TypeScript

●●●

Hunter Henrichsen

Software Development Workshop 2022

Agenda

- Setting Up
- Types in JavaScript
- Type Guards
- Generics
- Mapped Types
- Demo: Linked List
- TypeScript’s Type System is Turing Complete

TypeScript

Setting Up

```
>  npm init -y ts-test
>  npm install --save-dev typescript
>  touch tsconfig.json
```

Types in JavaScript

- `number` - any numeric value
- `string` - any text value
- `"Hunter"` - the literal value `"Hunter"`
- `{}` - an empty object
- `any` - any value
- `unknown` - a value with no type

Types in JavaScript | Any

- any means disabling all type checking on a given value
- any is contagious like NaN
- Generally it's good to avoid using any where possible
- Examples:

```
const myValue: any = undefined;
// type: any; no compile error
const ret = myValue.doMethod();
// type: any; no compile error
const other = ret.test;
// type: number; no error
const otherValue: number =
    myValue;
```

```
const myValue = {} as any;
// adding a method
myValue.get =
    (s: string) => `Hi, ${s}`;
// using the method
const greeting =
    myValue.get("Hunter");
```

Types in JavaScript | Unknown

- Instead of any, use unknown
- unknown is the same as saying that there is no type attached to a value
- Use a type guard and unknown instead of any
- Example:

```
const myValue: unknown =
     undefined;
// compile error
const ret = myValue.doMethod();
// compile error
const other = ret.test;
// compile error
const otherValue: number =
    myValue;
```

```
const myValue = {} as unknown;
// compile error
myValue.get =
    (s: string) => `Hi, ${s}`;
// compile error
const greeting =
    myValue.get("Hunter");
```

Types in TypeScript

Type Guards

- Used for objects and interfaces, not classes
- Classes can use the `instanceof` keyword

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

function isPoint(value: unknown): value is Point {
    const unsafeValue: any = value;
    if(typeof unsafeValue != 'object'
       || unsafeValue == undefined) {
        return false;
    }
    if (typeof unsafeValue.x == "number"
        && typeof unsafeValue.y == "number") {
        return true;
    }
    return false;
}
```

Types in JavaScript | Intersection Types

- One of my favorite features
- Allows dynamically extending/augmenting types
- “Intersection” of type members, not of type attributes

```
const myValue = {};
const augmented: typeof myValue & {
    get(name: string): string;
} = {
    ...myValue,
    get: (name: string) => `Hi, ${name}`
}

console.log(augmented.get("Hunter"));
```

Types in TypeScript

Union Types

- Allows accepting multiple types in one function
- “Union” of type members, not of type attributes

```typescript
interface Car {
    drive(environment: RoadEnvironments): void;
    fuel(units: number): void;
}

interface DuneBuggy {
    drive(environment: RoadEnvironments | "Beach"): void;
    fuel(units: number): void;
}

function gasStation(vehicle: Car | DuneBuggy) {
    vehicle.fuel(5);
}
```

Generics

Simple Example

```
function compare(value1: string, value2: string): number;
function compare(value1: number, value2: number): number;
function compare<C extends number | string>(
    value1: C,
    value2: C):
number {
    if (typeof value1 == "number"
        && typeof value2 == "number") {
        return value1 - value2;
    }
    if (typeof value1 == "string"
        && typeof value2 == "string") {
        if (value1 > value2) {
            return 1;
        }
        return value1 == value2 ? 0 : -1;
    }
    throw new Error("invalid comparison");
}
```

Generics

Advanced Example (Part 1)

```
class Key<K extends string, V> {
  private _val: V | undefined;

  constructor(public readonly key: K,
              private readonly _init: (val: string) => V) {}

  public init(s: string) {
    this._val = this._init(s);
  }

  public get value() {
    if (this._val) {
      return this._val;
    }
    throw new Error(
      `Getting value from key '${this.key}' without initializing it`
    );
  }
}
```

Mapped Types

Advanced Example (Part 2)

```
class KeyStore<Keys extends Array<Key<string, unknown>>> {
    private keyStore: {[K in Keys[number]["key"]]: Keys[number]["value"]}

    constructor(keys: Keys, init: Record<Keys[number]["key"], string>) {
        const keyStore: Record<string, unknown> = {};
        for (const key of keys) {
            key.init((init as Record<string, string>)[key.key]);
            keyStore[key.key] = key.value;
        };
        // bad but here for demonstration -- this should be a type guard
        this.keyStore = keyStore
            as {[K in Keys[number]["key"]]: Keys[number]["value"]};
    }

    public get<K extends string, T>(
        key: Key<K, T>
    ): K extends Keys[number]["key"] ? T : undefined {
        return this.keyStore[key.key]
            as K extends Keys[number]["key"] ? T : undefined;
    }
}
```

Demo

Linked List

```
export class LinkedList<ItemType> {
  private _head?: LinkedNode<ItemType>;
  private _tail?: LinkedNode<ItemType>;
  private _length = 0;

  constructor(...items: ItemType[]) {
    this.push(...items);
  }

  public [Symbol.iterator]() {
    let current = this._head;
    return {
      next: () => {
        const res = {
          value: current?.data,
          done: current == undefined
        };
        current = current?.next;
        return res;
      }
    };
  }

  public push(...items: ItemType[]) {
    for (const item of items) {
      this.insertBack(item);
    }
  }

  public unshift(...items: ItemType[]) {
    for (const item of items) {
      this.insertFront(item);
    }
  }
  // click me for the rest of the demo
```

Other Neat Features

- Default Arguments
- Constructor Shorthand

TypeScript's Type System is Turing Complete

```
type StringBool = "true"|"false";


interface AnyNumber { prev?: any, isZero: StringBool };
interface PositiveNumber { prev: any, isZero: "false" };

type IsZero<TNumber extends AnyNumber> = TNumber["isZero"];
type Next<TNumber extends AnyNumber> = { prev: TNumber, isZero: "false" };
type Prev<TNumber extends PositiveNumber> = TNumber["prev"];


type Add<T1 extends AnyNumber, T2> = { "true": T2, "false": Next<Add<Prev<T1>, T2>>
}[IsZero<T1>];

// Computes T1 * T2
type Mult<T1 extends AnyNumber, T2 extends AnyNumber> = MultAcc<T1, T2, _0>;
type MultAcc<T1 extends AnyNumber, T2, TAcc extends AnyNumber> =
        { "true": TAcc, "false": MultAcc<Prev<T1>, T2, Add<TAcc, T2>> }[IsZero<T1>];

type _0 = { isZero: "true" };
type _1 = Next<_0>;
type _2 = Next<_1>;
type _3 = Next<_2>;
type _4 = Next<_3>;
type _5 = Next<_4>;
type _6 = Next<_5>;
type _7 = Next<_6>;
type _8 = Next<_7>;
type _9 = Next<_8>;

type Digits = { 0: _0, 1: _1, 2: _2, 3: _3, 4: _4, 5: _5, 6: _6, 7: _7, 8: _8, 9: _9
};
type Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;
type NumberToType<TNumber extends Digit> = Digits[TNumber]; // I don't know why
typescript complains here.

type _10 = Next<_9>;
type _100 = Mult<_10, _10>;

type Dec2<T2 extends Digit, T1 extends Digit>
    = Add<Mult<_10, NumberToType<T2>>, NumberToType<T1>>;
```