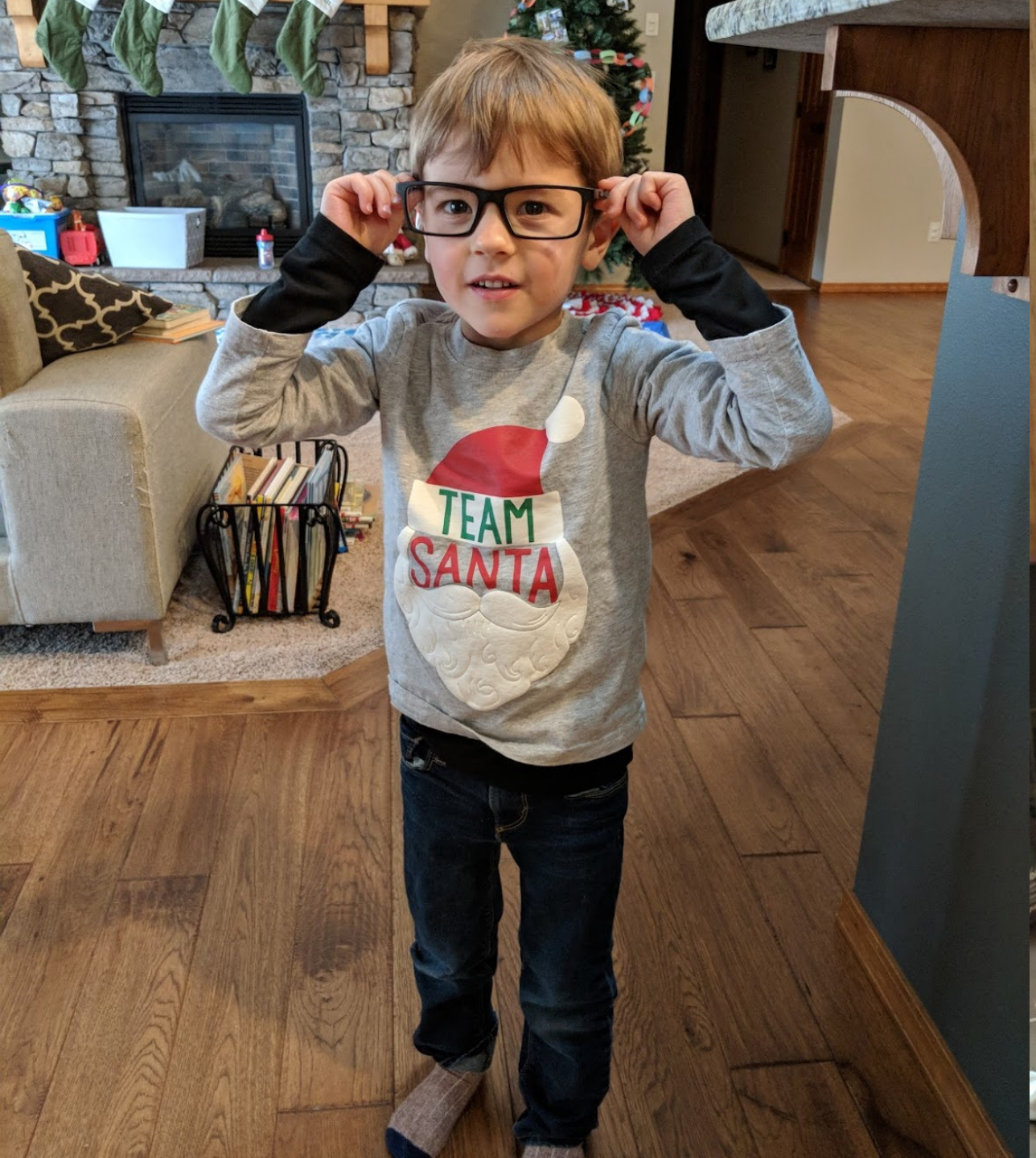

TYPESCRIPT
THE PATH TO BETTER JAVASCRIPT

INTRODUCTION

/usr/bin/whoami

Steve Byerly

WORK:	Senior Software Engineer @ Intellitect
GITHUB:	github.com/SteveByerly
TWITTER:	@SteveByerly
PYTHON SPOKANE:	github.com/python-spokane • @PythonSpokane


TEAM
SANTA


Citrus

Table of Contents

Housekeeping

Questions?

Please Ask!. Might take long questions offline.

Miss Something?

Please let me know. Slides and examples will be available.

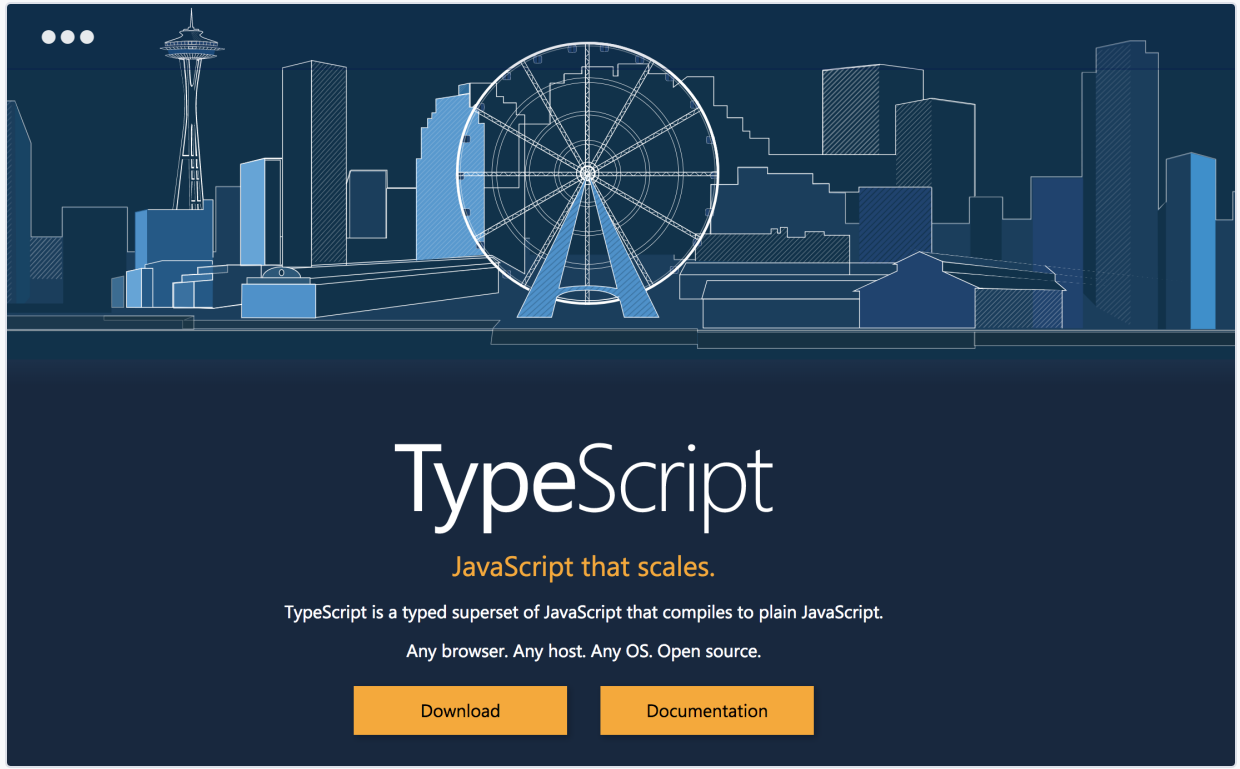
https://github.com/SteveByerly/typescript-demo

do/don't, should/shouldn't, can/can't

My opinions. Do what works best for you.

1 - INTRO TO TYPESCRIPT

BACKGROUND & PURPOSE



What is TypeScript?

More than just JavaScript with types

"JavaScript that scales"

"TypeScript is a typed superset of JavaScript that compiles to plain JavaScript"

"JavaScript that scales."

JavaScript is **hard**

There are many idiosyncrasies to JavaScript. Keeping track of everything when working on your own project is hard enough - exponentially harder with a team.

```
var username,
    totalFunds = 100,
    clickHandler;

var handleUser = function handler(name, debitFunds) {
  var username = name;

  function setupUser(name) {
    clickHandler = userHandler;
    totalFunds = totalFunds - debitFunds;
    username = name;
  }

  setupUser(name);
}

handleUser('steve', 50);
handleUser(50, 'mark');
userHandler();
```

Demo: `./intro/js-is-hard.js`

"TypeScript is a typed superset of JavaScript that compiles to plain JavaScript."

All JavaScript is TypeScript.

TypeScript, the language, is quite similar to plain JavaScript. It changes the more you opt-in to additional features - some of which are not unique to TypeScript.

In the end, your TypeScript is compiled back to JavaScript

Usually better JavaScript than you or I would write

```
// app.js
var username = 'steve';
function sayHi() {
  return 'Hi, ' + username;
}
```

```
// app.ts
var username = 'steve';
function sayHi() {
  return 'Hi, ' + username;
}
```

FEATURES

Features at a glance

Types

- Base

  `boolean, number, string`

  `Object, null, undefined`
- Collections

  `array, enum, "tuple"`
- Special

  `any, void, never`

Components

- Types
- Interfaces
- Enums
- Classes
  - Inheritance
  - Mixins
- Modules
- Namespaces

ES-next Features

- Spread / Destructuring

- Iterators / Generators
- Async / Await
- Async / Dynamic Imports
- Decorators

TypeScipt In-Use

JavaScript

```
var User = (function () {
  function User(username) {
    this.isLoggedIn = true;
    this.username = username;
  }

  User.prototype.logout = function () {
    this.isLoggedIn = false;
    return "Goodbye, " + this.username;
  };

  return User;
}());

var user = new User("steve");
```

TypeScript

```
class User {
  private username: string;
  private isLoggedIn = true;

  constructor(username: string) {
    this.username = username;
  }

  private logout() {
    this.isLoggedIn = false;
    return `Goodbye, ${this.username}`;
  }
}

const user = new User("steve");
```

These are equivalent

THE BAD PARTS

Haters gonna hate

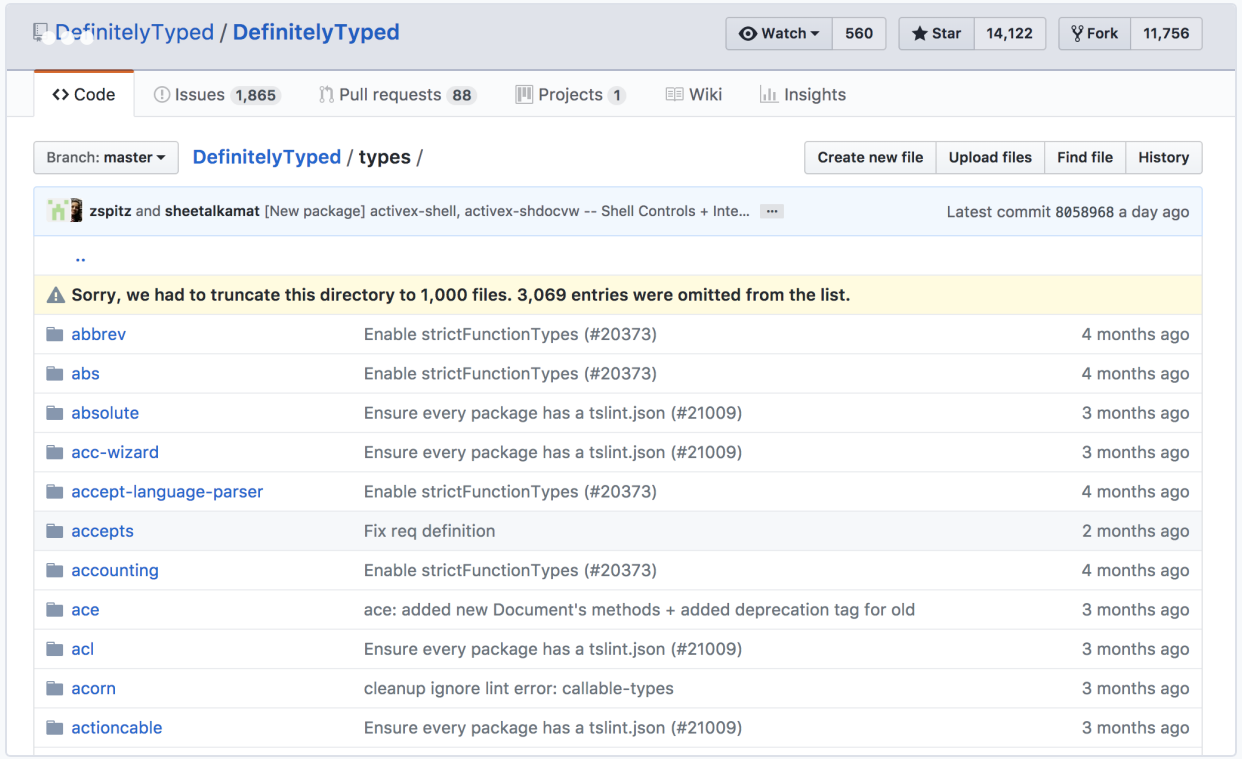
> Typescript is to .NET as Coffeescript was to Ruby.
> A bunch of developers who didn't know JavaScript and wanted to find the closest thing they could to what they already knew.

> I'll keep writing in the language it compiles down to.
> Which would be JavaScript, of course.

> If you like typescript so much you might as well stop pretending you write JavaScript and just write in C# like you really want to.

JS Fatigue ++

- New/Additional Tooling
  - Compilers
  - Linters
  - Build Pipelines
  - IDE configuration
- More dependencies to produce the same output
- Extra effort to keep your project and dependencies updated
- Interoperability with other systems and frameworks



3rd-Party Type Definitions

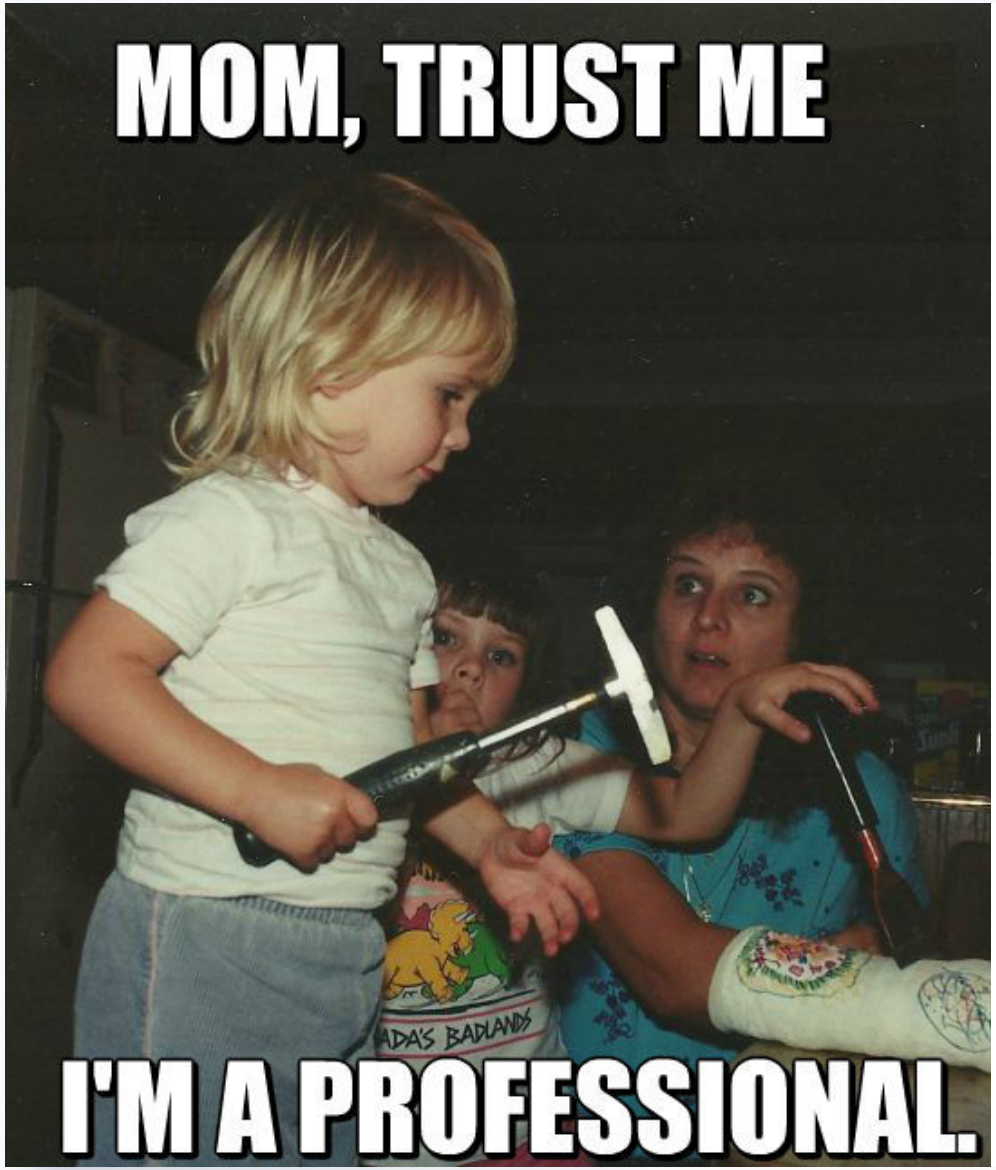
https://github.com/DefinitelyTyped/DefinitelyTyped

- Repo size makes it hard to:
  - Find relative issues/bugs
  - Search through commit history
  - Contribute
- Often difficult to match versions against library versions
- No cler timeline for updates / releases
- Documentation lacking (usually non-existent)

This is seriously awful

Unit Testing

- How to test?
  - Are tests written in TypeScript?
  - Does compilation use the same config as source?
- What to test?
  - Reference your TypeScript files directly?
  - Import your compiled library / app?
- Setup build pipelines for both code and tests
  - Loaders for src and test files
  - Ensure proper sourcemap configuration



Developers

When you tell the compiler "trust me, everything's cool"

...but not your team

...and you don't add test cases

```
const percentUsed = amtRemaining / (<any>initialAmt);
const userName = user.profile!.username.strip();
```

TypeScript provides us a way to avoid many of the pitfalls and unforseen errors in our normal JavaScript.

However...

You still need to be disciplined and maintain good practices, avoiding 'easy' routes as they come up.

USING TYPESCRIPT

INSTALLATION & BASIC USAGE

Installation

*Node.js and NPM are required

Most examples and tutorials have you install TypeScript globally:

```
$ npm install -g typescript
```

Make TypeScript a dev dependency of your project instead:

```
$ npm install --save-dev --save-exact typescript
```

- Version is now specific to your app
- All contributors will use the same version
- Saving exact version helps avoid unexpected changes

Basic Usage

*npx - execute binaries from npm packages

CLI - passing arguments directly

```
$ # global install
$ tsc srcFile.ts --outDir ./dist

$ # local installs
$ npx tsc srcFile.ts --outDir ./dist
$ ./node_modules/.bin/tsc srcFile.ts --outDir ./dist
```

CLI - using a 'project'

```
$ npx tsc -p ../tsconfig.json
```

That's it...

EXTENDED TOOLING

Extended Tooling

IDEs and Plugins

- VS Code
  - First-Class Support!
- Sublime Text
  - w/ TypeScript-Sublime-Plugin
- Atom
  - w/ atom-typescript

Formatters / Linters

- TSLint
  - Also Fixes/Formats!
- Prettier
- Various doc/code generators

Bundlers / Loaders

- ts-node

  Run your node app directly

- ts-loader

  With perf boost:
  - HappyPack
  - fork-ts-checker-webpack-plugin

TYPING EXISTING JAVASCRIPT

```
// @ts-check

/** @type {string} */
var username;

/**
 * @param {string} user
 * @param {boolean} isActive
*/
var login = function(user, isActive) {
  username = user.name;
}
```

Type Checking JS

TypeScript 2.3 added support for type-checking and reporting errors in `.js` files with the `--checkJs` compiler option.

Types can often be inferred by their assignment. When this fails, types can be specified using JSDoc annotations similar to TS annotations.

The TS compiler is surprisingly good at checking JS, but this process is tedious and still prone to error.

Demo: `./intro/js-type-checked.js`

CONFIGURATION & COMPILING TO JAVASCRIPT

```
{
  "compilerOptions": {
    "allowJs": true,
    "baseUrl": "./src",
    "forceConsistentCasingInFileNames": true,
    "lib": ["dom", "es2015.promise", "es5"],
    "module": "esnext",
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "outDir": "./dist/",
    "pretty": true,
    "removeComments": false,
    "sourceMap": true,
    "strict": true,
    "target": "es5"
  },
  "exclude": ["**/*.spec*"],
  "include": ["./src/**/*"]
}
```

Configuration

TypeScript projects use a configuration file (`tsconfig.json`) to determine how the source will be compiled.

Several of these settings will vary based on the type of application.

Recommended Universal Settings:

- strict - strict(null/func/prop) & noImplicit(this/any)
- noUnusedLocals
- noUnusedParameters
- forceConsistentCasingInFileNames

Compiled JavaScript

ES5 `npx tsc --target es5 --removeComments *.ts`

```
var User = (function () {
  function User(username) {
    this.isLoggedIn = true;
    this.username = username;
  }

  User.prototype.logout = function () {
    this.isLoggedIn = false;
    return "Goodbye, " + this.username;
  };

  return User;
}());

var user = new User("steve");
```

ES6 `npx tsc --target es6 *.ts`

```
class User {
  constructor(username) {
    this.isLoggedIn = true;
    this.username = username;
  }

  // This Is Private!!!
  logout() {
    this.isLoggedIn = false;
    return `Goodbye, ${this.username}`;
  }
}

const user = new User("steve");
```

BASIC TYPESCRIPT

TYPES & INTERFACES

```ts
type NumberOrString = number | string;

type Direction = 'north' | 'south' | 'east' | 'west';

type OddHour = 1 | 3 | 5 | 7 | 9 | 11;

type FreeTime = undefined[];

let age: NumberOrString = '42'; // string
age = parseInt(age); // number
age = undefined // error

let name = 'steve';
name = 42; // error
```

Types

Types are the most fundamental and integral parts of TypeScript.

They allow us to define our variables, method parameters, object properties, etc so that we can build code with confidence.

The compiler is smart enough to generate implicit types for those values you do not explicitly type.

Lastly, we can generate our own types whenever & wherever they're needed.

Demo: `./basics/types.ts`

```ts
interface Student {
  name: string;
  studentId: number;
}

interface Employee {
  name: string;
  employeeId?: string; // Optional
}

interface Immutable {
  readonly name: string; // cannot re-set
  readonly studentId: number;
}
```

Interfaces

Interfaces are one of the most powerful features of TypeScript.

They provide two key benefits:

- Define the shape of our data without an implementation
- Provide a contract for our code and consumers of our code

❗ One very important difference from other interface implementations (e.g. c#) is that you do not need to explicitly implement the interface to adhere to it.

Demo: `./basics/interfaces.ts`

DESTRUCTURING & SPREAD

Destructuring

1. Taking elements from an array **by position** and re-assigning them to new variables.
2. Taking properties from an object **by name** and re-assigning them to new variables.

```
// Arrays
const heroes = ['iron man', 'superman', 'spiderman'];

const bestHero = heroes[0];
const mehHero = heroes[1];

// Objects
const hero = { name: 'batman', ability: 'rich guy' };

const name = hero.name;
const ability = hero.ability;
```

```
// Arrays
const heroes = ['iron man', 'superman', 'spiderman'];

const [bestHero, mehHero, worstHero] = heroes;

// Objects
const hero = { name: 'batman', ability: 'rich guy' };

const { name, ability } = hero;
```

```
// Arrays
const good = ['iron man', 'superman', 'spiderman'];
const bad = ['magneto', 'bane'];

const all = [...good, ...bad];

// Objects
const hero = { name: 'batman', ability: 'rich guy' };

const newHero = { ...hero, ability: 'super strength' };

const heroCopy = { ...hero }; // heroCopy != hero;
```

Spread Operator

The (`...`) spread operator allows you to 'spread' an array into another array, or an object into another object.

While the spread operator makes it easier to create new variables from existing variables, it has a very important benefit:

Immutability

Demo: `./basics/spread.ts`

CLASSES

```
class User extends Person implements Loggable {
  private username: string;
  private isLoggedIn = true;

  constructor(username: string) {
    this.username = username;
  }

  private logout() {
    this.isLoggedIn = false;
    return `Goodbye, ${this.username}`;
  }
}

const user = new User("steve");
```

Classes

Classes were a pseudo-construct in JavaScript, but have finally made their way as first-class JavaScript citizens (for better or worse).

Classes:

- Encapsulate logic and 'instance' values
- Can be used as both a value and type
- Can be inherited, mixed-in, abstract

ADVANCED TYPESCRIPT

SET THEORY AND ENUMS

```ts
interface Person {
  firstName: string;
  lastName: string;
}

interface SchoolInfo {
  studentId: number;
}

type Student = Person & SchoolInfo;

const getStudent = (p: Person, s: SchoolInfo): Student => {
  return { ...p, ...s };
}
```

Intersection Types

An intersection type combines multiple types into one.

This allows you to add together existing types to represent a single type that has all the features you need.

Uses:
Most commonly used for composition.
e.g. Generic functions that take multiple arguments and combine them into one entity.

Demo: `./advanced/intersection.ts`

```ts
interface Student {
  name: string;
  studentId: number;
}

interface Employee {
  name: string;
  employeeId: string;
}

type Person = Student | Employee;

const formatName = (person: Person): string => {
  return person.name.trim().split(',').join(' ');
}
```

Union Types

Allow you to specify that a value will be one of several types and contain those properties that are common to both.

You can both decrease the specificity of your function arguments and narrow their scope to use only those features necessary.

Uses:

- Handle similar, but un-related implementations
- Operate on variables that can take multiple value types
  e.g. numberic input changeHandler

Demo: `./sets/union.ts`

GENERICS AND PARTIALS

```ts
interface Dictionary<T> {
  [key: string]: T;
}

type Ranking = Dictionary<number>;

const rankings: Ranking = {
  'gold': 1,
  'silver': 2,
  'bronze': '3', // error
};

const sortBy = <TEntity>(entities: TEntity[]) => {
  return entities.sort();
};
```

Generics

Generics allow you to create dynamic components that can work over a variety of types.

This allows you, and potential external parties, to use these components in a way that is simpler and more expressive.

They also provide a way to create type-factories to generate reusable pieces of data structure, or rules for how you can interact with your objects.

Demo: `./advanced/generics.ts`

```ts
type Partial<T> = {
  [P in keyof T]?: T[P];
}

interface User {
  name: string;
  id: number;
}

const user1: User = {
  name: 'steve',
  id: 1, // error
};
```

Partials

Partials are a form of generics that alter an existing type.

They provide a mechanism for quickly altering existing structures with well-defined, new behaviors and can also supplement existing structures with new properties.

Demo: `./advanced/partials.ts`

TYPE NARROWING & TYPE GUARDS

```ts
const isString = (value: any): value is string => {
  if (!value) {
    return false;
  }

  return typeof value === 'string'
      || value instanceof String;
};

const toNumber = (value: string | number): number => {
  if (isString(value)) {
    return parseInt(value, 10);
  }

  return value;
}
```

Type Narrowing

A type guard is some expression that performs a runtime check that guarantees the type in some scope.

To define a type guard, you define a function whose return type is a type predicate.

Demo: `./advanced/type-narrowing.ts`


QUESTIONS