Using TypeScript with the ArcGIS API for JavaScript

Nick Senger & Jesse van den Kieboom

Agenda

- TypeScript?
- Fundamentals
- Advanced types
- Development tooling & setup
- Working with the 4.x JS API
- Accessor, decorators, and advanced concepts

TypeScript?

Superset of JavaScript

- *Transpiles* to JavaScript
- ESNext features (import, =>, rest/spread, async/await, etc)
- Types
- Compatible with existing JavaScript

Benefits of TypeScript

- Easier for multiple people to work on
- Easier to refactor
- Easier to test
- Can help prevent technical debt

Fundamentals

Primitive (Basic) Types

- `boolean`, `number`, `string`, `[]`, `{}`
- `any`

```
type Foo = number;

const foo: Foo = 8;
const bar: string = "Lorem ipsum";

// Here be dragons
const waldo: any = {
  doStuff: (things: any) => something
};
```

Type Inference

- TypeScript compiler can infer types automatically

```
let foo = 8; // number type is inferred

foo = 12; // Ok

foo = "12"; // Error!
```

Interfaces

- Define contracts between parts of an application

```
type Foo = number;
type Bar = string;

interface Foobar {
  foo: Foo,
  bar: Bar
}

const baz: Foobar = { foo: 8, bar: "Lorem ipsum" }; // Ok
const qux: Foobar = { foo: "12", bar: "Lorem ipsum" } // Error!
```

- Interfaces facilitate predictable behavior

```
interface Foobar {
  foo: number,
  bar: string
}

const waldo = {
  doStuff: (things: Foobar): Foobar => ({
    foo: things.foo + 1,
    bar: `${things.bar}!`
  })
};

waldo.doStuff({ foo: 1, bar: "a" }); // Ok, { foo: 2, bar: "a!" }
waldo.doStuff(1, "a"); // Error!
```

Classes

```
class Waldo {
  public doStuff(things: Foobar): Foobar { ... }

  private iterateNumber(num: number) {
    return num + 1;
  }

  private addExclamationPoint(str: string) {
    return `${str}!`;
  }
}

const testWaldo = new Waldo(); // Create a Waldo instance
testWaldo.iterateNumber(2); // Error!
```

Extension

- Interfaces can extend other interfaces *or* classes
- Classes can extend other classes and *implement* interfaces

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

interface Point3d extends Point { z: number; }

class MyPoint implements Point3d {
  x = 0;
  y = 0;
  z = 0;
}

class My4dPoint extends MyPoint {
  time = Date.now();
}
```

Advanced types

Union types

- Type something as `one of` multiple choices of a type

```
// Set a size as either a number, of a string like "1px",
// "2em" etc
function setSize(v: number | string) {
  // ...
}
```

Intersection types

- Combining multiple types to a single type representing all of the features

```
function mixin<U, T>(u: U, t: T): T & U {
  // ...
}
```

Type guards

- Type guards allow TS to infer a specific type when a value may take multiple types (`union`)
- Types are `narrowed` to a more specific set by type guards
- Builtin type guards like `typeof` , `instanceof` or `tagged unions`

```
function foo(v: number | string) {
  if (typeof v === "number") {
    // TS infers that v: number
    return v + 1;
  }
  else {
    // TS infers that v: string
    return `${v} + 1`;
  }
}
```

Type guards

- Tagged unions are very useful to discriminate unions of interfaces

```
interface Foo {
  type: "foo";
  foo: string;
}

interface Bar {
  type: "bar";
  bar: string;
}

function func(v: Foo | Bar) {
  if (v.type === "foo") {
    // TS infers that v: Foo
    return v.foo;
  }
  else {
    // TS infers that v: Bar
    return v.bar;
  }
}
```

Generics

- Like in C# or Java (*not* like metaprogramming with templates in C++)
- "Generalizes" types over type parameters

```
class List<T> {
  constructor(private data?: T[]) {
  }

  find(f: (item: T) => boolean): T {
    // ...
  }
}

// Fails
const list = new List<number>(["1", "2"]);

// OK
const list = new List<number>([1, 2]);

// TS infers v to be of type number
list.find(v => v > 1);
```

keyof

- `keyof` allows for "dynamic" type creation
- Can help making types flexible but keeping them strict

```
interface Options {
  shouldQuery: boolean;
  returnGeometry: boolean;
}

const defaultOptions: Options = {
  shouldQuery: false,
  returnGeometry: false
};

function withOptions(options?: Partial<Options>) {
  const withDefaults = { ...defaultOptions, ...options };
}
```

keyof

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

type PartialOptions = Partial<Options>;

//
// Equivalent to
//

interface PartialOptions {
  shouldQuery?: boolean;
  returnGeometry?: boolean;
}
```

Async/await

- Makes asynchronous programming easy again (mostly)
- Internally based on promises
- Typescript polyfills async/await when targetting ES5
- Code

Development tooling

Essentials

- typescript: `npm install --save-dev typescript`
- JS API 4.x typings: `npm install --save-dev @types/arcgis-js-api`
- JS API 3.x typings: `npm install --save-dev @types/arcgis-js-api@3`

Recommended

- Visual Studio Code
- tslint: `npm install --save-dev tslint`
- dojo typings: `npm install --save-dev dojo-typings`

Setting Up

- developers.arcgis.com/javascript/latest/guide/typescript-setup

Working with the API

Imports

- JS API is currently strictly AMD
- Conventionally classes are exported directly
- Requires the use of `require` style imports
  - `import MapView = require("esri/views/MapView")`
  - Or, use `esModuleInterop` with typescript 2.7.2

Auto-cast

- Due to nature of types, auto-cast does not type-check
  - `get` and `set` must have the same type
- Auto-casting is supported in constructor signatures only
  - Still helps in lots of cases
  - For setting properties, need to import the relevant modules

Typing improvements

- Use of generics where possible `Collection<T>`
- Strictly type events (`MapView.on("mouse-wheel", ...)`))
- "Advanced" auto-casts like colors (`"red"`), screen sizes (`"5px"`) and basemaps `"streets"`

Advanced API concepts

Promises

- In 4.7, promises are more compatible with native promises
- Replaced `then` with `when` for `esri/core/Promise`
- Typings are more compatible (although not fully compatible)
- General advice is to wrap API promises in native if needed until JS API switches to native promises

Writing Accessor based classes

- Can be useful to use Accessor based classes in your app
- Also required for creating custom API based widgets
- API classes are using dojo declare, requires some additional work to integrate with TS
- Code

Multiple inheritance

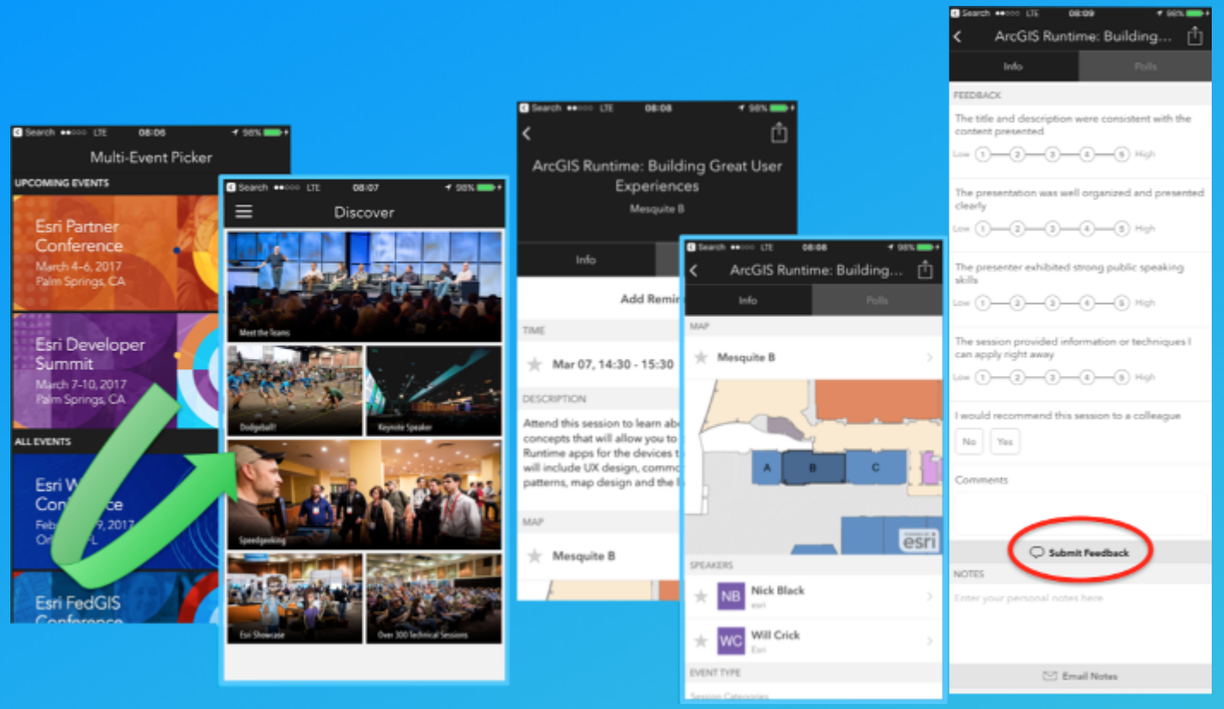
- Multiple inheritance possible with dojo declare
- Supported in typescript at runtime and strictly type-checked
- Uses declaration merging
- Code

Extending the API typings

- API typings are not always as strict as they can be
- In rare occasions typings are missing or imprecise
- Typings can be externally "patched" through declaration merging
- Code

Questions?

Help us to improve by filling out the survey



Slides & Demos: github.com/nicksenger/2018-TS-DS

esri | THE SCIENCE OF WHERE