Angular

I Lecture

KindGeek

TypeScript (TS)

- Types
- Interfaces
- Decorators

Types

- `boolean` (true/false)
- `number` integers, floats, `Infinity` and `NaN`
- `string` characters and strings of characters
- `[]` Arrays of other types, like `number[]` or `boolean[]`
- `{}` Object literal
- `undefined` not set

TypeScript also adds

- `enum` enumerations like `{ Red, Blue, Green }`
- `any` use any type
- `void` nothing

KindGeek School

Example

```typescript
let isDone: boolean = false;
let height: number = 6;
let name: string = "bob";
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
enum Color {Red, Green, Blue};
let c: Color = Color.Green;
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean

function showMessage(data: string): void {
  alert(data);
}
showMessage('hello');
```

KindGeek School

Classes

```
class Foo { foo: number; }
class Bar { bar: string; }

class Baz {
  constructor(foo: Foo, bar: Bar) { }
}

let baz = new Baz(new Foo(), new Bar()); // valid
baz = new Baz(new Bar(), new Foo());     // tsc errors


class Person {
  name: string;
  nickName?: string;
}
```

Interfaces

```
interface Callback {
  (error: Error, data: any): void;
}

function callServer(callback: Callback) {
  callback(null, 'hi');
}

callServer((error, data) => console.log(data));  // 'hi'
callServer('hi');                          // tsc error
```

```
interface Action {
  type: string;
}

let a: Action = {
    type: 'literal'
}
```

```
interface PrintOutput {
  (message: string): void;    // common case
  (message: string[]): void;  // less common
case
}

let printOut: PrintOutput = (message) => {
  if (Array.isArray(message)) {
    console.log(message.join(', '));
  } else {
    console.log(message);
  }
}

printOut('hello');       // 'hello'
printOut(['hi', 'bye']); // 'hi, bye'
```

KindGeek School

Decorators

```
class: declare type ClassDecorator = <TFunction extends
Function>(target: TFunction) => TFunction | void;

property: declare type PropertyDecorator = (target: Object,
propertyKey: string | symbol) => void;

method: declare type MethodDecorator = <T>(target: Object,
propertyKey: string | symbol, descriptor: TypedPropertyDescriptor<T>)
=> TypedPropertyDescriptor<T> | void;

parameter: declare type ParameterDecorator = (target: Object,
propertyKey: string | symbol, parameterIndex: number) => void;
```

Bootstrapping an Angular Application

KindGeek School

Understanding the File Structure

- `app/app.component.ts` - this is where we define our root component
- `app/app.module.ts` - the entry Angular Module to be bootstrapped
- `index.html` - this is the page the component will be rendered in
- `app/main.ts` - is the glue that combines the component and page together

Bootstrapping Providers

```
import { BrowserModule }  from '@angular/platform-browser';
import { NgModule } '@angular/core';
import { AppComponent } from './app.component'
import { GreeterService } from './greeter.service';

@NgModule({
  imports: [BrowserModule],
  providers: [GreeterService],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

KindGeek School

Creating Components

```
import { Component } from '@angular/core';

@Component({
  selector: 'rio-hello',
  template: '<p>Hello, {{name}}!</p>',
})
export class HelloComponent {
  name: string;

  constructor() {
    this.name = 'World';
  }
}
```

KindGeek School

Application Structure with Components

```
<rio-todo-app>
  <rio-todo-list>
    <rio-todo-item></rio-todo-item>
    <rio-todo-item></rio-todo-item>
    <rio-todo-item></rio-todo-item>
  </rio-todo-list>
  <rio-todo-form></rio-todo-form>
</rio-todo-app>
```

KindGeek School

Passing Data into a Component

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'rio-hello',
  template: '<p>Hello, {{name}}!</p>',
})
export class HelloComponent {
  @Input() name: string;
}
```

```
<!-- To bind to a raw string -->
<rio-hello name="World"></rio-hello>
<!-- To bind to a variable in the parent scope -->
<rio-hello [name]="helloName"></rio-hello>
```

KindGeek School

```
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'rio-counter',
  templateUrl: 'app/counter.component.html'
})
export class CounterComponent {
  @Input()  count = 0;
  @Output() result = new EventEmitter<number>();

  increment() {
    this.count++;
    this.result.emit(this.count);
  }
}
```

```
import { Component, OnChange } from '@angular/core';

@Component({
  selector: 'rio-app',
  templateUrl: 'app/app.component.html'
})
export class AppComponent implements OnChange {
  num = 0;
  parentCount = 0;

  ngOnChange(val: number) {
    this.parentCount = val;
  }
}
```

```
<div>
  <p>Count: {{ count }}</p>
  <button (click)="increment()">Increment</button>
</div>
```

```
<div>
  Parent Num: {{ num }}<br>
  Parent Count: {{ parentCount }}
  <rio-counter [count]="num" (result)="ngOnChange($event)">
  </rio-counter>
</div>
```

KindGeek School

Responding to Component Events

Using Two-Way Data Binding

```typescript
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'rio-counter',
  templateUrl: 'app/counter.component.html'
})
export class CounterComponent {
  @Input() count = 0;
  @Output() countChange = EventEmitter<number>();

  increment() {
    this.count++;
    this.countChange.emit(this.count);
  }
}
```

```html
<div>
  <p>
    <ng-content></ng-content>
    Count: {{ count }} -
    <button (click)="increment()">Increment</button>
  </p>
</div>
```

KindGeek School

Structuring Applications with Components

- Smart / Container components are application-specific, higher-level, container components, with access to the application's domain model.
- Dumb / Presentational components are components responsible for UI rendering and/or behavior of specific entities passed in via components API (i.e component properties and events). Those components are more in-line with the upcoming Web Component standards.

Home task

Create:

Using the Angular CLI, generate a new component named school.

Add a school property to the SchoolComponent for a school named "Cambridge school", show school property on template.

Create School class with property name, id, place, crate object and display it on template.