recursively. For example the following is an error.

Example

```
type ElementType<T> = T extends any[] ? ElementType<T[number]> : T;  // Error
```

Type inference in conditional types

Within the `extends` clause of a conditional type, it is now possible to have `infer` declarations that introduce a type variable to be inferred. Such inferred type variables may be referenced in the true branch of the conditional type. It is possible to have multiple `infer` locations for the same type variable.

For example, the following extracts the return type of a function type:

```
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;
```

Conditional types can be nested to form a sequence of pattern matches that are evaluated in order:

```
type Unpacked<T> =
    T extends (infer U)[] ? U :
    T extends (...args: any[]) => infer U ? U :
    T extends Promise<infer U> ? U :
    T;

type T0 = Unpacked<string>;  // string
type T1 = Unpacked<string[]>;  // string
type T2 = Unpacked<() => string>;  // string
type T3 = Unpacked<Promise<string>>;  // string
type T4 = Unpacked<Promise<string>[]>;  // Promise<string>
type T5 = Unpacked<Unpacked<Promise<string>[]>>;  // string
```

The following example demonstrates how multiple candidates for the same type variable in co-variant positions causes a union type to be inferred:

```
type Foo<T> = T extends { a: infer U, b: infer U } ? U : never;
type T10 = Foo<{ a: string, b: string }>;  // string
type T11 = Foo<{ a: string, b: number }>;  // string | number
```

Likewise, multiple candidates for the same type variable in contra-variant positions causes an intersection type to be inferred:

```
type Bar<T> = T extends { a: (x: infer U) => void, b: (x: infer U) => void } ? U : never;
type T20 = Bar<{ a: (x: string) => void, b: (x: string) => void }>;  // string
type T21 = Bar<{ a: (x: string) => void, b: (x: number) => void }>;  // string & number
```

When inferring from a type with multiple call signatures (such as the type of an overloaded function), inferences are made from the *last* signature (which, presumably, is the most permissive catch-all case). It is not possible to perform overload resolution based on a list of argument types.

```
declare function foo(x: string): number;
declare function foo(x: number): string;
declare function foo(x: string | number): string | number;
type T30 = ReturnType<typeof foo>;  // string | number
```

It is not possible to use `infer` declarations in constraint clauses for regular type parameters:

```
type ReturnType<T extends (...args: any[]) => infer R> = R;  // Error, not supported
```

However, much the same effect can be obtained by erasing the type variables in the constraint and instead specifying a conditional type:

```
type AnyFunction = (...args: any[]) => any;
type ReturnType<T extends AnyFunction> = T extends (...args: any[]) => infer R ? R : any;
```

Predefined conditional types

TypeScript 2.8 adds several predefined conditional types to `lib.d.ts`:

- `Exclude<T, U>` -- Exclude from `T` those types that are assignable to `U`.
- `Extract<T, U>` -- Extract from `T` those types that are assignable to `U`.
- `NonNullable<T>` -- Exclude `null` and `undefined` from `T`.
- `ReturnType<T>` -- Obtain the return type of a function type.
- `InstanceType<T>` -- Obtain the instance type of a constructor function type.

Example

```
type T00 = Exclude<"a" | "b" | "c" | "d", "a" | "c" | "f">;  // "b" | "d"
type T01 = Extract<"a" | "b" | "c" | "d", "a" | "c" | "f">;  // "a" | "c"

type T02 = Exclude<string | number | (() => void), Function>;  // string | number
type T03 = Extract<string | number | (() => void), Function>;  // () => void

type T04 = NonNullable<string | number | undefined>;  // string | number
type T05 = NonNullable<(() => string) | string[] | null | undefined>;  // (() => string) | string[]

function f1(s: string) {
    return { a: 1, b: s };
}

class C {
    x = 0;
    y = 0;
}

type T10 = ReturnType<() => string>;  // string
type T11 = ReturnType<(s: string) => void>;  // void
type T12 = ReturnType<(<T>() => T)>;  // {}
type T13 = ReturnType<(<T extends U, U extends number[]>() => T)>;  // number[]
type T14 = ReturnType<typeof f1>;  // { a: number, b: string }
type T15 = ReturnType<any>;  // any
type T16 = ReturnType<never>;  // any
type T17 = ReturnType<string>;  // Error
type T18 = ReturnType<Function>;  // Error

type T20 = InstanceType<typeof C>;  // C
type T21 = InstanceType<any>;  // any
type T22 = InstanceType<never>;  // any
type T23 = InstanceType<string>;  // Error
type T24 = InstanceType<Function>;  // Error
```

> Note: The `Exclude` type is a proper implementation of the `Diff` type suggested here. We've used the name `Exclude` to avoid breaking existing code that defines a `Diff`, plus we feel that name better conveys the semantics of the type. We did not include the `Omit<T, K>` type because it is trivially written as `Pick<T, Exclude<keyof T, K>>`.

11_Symbols.md

Introduction

Starting with ECMAScript 2015, `symbol` is a primitive data type, just like `number` and `string`.

`symbol` values are created by calling the `Symbol` constructor.

```
let sym1 = Symbol();

let sym2 = Symbol("key"); // optional string key
```

Symbols are immutable, and unique.

```
let sym2 = Symbol("key");
let sym3 = Symbol("key");

sym2 === sym3; // false, symbols are unique
```

Just like strings, symbols can be used as keys for object properties.

```
let sym = Symbol();

let obj = {
    [sym]: "value"
};
```

```
console.log(obj[sym]); // "value"
```

Symbols can also be combined with computed property declarations to declare object properties and class members.

```
const getClassNameSymbol = Symbol();

class C {
    [getClassNameSymbol](){
       return "C";
    }
}

let c = new C();
let className = c[getClassNameSymbol](); // "C"
```

Well-known Symbols

In addition to user-defined symbols, there are well-known built-in symbols. Built-in symbols are used to represent internal language behaviors.

Here is a list of well-known symbols:

`Symbol.hasInstance`

A method that determines if a constructor object recognizes an object as one of the constructor's instances. Called by the semantics of the instanceof operator.

`Symbol.isConcatSpreadable`

A Boolean value indicating that an object should be flattened to its array elements by Array.prototype.concat.

`Symbol.iterator`

A method that returns the default iterator for an object. Called by the semantics of the for-of statement.

`Symbol.match`

A regular expression method that matches the regular expression against a string. Called by the `String.prototype.match` method.

`Symbol.replace`

A regular expression method that replaces matched substrings of a string. Called by the `String.prototype.replace` method.

`Symbol.search`

A regular expression method that returns the index within a string that matches the regular expression. Called by the `String.prototype.search` method.

`Symbol.species`

A function valued property that is the constructor function that is used to create derived objects.

`Symbol.split`

A regular expression method that splits a string at the indices that match the regular expression. Called by the `String.prototype.split` method.

`Symbol.toPrimitive`

A method that converts an object to a corresponding primitive value. Called by the `ToPrimitive` abstract operation.

`Symbol.toStringTag`

A String value that is used in the creation of the default string description of an object. Called by the built-in method `Object.prototype.toString`.

`Symbol.unscopables`

An Object whose own property names are property names that are excluded from the 'with' environment bindings of the associated objects.

12_Iterators and Generators.md

Iterables

An object is deemed iterable if it has an implementation for the `Symbol.iterator` property. Some built-in types like `Array`, `Map`, `Set`, `String`, `Int32Array`, `Uint32Array`, etc. have their `Symbol.iterator` property already implemented. `Symbol.iterator` function on an object is responsible for returning the list of values to iterate on.

`for..of` statements

`for..of` loops over an iterable object, invoking the `Symbol.iterator` property on the object. Here is a simple `for..of` loop on an array:

```
let someArray = [1, "string", false];

for (let entry of someArray) {
    console.log(entry); // 1, "string", false
}
```

`for..of` vs. `for..in` statements

Both `for..of` and `for..in` statements iterate over lists; the values iterated on are different though, `for..in` returns a list of *keys* on the object being iterated, whereas `for..of` returns a list of *values* of the numeric properties of the object being iterated.

Here is an example that demonstrates this distinction:

```
let list = [4, 5, 6];

for (let i in list) {
   console.log(i); // "0", "1", "2",
}

for (let i of list) {
   console.log(i); // "4", "5", "6"
}
```

Another distinction is that `for..in` operates on any object; it serves as a way to inspect properties on this object. `for..of` on the other hand, is mainly interested in values of iterable objects. Built-in objects like `Map` and `Set` implement `Symbol.iterator` property allowing access to stored values.

```
let pets = new Set(["Cat", "Dog", "Hamster"]);
pets["species"] = "mammals";

for (let pet in pets) {
   console.log(pet); // "species"
}

for (let pet of pets) {
    console.log(pet); // "Cat", "Dog", "Hamster"
}
```

Code generation

Targeting ES5 and ES3

When targeting an ES5 or ES3, iterators are only allowed on values of `Array` type. It is an error to use `for..of` loops on non-Array values, even if these non-Array values implement the `Symbol.iterator` property.

The compiler will generate a simple `for` loop for a `for..of` loop, for instance:

```
let numbers = [1, 2, 3];
for (let num of numbers) {
    console.log(num);
}
```

will be generated as:

```
var numbers = [1, 2, 3];
for (var _i = 0; _i < numbers.length; _i++) {
    var num = numbers[_i];
    console.log(num);
}
```

Targeting ECMAScript 2015 and higher

When targeting an ECMAScipt 2015-compliant engine, the compiler will generate `for..of` loops to target the built-in iterator implementation in the engine.

13_Modules.md

> **A note about terminology:** It's important to note that in TypeScript 1.5, the nomenclature has changed. "Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with ECMAScript 2015's terminology, (namely that `module X {` is equivalent to the now-preferred `namespace X {`).

Introduction

Starting with ECMAScript 2015, JavaScript has a concept of modules. TypeScript shares this concept.

Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the `export` forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the `import` forms.

Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.

Modules import one another using a module loader. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it. Well-known modules loaders used in JavaScript are the CommonJS module loader for Node.js and require.js for Web applications.

In TypeScript, just as in ECMAScript 2015, any file containing a top-level `import` or `export` is considered a module. Conversely, a file without any top-level `import` or `export` declarations is treated as a script whose contents are available in the global scope (and therefore to modules as well).

Export

Exporting a declaration

Any declaration (such as a variable, function, class, type alias, or interface) can be exported by adding the `export` keyword.

Validation.ts

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

ZipCodeValidator.ts

```
export const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Export statements

Export statements are handy when exports need to be renamed for consumers, so the above example can be written as:

```
class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export { ZipCodeValidator };
export { ZipCodeValidator as mainValidator };
```

Re-exports

Often modules extend other modules, and partially expose some of their features. A re-export does not import it locally, or introduce a local variable.

ParseIntBasedZipCodeValidator.ts

```
export class ParseIntBasedZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && parseInt(s).toString() === s;
    }
}

// Export original validator but rename it
export {ZipCodeValidator as RegExpBasedZipCodeValidator} from
"./ZipCodeValidator";
```

Optionally, a module can wrap one or more modules and combine all their exports using `export * from "module"` syntax.

AllValidators.ts

```
export * from "./StringValidator"; // exports interface 'StringValidator'
export * from "./LettersOnlyValidator"; // exports class 'LettersOnlyValidator'
export * from "./ZipCodeValidator";  // exports class 'ZipCodeValidator'
```

Import

Importing is just about as easy as exporting from a module. Importing an exported declaration is done through using one of the `import` forms below:

Import a single export from a module

```
import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();
```

imports can also be renamed

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

Import the entire module into a single variable, and use it to access the module exports

```
import * as validator from "./ZipCodeValidator";
let myValidator = new validator.ZipCodeValidator();
```

Import a module for side-effects only

Though not recommended practice, some modules set up some global state that can be used by other modules. These modules may not have any exports, or the consumer is not interested in any of their exports. To import these modules, use:

```
import "./my-module.js";
```

Default exports

Each module can optionally export a `default` export. Default exports are marked with the keyword `default`; and there can only be one `default` export per module. `default` exports are imported using a different import form.

`default` exports are really handy. For instance, a library like JQuery might have a default export of `jQuery` or `$`, which we'd probably also import under the name `$` or `jQuery`.

JQuery.d.ts

```
declare let $: JQuery;
export default $;
```

App.ts

```
import $ from "JQuery";

$("button.continue").html( "Next Step..." );
```

Classes and function declarations can be authored directly as default exports. Default export class and function declaration names are optional.

ZipCodeValidator.ts

```
export default class ZipCodeValidator {
    static numberRegexp = /^[0-9]+$/;
    isAcceptable(s: string) {
        return s.length === 5 && ZipCodeValidator.numberRegexp.test(s);
    }
}
```

Test.ts

```
import validator from "./ZipCodeValidator";

let myValidator = new validator();
```

or

StaticZipCodeValidator.ts

```
const numberRegexp = /^[0-9]+$/;

export default function (s: string) {
    return s.length === 5 && numberRegexp.test(s);
}
```

Test.ts

```
import validate from "./StaticZipCodeValidator";

let strings = ["Hello", "98052", "101"];

// Use function validate
strings.forEach(s => {
  console.log(`"${s}" ${validate(s) ? " matches" : " does not match"}`);
});
```

`default` exports can also be just values:

OneTwoThree.ts

```
export default "123";
```

Log.ts

```
import num from "./OneTwoThree";

console.log(num); // "123"
```

`export =` and `import = require()`

Both CommonJS and AMD generally have the concept of an `exports` object which contains all exports from a module.

They also support replacing the `exports` object with a custom single object. Default exports are meant to act as a replacement for this behavior; however, the two are incompatible. TypeScript supports `export =` to model the traditional CommonJS and AMD workflow.

The `export =` syntax specifies a single object that is exported from the module. This can be a class, interface, namespace, function, or enum.

When exporting a module using `export =`, TypeScript-specific `import module = require("module")` must be used to import the module.

ZipCodeValidator.ts

```
let numberRegexp = /^[0-9]+$/;
class ZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export = ZipCodeValidator;
```

Test.ts

```
import zip = require("./ZipCodeValidator");

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validator = new zip();

// Show whether each string passed each validator
strings.forEach(s => {
  console.log(`"${ s }" - ${ validator.isAcceptable(s) ? "matches" : "does not
match" }`);
});
```

Code Generation for Modules

Depending on the module target specified during compilation, the compiler will generate appropriate code for Node.js (CommonJS), require.js (AMD), UMD, SystemJS, or ECMAScript 2015 native modules (ES6) module-loading systems. For more information on what the `define`, `require` and `register` calls in the generated code do, consult the documentation for each module loader.

This simple example shows how the names used during importing and exporting get translated into the module loading code.

SimpleModule.ts

```
import m = require("mod");
export let t = m.something + 1;
```

AMD / RequireJS SimpleModule.js

```
define(["require", "exports", "./mod"], function (require, exports, mod_1) {
    exports.t = mod_1.something + 1;
});
```

CommonJS / Node SimpleModule.js

```
var mod_1 = require("./mod");
exports.t = mod_1.something + 1;
```

UMD SimpleModule.js

```
(function (factory) {
    if (typeof module === "object" && typeof module.exports === "object") {
        var v = factory(require, exports); if (v !== undefined) module.exports =
v;
    }
    else if (typeof define === "function" && define.amd) {
        define(["require", "exports", "./mod"], factory);
    }
})(function (require, exports) {
    var mod_1 = require("./mod");
    exports.t = mod_1.something + 1;
});
```

System SimpleModule.js

```
System.register(["./mod"], function(exports_1) {
    var mod_1;
    var t;
    return {
        setters:[
            function (mod_1_1) {
                mod_1 = mod_1_1;
            }],
        execute: function() {
            exports_1("t", t = mod_1.something + 1);
        }
    }
});
```

Native ECMAScript 2015 modules SimpleModule.js

```
import { something } from "./mod";
export var t = something + 1;
```

Simple Example

Below, we've consolidated the Validator implementations used in previous examples to only export a single named export from each module.

To compile, we must specify a module target on the command line. For Node.js, use `--module commonjs`; for require.js, use `--module amd`. For example:

```
tsc --module commonjs Test.ts
```

When compiled, each module will become a separate `.js` file. As with reference tags, the compiler will follow `import` statements to compile dependent files.

Validation.ts

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

LettersOnlyValidator.ts

```
import { StringValidator } from "./Validation";

const lettersRegexp = /^[A-Za-z]+$/;

export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}
```

ZipCodeValidator.ts

```
import { StringValidator } from "./Validation";

const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Test.ts

```
import { StringValidator } from "./Validation";
import { ZipCodeValidator } from "./ZipCodeValidator";
import { LettersOnlyValidator } from "./LettersOnlyValidator";

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// Show whether each string passed each validator
strings.forEach(s => {
    for (let name in validators) {
        console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches" :
"does not match" } ${ name }`);
    }
});
```

Optional Module Loading and Other Advanced Loading Scenarios

In some cases, you may want to only load a module under some conditions. In TypeScript, we can use the pattern shown below to implement this and other advanced loading scenarios to directly invoke the module loaders without losing type safety.

The compiler detects whether each module is used in the emitted JavaScript. If a module identifier is only ever used as part of a type annotations and never as an expression, then no `require` call is emitted for that module. This elision of unused references is a good performance optimization, and also allows for optional loading of those modules.

The core idea of the pattern is that the `import id = require("...")` statement gives us access to the types exposed by the module. The module loader is invoked (through `require`) dynamically, as shown in the `if` blocks below. This leverages the reference-elision optimization so that the module is only loaded when needed. For this pattern to work, it's important that the symbol defined via an `import` is only used in type positions (i.e. never in a position that would be emitted into the JavaScript).

To maintain type safety, we can use the `typeof` keyword. The `typeof` keyword, when used in a type position, produces the type of a value, in this case the type of the module.

Dynamic Module Loading in Node.js

```
declare function require(moduleName: string): any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    let ZipCodeValidator: typeof Zip = require("./ZipCodeValidator");
    let validator = new ZipCodeValidator();
    if (validator.isAcceptable("...")) { /* ... */ }
}
```

Sample: Dynamic Module Loading in require.js

```
declare function require(moduleNames: string[], onLoad: (...args: any[]) => void): void;

import * as Zip from "./ZipCodeValidator";

if (needZipValidation) {
    require(["./ZipCodeValidator"], (ZipCodeValidator: typeof Zip) => {
        let validator = new ZipCodeValidator.ZipCodeValidator();
        if (validator.isAcceptable("...")) { /* ... */ }
    });
}
```

Sample: Dynamic Module Loading in System.js

```
declare const System: any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";
```

```
if (needZipValidation) {
    System.import("./ZipCodeValidator").then((ZipCodeValidator: typeof Zip) => {
        var x = new ZipCodeValidator();
        if (x.isAcceptable("...")) { /* ... */ }
    });
}
```

Working with Other JavaScript Libraries

To describe the shape of libraries not written in TypeScript, we need to declare the API that the library exposes.

We call declarations that don't define an implementation "ambient". Typically, these are defined in `.d.ts` files. If you're familiar with C/C++, you can think of these as `.h` files. Let's look at a few examples.

Ambient Modules

In Node.js, most tasks are accomplished by loading one or more modules. We could define each module in its own `.d.ts` file with top-level export declarations, but it's more convenient to write them as one larger `.d.ts` file. To do so, we use a construct similar to ambient namespaces, but we use the `module` keyword and the quoted name of the module which will be available to a later import. For example:

node.d.ts (simplified excerpt)

```
declare module "url" {
    export interface Url {
        protocol?: string;
        hostname?: string;
        pathname?: string;
    }

    export function parse(urlStr: string, parseQueryString?, slashesDenoteHost?):
Url;
}

declare module "path" {
    export function normalize(p: string): string;
    export function join(...paths: any[]): string;
    export var sep: string;
}
```

Now we can `/// <reference> node.d.ts` and then load the modules using `import url = require("url");` or `import * as URL from "url"`.

```
/// <reference path="node.d.ts"/>
import * as URL from "url";
```

```
let myUrl = URL.parse("http://www.typescriptlang.org");
```

Shorthand ambient modules

If you don't want to take the time to write out declarations before using a new module, you can use a shorthand declaration to get started quickly.

declarations.d.ts

```
declare module "hot-new-module";
```

All imports from a shorthand module will have the `any` type.

```
import x, {y} from "hot-new-module";
x(y);
```

Wildcard module declarations

Some module loaders such as SystemJS and AMD allow non-JavaScript content to be imported. These typically use a prefix or suffix to indicate the special loading semantics. Wildcard module declarations can be used to cover these cases.

```
declare module "*!text" {
    const content: string;
    export default content;
}
// Some do it the other way around.
declare module "json!*" {
    const value: any;
    export default value;
}
```

Now you can import things that match `"*!text"` or `"json!*"`.

```
import fileContent from "./xyz.txt!text";
import data from "json!http://example.com/data.json";
console.log(data, fileContent);
```

UMD modules

Some libraries are designed to be used in many module loaders, or with no module loading (global variables). These are known as UMD modules. These libraries can be accessed through either an import or a global variable. For example:

math-lib.d.ts

```
export function isPrime(x: number): boolean;
export as namespace mathLib;
```

The library can then be used as an import within modules:

```
import { isPrime } from "math-lib";
isPrime(2);
mathLib.isPrime(2); // ERROR: can't use the global definition from inside a module
```

It can also be used as a global variable, but only inside of a script. (A script is a file with no imports or exports.)

```
mathLib.isPrime(2);
```

Guidance for structuring modules

Export as close to top-level as possible

Consumers of your module should have as little friction as possible when using things that you export. Adding too many levels of nesting tends to be cumbersome, so think carefully about how you want to structure things.

Exporting a namespace from your module is an example of adding too many layers of nesting. While namespaces sometime have their uses, they add an extra level of indirection when using modules. This can quickly become a pain point for users, and is usually unnecessary.

Static methods on an exported class have a similar problem - the class itself adds a layer of nesting. Unless it increases expressivity or intent in a clearly useful way, consider simply exporting a helper function.

If you're only exporting a single `class` or `function`, use `export default`

Just as "exporting near the top-level" reduces friction on your module's consumers, so does introducing a default export. If a module's primary purpose is to house one specific export, then you should consider exporting it as a default export. This makes both importing and actually using the import a little easier. For example:

MyClass.ts

```
export default class SomeType {
  constructor() { ... }
}
```

MyFunc.ts

```
export default function getThing() { return "thing"; }
```

Consumer.ts

```
import t from "./MyClass";
import f from "./MyFunc";
let x = new t();
console.log(f());
```

This is optimal for consumers. They can name your type whatever they want (`t` in this case) and don't have to do any excessive dotting to find your objects.

If you're exporting multiple objects, put them all at top-level

MyThings.ts

```
export class SomeType { /* ... */ }
export function someFunc() { /* ... */ }
```

Conversely when importing:

Explicitly list imported names

Consumer.ts

```
import { SomeType, someFunc } from "./MyThings";
let x = new SomeType();
let y = someFunc();
```

Use the namespace import pattern if you're importing a large number of things

MyLargeModule.ts

```
export class Dog { ... }
export class Cat { ... }
export class Tree { ... }
export class Flower { ... }
```

Consumer.ts

```
import * as myLargeModule from "./MyLargeModule.ts";
let x = new myLargeModule.Dog();
```

Re-export to extend

Often you will need to extend functionality on a module. A common JS pattern is to augment the original object with *extensions*, similar to how JQuery extensions work. As we've mentioned before, modules do not *merge* like global namespace objects would. The recommended solution is to *not* mutate the original object, but rather export a new entity that provides the new functionality.

Consider a simple calculator implementation defined in module `Calculator.ts`. The module also exports a helper function to test the calculator functionality by passing a list of input strings and writing the result at the end.

Calculator.ts

```
export class Calculator {
    private current = 0;
    private memory = 0;
    private operator: string;

    protected processDigit(digit: string, currentValue: number) {
        if (digit >= "0" && digit <= "9") {
            return currentValue * 10 + (digit.charCodeAt(0) - "0".charCodeAt(0));
        }
    }

    protected processOperator(operator: string) {
        if (["+", "-", "*", "/"].indexOf(operator) >= 0) {
            return operator;
        }
    }

    protected evaluateOperator(operator: string, left: number, right: number):
number {
        switch (this.operator) {
            case "+": return left + right;
            case "-": return left - right;
            case "*": return left * right;
            case "/": return left / right;
        }
    }
    private evaluate() {
        if (this.operator) {
            this.memory = this.evaluateOperator(this.operator, this.memory, this.current);
        }
        else {
            this.memory = this.current;
        }
        this.current = 0;
    }

    public handleChar(char: string) {
        if (char === "=") {
            this.evaluate();
            return;
        }
        else {
            let value = this.processDigit(char, this.current);
            if (value !== undefined) {
                this.current = value;
                return;
            }
            else {
                let value = this.processOperator(char);
                if (value !== undefined) {
                    this.evaluate();
                    this.operator = value;
                    return;
                }
            }
        }
        throw new Error(`Unsupported input: '${char}'`);
    }

    public getResult() {
        return this.memory;
    }
}

export function test(c: Calculator, input: string) {
    for (let i = 0; i < input.length; i++) {
        c.handleChar(input[i]);
    }

    console.log(`result of '${input}' is '${c.getResult()}'`);
}
```

Here is a simple test for the calculator using the exposed `test` function.

TestCalculator.ts

```
import { Calculator, test } from "./Calculator";


let c = new Calculator();
test(c, "1+2*33/11="); // prints 9
```

Now to extend this to add support for input with numbers in bases other than 10, let's create `ProgrammerCalculator.ts`

ProgrammerCalculator.ts

```
import { Calculator } from "./Calculator";

class ProgrammerCalculator extends Calculator {
    static digits = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B",
"C", "D", "E", "F"];

    constructor(public base: number) {
        super();
        const maxBase = ProgrammerCalculator.digits.length;
        if (base <= 0 || base > maxBase) {
            throw new Error(`base has to be within 0 to ${maxBase} inclusive.`);
        }
    }

    protected processDigit(digit: string, currentValue: number) {
        if (ProgrammerCalculator.digits.indexOf(digit) >= 0) {
            return currentValue * this.base +
ProgrammerCalculator.digits.indexOf(digit);
        }
    }
}

// Export the new extended calculator as Calculator
export { ProgrammerCalculator as Calculator };

// Also, export the helper function
export { test } from "./Calculator";
```

The new module `ProgrammerCalculator` exports an API shape similar to that of the original `Calculator` module, but does not augment any objects in the original module. Here is a test for our ProgrammerCalculator class:

TestProgrammerCalculator.ts

```
import { Calculator, test } from "./ProgrammerCalculator";
```

```
let c = new Calculator(2);
test(c, "001+010="); // prints 3
```

Do not use namespaces in modules

When first moving to a module-based organization, a common tendency is to wrap exports in an additional layer of namespaces. Modules have their own scope, and only exported declarations are visible from outside the module. With this in mind, namespace provide very little, if any, value when working with modules.

On the organization front, namespaces are handy for grouping together logically-related objects and types in the global scope. For example, in C#, you're going to find all the collection types in System.Collections. By organizing our types into hierarchical namespaces, we provide a good "discovery" experience for users of those types. Modules, on the other hand, are already present in a file system, necessarily. We have to resolve them by path and filename, so there's a logical organization scheme for us to use. We can have a /collections/generic/ folder with a list module in it.

Namespaces are important to avoid naming collisions in the global scope. For example, you might have `My.Application.Customer.AddForm` and `My.Application.Order.AddForm` -- two types with the same name, but a different namespace. This, however, is not an issue with modules. Within a module, there's no plausible reason to have two objects with the same name. From the consumption side, the consumer of any given module gets to pick the name that they will use to refer to the module, so accidental naming conflicts are impossible.

> For more discussion about modules and namespaces see [Namespaces and Modules](./Namespaces and Modules.md).

Red Flags

All of the following are red flags for module structuring. Double-check that you're not trying to namespace your external modules if any of these apply to your files:

- A file whose only top-level declaration is `export namespace Foo { ... }` (remove `Foo` and move everything 'up' a level)
- A file that has a single `export class` or `export function` (consider using `export default`)
- Multiple files that have the same `export namespace Foo {` at top-level (don't think that these are going to combine into one `Foo`!)

14_Namespaces.md

> **A note about terminology:** It's important to note that in TypeScript 1.5, the nomenclature has changed. "Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with ECMAScript 2015's terminology, (namely that `module X {` is equivalent to the now-preferred `namespace X {`).

Introduction

This post outlines the various ways to organize your code using namespaces (previously "internal modules") in TypeScript. As we alluded in our note about terminology, "internal modules" are now referred to as "namespaces". Additionally, anywhere the `module` keyword was used when declaring an internal module, the `namespace` keyword can and should be used instead. This avoids confusing new users by overloading them with similarly named terms.

First steps

Let's start with the program we'll be using as our example throughout this page. We've written a small set of simplistic string validators, as you might write to check a user's input on a form in a webpage or check the format of an externally-provided data file.

Validators in a single file

```
interface StringValidator {
    isAcceptable(s: string): boolean;
}

let lettersRegexp = /^[A-Za-z]+$/;
let numberRegexp = /^[0-9]+$/;

class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}

class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        let isMatch = validators[name].isAcceptable(s);
        console.log(`'${ s }' ${ isMatch ? "matches" : "does not match" } '${ name
}'.`);
    }
}
```

Namespacing

As we add more validators, we're going to want to have some kind of organization scheme so that we can keep track of our types and not worry about name collisions with other objects. Instead of putting lots of different names into the global namespace, let's wrap up our objects into a namespace.

In this example, we'll move all validator-related entities into a namespace called `Validation`. Because we want the interfaces and classes here to be visible outside the namespace, we preface them with `export`. Conversely, the variables `lettersRegexp` and `numberRegexp` are implementation details, so they are left unexported and will not be visible to code outside the namespace. In the test code at the bottom of the file, we now need to qualify the names of the types when used outside the namespace, e.g. `Validation.LettersOnlyValidator`.

Namespaced Validators

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }

    const lettersRegexp = /^[A-Za-z]+$/;
    const numberRegexp = /^[0-9]+$/;

    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }

    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches" :
"does not match" } ${ name }`);
```

```
    }
}
```

Splitting Across Files

As our application grows, we'll want to split the code across multiple files to make it easier to maintain.

Multi-file namespaces

Here, we'll split our `Validation` namespace across many files. Even though the files are separate, they can each contribute to the same namespace and can be consumed as if they were all defined in one place. Because there are dependencies between files, we'll add reference tags to tell the compiler about the relationships between the files. Our test code is otherwise unchanged.

Validation.ts

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }
}
```

LettersOnlyValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
    const lettersRegexp = /^[A-Za-z]+$/;
    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }
}
```

ZipCodeValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
    const numberRegexp = /^[0-9]+$/;
    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}
```

Test.ts

```
/// <reference path="Validation.ts" />
/// <reference path="LettersOnlyValidator.ts" />
/// <reference path="ZipCodeValidator.ts" />

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches" :
"does not match" } ${ name }`);
    }
}
```

Once there are multiple files involved, we'll need to make sure all of the compiled code gets loaded. There are two ways of doing this.

First, we can use concatenated output using the `--outFile` flag to compile all of the input files into a single JavaScript output file:

```
tsc --outFile sample.js Test.ts
```

The compiler will automatically order the output file based on the reference tags present in the files. You can also specify each file individually:

```
tsc --outFile sample.js Validation.ts LettersOnlyValidator.ts ZipCodeValidator.ts
Test.ts
```

Alternatively, we can use per-file compilation (the default) to emit one JavaScript file for each input file. If multiple JS files get produced, we'll need to use `<script>` tags on our webpage to load each emitted file in the appropriate order, for example:

MyTestPage.html (excerpt)

```
    <script src="Validation.js" type="text/javascript" />
    <script src="LettersOnlyValidator.js" type="text/javascript" />
    <script src="ZipCodeValidator.js" type="text/javascript" />
    <script src="Test.js" type="text/javascript" />
```

Aliases

Another way that you can simplify working with of namespaces is to use `import q = x.y.z` to create shorter names for commonly-used objects. Not to be confused with the `import x = require("name")` syntax used to load modules, this syntax simply creates an alias for the specified symbol. You can use these sorts of imports (commonly referred to as aliases) for any kind of identifier, including objects created from module imports.

```
namespace Shapes {
    export namespace Polygons {
        export class Triangle { }
        export class Square { }
    }
}

import polygons = Shapes.Polygons;
let sq = new polygons.Square(); // Same as 'new Shapes.Polygons.Square()'
```

Notice that we don't use the `require` keyword; instead we assign directly from the qualified name of the symbol we're importing. This is similar to using `var`, but also works on the type and namespace meanings of the imported symbol. Importantly, for values, `import` is a distinct reference from the original symbol, so changes to an aliased `var` will not be reflected in the original variable.

Working with Other JavaScript Libraries

To describe the shape of libraries not written in TypeScript, we need to declare the API that the library exposes. Because most JavaScript libraries expose only a few top-level objects, namespaces are a good way to represent them.

We call declarations that don't define an implementation "ambient". Typically these are defined in `.d.ts` files. If you're familiar with C/C++, you can think of these as `.h` files. Let's look at a few examples.

Ambient Namespaces

The popular library D3 defines its functionality in a global object called `d3`. Because this library is loaded through a `<script>` tag (instead of a module loader), its declaration uses namespaces to define its shape. For the TypeScript compiler to see this shape, we use an ambient namespace declaration. For example, we could begin writing it as follows:

D3.d.ts (simplified excerpt)

```
declare namespace D3 {
    export interface Selectors {
        select: {
            (selector: string): Selection;
            (element: EventTarget): Selection;
        };
    }

    export interface Event {
        x: number;
        y: number;
    }

    export interface Base extends Selectors {
        event: Event;
    }
}

declare var d3: D3.Base;
```

15_Namespaces and Modules.md

> **A note about terminology:** It's important to note that in TypeScript 1.5, the nomenclature has changed. "Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with ECMAScript 2015's terminology, (namely that `module X {` is equivalent to the now-preferred `namespace X {`).

Introduction

This post outlines the various ways to organize your code using namespaces and modules in TypeScript. We'll also go over some advanced topics of how to use namespaces and modules, and address some common pitfalls when using them in TypeScript.

See the Modules documentation for more information about modules. See the Namespaces documentation for more information about namespaces.

Using Namespaces

Namespaces are simply named JavaScript objects in the global namespace. This makes namespaces a very simple construct to use. They can span multiple files, and can be concatenated using `--outFile`. Namespaces can be a good way to structure your code in a Web Application, with all dependencies included as `<script>` tags in your HTML page.

Just like all global namespace pollution, it can be hard to identify component dependencies, especially in a large application.

Using Modules

Just like namespaces, modules can contain both code and declarations. The main difference is that modules *declare* their dependencies.

Modules also have a dependency on a module loader (such as CommonJs/Require.js). For a small JS application this might not be optimal, but for larger applications, the cost comes with long term modularity and maintainability benefits. Modules provide for better code reuse, stronger isolation and better tooling support for bundling.

It is also worth noting that, for Node.js applications, modules are the default and the recommended approach to structure your code.

Starting with ECMAScript 2015, modules are native part of the language, and should be supported by all compliant engine implementations. Thus, for new projects modules would be the recommended code organization mechanism.

Pitfalls of Namespaces and Modules

In this section we'll describe various common pitfalls in using namespaces and modules, and how to avoid them.

`/// <reference>`-ing a module

A common mistake is to try to use the `/// <reference ... />` syntax to refer to a module file, rather than using an `import` statement. To understand the distinction, we first need to understand how the compiler can locate the type information for a module based on the path of an `import` (e.g. the `...` in `import x from "...";`, `import x = require("...");`, etc.) path.

The compiler will try to find a `.ts`, `.tsx`, and then a `.d.ts` with the appropriate path. If a specific file could not be found, then the compiler will look for an *ambient module declaration*. Recall that these need to be declared in a `.d.ts` file.

- `myModules.d.ts`

  ```
  // In a .d.ts file or .ts file that is not a module:
  declare module "SomeModule" {
      export function fn(): string;
  }
  ```

- `myOtherModule.ts`

```
/// <reference path="myModules.d.ts" />
import * as m from "SomeModule";
```

The reference tag here allows us to locate the declaration file that contains the declaration for the ambient module. This is how the `node.d.ts` file that several of the TypeScript samples use is consumed.

Needless Namespacing

If you're converting a program from namespaces to modules, it can be easy to end up with a file that looks like this:

- `shapes.ts`

```
export namespace Shapes {
    export class Triangle { /* ... */ }
    export class Square { /* ... */ }
}
```

The top-level module here `Shapes` wraps up `Triangle` and `Square` for no reason. This is confusing and annoying for consumers of your module:

- `shapeConsumer.ts`

```
import * as shapes from "./shapes";
let t = new shapes.Shapes.Triangle(); // shapes.Shapes?
```

A key feature of modules in TypeScript is that two different modules will never contribute names to the same scope. Because the consumer of a module decides what name to assign it, there's no need to proactively wrap up the exported symbols in a namespace.

To reiterate why you shouldn't try to namespace your module contents, the general idea of namespacing is to provide logical grouping of constructs and to prevent name collisions. Because the module file itself is already a logical grouping, and its top-level name is defined by the code that imports it, it's unnecessary to use an additional module layer for exported objects.

Here's a revised example:

- `shapes.ts`

```
export class Triangle { /* ... */ }
export class Square { /* ... */ }
```

- `shapeConsumer.ts`

```
import * as shapes from "./shapes";
let t = new shapes.Triangle();
```

Trade-offs of Modules

Just as there is a one-to-one correspondence between JS files and modules, TypeScript has a one-to-one correspondence between module source files and their emitted JS files. One effect of this is that it's not possible to concatenate multiple module source files depending on the module system you target. For instance, you can't use the `outFile` option while targeting `commonjs` or `umd`, but with TypeScript 1.8 and later, it's possible to use `outFile` when targeting `amd` or `system`.

16_Module Resolution.md

> This section assumes some basic knowledge about modules. Please see the Modules documentation for more information.

Module resolution is the process the compiler uses to figure out what an import refers to. Consider an import statement like `import { a } from "moduleA"`; in order to check any use of `a`, the compiler needs to know exactly what it represents, and will need to check its definition `moduleA`.

At this point, the compiler will ask "what's the shape of `moduleA`?" While this sounds straightforward, `moduleA` could be defined in one of your own `.ts`/`.tsx` files, or in a `.d.ts` that your code depends on.

First, the compiler will try to locate a file that represents the imported module. To do so the compiler follows one of two different strategies: Classic or Node. These strategies tell the compiler *where* to look for `moduleA`.

If that didn't work and if the module name is non-relative (and in the case of `"moduleA"`, it is), then the compiler will attempt to locate an ambient module declaration. We'll cover non-relative imports next.

Finally, if the compiler could not resolve the module, it will log an error. In this case, the error would be something like `error TS2307: Cannot find module 'moduleA'.`

Relative vs. Non-relative module imports

Module imports are resolved differently based on whether the module reference is relative or non-relative.

A *relative import* is one that starts with `/`, `./` or `../`. Some examples include:

- `import Entry from "./components/Entry";`
- `import { DefaultHeaders } from "../constants/http";`
- `import "/mod";`

Any other import is considered **non-relative**. Some examples include:

- `import * as $ from "jquery";`
- `import { Component } from "@angular/core";`

A relative import is resolved relative to the importing file and *cannot* resolve to an ambient module declaration. You should use relative imports for your own modules that are guaranteed to maintain their relative location at runtime.

A non-relative import can be resolved relative to `baseUrl`, or through path mapping, which we'll cover below. They can also resolve to ambient module declarations. Use non-relative paths when importing any of your external dependencies.

Module Resolution Strategies

There are two possible module resolution strategies: Node and Classic. You can use the `--moduleResolution` flag to specify the module resolution strategy. If not specified, the default is Classic for `--module AMD | System | ES2015` or Node otherwise.

Classic

This used to be TypeScript's default resolution strategy. Nowadays, this strategy is mainly present for backward compatibility.

A relative import will be resolved relative to the importing file. So `import { b } from "./moduleB"` in source file `/root/src/folder/A.ts` would result in the following lookups:

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`

For non-relative module imports, however, the compiler walks up the directory tree starting with the directory containing the importing file, trying to locate a matching definition file.

For example:

A non-relative import to `moduleB` such as `import { b } from "moduleB"`, in a source file `/root/src/folder/A.ts`, would result in attempting the following locations for locating `"moduleB"`:

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`
3. `/root/src/moduleB.ts`
4. `/root/src/moduleB.d.ts`
5. `/root/moduleB.ts`
6. `/root/moduleB.d.ts`
7. `/moduleB.ts`
8. `/moduleB.d.ts`

Node

This resolution strategy attempts to mimic the Node.js module resolution mechanism at runtime. The full Node.js resolution algorithm is outlined in Node.js module documentation.

How Node.js resolves modules

To understand what steps the TS compiler will follow, it is important to shed some light on Node.js modules. Traditionally, imports in Node.js are performed by calling a function named `require`. The behavior Node.js takes will differ depending on if `require` is given a relative path or a non-relative path.

Relative paths are fairly straightforward. As an example, let's consider a file located at `/root/src/moduleA.js`, which contains the import `var x = require("./moduleB");` Node.js resolves that import in the following order:

1. Ask the file named `/root/src/moduleB.js`, if it exists.
2. Ask the folder `/root/src/moduleB` if it contains a file named `package.json` that specifies a `"main"` module. In our example, if Node.js found the file `/root/src/moduleB/package.json` containing `{ "main": "lib/mainModule.js" }`, then Node.js will refer to `/root/src/moduleB/lib/mainModule.js`.
3. Ask the folder `/root/src/moduleB` if it contains a file named `index.js`. That file is implicitly considered that folder's "main" module.

You can read more about this in Node.js documentation on file modules and folder modules.

However, resolution for a non-relative module name is performed differently. Node will look for your modules in special folders named `node_modules`. A `node_modules` folder can be on the same level as the current file, or higher up in the directory chain. Node will walk up the directory chain, looking through each `node_modules` until it finds the module you tried to load.

Following up our example above, consider if `/root/src/moduleA.js` instead used a non-relative path and had the import `var x = require("moduleB");`. Node would then try to resolve `moduleB` to each of the locations until one worked.

1. `/root/src/node_modules/moduleB.js`
2. `/root/src/node_modules/moduleB/package.json` (if it specifies a `"main"` property)
3. `/root/src/node_modules/moduleB/index.js`

4. `/root/node_modules/moduleB.js`
5. `/root/node_modules/moduleB/package.json` (if it specifies a `"main"` property)
6. `/root/node_modules/moduleB/index.js`

7. `/node_modules/moduleB.js`
8. `/node_modules/moduleB/package.json` (if it specifies a `"main"` property)
9. `/node_modules/moduleB/index.js`

Notice that Node.js jumped up a directory in steps (4) and (7).

You can read more about the process in Node.js documentation on loading modules from `node_modules`.

How TypeScript resolves modules

TypeScript will mimic the Node.js run-time resolution strategy in order to locate definition files for modules at compile-time. To accomplish this, TypeScript overlays the TypeScript source file extensions (`.ts`, `.tsx`, and `.d.ts`) over the Node's resolution logic. TypeScript will also use a field in `package.json` named `"types"` to mirror the purpose of `"main"` - the compiler will use it to find the "main" definition file to consult.

For example, an import statement like `import { b } from "./moduleB"` in `/root/src/moduleA.ts` would result in attempting the following locations for locating `"./moduleB"`:

1. `/root/src/moduleB.ts`
2. `/root/src/moduleB.tsx`
3. `/root/src/moduleB.d.ts`
4. `/root/src/moduleB/package.json` (if it specifies a `"types"` property)
5. `/root/src/moduleB/index.ts`
6. `/root/src/moduleB/index.tsx`
7. `/root/src/moduleB/index.d.ts`

Recall that Node.js looked for a file named `moduleB.js`, then an applicable `package.json`, and then for an `index.js`.

Similarly a non-relative import will follow the Node.js resolution logic, first looking up a file, then looking up an applicable folder. So `import { b } from "moduleB"` in source file `/root/src/moduleA.ts` would result in the following lookups:

1. `/root/src/node_modules/moduleB.ts`
2. `/root/src/node_modules/moduleB.tsx`
3. `/root/src/node_modules/moduleB.d.ts`
4. `/root/src/node_modules/moduleB/package.json` (if it specifies a `"types"` property)
5. `/root/src/node_modules/moduleB/index.ts`
6. `/root/src/node_modules/moduleB/index.tsx`
7. `/root/src/node_modules/moduleB/index.d.ts`

8. `/root/node_modules/moduleB.ts`
9. `/root/node_modules/moduleB.tsx`
10. `/root/node_modules/moduleB.d.ts`
11. `/root/node_modules/moduleB/package.json` (if it specifies a `"types"` property)
12. `/root/node_modules/moduleB/index.ts`
13. `/root/node_modules/moduleB/index.tsx`
14. `/root/node_modules/moduleB/index.d.ts`

15. `/node_modules/moduleB.ts`
16. `/node_modules/moduleB.tsx`
17. `/node_modules/moduleB.d.ts`
18. `/node_modules/moduleB/package.json` (if it specifies a `"types"` property)
19. `/node_modules/moduleB/index.ts`
20. `/node_modules/moduleB/index.tsx`
21. `/node_modules/moduleB/index.d.ts`

Don't be intimidated by the number of steps here - TypeScript is still only jumping up directories twice at steps (8) and (15). This is really no more complex than what Node.js itself is doing.

Additional module resolution flags

A project source layout sometimes does not match that of the output. Usually a set of build steps result in generating the final output. These include compiling `.ts` files into `.js`, and copying dependencies from different source locations to a single output location. The net result is that modules at runtime may have different names than the source files containing their definitions. Or module paths in the final output may not match their corresponding source file paths at compile time.

The TypeScript compiler has a set of additional flags to *inform* the compiler of transformations that are expected to happen to the sources to generate the final output.

It is important to note that the compiler will *not* perform any of these transformations; it just uses these pieces of information to guide the process of resolving a module import to its definition file.

Base URL

Using a `baseUrl` is a common practice in applications using AMD module loaders where modules are "deployed" to a single folder at run-time. The sources of these modules can live in different directories, but a build script will put them all together.

Setting `baseUrl` informs the compiler where to find modules. All module imports with non-relative names are assumed to be relative to the `baseUrl`.

Value of *baseUrl* is determined as either:

- value of *baseUrl* command line argument (if given path is relative, it is computed based on current directory)
- value of *baseUrl* property in 'tsconfig.json' (if given path is relative, it is computed based on the location of 'tsconfig.json')

Note that relative module imports are not impacted by setting the baseUrl, as they are always resolved relative to their importing files.

You can find more documentation on baseUrl in RequireJS and SystemJS documentation.

Path mapping

Sometimes modules are not directly located under *baseUrl*. For instance, an import to a module `"jquery"` would be translated at runtime to `"node_modules/jquery/dist/jquery.slim.min.js"`. Loaders use a mapping configuration to map module names to files at run-time, see RequireJs documentation and SystemJS documentation.

The TypeScript compiler supports the declaration of such mappings using `"paths"` property in `tsconfig.json` files. Here is an example for how to specify the `"paths"` property for `jquery`.

```
{
  "compilerOptions": {
    "baseUrl": ".", // This must be specified if "paths" is.
    "paths": {
      "jquery": ["node_modules/jquery/dist/jquery"] // This mapping is relative to "baseUrl"
    }
  }
}
```

Please notice that `"paths"` are resolved relative to `"baseUrl"`. When setting `"baseUrl"` to another value than `"."`, i.e. the directory of `tsconfig.json`, the mappings must be changed accordingly. Say, you set `"baseUrl": "./src"` in the above example, then jquery should be mapped to `"../node_modules/jquery/dist/jquery"`.

Using `"paths"` also allows for more sophisticated mappings including multiple fall back locations. Consider a project configuration where only some modules are available in one location, and the rest are in another. A build step would put them all together in one place. The project layout may look like:

```
projectRoot
├── folder1
│   ├── file1.ts (imports 'folder1/file2' and 'folder2/file3')
│   └── file2.ts
├── generated
│   ├── folder1
│   └── folder2
│       └── file3.ts
└── tsconfig.json
```

The corresponding `tsconfig.json` would look like:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "*": [
        "*",
        "generated/*"
      ]
    }
  }
}
```

This tells the compiler for any module import that matches the pattern `"*"` (i.e. all values), to look in two locations:

1. `"*"`: meaning the same name unchanged, so map `<moduleName>` => `<baseUrl>/<moduleName>`
2. `"generated/*"` meaning the module name with an appended prefix "generated", so map `<moduleName>` => `<baseUrl>/generated/<moduleName>`

Following this logic, the compiler will attempt to resolve the two imports as such:

- import 'folder1/file2'
    1. pattern '*' is matched and wildcard captures the whole module name
    2. try first substitution in the list: '*' -> `folder1/file2`
    3. result of substitution is non-relative name - combine it with *baseUrl* -> `projectRoot/folder1/file2.ts`.
    4. File exists. Done.

- import 'folder2/file3'
    1. pattern '*' is matched and wildcard captures the whole module name
    2. try first substitution in the list: '*' -> `folder2/file3`
    3. result of substitution is non-relative name - combine it with *baseUrl* -> `projectRoot/folder2/file3.ts`.
    4. File does not exist, move to the second substitution
    5. second substitution 'generated/*' -> `generated/folder2/file3`
    6. result of substitution is non-relative name - combine it with *baseUrl* -> `projectRoot/generated/folder2/file3.ts`.
    7. File exists. Done.

Virtual Directories with `rootDirs`

Sometimes the project sources from multiple directories at compile time are all combined to generate a single output directory. This can be viewed as a set of source directories create a "virtual" directory.

Using 'rootDirs', you can inform the compiler of the *roots* making up this "virtual" directory; and thus the compiler can resolve relative modules imports within these "virtual" directories *as if* were merged together in one directory.

For example consider this project structure:

```
src
└── views
    └── view1.ts (imports './template1')
    └── view2.ts

generated
└── templates
        └── views
            └── template1.ts (imports './view2')
```

Files in `src/views` are user code for some UI controls. Files in `generated/templates` are UI template binding code auto-generated by a template generator as part of the build. A build step will copy the files in `/src/views` and `/generated/templates/views` to the same directory in the output. At run-time, a view can expect its template to exist next to it, and thus should import it using a relative name as `"./template"`.

To specify this relationship to the compiler, use`"rootDirs"`. `"rootDirs"` specify a list of *roots* whose contents are expected to merge at run-time. So following our example, the `tsconfig.json` file should look like:

```
{
  "compilerOptions": {
    "rootDirs": [
      "src/views",
      "generated/templates/views"
    ]
  }
}
```

Every time the compiler sees a relative module import in a subfolder of one of the `rootDirs`, it will attempt to look for this import in each of the entries of `rootDirs`.

The flexibility of `rootDirs` is not limited to specifying a list of physical source directories that are logically merged. The supplied array may include any number of ad hoc, arbitrary directory names, regardless of whether they exist or not. This allows the compiler to capture sophisticated bundling and runtime features such as conditional inclusion and project specific loader plugins in a type safe way.

Consider an internationalization scenario where a build tool automatically generates locale specific bundles by interpolating a special path token, say `#{locale}`, as part of a relative module path such as `./#{locale}/messages`. In this hypothetical setup the tool enumerates supported locales, mapping the abstracted path into `./zh/messages`, `./de/messages`, and so forth.

Assume that each of these modules exports an array of strings. For example `./zh/messages` might contain:

```
export default [
    "您好吗",
    "很高兴认识你"
];
```

By leveraging `rootDirs` we can inform the compiler of this mapping and thereby allow it to safely resolve `./#{locale}/messages`, even though the directory will never exist. For example, with the following `tsconfig.json`:

```
{
  "compilerOptions": {
    "rootDirs": [
      "src/zh",
      "src/de",
      "src/#{locale}"
    ]
  }
}
```

The compiler will now resolve `import messages from './#{locale}/messages'` to `import messages from './zh/messages'` for tooling purposes, allowing development in a locale agnostic manner without compromising design time support.

Tracing module resolution

As discussed earlier, the compiler can visit files outside the current folder when resolving a module. This can be hard when diagnosing why a module is not resolved, or is resolved to an incorrect definition. Enabling the compiler module resolution tracing using `--traceResolution` provides insight in what happened during the module resolution process.

Let's say we have a sample application that uses the `typescript` module. `app.ts` has an import like `import * as ts from "typescript"`.

```
│   tsconfig.json
├───node_modules
│   └───typescript
│       └───lib
│               typescript.d.ts
└───src
        app.ts
```

Invoking the compiler with `--traceResolution`

```
tsc --traceResolution
```

Results in an output such as:

```
======== Resolving module 'typescript' from 'src/app.ts'. ========
Module resolution kind is not specified, using 'NodeJs'.
Loading module 'typescript' from 'node_modules' folder.
File 'src/node_modules/typescript.ts' does not exist.
File 'src/node_modules/typescript.tsx' does not exist.
File 'src/node_modules/typescript.d.ts' does not exist.
File 'src/node_modules/typescript/package.json' does not exist.
File 'node_modules/typescript.ts' does not exist.
File 'node_modules/typescript.tsx' does not exist.
File 'node_modules/typescript.d.ts' does not exist.
Found 'package.json' at 'node_modules/typescript/package.json'.
'package.json' has 'types' field './lib/typescript.d.ts' that references
'node_modules/typescript/lib/typescript.d.ts'.
File 'node_modules/typescript/lib/typescript.d.ts' exist - use it as a module
resolution result.
======== Module name 'typescript' was successfully resolved to
'node_modules/typescript/lib/typescript.d.ts'. ========
```

Things to look out for

- Name and location of the import

> ======== Resolving module **'typescript'** from **'src/app.ts'**. ========

- The strategy the compiler is following

> Module resolution kind is not specified, using **'NodeJs'**.

- Loading of types from npm packages

> 'package.json' has **'types'** field './lib/typescript.d.ts' that references 'node_modules/typescript/lib/typescript.d.ts'.

- Final result

> ======== Module name 'typescript' was **successfully resolved** to 'node_modules/typescript/lib/typescript.d.ts'. ========

Using `--noResolve`

Normally the compiler will attempt to resolve all module imports before it starts the compilation process. Every time it successfully resolves an `import` to a file, the file is added to the set of files the compiler will process later on.

The `--noResolve` compiler options instructs the compiler not to "add" any files to the compilation that were not passed on the command line. It will still try to resolve the module to files, but if the file is not specified, it will not be included.

For instance:

app.ts

```
import * as A from "moduleA" // OK, 'moduleA' passed on the command-line
import * as B from "moduleB" // Error TS2307: Cannot find module 'moduleB'.
```

```
tsc app.ts moduleA.ts --noResolve
```

Compiling `app.ts` using `--noResolve` should result in:

- Correctly finding `moduleA` as it was passed on the command-line.
- Error for not finding `moduleB` as it was not passed.

Common Questions

Why does a module in the exclude list still get 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. That does not embed module resolution as discussed above. If the compiler identified a file as a target of a module import, it will be included in the compilation regardless if it was excluded in the previous steps.

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

17_Declaration Merging.md

Introduction

Some of the unique concepts in TypeScript describe the shape of JavaScript objects at the type level. One example that is especially unique to TypeScript is the concept of 'declaration merging'. Understanding this concept will give you an advantage when working with existing JavaScript. It also opens the door to more advanced abstraction concepts.

For the purposes of this article, "declaration merging" means that the compiler merges two separate declarations declared with the same name into a single definition. This merged definition has the features of both of the original declarations. Any number of declarations can be merged; it's not limited to just two declarations.

Basic Concepts

In TypeScript, a declaration creates entities in at least one of three groups: namespace, type, or value. Namespace-creating declarations create a namespace, which contains names that are accessed using a dotted notation. Type-creating declarations do just that: they create a type that is visible with the declared shape and bound to the given name. Lastly, value-creating declarations create values that are visible in the output JavaScript.

Declaration Type	Namespace	Type	Value
Namespace	X		X
Class		X	X
Enum		X	X
Interface		X	
Type Alias		X	
Function			X
Variable			X

Understanding what is created with each declaration will help you understand what is merged when you perform a declaration merge.

Merging Interfaces

The simplest, and perhaps most common, type of declaration merging is interface merging. At the most basic level, the merge mechanically joins the members of both declarations into a single interface with the same name.

```
interface Box {
    height: number;
    width: number;
}

interface Box {
    scale: number;
}

let box: Box = {height: 5, width: 6, scale: 10};
```

Non-function members of the interfaces should be unique. If they are not unique, they must be of the same type. The compiler will issue an error if the interfaces both declare a non-function member of the same name, but of different types.

For function members, each function member of the same name is treated as describing an overload of the same function. Of note, too, is that in the case of interface `A` merging with later interface `A`, the second interface will have a higher precedence than the first.

That is, in the example:

```
interface Cloner {
    clone(animal: Animal): Animal;
}

interface Cloner {
    clone(animal: Sheep): Sheep;
}

interface Cloner {
    clone(animal: Dog): Dog;
    clone(animal: Cat): Cat;
}
```

The three interfaces will merge to create a single declaration as so:

```
interface Cloner {
    clone(animal: Dog): Dog;
    clone(animal: Cat): Cat;
    clone(animal: Sheep): Sheep;
    clone(animal: Animal): Animal;
}
```

Notice that the elements of each group maintains the same order, but the groups themselves are merged with later overload sets ordered first.

One exception to this rule is specialized signatures. If a signature has a parameter whose type is a *single* string literal type (e.g. not a union of string literals), then it will be bubbled toward the top of its merged overload list.

For instance, the following interfaces will merge together:

```
interface Document {
    createElement(tagName: any): Element;
}
interface Document {
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
}
interface Document {
    createElement(tagName: string): HTMLElement;
    createElement(tagName: "canvas"): HTMLCanvasElement;
}
```

The resulting merged declaration of `Document` will be the following:

```
interface Document {
    createElement(tagName: "canvas"): HTMLCanvasElement;
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
    createElement(tagName: string): HTMLElement;
    createElement(tagName: any): Element;
}
```

Merging Namespaces

Similarly to interfaces, namespaces of the same name will also merge their members. Since namespaces create both a namespace and a value, we need to understand how both merge.

To merge the namespaces, type definitions from exported interfaces declared in each namespace are themselves merged, forming a single namespace with merged interface definitions inside.

To merge the namespace value, at each declaration site, if a namespace already exists with the given name, it is further extended by taking the existing namespace and adding the exported members of the second namespace to the first.

The declaration merge of `Animals` in this example:

```
namespace Animals {
    export class Zebra { }
}
```

```
namespace Animals {
    export interface Legged { numberOfLegs: number; }
    export class Dog { }
}
```

is equivalent to:

```
namespace Animals {
    export interface Legged { numberOfLegs: number; }

    export class Zebra { }
    export class Dog { }
}
```

This model of namespace merging is a helpful starting place, but we also need to understand what happens with non-exported members. Non-exported members are only visible in the original (un-merged) namespace. This means that after merging, merged members that came from other declarations cannot see non-exported members.

We can see this more clearly in this example:

```
namespace Animal {
    let haveMuscles = true;

    export function animalsHaveMuscles() {
        return haveMuscles;
    }
}

namespace Animal {
    export function doAnimalsHaveMuscles() {
        return haveMuscles;  // Error, because haveMuscles is not accessible here
    }
}
```

Because `haveMuscles` is not exported, only the `animalsHaveMuscles` function that shares the same un-merged namespace can see the symbol. The `doAnimalsHaveMuscles` function, even though it's part of the merged `Animal` namespace can not see this un-exported member.

Merging Namespaces with Classes, Functions, and Enums

Namespaces are flexible enough to also merge with other types of declarations. To do so, the namespace declaration must follow the declaration it will merge with. The resulting declaration has properties of both declaration types. TypeScript uses this capability to model some of the patterns in JavaScript as well as other programming languages.

Merging Namespaces with Classes

This gives the user a way of describing inner classes.

```
class Album {
    label: Album.AlbumLabel;
}
namespace Album {
    export class AlbumLabel { }
}
```

The visibility rules for merged members is the same as described in the 'Merging Namespaces' section, so we must export the `AlbumLabel` class for the merged class to see it. The end result is a class managed inside of another class. You can also use namespaces to add more static members to an existing class.

In addition to the pattern of inner classes, you may also be familiar with JavaScript practice of creating a function and then extending the function further by adding properties onto the function. TypeScript uses declaration merging to build up definitions like this in a type-safe way.

```
function buildLabel(name: string): string {
    return buildLabel.prefix + name + buildLabel.suffix;
}

namespace buildLabel {
    export let suffix = "";
    export let prefix = "Hello, ";
}

console.log(buildLabel("Sam Smith"));
```

Similarly, namespaces can be used to extend enums with static members:

```
enum Color {
    red = 1,
    green = 2,
    blue = 4
}

namespace Color {
    export function mixColor(colorName: string) {
        if (colorName == "yellow") {
            return Color.red + Color.green;
        }
        else if (colorName == "white") {
            return Color.red + Color.green + Color.blue;
        }
        else if (colorName == "magenta") {
            return Color.red + Color.blue;
        }
        else if (colorName == "cyan") {
            return Color.green + Color.blue;
        }
    }
}
```

Disallowed Merges

Not all merges are allowed in TypeScript. Currently, classes can not merge with other classes or with variables. For information on mimicking class merging, see the Mixins in TypeScript section.

Module Augmentation

Although JavaScript modules do not support merging, you can patch existing objects by importing and then updating them. Let's look at a toy Observable example:

```
// observable.js
export class Observable<T> {
    // ... implementation left as an exercise for the reader ...
}

// map.js
import { Observable } from "./observable";
Observable.prototype.map = function (f) {
    // ... another exercise for the reader
}
```

This works fine in TypeScript too, but the compiler doesn't know about `Observable.prototype.map`. You can use module augmentation to tell the compiler about it:

```
// observable.ts stays the same
// map.ts
import { Observable } from "./observable";
declare module "./observable" {
    interface Observable<T> {
        map<U>(f: (x: T) => U): Observable<U>;
    }
}
Observable.prototype.map = function (f) {
    // ... another exercise for the reader
}
```

```
// consumer.ts
import { Observable } from "./observable";
import "./map";
let o: Observable<number>;
o.map(x => x.toFixed());
```

The module name is resolved the same way as module specifiers in `import`/`export`. See Modules for more information. Then the declarations in an augmentation are merged as if they were declared in the same file as the original. However, you can't declare new top-level declarations in the augmentation -- just patches to existing declarations.

Global augmentation

You can also add declarations to the global scope from inside a module:

```
// observable.ts
export class Observable<T> {
    // ... still no implementation ...
}

declare global {
    interface Array<T> {
        toObservable(): Observable<T>;
    }
}

Array.prototype.toObservable = function () {
    // ...
}
```

Global augmentations have the same behavior and limits as module augmentations.

18_JSX.md

Introduction

JSX is an embeddable XML-like syntax. It is meant to be transformed into valid JavaScript, though the semantics of that transformation are implementation-specific. JSX rose to popularity with the React framework, but has since seen other implementations as well. TypeScript supports embedding, type checking, and compiling JSX directly to JavaScript.

Basic usage

In order to use JSX you must do two things.

1. Name your files with a `.tsx` extension
2. Enable the `jsx` option

TypeScript ships with three JSX modes: `preserve`, `react`, and `react-native`. These modes only affect the emit stage - type checking is unaffected. The `preserve` mode will keep the JSX as part of the output to be further consumed by another transform step (e.g. Babel). Additionally the output will have a `.jsx` file extension. The `react` mode will emit `React.createElement`, does not need to go through a JSX transformation before use, and the output will have a `.js` file extension. The `react-native` mode is the equivalent of `preserve` in that it keeps all JSX, but the output will instead have a `.js` file extension.

Mode	Input	Output	Output File Extension
`preserve`	`<div />`	`<div />`	`.jsx`
`react`	`<div />`	`React.createElement("div")`	`.js`
`react-native`	`<div />`	`<div />`	`.js`

You can specify this mode using either the `--jsx` command line flag or the corresponding option in your tsconfig.json file.

> *Note: The identifier `React` is hard-coded, so you must make React available with an uppercase R.*

The `as` operator

Recall how to write a type assertion:

```
var foo = <foo>bar;
```

This asserts the variable `bar` to have the type `foo`. Since TypeScript also uses angle brackets for type assertions, combining it with JSX's syntax would introduce certain parsing difficulties. As a result, TypeScript disallows angle bracket type assertions in `.tsx` files.

Since the above syntax cannot be used in `.tsx` files, an alternate type assertion operator should be used: `as`. The example can easily be rewritten with the `as` operator.

```
var foo = bar as foo;
```

The `as` operator is available in both `.ts` and `.tsx` files, and is identical in behavior to the angle-bracket type assertion style.

Type Checking

In order to understand type checking with JSX, you must first understand the difference between intrinsic elements and value-based elements. Given a JSX expression `<expr />`, `expr` may either refer to something intrinsic to the environment (e.g. a `div` or `span` in a DOM environment) or to a custom component that you've created. This is important for two reasons:

1. For React, intrinsic elements are emitted as strings (`React.createElement("div")`), whereas a component you've created is not (`React.createElement(MyComponent)`).
2. The types of the attributes being passed in the JSX element should be looked up differently. Intrinsic element attributes should be known *intrinsically* whereas components will likely want to specify their own set of attributes.

TypeScript uses the same convention that React does for distinguishing between these. An intrinsic element always begins with a lowercase letter, and a value-based element always begins with an uppercase letter.

Intrinsic elements

Intrinsic elements are looked up on the special interface `JSX.IntrinsicElements`. By default, if this interface is not specified, then anything goes and intrinsic elements will not be type checked. However, if this interface *is* present, then the name of the intrinsic element is looked up as a property on the `JSX.IntrinsicElements` interface. For example:

```
declare namespace JSX {
    interface IntrinsicElements {
        foo: any
    }
}

<foo />; // ok
<bar />; // error
```

In the above example, `<foo />` will work fine but `<bar />` will result in an error since it has not been specified on `JSX.IntrinsicElements`.

> Note: You can also specify a catch-all string indexer on `JSX.IntrinsicElements` as follows:
>
> ```
> declare namespace JSX {
>    interface IntrinsicElements {
>        [elemName: string]: any;
>    }
> }
> ```

Value-based elements

Value based elements are simply looked up by identifiers that are in scope.

```
import MyComponent from "./myComponent";

<MyComponent />; // ok
<SomeOtherComponent />; // error
```

There are two ways to define a value-based element:

1. Stateless Functional Component (SFC)
2. Class Component

Because these two types of value-based elements are indistinguishable from each other in a JSX expression, first TS tries to resolve the expression as Stateless Functional Component using overload resolution. If the process succeeds, then TS finishes resolving the expression to its declaration. If the value fails to resolve as SFC, TS will then try to resolve it as a class component. If that fails, TS will report an error.

Stateless Functional Component

As the name suggests, the component is defined as JavaScript function where its first argument is a `props` object. TS enforces that its return type must be assignable to `JSX.Element`.

```
interface FooProp {
  name: string;
  X: number;
  Y: number;
}

declare function AnotherComponent(prop: {name: string});
function ComponentFoo(prop: FooProp) {
  return <AnotherComponent name={prop.name} />;
}

const Button = (prop: {value: string}, context: { color: string }) => <button>
```

Because an SFC is simply a JavaScript function, function overloads may be used here as well:

```
interface ClickableProps {
  children: JSX.Element[] | JSX.Element
}

interface HomeProps extends ClickableProps {
  home: JSX.Element;
}

interface SideProps extends ClickableProps {
  side: JSX.Element | string;
}

function MainButton(prop: HomeProps): JSX.Element;
function MainButton(prop: SideProps): JSX.Element {
  ...
}
```

Class Component

It is possible to define the type of a class component. However, to do so it is best to understand two new terms: the *element class type* and the *element instance type*.

Given `<Expr />`, the *element class type* is the type of `Expr`. So in the example above, if `MyComponent` was an ES6 class the class type would be that class's constructor and statics. If `MyComponent` was a factory function, the class type would be that function.

Once the class type is established, the instance type is determined by the union of the return types of the class type's construct or call signatures (whichever is present). So again, in the case of an ES6 class, the instance type would be the type of an instance of that class, and in the case of a factory function, it would be the type of the value returned from the function.

```
class MyComponent {
  render() {}
}

// use a construct signature
var myComponent = new MyComponent();

// element class type => MyComponent
// element instance type => { render: () => void }

function MyFactoryFunction() {
  return {
    render: () => {
    }
  }
}

// use a call signature
var myComponent = MyFactoryFunction();

// element class type => FactoryFunction
// element instance type => { render: () => void }
```

The element instance type is interesting because it must be assignable to `JSX.ElementClass` or it will result in an error. By default `JSX.ElementClass` is `{}`, but it can be augmented to limit the use of JSX to only those types that conform to the proper interface.

```
declare namespace JSX {
  interface ElementClass {
    render: any;
  }
}

class MyComponent {
  render() {}
}
function MyFactoryFunction() {
  return { render: () => {} }
}

<MyComponent />; // ok
<MyFactoryFunction />; // ok

class NotAValidComponent {}
function NotAValidFactoryFunction() {
  return {};
}

<NotAValidComponent />; // error
<NotAValidFactoryFunction />; // error
```

Attribute type checking

The first step to type checking attributes is to determine the *element attributes type*. This is slightly different between intrinsic and value-based elements.

For intrinsic elements, it is the type of the property on `JSX.IntrinsicElements`

```
declare namespace JSX {
  interface IntrinsicElements {
    foo: { bar?: boolean }
  }
}

// element attributes type for 'foo' is '{bar?: boolean}'
<foo bar />;
```

For value-based elements, it is a bit more complex. It is determined by the type of a property on the *element instance type* that was previously determined. Which property to use is determined by `JSX.ElementAttributesProperty`. It should be declared with a single property. The name of that property is then used. As of TypeScript 2.8, if `JSX.ElementAttributesProperty` is not provided, the type of first parameter of the class element's constructor or SFC's call will be used instead.

```
declare namespace JSX {
  interface ElementAttributesProperty {
    props; // specify the property name to use
  }
}
```

```
class MyComponent {
  // specify the property on the element instance type
  props: {
    foo?: string;
  }
}

// element attributes type for 'MyComponent' is '{foo?: string}'
<MyComponent foo="bar" />
```

The element attribute type is used to type check the attributes in the JSX. Optional and required properties are supported.

```
declare namespace JSX {
  interface IntrinsicElements {
    foo: { requiredProp: string; optionalProp?: number }
  }
}

<foo requiredProp="bar" />; // ok
<foo requiredProp="bar" optionalProp={0} />; // ok
<foo />; // error, requiredProp is missing
<foo requiredProp={0} />; // error, requiredProp should be a string
<foo requiredProp="bar" unknownProp />; // error, unknownProp does not exist
<foo requiredProp="bar" some-unknown-prop />; // ok, because 'some-unknown-prop'
is not a valid identifier
```

> Note: If an attribute name is not a valid JS identifier (like a `data-*` attribute), it is not considered to be an error if it is not found in the element attributes type.

Additionally, the `JSX.IntrinsicAttributes` interface can be used to specify extra properties used by the JSX framework which are not generally used by the components' props or arguments - for instance `key` in React. Specializing further, the generic `JSX.IntrinsicClassAttributes<T>` type may also be used to specify the same kind of extra attributes just for class components (and not SFCs). In this type, the generic parameter corresponds to the class instance type. In React, this is used to allow the `ref` attribute of type `Ref<T>`. Generally speaking, all of the properties on these interfaces should be optional, unless you intend that users of your JSX framework need to provide some attribute on every tag.

The spread operator also works:

```
var props = { requiredProp: "bar" };
<foo {...props} />; // ok

var badProps = {};
<foo {...badProps} />; // error
```

Children Type Checking

In TypeScript 2.3, TS introduced type checking of *children*. *children* is a special property in an *element attributes type* where child *JSXExpression*s are taken to be inserted into the attributes. Similar to how TS uses `JSX.ElementAttributesProperty` to determine the name of *props*, TS uses `JSX.ElementChildrenAttribute` to determine the name of *children* within those props. `JSX.ElementChildrenAttribute` should be declared with a single property.

```
declare namespace JSX {
  interface ElementChildrenAttribute {
    children: {};  // specify children name to use
  }
}
```

```
<div>
  <h1>Hello</h1>
</div>;

<div>
  <h1>Hello</h1>
  World
</div>;

const CustomComp = (props) => <div>props.children</div>
<CustomComp>
  <div>Hello World</div>
  {"This is just a JS expression..." + 1000}
</CustomComp>
```

You can specify the type of *children* like any other attribute. This will override the default type from, eg the React typings if you use them.

```
interface PropsType {
  children: JSX.Element
  name: string
}

class Component extends React.Component<PropsType, {}> {
  render() {
    return (
      <h2>
        {this.props.children}
      </h2>
    )
  }
}
```

```
// OK
<Component>
  <h1>Hello World</h1>
</Component>

// Error: children is of type JSX.Element not array of JSX.Element
<Component>
  <h1>Hello World</h1>
  <h2>Hello World</h2>
</Component>

// Error: children is of type JSX.Element not array of JSX.Element or string.
<Component>
  <h1>Hello</h1>
  World
</Component>
```

The JSX result type

By default the result of a JSX expression is typed as `any`. You can customize the type by specifying the `JSX.Element` interface. However, it is not possible to retrieve type information about the element, attributes or children of the JSX from this interface. It is a black box.

Embedding Expressions

JSX allows you to embed expressions between tags by surrounding the expressions with curly braces (`{ }`).

```
var a = <div>
  {["foo", "bar"].map(i => <span>{i / 2}</span>)}
</div>
```

The above code will result in an error since you cannot divide a string by a number. The output, when using the `preserve` option, looks like:

```
var a = <div>
  {["foo", "bar"].map(function (i) { return <span>{i / 2}</span>; })}
</div>
```

React integration

To use JSX with React you should use the React typings. These typings define the `JSX` namespace appropriately for use with React.

```
/// <reference path="react.d.ts" />

interface Props {
  foo: string;
}

class MyComponent extends React.Component<Props, {}> {
  render() {
    return <span>{this.props.foo}</span>
  }
}

<MyComponent foo="bar" />; // ok
<MyComponent foo={0} />; // error
```

Factory Functions

The exact factory function used by the `jsx: react` compiler option is configurable. It may be set using either the `jsxFactory` command line option, or an inline `@jsx` comment pragma to set it on a per-file basis. For example, if you set `jsxFactory` to `createElement`, `<div />` will emit as `createElement("div")` instead of `React.createElement("div")`.

The comment pragma version may be used like so (in TypeScript 2.8):

```
import preact = require("preact");
/* @jsx preact.h */
const x = <div />;
```

emits as:

```
const preact = require("preact");
const x = preact.h("div", null);
```

The factory chosen will also affect where the `JSX` namespace is looked up (for type checking information) before falling back to the global one. If the factory is defined as `React.createElement` (the default), the compiler will check for `React.JSX` before checking for a global `JSX`. If the factory is defined as `h`, it will check for `h.JSX` before a global `JSX`.

19_Decorators.md

Introduction

With the introduction of Classes in TypeScript and ES6, there now exist certain scenarios that require additional features to support annotating or modifying classes and class members. Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members. Decorators are a stage 2 proposal for JavaScript and are available as an experimental feature of TypeScript.

> NOTE Decorators are an experimental feature that may change in future releases.

To enable experimental support for decorators, you must enable the `experimentalDecorators` compiler option either on the command line or in your `tsconfig.json`:

Command Line:

```
tsc --target ES5 --experimentalDecorators
```

tsconfig.json:

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true
    }
}
```

Decorators

A *Decorator* is a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter. Decorators use the form `@expression`, where `expression` must evaluate to a function that will be called at runtime with information about the decorated declaration.

For example, given the decorator `@sealed` we might write the `sealed` function as follows:

```
function sealed(target) {
    // do something with 'target' ...
}
```

> NOTE You can see a more detailed example of a decorator in Class Decorators, below.

Decorator Factories

If we want to customize how a decorator is applied to a declaration, we can write a decorator factory. A *Decorator Factory* is simply a function that returns the expression that will be called by the decorator at runtime.

We can write a decorator factory in the following fashion:

```
function color(value: string) { // this is the decorator factory
    return function (target) { // this is the decorator
        // do something with 'target' and 'value'...
    }
}
```

> NOTE You can see a more detailed example of a decorator factory in Method Decorators, below.

Decorator Composition

Multiple decorators can be applied to a declaration, as in the following examples:

- On a single line:

  ```
  @f @g x
  ```

- On multiple lines:

  ```
  @f
  @g
  x
  ```

When multiple decorators apply to a single declaration, their evaluation is similar to function composition in mathematics. In this model, when composing functions *f* and *g*, the resulting composite ($f \circ g$)(x) is equivalent to $f(g(x))$.

As such, the following steps are performed when evaluating multiple decorators on a single declaration in TypeScript:

1. The expressions for each decorator are evaluated top-to-bottom.
2. The results are then called as functions from bottom-to-top.

If we were to use decorator factories, we can observe this evaluation order with the following example:

```
function f() {
    console.log("f(): evaluated");
    return function (target, propertyKey: string, descriptor: PropertyDescriptor)
{
        console.log("f(): called");
    }
}

function g() {
    console.log("g(): evaluated");
    return function (target, propertyKey: string, descriptor: PropertyDescriptor)
{
```

```
        console.log("g(): called");
    }
}

class C {
    @f()
    @g()
    method() {}
}
```

Which would print this output to the console:

```
f(): evaluated
g(): evaluated
g(): called
f(): called
```

Decorator Evaluation

There is a well defined order to how decorators applied to various declarations inside of a class are applied:

1. *Parameter Decorators*, followed by *Method*, *Accessor*, or *Property Decorators* are applied for each instance member.
2. *Parameter Decorators*, followed by *Method*, *Accessor*, or *Property Decorators* are applied for each static member.
3. *Parameter Decorators* are applied for the constructor.
4. *Class Decorators* are applied for the class.

Class Decorators

A *Class Decorator* is declared just before a class declaration. The class decorator is applied to the constructor of the class and can be used to observe, modify, or replace a class definition. A class decorator cannot be used in a declaration file, or in any other ambient context (such as on a `declare` class).

The expression for the class decorator will be called as a function at runtime, with the constructor of the decorated class as its only argument.

If the class decorator returns a value, it will replace the class declaration with the provided constructor function.

> NOTE Should you chose to return a new constructor function, you must take care to maintain the original prototype. The logic that applies decorators at runtime will **not** do this for you.

The following is an example of a class decorator (`@sealed`) applied to the `Greeter` class:

```
@sealed
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
```

We can define the `@sealed` decorator using the following function declaration:

```
function sealed(constructor: Function) {
    Object.seal(constructor);
    Object.seal(constructor.prototype);
}
```

When `@sealed` is executed, it will seal both the constructor and its prototype.

Next we have an example of how to override the constructor.

```
function classDecorator<T extends {new(...args:any[]):{}}>(constructor:T) {
    return class extends constructor {
        newProperty = "new property";
        hello = "override";
    }
}

@classDecorator
class Greeter {
    property = "property";
    hello: string;
    constructor(m: string) {
        this.hello = m;
    }
}

console.log(new Greeter("world"));
```

Method Decorators

A *Method Decorator* is declared just before a method declaration. The decorator is applied to the *Property Descriptor* for the method, and can be used to observe, modify, or replace a method definition. A method decorator cannot be used in a declaration file, on an overload, or in any other ambient context (such as in a `declare` class).

The expression for the method decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The *Property Descriptor* for the member.

> NOTE The *Property Descriptor* will be `undefined` if your script target is less than `ES5`.

If the method decorator returns a value, it will be used as the *Property Descriptor* for the method.

> NOTE The return value is ignored if your script target is less than `ES5`.

The following is an example of a method decorator (`@enumerable`) applied to a method on the `Greeter` class:

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }

    @enumerable(false)
    greet() {
        return "Hello, " + this.greeting;
    }
}
```

We can define the `@enumerable` decorator using the following function declaration:

```
function enumerable(value: boolean) {
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        descriptor.enumerable = value;
    };
}
```

The `@enumerable(false)` decorator here is a decorator factory. When the `@enumerable(false)` decorator is called, it modifies the `enumerable` property of the property descriptor.

Accessor Decorators

An *Accessor Decorator* is declared just before an accessor declaration. The accessor decorator is applied to the *Property Descriptor* for the accessor and can be used to observe, modify, or replace an accessor's definitions. An accessor decorator cannot be used in a declaration file, or in any other ambient context (such as in a `declare` class).

> NOTE TypeScript disallows decorating both the `get` and `set` accessor for a single member. Instead, all decorators for the member must be applied to the first accessor specified in document order. This is because decorators apply to a *Property Descriptor*, which combines both the `get` and `set` accessor, not each declaration separately.

The expression for the accessor decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The *Property Descriptor* for the member.

> NOTE The *Property Descriptor* will be `undefined` if your script target is less than `ES5`.

If the accessor decorator returns a value, it will be used as the *Property Descriptor* for the member.

> NOTE The return value is ignored if your script target is less than `ES5`.

The following is an example of an accessor decorator (`@configurable`) applied to a member of the `Point` class:

```
class Point {
    private _x: number;
    private _y: number;
    constructor(x: number, y: number) {
        this._x = x;
        this._y = y;
    }

    @configurable(false)
    get x() { return this._x; }

    @configurable(false)
    get y() { return this._y; }
}
```

We can define the `@configurable` decorator using the following function declaration:

```
function configurable(value: boolean) {
    return function (target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
        descriptor.configurable = value;
    };
}
```

Property Decorators

A *Property Decorator* is declared just before a property declaration. A property decorator cannot be used in a declaration file, or in any other ambient context (such as in a `declare` class).

The expression for the property decorator will be called as a function at runtime, with the following two arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.

> NOTE A *Property Descriptor* is not provided as an argument to a property decorator due to how property decorators are initialized in TypeScript. This is because there is currently no mechanism to describe an instance property when defining members of a prototype, and no way to observe or modify the initializer for a property. The return value is ignored too. As such, a property decorator can only be used to observe that a property of a specific name has been declared for a class.

We can use this information to record metadata about the property, as in the following example:

```
class Greeter {
    @format("Hello, %s")
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        let formatString = getFormat(this, "greeting");
        return formatString.replace("%s", this.greeting);
    }
}
```

We can then define the `@format` decorator and `getFormat` functions using the following function declarations:

```
import "reflect-metadata";

const formatMetadataKey = Symbol("format");

function format(formatString: string) {
    return Reflect.metadata(formatMetadataKey, formatString);
}

function getFormat(target: any, propertyKey: string) {
    return Reflect.getMetadata(formatMetadataKey, target, propertyKey);
}
```

The `@format("Hello, %s")` decorator here is a decorator factory. When `@format("Hello, %s")` is called, it adds a metadata entry for the property using the `Reflect.metadata` function from the `reflect-metadata` library. When `getFormat` is called, it reads the metadata value for the format.

> NOTE This example requires the `reflect-metadata` library. See Metadata for more information about the `reflect-metadata` library.

Parameter Decorators

A *Parameter Decorator* is declared just before a parameter declaration. The parameter decorator is applied to the function for a class constructor or method declaration. A parameter decorator cannot be used in a declaration file, an overload, or in any other ambient context (such as in a `declare` class).

The expression for the parameter decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The ordinal index of the parameter in the function's parameter list.

> NOTE A parameter decorator can only be used to observe that a parameter has been declared on a method.

The return value of the parameter decorator is ignored.

The following is an example of a parameter decorator (`@required`) applied to parameter of a member of the `Greeter` class:

```
class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    @validate
    greet(@required name: string) {
        return "Hello " + name + ", " + this.greeting;
    }
}
```

We can then define the `@required` and `@validate` decorators using the following function declarations:

```
import "reflect-metadata";

const requiredMetadataKey = Symbol("required");

function required(target: Object, propertyKey: string | symbol, parameterIndex:
number) {
    let existingRequiredParameters: number[] =
Reflect.getOwnMetadata(requiredMetadataKey, target, propertyKey) || [];
    existingRequiredParameters.push(parameterIndex);
    Reflect.defineMetadata(requiredMetadataKey, existingRequiredParameters,
target, propertyKey);
}

function validate(target: any, propertyName: string, descriptor:
TypedPropertyDescriptor<Function>) {
    let method = descriptor.value;
    descriptor.value = function () {
        let requiredParameters: number[] =
Reflect.getOwnMetadata(requiredMetadataKey, target, propertyName);
        if (requiredParameters) {
            for (let parameterIndex of requiredParameters) {
                if (parameterIndex >= arguments.length ||
arguments[parameterIndex] === undefined) {
                    throw new Error("Missing required argument.");
                }
            }
        }

        return method.apply(this, arguments);
    }
}
```

The `@required` decorator adds a metadata entry that marks the parameter as required. The `@validate` decorator then wraps the existing `greet` method in a function that validates the arguments before invoking the original method.

> NOTE This example requires the `reflect-metadata` library. See Metadata for more information about the `reflect-metadata` library.

Metadata

Some examples use the `reflect-metadata` library which adds a polyfill for an experimental metadata API. This library is not yet part of the ECMAScript (JavaScript) standard. However, once decorators are officially adopted as part of the ECMAScript standard these extensions will be proposed for adoption.

You can install this library via npm:

```
npm i reflect-metadata --save
```

TypeScript includes experimental support for emitting certain types of metadata for declarations that have decorators. To enable this experimental support, you must set the `emitDecoratorMetadata` compiler option either on the command line or in your `tsconfig.json`:

Command Line:

```
tsc --target ES5 --experimentalDecorators --emitDecoratorMetadata
```

tsconfig.json:

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true,
        "emitDecoratorMetadata": true
    }
}
```

When enabled, as long as the `reflect-metadata` library has been imported, additional design-time type information will be exposed at runtime.

We can see this in action in the following example:

```
import "reflect-metadata";

class Point {
    x: number;
    y: number;
}

class Line {
    private _p0: Point;
    private _p1: Point;

    @validate
    set p0(value: Point) { this._p0 = value; }
    get p0() { return this._p0; }

    @validate
    set p1(value: Point) { this._p1 = value; }
    get p1() { return this._p1; }
}

function validate<T>(target: any, propertyKey: string, descriptor:
TypedPropertyDescriptor<T>) {
    let set = descriptor.set;
    descriptor.set = function (value: T) {
        let type = Reflect.getMetadata("design:type", target, propertyKey);
        if (!(value instanceof type)) {
            throw new TypeError("Invalid type.");
        }
        set(value);
    }
}
```

The TypeScript compiler will inject design-time type information using the `@Reflect.metadata` decorator. You could consider it the equivalent of the following TypeScript:

```
class Line {
    private _p0: Point;
    private _p1: Point;
    @validate
    @Reflect.metadata("design:type", Point)
    set p0(value: Point) { this._p0 = value; }
    get p0() { return this._p0; }

    @validate
    @Reflect.metadata("design:type", Point)
    set p1(value: Point) { this._p1 = value; }
    get p1() { return this._p1; }
}
```

> NOTE Decorator metadata is an experimental feature and may introduce breaking changes in future releases.

20_Mixins.md

Introduction

Along with traditional OO hierarchies, another popular way of building up classes from reusable components is to build them by combining simpler partial classes. You may be familiar with the idea of mixins or traits for languages like Scala, and the pattern has also reached some popularity in the JavaScript community.

Mixin sample

In the code below, we show how you can model mixins in TypeScript. After the code, we'll break down how it works.

```
// Disposable Mixin
class Disposable {
    isDisposed: boolean;
    dispose() {
        this.isDisposed = true;
    }

}

// Activatable Mixin
class Activatable {
    isActive: boolean;
    activate() {
        this.isActive = true;
    }
    deactivate() {
        this.isActive = false;
    }
}

class SmartObject implements Disposable, Activatable {
    constructor() {
        setInterval(() => console.log(this.isActive + " : " + this.isDisposed),
500);
    }

    interact() {
        this.activate();
    }

    // Disposable
    isDisposed: boolean = false;
    dispose: () => void;
    // Activatable
    isActive: boolean = false;
    activate: () => void;
    deactivate: () => void;
}
applyMixins(SmartObject, [Disposable, Activatable]);

let smartObj = new SmartObject();
setTimeout(() => smartObj.interact(), 1000);

////////////////////////////////////////
// In your runtime library somewhere
////////////////////////////////////////

function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        });
    });
}
```

Understanding the sample

The code sample starts with the two classes that will act as our mixins. You can see each one is focused on a particular activity or capability. We'll later mix these together to form a new class from both capabilities.

```
// Disposable Mixin
class Disposable {
    isDisposed: boolean;
    dispose() {
        this.isDisposed = true;
    }

}
```

```
// Activatable Mixin
class Activatable {
    isActive: boolean;
    activate() {
        this.isActive = true;
    }
    deactivate() {
        this.isActive = false;
    }
}
```

Next, we'll create the class that will handle the combination of the two mixins. Let's look at this in more detail to see how it does this:

```
class SmartObject implements Disposable, Activatable {
```

The first thing you may notice in the above is that instead of using `extends`, we use `implements`. This treats the classes as interfaces, and only uses the types behind Disposable and Activatable rather than the implementation. This means that we'll have to provide the implementation in class. Except, that's exactly what we want to avoid by using mixins.

To satisfy this requirement, we create stand-in properties and their types for the members that will come from our mixins. This satisfies the compiler that these members will be available at runtime. This lets us still get the benefit of the mixins, albeit with some bookkeeping overhead.

```
// Disposable
isDisposed: boolean = false;
dispose: () => void;
// Activatable
isActive: boolean = false;
activate: () => void;
deactivate: () => void;
```

Finally, we mix our mixins into the class, creating the full implementation.

```
applyMixins(SmartObject, [Disposable, Activatable]);
```

Lastly, we create a helper function that will do the mixing for us. This will run through the properties of each of the mixins and copy them over to the target of the mixins, filling out the stand-in properties with their implementations.

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        });
    });
}
```

21_Triple-Slash Directives.md

Triple-slash directives are single-line comments containing a single XML tag. The contents of the comment are used as compiler directives.

Triple-slash directives are **only** valid at the top of their containing file. A triple-slash directive can only be preceded by single or multi-line comments, including other triple-slash directives. If they are encountered following a statement or a declaration they are treated as regular single-line comments, and hold no special meaning.

`/// <reference path="..." />`

The `/// <reference path="..." />` directive is the most common of this group. It serves as a declaration of *dependency* between files.

Triple-slash references instruct the compiler to include additional files in the compilation process.

They also serve as a method to order the output when using `--out` or `--outFile`. Files are emitted to the output file location in the same order as the input after preprocessing pass.

Preprocessing input files

The compiler performs a preprocessing pass on input files to resolve all triple-slash reference directives. During this process, additional files are 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 preprocessed in the same order they are specified. Before a file is added to the list, all triple-slash references in it are processed, and their targets included. Triple-slash references are resolved in a depth first manner, in the order they have been seen in the file.

A triple-slash reference path is resolved relative to the containing file, if unrooted.

Errors

It is an error to reference a file that does not exist. It is an error for a file to have a triple-slash reference to itself.

Using `--noResolve`

If the compiler flag `--noResolve` is specified, triple-slash references are ignored; they neither result in adding new files, nor change the order of the files provided.

`/// <reference types="..." />`

Similar to a `/// <reference path="..." />` directive, this directive serves as a declaration of *dependency*; a `/// <reference types="..." />` directive, however, declares a dependency on a package.

The process of resolving these package names is similar to the process of resolving module names in an `import` statement. An easy way to think of triple-slash-reference-types directives are as an `import` for declaration packages.

For example, including `/// <reference types="node" />` in a declaration file declares that this file uses names declared in `@types/node/index.d.ts`; and thus, this package needs to be included in the compilation along with the declaration file.

Use these directives only when you're authoring a `d.ts` file by hand.

For declaration files generated during compilation, the compiler will automatically add `/// <reference types="..." />` for you; A `/// <reference types="..." />` in a generated declaration file is added *if and only if* the resulting file uses any declarations from the referenced package.

For declaring a dependency on an `@types` package in a `.ts` file, use `--types` on the command line or in your `tsconfig.json` instead. See using `@types`, `typeRoots` and `types` in `tsconfig.json` files for more details.

`/// <reference lib="..." />`

This directive allows a file to explicitly include an existing built-in *lib* file.

Built-in *lib* files are referenced in the same fashion as the `"lib"` compiler option in *tsconfig.json* (e.g. use `lib="es2015"` and not `lib="lib.es2015.d.ts"`, etc.).

For declaration file authors who relay on built-in types, e.g. DOM APIs or built-in JS run-time constructors like `Symbol` or `Iterable`, triple-slash-reference lib directives are the recommended. Previously these .d.ts files had to add forward/duplicate declarations of such types.

For example, adding `/// <reference lib="es2017.string" />` to one of the files in a compilation is equivalent to compiling with `--lib es2017.string`.

```
/// <reference lib="es2017.string" />

"foo".padStart(4);
```

`/// <reference no-default-lib="true"/>`

This directive marks a file as a *default library*. You will see this comment at the top of `lib.d.ts` and its different variants.

This directive instructs the compiler to *not* include the default library (i.e. `lib.d.ts`) in the compilation. The impact here is similar to passing `--noLib` on the command line.

Also note that when passing `--skipDefaultLibCheck`, the compiler will only skip checking files with `/// <reference no-default-lib="true"/>`.

`/// <amd-module />`

By default AMD modules are generated anonymous. This can lead to problems when other tools are used to process the resulting modules, such as bundlers (e.g. `r.js`).

The `amd-module` directive allows passing an optional module name to the compiler:

amdModule.ts

```
///<amd-module name="NamedModule"/>
export class C {
}
```

Will result in assigning the name `NamedModule` to the module as part of calling the AMD `define`:

amdModule.js

```
define("NamedModule", ["require", "exports"], function (require, exports) {
    var C = (function () {
        function C() {
        }
        return C;
    })();
    exports.C = C;
});
```

`/// <amd-dependency />`

> **Note**: this directive has been deprecated. Use `import "moduleName";` statements instead.

`/// <amd-dependency path="x" />` informs the compiler about a non-TS module dependency that needs to be injected in the resulting module's require call.

The `amd-dependency` directive can also have an optional `name` property; this allows passing an optional name for an amd-dependency:

```
/// <amd-dependency path="legacy/moduleA" name="moduleA"/>
declare var moduleA:MyType
moduleA.callStuff()
```

Generated JS code:

```
define(["require", "exports", "legacy/moduleA"], function (require, exports,
moduleA) {
    moduleA.callStuff()
});
```

22_Type Checking JavaScript Files.md

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

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

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

JSDoc types are used for type information

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

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

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

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

You can find the full list of supported JSDoc patterns in the JSDoc support in JavaScript documentation.

Properties are inferred from assignments in class bodies

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

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

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

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

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


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

Constructor functions are equivalent to classes

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

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

CommonJS modules are supported

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

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

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

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

Classes, functions, and object literals are namespaces

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

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

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

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

It can also be used to create simple namespaces:

```
var ns = {}
ns.C = class {
}
```

```
ns.func = function() {
}
```

Other variants are allowed as well:

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

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

Object literals are open-ended

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

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

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

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

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

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

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

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

Function parameters are optional by default

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

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

For instance:

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

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

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

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

sayHello();
```

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

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

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

Unspecified type parameters default to `any`

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

In extends clause:

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

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

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

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

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

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

In JSDoc references

An unspecified type argument in JSDoc defaults to any:

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

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


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

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

In function calls

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

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

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

Supported JSDoc

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

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

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

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

`@type`

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

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

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

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

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

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

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

Note that parentheses are optional for union types.

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

You can specify array types using a variety of syntaxes:

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

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

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

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

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

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

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

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

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

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

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

Other types from Closure also work:

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

Casts

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

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

Import types

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

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

import types can also be used in type alias declarations:

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

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

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

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

`@param` and `@returns`

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

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

Likewise, for the return type of a function:

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

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

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

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

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

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

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

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

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

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

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

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

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

`@template`

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

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

Use comma or multiple tags to declare multiple type parameters:

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

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

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

`@constructor`

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

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

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

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

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

`@this`

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

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

`@extends`

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

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

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

`@enum`

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

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

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

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

More examples

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

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

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

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

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

/**
```

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

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

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

Patterns that are known NOT to be supported

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

```
function aNormalFunction() {

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

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

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

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

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

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

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

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