1. TypeScript Handbook
2. Table of Contents
3. The TypeScript Handbook
4. Basic Types
5. Interfaces
6. Functions
7. Literal Types
8. Unions and Intersection Types
9. Classes
10. Enums
11. Generics

This copy of the TypeScript handbook was generated on Tuesday, June 9, 2020 against commit `d9d3fe` with TypeScript 3.9.

The TypeScript Handbook

About this Handbook

Over 20 years after its introduction to the programming community, JavaScript is now one of the most widespread cross-platform languages ever created. Starting as a small scripting language for adding trivial interactivity to webpages, JavaScript has grown to be a language of choice for both frontend and backend applications of every size. While the size, scope, and complexity of programs written in JavaScript has grown exponentially, the ability of the JavaScript language to express the relationships between different units of code has not. Combined with JavaScript's rather peculiar runtime semantics, this mismatch between language and program complexity has made JavaScript development a difficult task to manage at scale.

The most common kinds of errors that programmers write can be described as type errors: a certain kind of value was used where a different kind of value was expected. This could be due to simple typos, a failure to understand the API surface of a library, incorrect assumptions about runtime behavior, or other errors. The goal of TypeScript is to be a static typechecker for JavaScript programs - in other words, a tool that runs before your code runs (static) and ensure that the types of the program are correct (typechecked).

If you are coming to TypeScript without a JavaScript background, with the intention of TypeScript being your first language, we recommend you first start reading the documentation on JavaScript at the Mozilla Web Docs. If you have experience in other languages, you should be able to pick up JavaScript syntax quite quickly by reading the handbook.

How is this Handbook Structured

The handbook is split into two sections:

- **The Handbook**

  The TypeScript Handbook is intended to be a comprehensive document that explains TypeScript to everyday programmers. You can read the handbook by going from top to bottom in the left-hand navigation.

  You should expect each chapter or page to provide you with a strong understanding of the given concepts. The TypeScript Handbook is not a complete language specification, but it is intended to be a comprehensive guide to all of the language's features and behaviors.

  A reader who completes the walkthrough should be able to:

  - Read and understand commonly-used TypeScript syntax and patterns
  - Explain the effects of important compiler options
  - Correctly predict type system behavior in most cases
  - Write a .d.ts declaration for a simple function, object, or class

  In the interests of clarity and brevity, the main content of the Handbook will not explore every edge case or minutiae of the features being covered. You can find more details on particular concepts in the reference articles.

- **The Handbook Reference**

  The handbook reference is built to provide a richer understanding of how a particular part of TypeScript works. You can read it top-to-bottom, but each section aims to provide a deeper explanation of a single concept - meaning there is no aim for continuity.

Non-Goals

Non-Goals

The Handbook is also intended to be a concise document that can be comfortably read in a few hours. Certain topics won't be covered in order to keep things short.

Specifically, the Handbook does not fully introduce core JavaScript basics like functions, classes, and closures. Where appropriate, we'll include links to background reading that you can use to read up on those concepts.

The Handbook also isn't intended to be a replacement for a language specification. In some cases, edge cases or formal descriptions of behavior will be skipped in favor of high-level, easier-to-understand explanations. Instead, there are separate reference pages that more precisely and formally describe many aspects of TypeScript's behavior. The reference pages are not intended for readers unfamiliar with TypeScript, so they may use advanced terminology or reference topics you haven't read about yet.

Finally, the Handbook won't cover how TypeScript interacts with other tools, except where necessary. Topics like how to configure TypeScript with webpack, rollup, parcel, react, babel, closure, lerna, rush, bazel, preact, vue, angular, svelte, jquery, yarn, or npm are out of scope - you can find these resources elsewhere on the web.

Get Started

Before getting started with Basic Types, we recommend reading one of the following introductory pages. These introductions are intended to highlight key similarities and differences between TypeScript and your favored programming language, and clear up common misconceptions specific to those languages.

- TypeScript for New Programmers
- TypeScript for JavaScript Programmers
- TypeScript for OOP Programmers
- TypeScript for Functional Programmers

Basic Types

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 either floating point values or BigIntegers. These floating point numbers get the type `number`, while BigIntegers get the type `bigint`. 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;
let big: bigint = 100n;
```

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 with a fixed number of elements whose types are 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
```

```
Type 'number' is not assignable to type 'string'.
Type 'string' is not assignable to type 'number'.
```

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

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

```
Property 'substring' does not exist on type 'number'.
```

Accessing an element outside the set of known indices fails with an error:

```
x[3] = "world"; // Error, Property '3' does not exist on type '[string
```

```
Tuple type '[string, number]' of length '2' has no element at index '3'
```

```
console.log(x[5].toString()); // Error, Property '5' does not exist on
```

```
Object is possibly 'undefined'.
Tuple type '[string, number]' of length '2' has no element at index '5'
```

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. However, 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 c

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

```
Property 'toFixed' does not 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 `null` (only if `--strictNullChecks` is not specified, see next section) or `undefined` to them:

```
let unusable: void = undefined;
unusable = null; // OK if `--strictNullChecks` is not given
```

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 `any` and their respective types (the one exception being that `undefined` is also assignable to `void`). 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`.

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

> 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. anything 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
Argument of type '42' is not assignable to parameter of type 'object |
create("string"); // Error
Argument of type '"string"' is not assignable to parameter of type 'obj
create(false); // Error
Argument of type 'false' is not assignable to parameter of type 'object
create(undefined); // Error
Argument of type 'undefined' is not assignable to parameter of type 'ob
```

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 have 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.

Interfaces

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(labeledObj: { label: string }) {
  console.log(labeledObj.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 LabeledValue {
  label: string;
}

function printLabel(labeledObj: LabeledValue) {
  console.log(labeledObj.label);
}

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

```
printLabel(myObj);
```

The interface `LabeledValue` 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: nu
  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: nu
  let newSquare = { color: "white", area: 100 };
  if (config.clor) {
Property 'clor' does not exist on type 'SquareConfig'. Did you mean 'co
    // Error: Property 'clor' does not exist on type 'SquareConfig'
    newSquare.color = config.clor;
Property 'clor' does not exist on type 'SquareConfig'. Did you mean 'co
  }
  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!
```

```
Cannot assign to 'x' because it is a read-only property.
```

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!
```

```
Index signature in type 'readonly number[]' only permits reading.
```

```
ro.push(5); // error!
```

```
Property 'push' does not exist on type 'readonly number[]'.
```

```
ro.length = 100; // error!
```

```
Cannot assign to 'length' because it is a read-only property.
```

```
a = ro; // error!
```

```
The type 'readonly number[]' is 'readonly' and cannot be assigned to th
```

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:

```
let a: number[] = [1, 2, 3, 4];
let ro: ReadonlyArray<number> = a;

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 allow an error to sneak in. For example, taking our last example using `createSquare`:

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

function createSquare(config: SquareConfig): { color: string; area: nu
  return { color: config.color || "red", area: config.width || 20 };
}

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

```
Argument of type '{ colour: string; width: number; }' is not assignable
  Object literal may only specify known properties, but 'colour' does n
```

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:

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

```
Argument of type '{ colour: string; width: number; }' is not assignable
  Object literal may only specify known properties, but 'colour' does n
```

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 SquareConf
```

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);
```

The above workaround will work as long as you have a common property between `squareOptions` and `SquareConfig`. In this example, it was the property `width`. It will however, fail if the variable does not have any common object property. For example:

```
let squareOptions = { colour: "red" };
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`).

```
let mySearch: SearchFunc;

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

Had the function expression returned numbers or strings, the type checker would have made an error that indicates return type doesn't match the return type described in the `SearchFunc` interface.

```
let mySearch: SearchFunc;

mySearch = function (src, sub) {
Type '(src: string, sub: string) => string' is not assignable to type '
  Type 'string' is not assignable to type 'boolean'.
  let result = src.search(sub);
  return "string";
};
```

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.

```
interface Animal {
  name: string;
}

interface Dog extends Animal {
  breed: string;
}

// Error: indexing with a numeric string might get you a completely se
interface NotOkay {
  [x: number]: Animal;
Numeric index type 'Animal' is not assignable to string index type 'Dog
  [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 i
Property 'name' of type 'string' is not assignable to string index type
}
```

However, properties of different types are acceptable if the index signature is a union of the property types:

```
interface NumberOrStringDictionary {
  [index: string]: number | string;
  length: number; // ok, length is a number
  name: string; // ok, name is a string
```

```
}
```

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!
```

```
Index signature in type 'ReadonlyStringArray' only permits reading.
```

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 = new 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): void;
}
```

```
class Clock implements ClockInterface {
  currentTime: Date = new 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 {
Class 'Clock' incorrectly implements interface 'ClockConstructor'.
  Type 'Clock' provides no match for the signature 'new (hour: number,
  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(): void;
}

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.

Another simple way is to use class expressions:

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

interface ClockInterface {
  tick();
}

const Clock: ClockConstructor = class Clock implements ClockInterface
  constructor(h: number, m: number) {}
```

```
  tick() {
    console.log("beep beep");
  }
};
```

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 = {} as 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 = {} as 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 = function (start: number) {} as Counter;
  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() {}
}

class ImageControl implements SelectableControl {
Class 'ImageControl' incorrectly implements interface 'SelectableContro
  Types have separate declarations of a private property 'state'.
  private state: any;
  select() {}
}
```

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). The `ImageControl` class has it's own `state` private member rather than extending `Control`, so it cannot implement `SelectableControl`.

Functions

Introduction

Functions are the fundamental building block of any application 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) is outside of the scope of this article, having a firm understanding how this mechanic works 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

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 even if you only have types on one side of the equation:

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

// myAdd has the full function type
let myAdd: (baseValue: number, increment: number) => number = function
  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

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 para
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 para
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 Sm
let result2 = buildName("Bob", undefined); // still works, also returns
let result3 = buildName("Bob", "Adams", "Sr."); // error, too many para
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 para
let result3 = buildName("Bob", "Adams"); // okay and returns "Bob Adams
let result4 = buildName(undefined, "Adams"); // okay and returns "Will
```

Rest Parameters

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(" ");
}

// employeeName will be "Joseph Samuel Lucas MacKinzie"
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 = buildN
```

`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 ca
    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
  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 runt
    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 use the outer `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): 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 type check, 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.

Literal Types

A literal is a more concrete sub-type of a collective type. What this means is that `"Hello World"` is a `string`, but a `string` is not `"Hello World"` inside the type system.

There are two sets of literal types available in TypeScript today, strings and numbers, by using literal types you can allow an exact value which a string or number must have.

Literal Narrowing

When you declare a variable via `var` or `let`, you are telling the compiler that there is the chance that this variable will change its contents. In contrast, using `const` to declare a variable will inform TypeScript that this object will never change.

```
// We're making a guarantee that this variable
// helloWorld will never change, by using const.

// So, TypeScript sets the type to be "Hello World" not string
const helloWorld = "Hello World";

// On the other hand, a let can change, and so the compiler declares i
let hiWorld = "Hi World";
```

The process of going from an infinite number of potential cases (there are an infinite number of possible string values) to a smaller, finite number of potential case (in `helloWorld`'s case: 1) is called narrowing.

String Literal Types

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 {
      // It's possible that someone could reach this
      // by ignoring your types though.
    }
  }
}

let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy");
```

```
Argument of type '"uneasy"' is not assignable to parameter of type 'Eas
```

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 '"eas
```

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, which act the same as the string literals above.

```
function rollDice(): 1 | 2 | 3 | 4 | 5 | 6 {
  return (Math.floor(Math.random() * 6) + 1) as 1 | 2 | 3 | 4 | 5 | 6;
}

const result = rollDice();
```

A common case for their use is for describing config values:

```
interface MapConfig {
  lng: number;
  lat: number;
  tileSize: 8 | 16 | 32;
}

setupMap({ lng: -73.935242, lat: 40.73061, tileSize: 16 });
```

Unions and Intersection Types

So far, the handbook has covered types which are atomic objects. However, as you model more types you find yourself looking for tools which let you compose or combine existing types instead of creating them from scratch.

Intersection and Union types are one of the ways in which you can compose types.

Union Types

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 si
 * If 'padding' is a number, then that number of spaces is added to th
 */
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` in the above example 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.

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

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 si
 * If 'padding' is a number, then that number of spaces is added to th
 */
function padLeft(value: string, padding: string | number) {
  // ...
}

let indentedString = padLeft("Hello world", true);
Argument of type 'true' is not assignable to parameter of type 'string
```

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`.

Unions with Common Fields

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

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

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

declare function getSmallPet(): Fish | Bird;

let pet = getSmallPet();
```

```
pet.layEggs();

// Only available in one of the two possible types
pet.swim();
```

```
Property 'swim' does not exist on type 'Bird | Fish'.
  Property 'swim' does not exist on type 'Bird'.
```

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.

Discriminating Unions

A common technique for working with unions is to have a single field which uses literal types which you can use to let TypeScript narrow down the possible current type. For example, we're going to create a union of three types which have a single shared field.

```
type NetworkLoadingState = {
  state: "loading";
};

type NetworkFailedState = {
  state: "failed";
  code: number;
};

type NetworkSuccessState = {
  state: "success";
  response: {
    title: string;
    duration: number;
    summary: string;
  };
};

// Create a type which represents only one of the above types
// but you aren't sure which it is yet.
type NetworkState =
  | NetworkLoadingState
```

```
    | NetworkFailedState
    | NetworkSuccessState;
```

All of the above types have a field named `state`, and then they also have their own fields:

NetworkLoadingState	NetworkFailedState	NetworkSuccessState
state	state	state
	code	response

Given the `state` field is common in every type inside `NetworkState` - it is safe for your code to access without an existence check.

With `state` as a literal type, you can compare the value of `state` to the equivalent string and TypeScript will know which type is currently being used.

NetworkLoadingState	NetworkFailedState	NetworkSuccessState
`"loading"`	`"failed"`	`"success"`

In this case, you can use a `switch` statement to narrow down which type is represented at runtime:

```
type NetworkState =
  | NetworkLoadingState
  | NetworkFailedState
  | NetworkSuccessState;

function networkStatus(state: NetworkState): string {
  // Right now TypeScript does not know which of the three
  // potential types state could be.

  // Trying to access a property which isn't shared
  // across all types will raise an error
  state.code;
Property 'code' does not exist on type 'NetworkState'.
  Property 'code' does not exist on type 'NetworkLoadingState'.

  // By switching on state, TypeScript can narrow the union
  // down in code flow analysis
  switch (state.state) {
    case "loading":
      return "Downloading...";
    case "failed":
```

```
      // The type must be NetworkFailedState here,
      // so accessing the `code` field is safe
      return `Error ${state.code} downloading`;
    case "success":
      return `Downloaded ${state.response.title} - ${state.response.su
  }
}
```

Intersection Types

Intersection types are closely related to union types, but they are used very differently. 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 type which is all of `Person` *and* `Serializable` *and* `Loggable`. That means an object of this type will have all members of all three types.

For example, if you had networking requests with consistent error handling then you could separate out the error handling into it's own type which is merged with types which correspond to a single response type.

```
interface ErrorHandling {
  success: boolean;
  error?: { message: string };
}

interface ArtworksData {
  artworks: { title: string }[];
}

interface ArtistsData {
  artists: { name: string }[];
}

// These interfaces are composed to have
// consistent error handling, and their own data.

type ArtworksResponse = ArtworksData & ErrorHandling;
type ArtistsResponse = ArtistsData & ErrorHandling;

const handleArtistsResponse = (response: ArtistsResponse) => {
  if (response.error) {
    console.error(response.error.message);
    return;
  }

  console.log(response.artists);
};
```

Mixins via Intersections

Intersections are used to implement the mixin pattern:

```
class Person {
  constructor(public name: string) {}
}

interface Loggable {
  log(name: string): void;
}

class ConsoleLogger implements Loggable {
  log(name: string) {
    console.log(`Hello, I'm ${name}.`);
  }
}

// Takes two objects and merges them together
function extend<First extends {}, Second extends {}>(
  first: First,
  second: Second
): First & Second {
  const result: Partial<First & Second> = {};
  for (const prop in first) {
    if (first.hasOwnProperty(prop)) {
      (result as First)[prop] = first[prop];
    }
  }
  for (const prop in second) {
    if (second.hasOwnProperty(prop)) {
      (result as Second)[prop] = second[prop];
    }
  }
  return result as First & Second;
}

const jim = extend(new Person("Jim"), ConsoleLogger.prototype);
jim.log(jim.name);
```

Classes

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 can 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.`);
  }
}
```

ECMAScript Private Fields

With TypeScript 3.8, TypeScript supports the new JavaScript syntax for private fields:

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

new Animal("Cat").#name; // Property '#name' is not accessible outside
```

This syntax is built into the JavaScript runtime and can have better guarantees about the isolation of each private field. Right now, the best documentation for these private fields is in the TypeScript 3.8 release notes.

Understanding TypeScript's `private`

TypeScript also has it's own way to declare a member as being 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 compatibl
```

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.departı
  }
}

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.departı
  }
```

```
}

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

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. This is needed in order to have the value of `theName` accessible after the `Octopus` constructor is executed. *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, we may also want enforce some constraints when `fullName` is set.

In this version, we add a setter that checks the length of the `newName` to make sure it's compatible with the max-length of our backing database field. If it isn't we throw an error notifying client code that something went wrong.

To preserve existing functionality, we also add a simple getter that retrieves `fullName` unmodified.

```
const fullNameMaxLength = 10;

class Employee {
  private _fullName: string;

  get fullName(): string {
    return this._fullName;
  }

  set fullName(newName: string) {
```

```
    if (newName && newName.length > fullNameMaxLength) {
      throw new Error("fullName has a max length of " + fullNameMaxLen
    }

    this._fullName = newName;
  }
}

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 length of values, we can attempt to assign a name longer than 10 characters and verify that we get an error.

A couple of things to note about accessors:

First, accessors require you to set the compiler to output ECMAScript 5 or higher. Downleveling 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 cla
}

class AccountingDepartment extends Department {
  constructor() {
    super("Accounting and Auditing"); // constructors in derived class
  }
```

```
  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
department = new Department(); // error: cannot create an instance of a
department = new AccountingDepartment(); // ok to create and assign a r
department.printName();
department.printMeeting();
department.generateReports(); // error: method doesn't exist on declar
```

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()); // "Hello, world"
```

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()); // "Hello, world"
```

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()); // "Hello, there"

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

let greeter2: Greeter = new greeterMaker();
```

```
console.log(greeter2.greet()); // "Hey there!"
```

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 };
```

Enums

Enums are one of the few features TypeScript has which is not a type-level extension of JavaScript.

Enums allow a developer 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! Enum member must have 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.

```ts
// 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.

```ts
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! Type 'ShapeKind.Square' is not ass
  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 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:

```ts
enum E {
  Foo,
  Bar
}

function f(x: E) {
  if (x !== E.Foo || x !== E.Bar) {
    //             ~~~~~~~~~~~
    // Error! This condition will always return 'true' since the types
  }
}
```

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 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

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

can actually be passed around to functions

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

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

Enums at compile time

Even though Enums are real objects that exist at runtime, the `keyof` keyword

works differently than you might expect for typical objects. Instead, use `keyof typeof` to get a Type that represents all Enum keys as strings.

```
enum LogLevel {
  ERROR,
  WARN,
  INFO,
  DEBUG
}

/**
 * This is equivalent to:
 * type LogLevelStrings = 'ERROR' | 'WARN' | 'INFO' | 'DEBUG';
 */
type LogLevelStrings = keyof typeof LogLevel;

function printImportant(key: LogLevelStrings, message: string) {
  const num = LogLevel[key];
  if (num <= LogLevel.WARN) {
    console.log("Log level key is:", key);
    console.log("Log level value is:", num);
    console.log("Log level message is:", message);
  }
}
printImportant("ERROR", "This is a message");
```

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 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.

Generics

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 '
```

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

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, s
  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 t
```

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!
```