The Role of Classes in TypeScript

Classes act as containers for different members

TypeScript Class Members

Fields

Constructors

Properties

Functions

Defining a Class

```
class Car {
    //Fields

    //Constructor

    //Properties

    //Functions
}
```

Classes act as containers that encapsulate code

Defining Constructors

Constructors are used to initialize fields

```
class Car {
    engine: string;

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

Field

Constructor

Shorthand way to declare a field

```
class Car {
    constructor(public engine: string) { }
}
```

Adding Functions

```
class Car {
    engine: string;

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

    start() {
        return "Started " + this.engine;
    }

    stop() {
        return "Stopped " + this.engine;
    }
}
```

Class members are public by default

Defining Properties

```
class Car {
    private _engine: string;

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

    get engine(): string {
        return this._engine;
    }

    set engine(value: string) {
        if (value == undefined) throw 'Supply an Engine!';
        this._engine = value;
    }
}
```

Properties act as filters and can have get or set blocks

Using Complex Types

```
class Engine {
    constructor(public horsePower: number,
                public engineType: string) { }
}

class Car {
    private _engine: Engine;

    constructor(engine: Engine) {
       this.engine = engine;
    }

    ...
}
```

Instantiating a Type

Types are instantiated using the "new" keyword

```
var engine = new Engine(300, 'V8');
var car = new Car(engine);
```

Casting Types

This fails

```
var table : HTMLTableElement =
  document.createElement('table');
```

This succeeds

```
var table : HTMLTableElement =
  <HTMLTableElement>document.createElement('table');
```

Cast HTMLElement to HTMLTableElement

Type Definition Files

- As you work with the DOM or other libraries you need a Type Definition file (*.d.ts file)

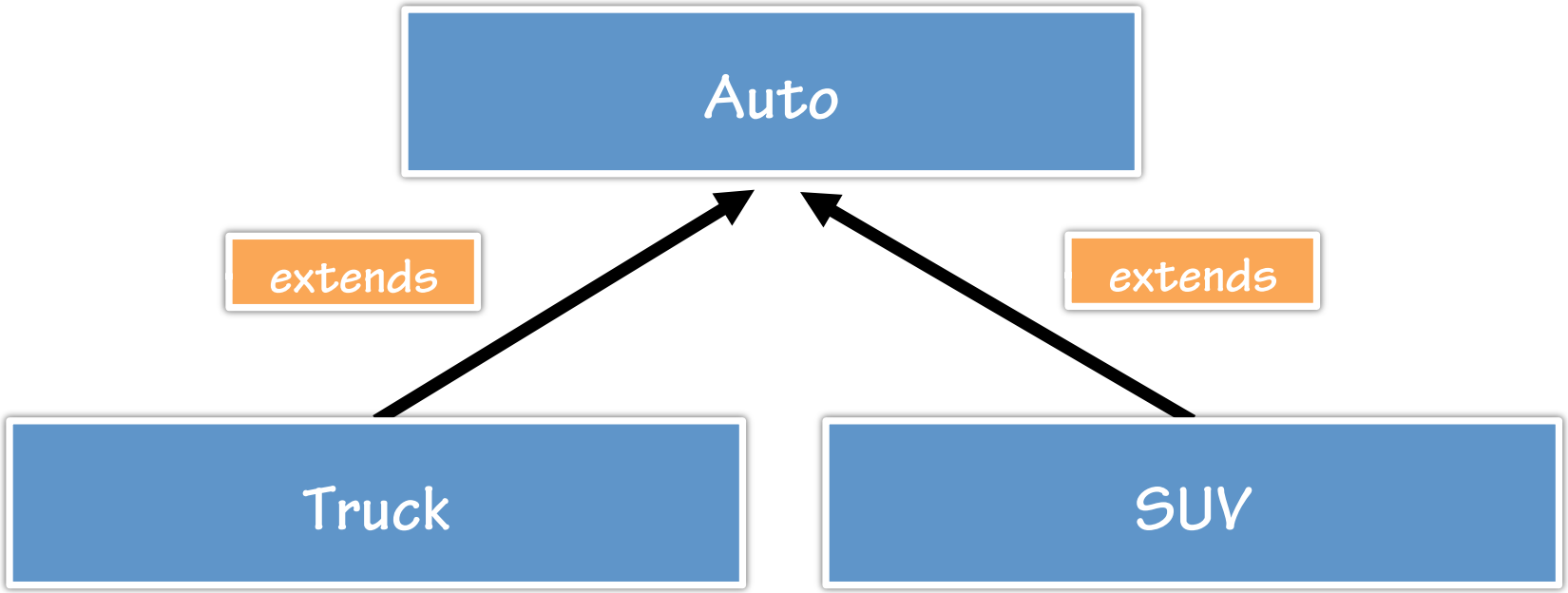
- lib.d.ts file is built-in out of the box for the DOM and JavaScript

- Additional Type Definition files for 3rd party scripts can be found at:

  https://github.com/borisyankov/DefinitelyTyped

  http://definitelytyped.org/

Extending Types with TypeScript



Extending a Type

Types can be extended using the TypeScript "extends" keyword

```
class ChildClass extends ParentClass {
    constructor() {
        super();
    }
}
```

Child class constructor must call base class (super) constructor

Type Extension Example

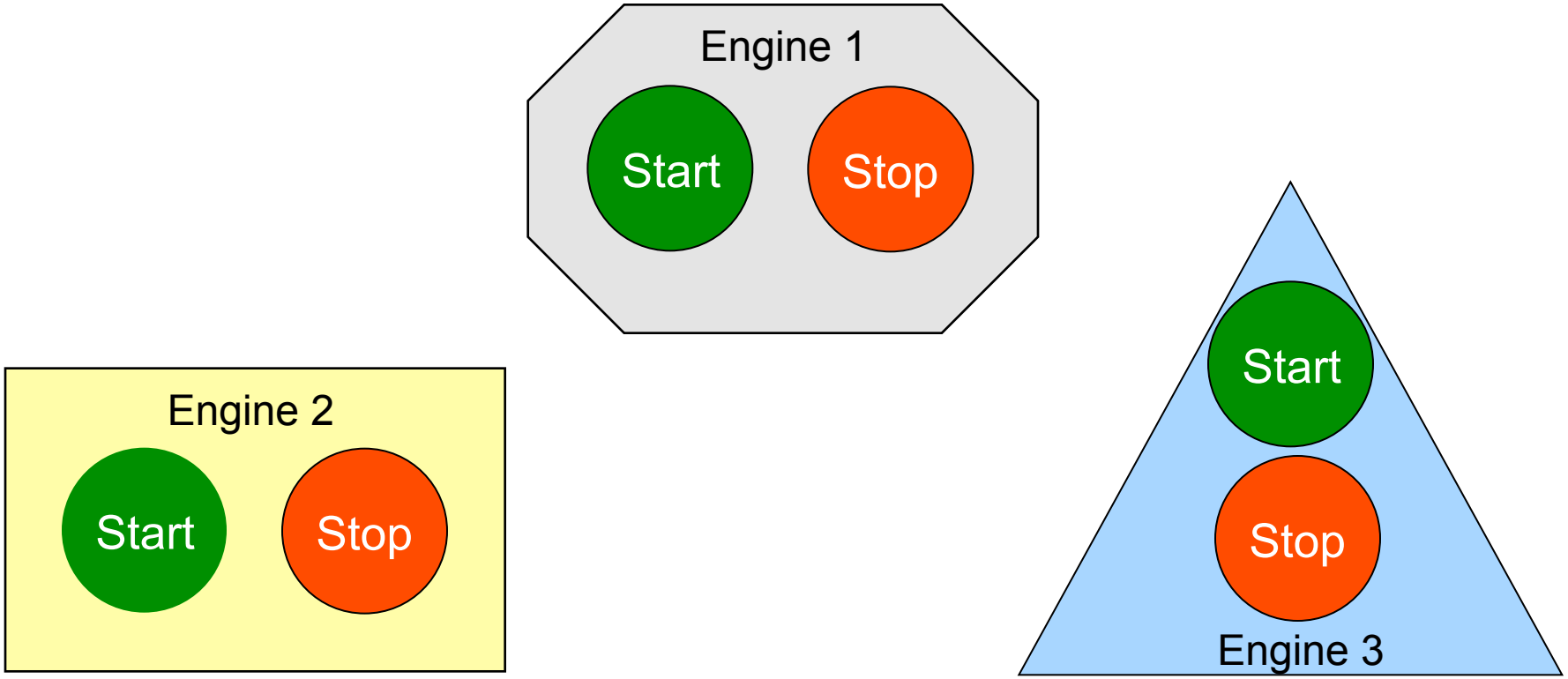
```
class Auto {
    engine: Engine;
    constructor(engine: Engine) {
        this.engine = engine;
    }
}

class Truck extends Auto {
    fourByFour: boolean;
    constructor(engine: Engine, fourByFour: boolean) {
        super(engine);

        this.fourByFour = fourByFour;
    }
}
```

Truck derives from Auto

Call base class constructor

What's an Interface?

- A factory requires that all engines being built have a standard "interface":



Defining an Interface

Interfaces provide a way to define a "contract" that other objects must implement

```
interface IEngine {
    start(callback: (startStatus: boolean,
          engineType: string) => void) : void;
    stop(callback: (stopStatus: boolean,
         engineType: string) => void) : void;
}
```

IEngine Interface defines 2 members

Understanding Functions in an Interface

```
interface IEngine {

    start(callback: (startStatus: boolean,
          engineType: string) => void) : void;

}
```

start() accepts a single parameter named callback

start() doesn't return any data

callback parameter must be a function that accepts a boolean and a string as parameters

callback() doesn't return any data

Optional Members in an Interface

```
interface IAutoOptions {
    engine: IEngine;
    basePrice: number;
    state: string;
    make?: string;
    model?: string;
    year?: number;
}
```

Implementing an Interface

```
class Engine implements IEngine {
    constructor(public horsePower: number,
                public engineType: string) { }

    start(callback: (startStatus: boolean,
          engineType: string) => void) {
        window.setTimeout(() => {
            callback(true, this.engineType);
        }, 1000);
    }
    stop(callback: (stopStatus: boolean,
         engineType: string) => void) {
        window.setTimeout(() => {
            callback(true, this.engineType);
        }, 1000);
    }
}
```

Interfaces provide a way to enforce a "contract"

Using an Interface as a Type

Interfaces help ensure that proper data is passed

```
class Auto {
    engine: IEngine;
    basePrice: number;
    //More fields…

    constructor(data: IAutoOptions) {
        this.engine = data.engine;
        this.basePrice = data.basePrice;
    }
}
```

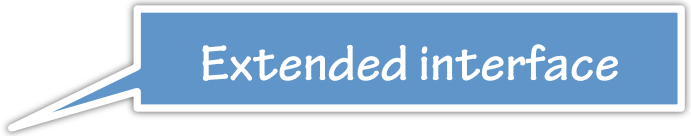
Extending an Interface

```
interface IAutoOptions {
    engine: IEngine;
    basePrice: number;
    state: string;
    make?: string;
    model?: string;
    year?: number;
}

interface ITruckOptions extends IAutoOptions {
    bedLength?: string;
    fourByFour: boolean;
}
```

Defines IAutoOptions members plus custom members

Using an Extended Interface

```
class Truck extends Auto {
    bedLength: string;
    fourByFour: boolean;

    constructor(data: ITruckOptions) {
        super(data);
        this.bedLength = data.bedLength;
        this.fourByFour = data.fourByFour;
    }
}
```



Summary

- TypeScript provides code encapsulation through classes
- Classes can inherit from other classes
- Interfaces provide a "code contract" to ensure consistency across objects
- Interfaces can extend other interfaces