TypeScript

and ES Modules

Risks of Dynamic Weak Type Checking with JavaScript

1. Surprising dynamic type behaviour
   - re-assignment to different types with no error!
   - operations assuming types work fine or "silently" fail!
2. Typo in function or method name
   - won't know until run time!

```
Uncaught ReferenceError: foo is not defined
```

3. Invalid function arguments
   - no checks on number or type of arguments!
4. Typo in property name
   - creates new property instead of failing!
5. Function doesn't always return a value
   - returns undefined in some cases!

... also uncalled functions, unreachable code, unused variables or parameters, switch statement fall through, and many more!

https://www.meziantou.net/detect-common-javascript-errors-with-typescript.htm

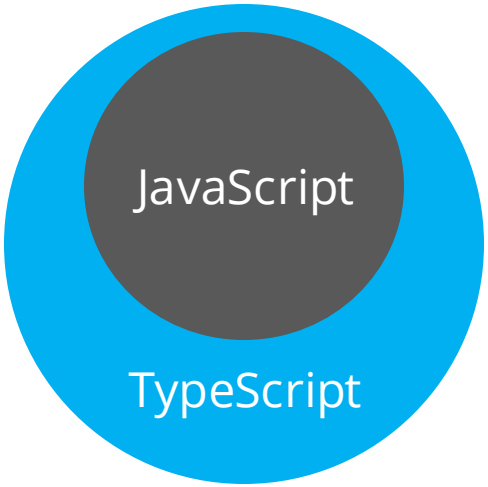
TypeScript

"JavaScript with Type Checking"

- TypeScript is a *superset* of JavaScript

- Developed by Microsoft starting in 2012
  - Led by Anders Hejlsberg (creator of C# and Turbo Pascal)
  - Took off in 2014 (when Angular 2 chose TypeScript)

- Main feature is adding **static types**
  - enables type checking
  - enables code completion



TypeScript "in" the Browser

- TypeScript is *transpiled* to JavaScript
  - remember, the browser only executes JavaScript
- Our Vite setup lets you debug TypeScript source
  - but you're debugging JavaScript from a TypeScript "source map"
- Most TypeScript tutorials show how to install the tsc compiler
  - Our Vite dev environment already transpiles to TypeScript, there is no need to install/call tsc yourself

Types

- TypeScript types are **annotations**
  - checked at "compile" time, *not* run time
- Examples

```
let a = 123;
a = a + "hello";
```

```
Type 'string' is not assignable to
type 'number'.ts(2322)
```

```
const obj = { a: 1, b: 2 };
obj.aa = 123;
```

```
Property 'aa' does not exist on type
'{ a: number; b: number; }'.ts(2339)
```

Primitive Types

- Explicit type annotation

```
let n: number = 123;
let b: boolean = true;
let s: string = "Hello";
```

- Implicit type inference

```
let n = 123;
let b = true;
let s = "Hello";
```

- **Caution**: implicit type inference can result in the **any** type

```
let x;
x = 123;
x = x + "ABC";
```

I want x to be a number, so this is a bug

you really want to avoid the **any** type

Array, Object, and Function Types

- Array
  - two equivalent type declarations

```
let arr1: number[];
let arr2: Array<number>; // generics!
arr1 = [1, 2, 3];
```

- Object

```
let obj: { a: number; b: string };
obj = { a: 1, b: "hello" };
```

- Function

```
function add(a: number, b: number): number {
  return a + b;
}
```

TypeScript has a **void** type for functions that don't return anything

Type Aliases

- Can define type annotation to re-use
- Array type alias example:
  - `type NumberArray = number[];`
- Function type alias example:
  - `type SimpleCallback = (data: string) => void;`
- Object type alias examples:

```
type Point = { x: number; y: number };
type Shape = { colour: string };
```

the convention is CamelCase for type aliases

Interfaces

- Alternate way to define *object types*, e.g.:

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

- An interface can be extended, for example:

```
interface Shape { colour: string };

interface Square extends Shape { size: number };

let square = { colour: "red", size: 10 } as Square;
```

a "type assertion"

https://blog.logrocket.com/types-vs-interfaces-typescript/

Intersection Type

- Creates new type by combining multiple existing types
  - new type has *all* features of the existing types
  - usually used with object types
  - equivalent to “extends” for interfaces
- For example:

```
type Shape = { colour: string };
type Square = Shape & { size: number };

let square = { colour: "red", size: 10 } as Square;
```

intersection

https://www.typescriptlang.org/docs/handbook/2/objects.html#intersection-types

Structural Type Checking

- Only the *structure* of a type matters
  - a `Square` has colour and size properties, anything else is irrelevant!

```
function renderSquare(square: Square) {
  console.log(
    `A ${square.colour} square of size  ${square.size}`
  );
}

let square = { colour: "red", size: 10 };
renderSquare(square); // still works!

let square2 = { colour: "blue", size: 20, alpha: 0.5 };
renderSquare(square2); // also works!
```

no type assertion!

an extra property that isn't in Square!

Union Types

- A union type describes a value that can be one of several types
- For example:

```ts
type NumberOrString = number | string;
```

union

- Commonly used in function arguments:

```ts
function printId(id: number | string) {
  console.log(`Your ID is: ${id}`);
}

printId(101); // ok
printId("202"); // ok
printId({ myID: 22342 }); // error!
```

https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#union-types

Type Narrowing

- Guarding is an example of *type narrowing:*

```
const el = document.getElementById("apple");
if (el)
  el.setAttribute("class", "red");
```

truthiness guard

- Type narrowing often needed with union types, e.g.:

```
function formatId(id: number | string): string {
  if (typeof id === "number") {
    return `${id}`;
  } else {
    return id.toUpperCase();
  }
}
```

https://www.typescriptlang.org/docs/handbook/2/narrowing.html

Optional Parameters

- Can specify function parameters as optional with default or ? (same approach applies to objects)
- Type narrowing is usually required

Optional parameter since it has a default value:

```
function add(a: number, b: number, c: number = 0) {
  return a + b + c;
}
```

Optional parameter defined with ?

```
function add(a: number, b: number, c?: number) {
  if (c) {
    return a + b + c;
  } else {
    return a + b;
  }
}
```

Using Object as Function Argument (aka “Props”)

- Defining props object type
- Creating an object with that type
- Destructure local variables from props object
- Using props object as argument to function
- **DEMO:** when all props optional and used as function argument
  - Need to provide default for props argument as empty object {}

Class with Types

```
class Shape {
  colour: string;
  constructor(colour: string) { this.colour = colour; }
  draw() { return `A ${this.colour} shape.`; }
}

class Square extends Shape {
  size: number;
  constructor(colour: string, size: number) {
    super(colour);
    this.size = size;
  }
  draw() {
    return `A ${this.colour} square size ${this.size}`;
  }
}
```

More Class Annotations

- Methods and classes can be **abstract**
- Properties can be **private**, **public**, **protected**
  - but this is only a compile-time annotation
- Public constructor parameters as property assignment shortcut

```
class Shape {
  constructor(public colour: string) {}
  draw() {
    return `A ${this.colour} shape.`;
  }
}
const shape = new Shape("red");
console.log(shape.draw()); // A red shape.
shape.colour = "blue";
console.log(shape.draw()); // A blue shape.
```

means colour is a property

mutate colour property

Property Getters and Setters

- "Hidden" property accessed with getter and/or setter
- Can make property readonly or add side effects when value set

```
class Shape {
  private _colour: string;
  get colour() {
    return this._colour;
  }
  set colour(c: string) {
    this._colour = c;
  }

constructor(colour: string) {
  this._colour = colour;
}
...
```

convention is to use an underscore prefix for "hidden" properties accessed by get/set

Enums versus Literal Types

- TypeScript introduces an enum type

```
enum State {
  Idle,
  Down,
  Up,
}

let state1: State = State.Idle;
```

I tend to use this method

- Same type-safe result using a *union* of **literal string types:**

```
let state2: "idle" | "down" | "up" = "idle";
```

Generics

- Create code that works over a variety of types
- Consider JavaScript “identity function”:

```
function identity(arg) {
  return arg;
}
const n = identity(123);
const s = identity("hello");
```

- Convert it to TypeScript using Generics

Good tutorial

https://www.typescriptlang.org/docs/handbook/2/generics.html

Circumventing TypeScript type safety

WARNING: You should avoid doing these things

- use the **any** type implicitly or explicitly

```
let a: any = 123;
a = a + "hello";
```

- tell the TypeScript compiler to ignore errors

```
let a = 123;
// @ts-ignore
a = a + "hello";
```

- use the non-null assertion operator `(!)`

```
function f(s: string | null) {
  s!.toUpperCase(); // DANGER!
}
```

relevant article

https://hackernoon.com/making-typescript-truly-strongly-typed

Resources for Learning TypeScript

- TypeScript for Java/C# Programmers
  - https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes-oop.html
- The TypeScript Handbook
  - https://www.typescriptlang.org/docs/handbook/intro.html
  - (can skip tsc compiler setup)
- TypeScript Tutorial for Beginners Video (Programming with Mosh)
  - https://youtu.be/d56mG7DezGs?si=hyzCpU2m0eoUqxOp
  - (can skip tsc compiler setup, 7:50 – 23:00)

ES6 Modules

ES Modules

- Use ES module

```
import { ... } from " ... ";
```

- All variables, functions, objects, types are local unless "exported"

```
export function getSecret() { ... }
```

- Exports from multiple files can be consolidated in "index" file

```
// in "index.ts"
export { ... } from " ... ";
```

- Force file to be a module if no imports or exports

```
export {}; // force file to be a module
```

- Load module into HTML document

```
<script type="module" src=" ... "></script>
```

ES Modules (ESM for short) are not the same as CommonJS Module (CJS for short)

https://www.typescriptlang.org/docs/handbook/2/modules.html

ES Module Example

mymodule.ts

function is exported

```
export function getSecret() {
  return s;
}

let s = "secret";
```

not exported, will be private to module

main.ts

relative path

```
import { getSecret } from "./mymodule";
console.log(getSecret());
```

Exercise

1. **Create a ”clean” TypeScript Vite project**
   - Delete everything in main.ts
   - Remove other files used by the “counter” demo
2. **Test out your Vite and VS Code TypeScript environment**
   - In main.ts, console.log some string and check that it appears in your browser console
   - Change the string you logged and check that your browser console updates
   - Create a simple variable with a string type, like `let myvar:string;`
   - Assign a value to it like “hello” and use your variable in the console.log
   - Save the file and check it shows up in your browser console
   - Now assign something other than a string to your variable: you should see red squiggly lines telling you the types are not compatible
3. **Do Sections 2 to 7 and section 8 of** https://www.typescripttutorial.net/
   - Use your own Vite environment to try out code
   - Don’t setup the tsc environment as they demo in Section 1