Supercharge Your Client-Side Applications with TypeScript

Dan Wahlin

About Me

Dan Wahlin

@DanWahlin

http://blog.codewithdan.com

Get the Slides and Content

http://codewithdan.me/ts-workshop

ES6 and TypeScript Demos

ES6 Demos

https://github.com/DanWahlin/ES6Samples

TypeScript Demos

https://github.com/DanWahlin/TypeScriptDemos

Angular and TypeScript Code

Angular 1 and TypeScript

https://github.com/DanWahlin/AngularIn20TypeScript

https://github.com/DanWahlin/AngularTypeScript

https://github.com/JohnPapa/hottowel-angular-typescript

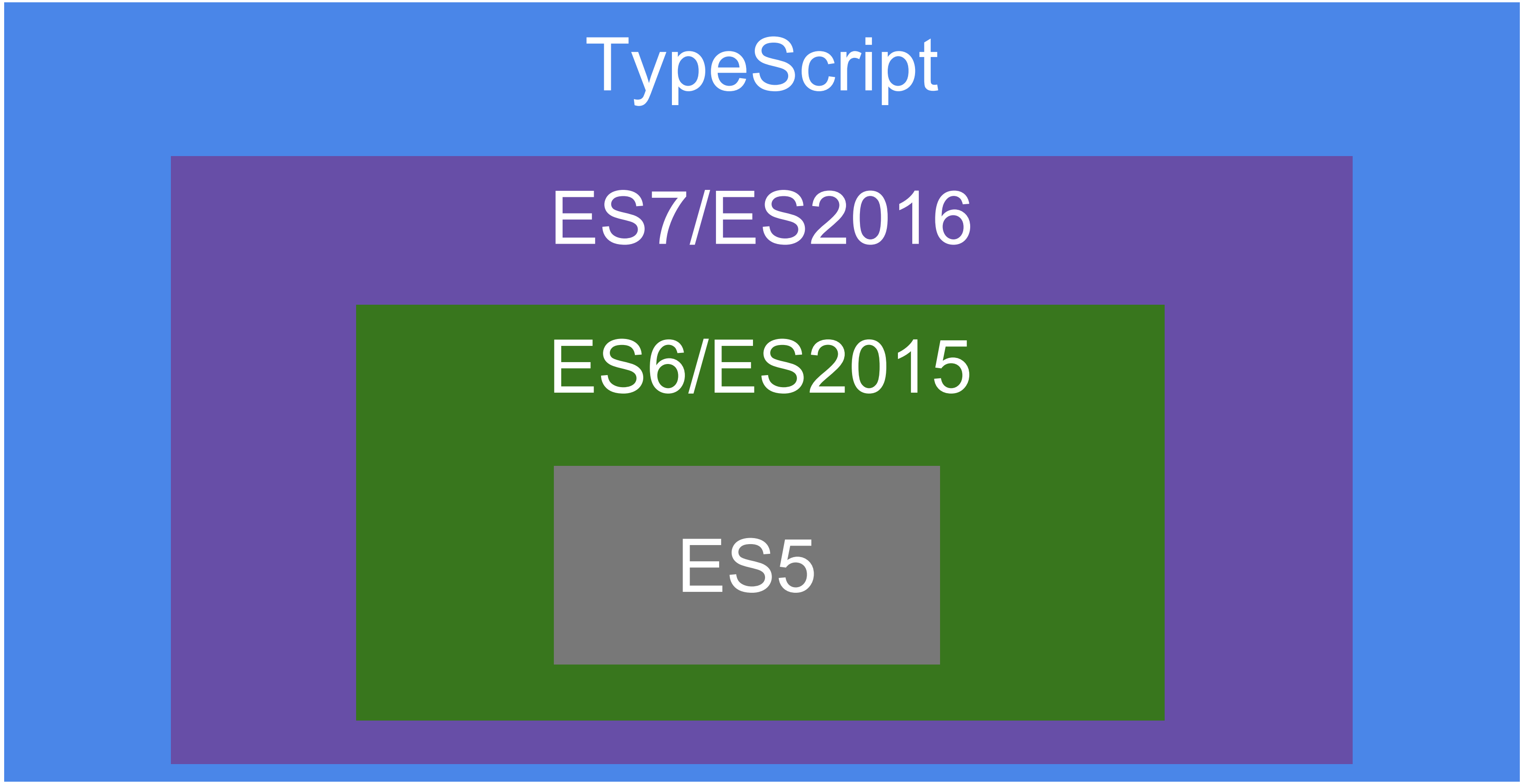
Angular 2 and TypeScript

https://github.com/DanWahlin/Angular2-JumpStart

https://github.com/DanWahlin/Angular2-BareBones

https://github.com/JohnPapa/angular2-tour-of-heroes

http://tinyurl.com/jspatternsguide


TypeScript
ES7/ES2016
ES6/ES2015
ES5

Major Benefits of ES6 / TypeScript

Tooling

Refactorings

Debugging (sourcemaps)

Find and fix issues early!

Any Editor - Any OS

ATOM

Visual Studio

Tooling / Refactoring Benefits

Intellisense

Parameter hints

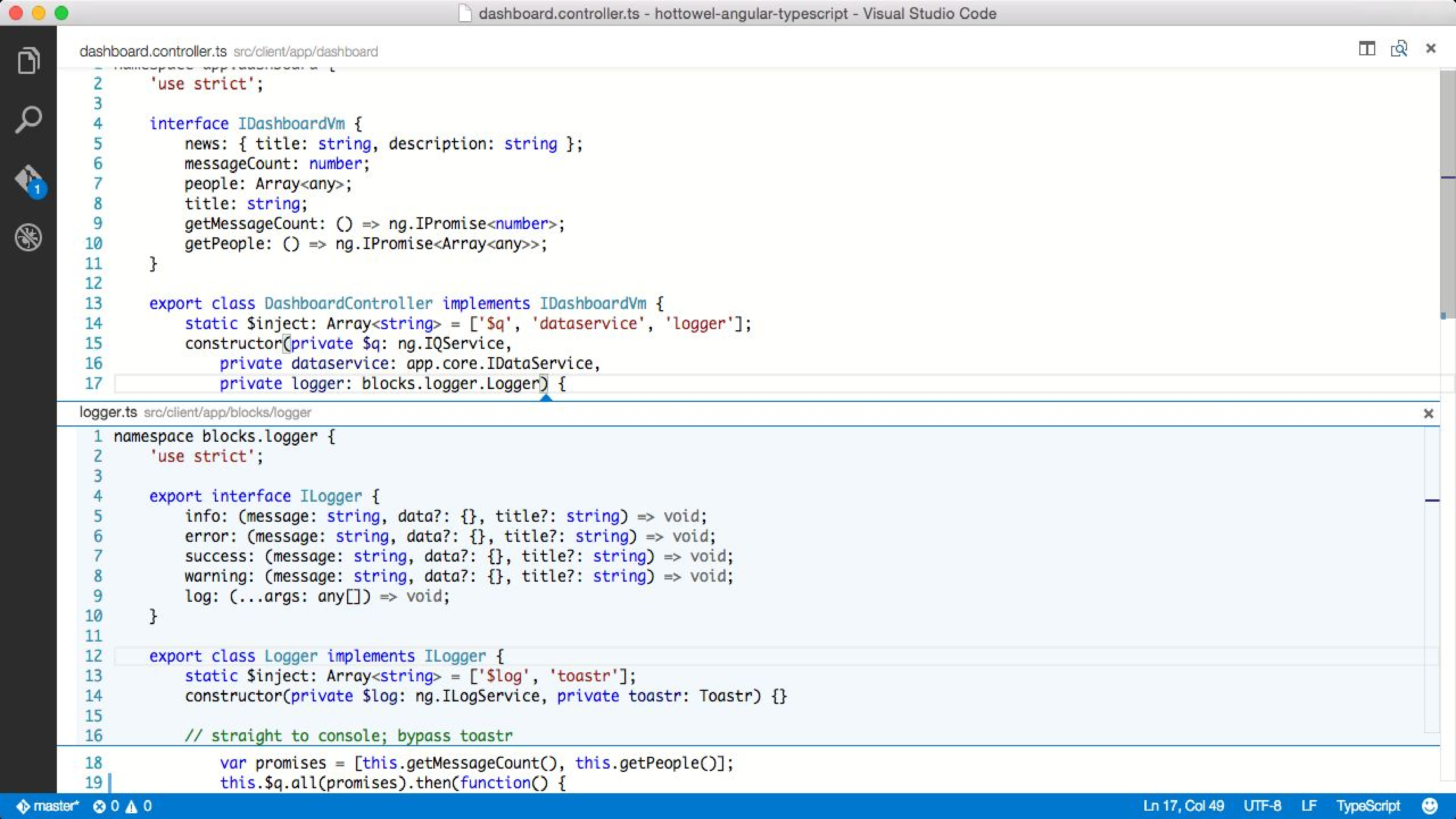
Go to definition or symbol

Peek

Hover

Renaming

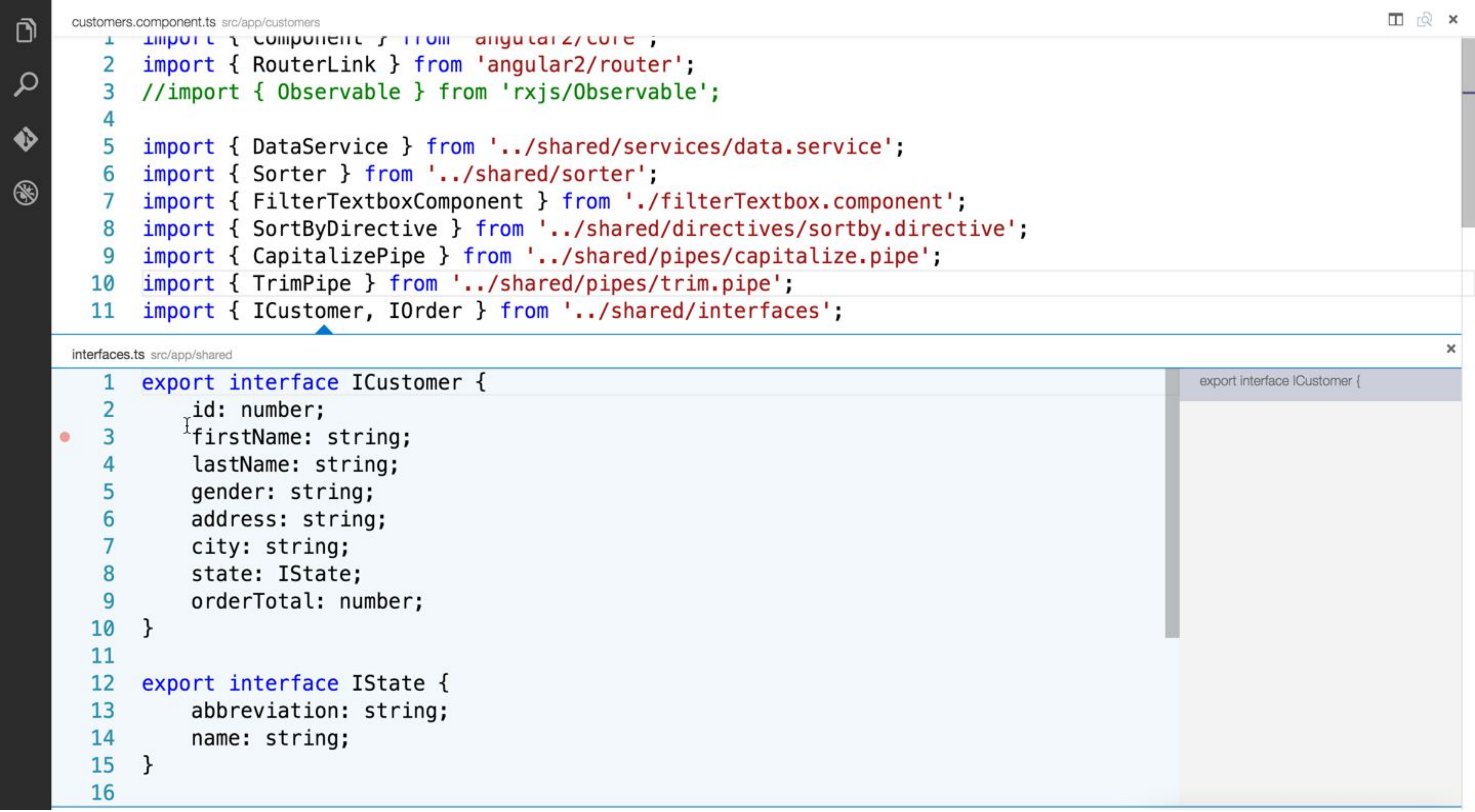
Errors / Warnings


dashboard.controller.ts - hottowel-angular-typescript - Visual Studio Code
dashboard.controller.ts src/client/app/dashboard
logger.ts src/client/app/blocks/logger
master* ⊗ 0 ⚠ 0
Ln 17, Col 49 UTF-8 LF TypeScript

```typescript
    'use strict';

    interface IDashboardVm {
        news: { title: string, description: string };
        messageCount: number;
        people: Array<any>;
        title: string;
        getMessageCount: () => ng.IPromise<number>;
        getPeople: () => ng.IPromise<Array<any>>;
    }

    export class DashboardController implements IDashboardVm {
        static $inject: Array<string> = ['$q', 'dataservice', 'logger'];
        constructor(private $q: ng.IQService,
            private dataservice: app.core.IDataService,
            private logger: blocks.logger.Logger) {
```

```typescript
namespace blocks.logger {
    'use strict';

    export interface ILogger {
        info: (message: string, data?: {}, title?: string) => void;
        error: (message: string, data?: {}, title?: string) => void;
        success: (message: string, data?: {}, title?: string) => void;
        warning: (message: string, data?: {}, title?: string) => void;
        log: (...args: any[]) => void;
    }

    export class Logger implements ILogger {
        static $inject: Array<string> = ['$log', 'toastr'];
        constructor(private $log: ng.ILogService, private toastr: Toastr) {}

        // straight to console; bypass toastr
```

```typescript
            var promises = [this.getMessageCount(), this.getPeople()];
            this.$q.all(promises).then(function() {
```



customers.component.ts src/app/customers

```
import { Component } from 'angular2/core';
import { RouterLink } from 'angular2/router';
//import { Observable } from 'rxjs/Observable';

import { DataService } from '../shared/services/data.service';
import { Sorter } from '../shared/sorter';
import { FilterTextboxComponent } from './filterTextbox.component';
import { SortByDirective } from '../shared/directives/sortby.directive';
import { CapitalizePipe } from '../shared/pipes/capitalize.pipe';
import { TrimPipe } from '../shared/pipes/trim.pipe';
import { ICustomer, IOrder } from '../shared/interfaces';
```

interfaces.ts src/app/shared

```
export interface ICustomer {
    id: number;
    firstName: string;
    lastName: string;
    gender: string;
    address: string;
    city: string;
    state: IState;
    orderTotal: number;
}

export interface IState {
    abbreviation: string;
    name: string;
}

```

tsconfig.json

```
{
 "compilerOptions": {
   "target": "es5",
   "module": "system",
   "moduleResolution": "node",
   "sourceMap": true,
   "emitDecoratorMetadata": true,
   "experimentalDecorators": true,
   "removeComments": false,
   "noImplicitAny": true,
   "suppressImplicitAnyIndexErrors": true,
   "allowJs": true
 }
}
```

Target to Transpile to

ES2016 / ES7 Features

Debugging

Debugging

Node

Launch configurations

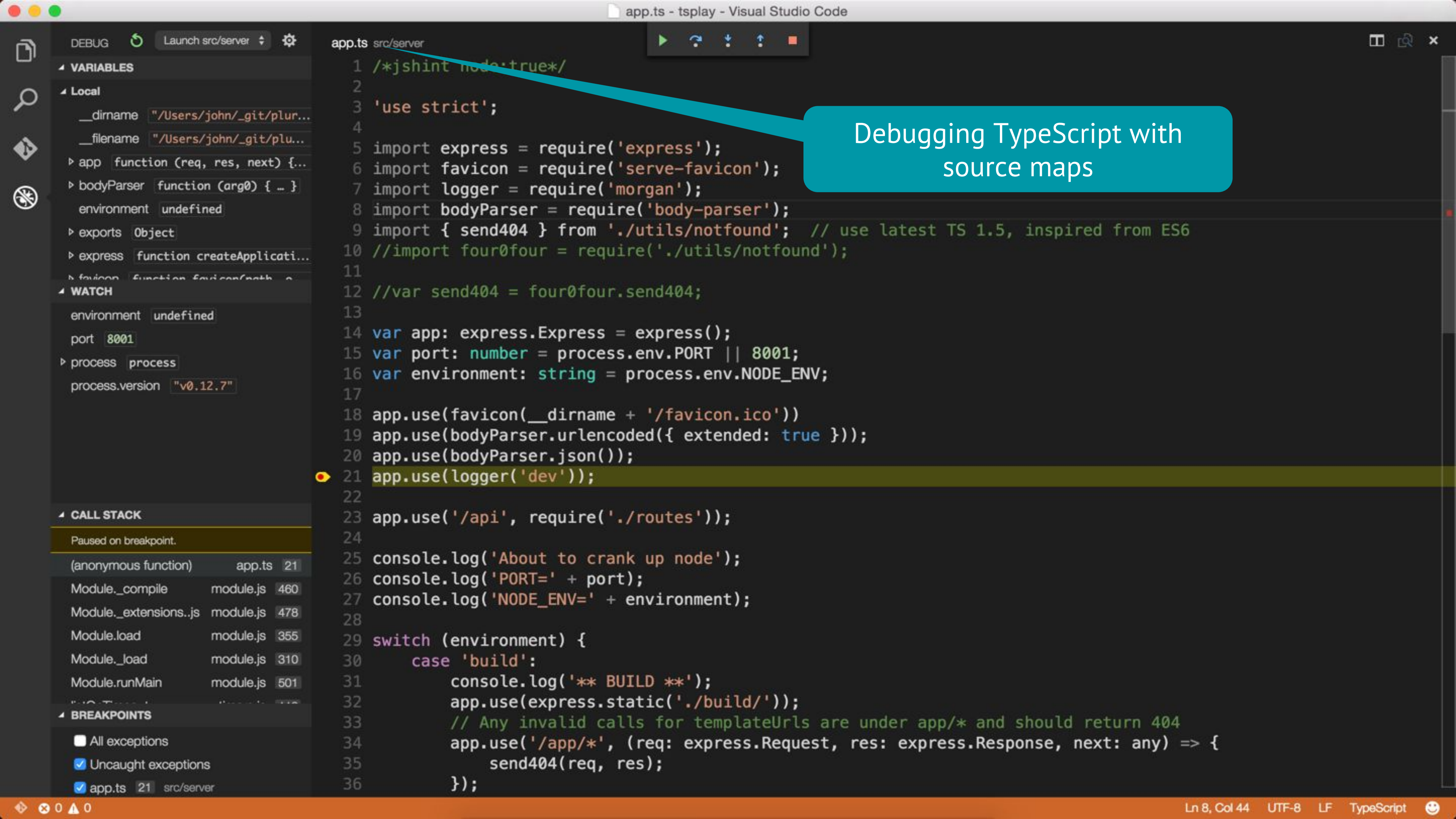
Debugging experience

Attaching to processes

Source maps

JavaScript and TypeScript

Debugging TypeScript with source maps



VARIABLES

Local

__dirname	"/Users/john/_git/plur…
__filename	"/Users/john/_git/plu…
app	function (req, res, next) {…
bodyParser	function (arg0) { … }
environment	undefined
exports	Object
express	function createApplicati…

WATCH

environment	undefined
port	8001
process	process
process.version	"v0.12.7"

CALL STACK

Paused on breakpoint.

(anonymous function)	app.ts	21
Module._compile	module.js	460
Module._extensions..js	module.js	478
Module.load	module.js	355
Module._load	module.js	310
Module.runMain	module.js	501

BREAKPOINTS

- [ ] All exceptions
- [x] Uncaught exceptions
- [x] app.ts 21 src/server

```typescript
/*jshint node:true*/

'use strict';

import express = require('express');
import favicon = require('serve-favicon');
import logger = require('morgan');
import bodyParser = require('body-parser');
import { send404 } from './utils/notfound';  // use latest TS 1.5, inspired from ES6
//import four0four = require('./utils/notfound');

//var send404 = four0four.send404;

var app: express.Express = express();
var port: number = process.env.PORT || 8001;
var environment: string = process.env.NODE_ENV;

app.use(favicon(__dirname + '/favicon.ico'))
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.use(logger('dev'));

app.use('/api', require('./routes'));

console.log('About to crank up node');
console.log('PORT=' + port);
console.log('NODE_ENV=' + environment);

switch (environment) {
    case 'build':
        console.log('** BUILD **');
        app.use(express.static('./build/'));
        // Any invalid calls for templateUrls are under app/* and should return 404
        app.use('/app/*', (req: express.Request, res: express.Response, next: any) => {
            send404(req, res);
        });
```

⊗ 0 ⚠ 0   Ln 8, Col 44 UTF-8 LF TypeScript

Getting Started with ES6

Key ES6 Features

Maps/Sets	Modules/Classes	Block Scope	Destructuring
Arrow Functions	Default Parameters	Rest Parameters	More…

Browser support: http://kangax.github.io/compat-table/es6/

Babel Transpiler

Babel

- Available at https://babeljs.io
- Supports a broad range of ES6 features
- Support for a variety of plugins (gulp, grunt, etc.)

Transpiling with Babel and Gulp

Gulp automates the process with Babel:

```
var gulp = require('gulp'),
    babel = require('gulp-babel');

gulp.task('babel', function () {
    gulp.src([es6Path])
        .pipe(babel())
        .pipe(gulp.dest(compilePath + '/babel'));
});

gulp.task('watch', function() {
    gulp.watch(es6Path, ['babel']);
});

gulp.task('default', ['babel', 'watch']);
```

ES6 Class Example

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

    get engine() {
        return this._engine;
    }

    set engine(val) {
        this._engine = val;
    }

    start() {
        console.log(this.engine);

    }
}
```

constructor

get/set property blocks

function

Using export and import with Modules

ES6 relies on **export** and **import** keywords:

```
// foobar.js
export var foo = 'foo';
export var bar = 'bar';
```

```
import { foo, bar } from 'foobar';
console.log(foo); // 'foo'
```

```
import * as foobar from 'foobar';
console.log(foobar.foo); // 'foo'
console.log(foobar.bar); // 'bar'
```

Maps and Sets

Maps store a collection of key/value pairs, with unique keys

Sets can store a collection of items (items must be unique)

Using Maps

```
var map = new Map();
map.set('Finance','Process bills');
map.set('HR', 'Human Resources and Healthcare');
//Duplicate ignored
map.set('HR', 'Human Resources and Healthcare');
console.log('Getting HR: ' + map.get('HR'));
console.log(map.size);
if (map.has('Finance')) console.log('Found it!');
map.delete('Finance'); //Delete single item
map.clear(); //Clear all items
```

Add key/value into Map

Using Sets

```
var set = new Set();
set.add('HR');
set.add('Finance');
set.add('Finance'); //Duplicate ignored
set.add({name: 'GIS', desc: 'Mapping'});
console.log(set.size);

if (set.has('Finance')) console.log('Found it!');
set.delete('Finance'); //Delete single item
set.clear(); //Clear all items
```

Add items into the Set

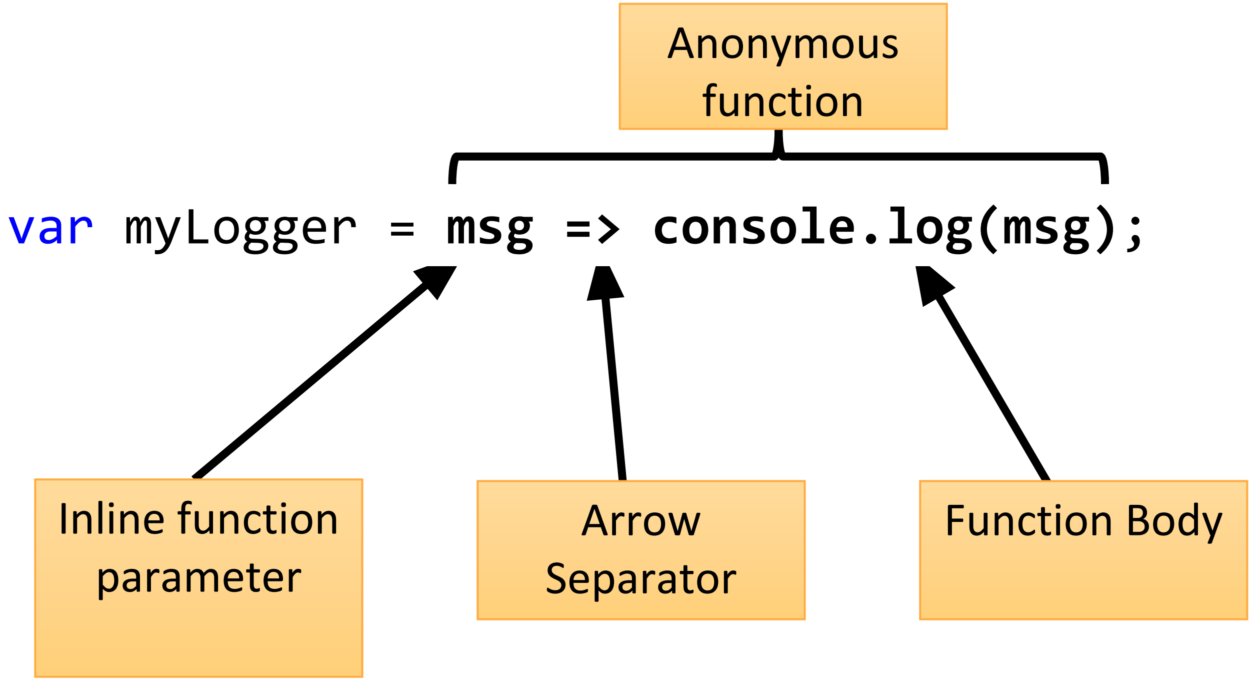
"size" not "length"

Arrow Functions

```
var myLogger = function(msg) {
    console.log(msg);
};
```

```
var myLogger = msg => console.log(msg);
```

Arrow Function Syntax



```
var myLogger = msg => console.log(msg);
```

Template Strings

Embed variables and expressions in string literals

```
`Hello ${firstName}`
```

Uses the **`** back-tick to start and end a string

Great for multi-line strings

Template Strings in Action

```
class Car {

    constructor(make, model, engine) {
        this._make = make;
        this._model = model;
        this._engine = engine;
    }

    start() {
        return `
            ${this._make} ${this._model} with a
            ${this._engine} engine is started!
           `;

    }

}
```

Template String

Destructuring

```
// Destructure object
var {total2, tax2} = {total:9.99, tax:.50};


// Destructure array
var [red, yellow, green] = ['red', 'yellow', 'green'];
console.log(`Destructuring colors: ${red} ${yellow} ${green}`);
```

Ignoring Specific Members

```
var [red2, , green2] = ['red', 'yellow', 'green'];
console.log(`Destructuring with an ignore: ${red2}
            ${green2}`);
```

Default Parameters

Assign default value to a parameter

```
class Car {

    currentYear() {
        return new Date().getFullYear();
    }

    //make, model, and year are "default parameters"
    setDetails(make = 'None', model = 'None',
               year = this.currentYear()) {

        console.log(make + ' ' + model + ' ' + year);

    }
}
```

...Rest Parameters

Pass indefinite number of parameters to a function

```
class Car {

    //accessories is "rest parameter"
    setDetails(make = 'No Make', ...accessories) {
        console.log(make);

        if (accessories) {
            for (var i = 0; i < accessories.length; i++) {
                console.log('\n' + accessories[i]);
            }
        }
    }
}
```

Rest Parameter

Getting Started with TypeScript

Why use TypeScript?

Use Existing JavaScript Code
(use ES3/ES5 code)

Strong Typing
(structural typing + type inference)

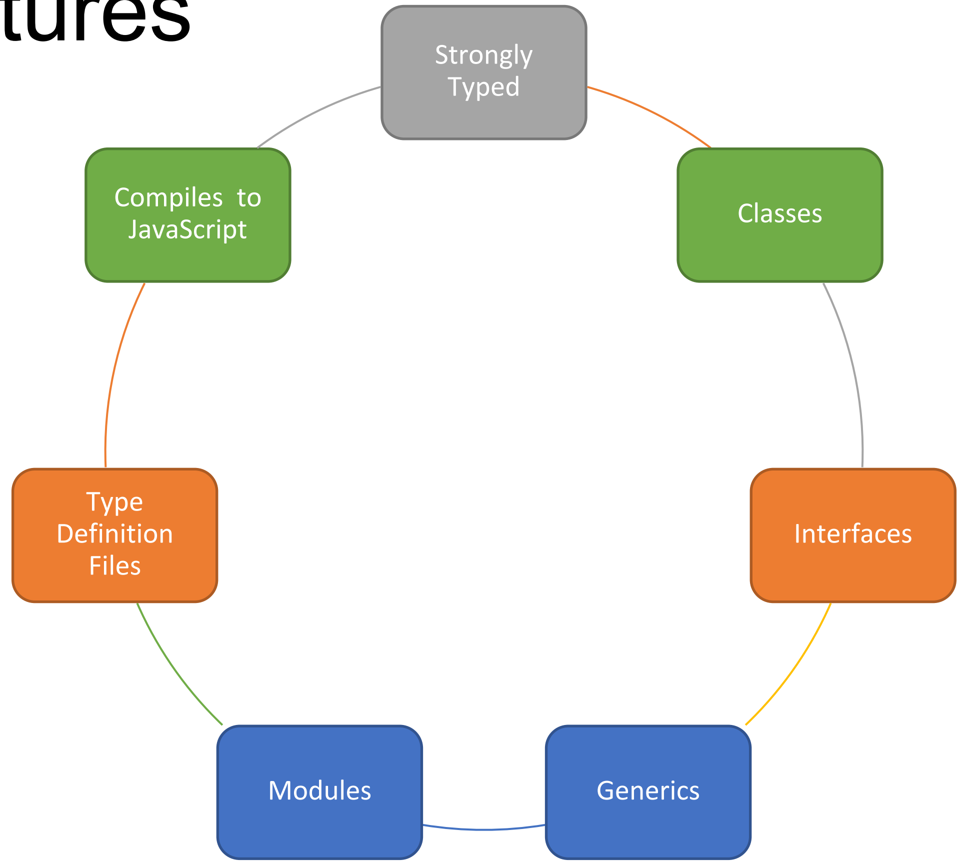
Modular
(CommonJS and AMD)

Tooling Support
(Visual Studio, WebStorm, more)

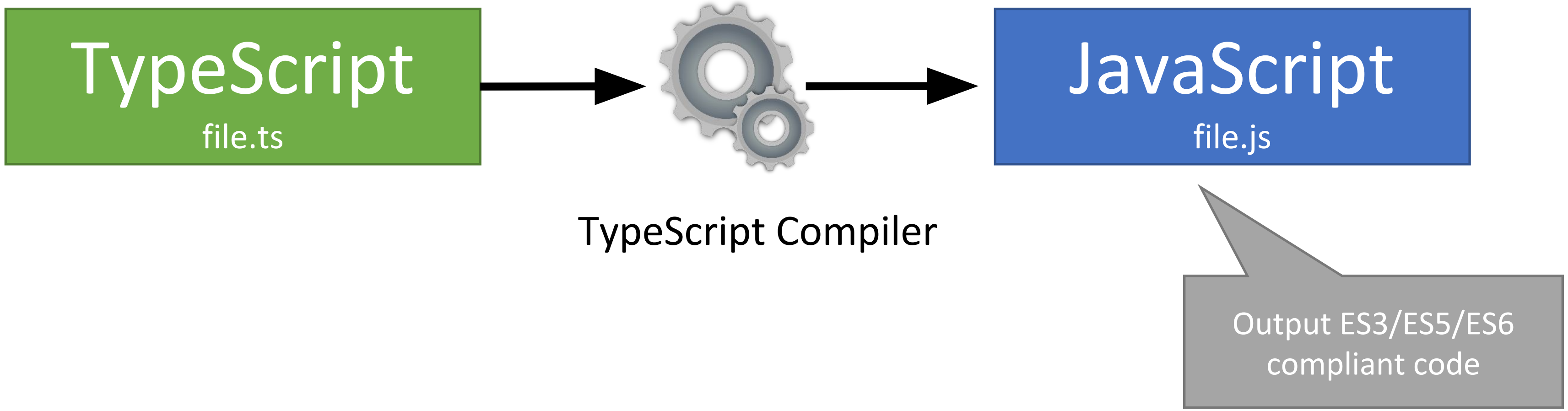
Scalable Application Structure
(support large code bases)

ES6 Standards
(classes, arrow functions, more)

Key Features



How Does TypeScript Work?



TypeScript Playground and Help Documentation

Automating TypeScript Builds

- JavaScript task runners automate various tasks
  - tsc
  - Gulp
  - Grunt

- A TypeScript Gulp/Grunt task can compile .ts to .js

* Gulp and Grunt rely on Node.js

Checkpoint

TypeScript is a Superset of JavaScript

Strong Typing

ES6 Functionality

Simplify Application Maintenance

Catch Issues Early

Basic Types

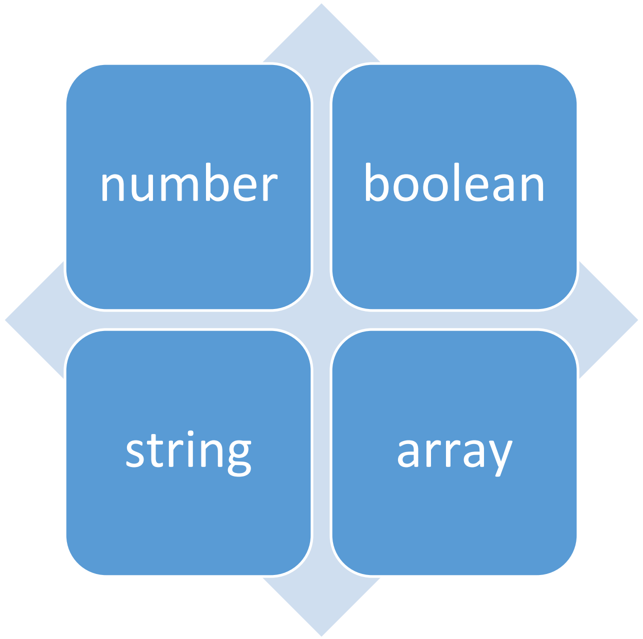
Types in JavaScript?

What is the result of the following code?

```
function total(x, y) {
    alert( x + y );
}

total('1', 2);
```

Returns '12';

TypeScript Types



any and void

Defining Typed Variables

```
var variableName: typeScriptType = value;
```

```
var age: number = 5;
var name: string = 'Anders';
var isLoaded: boolean = false;
var pets: string[] = ['Fido', 'Lassie', 'Rover'];
```

Typed Parameters

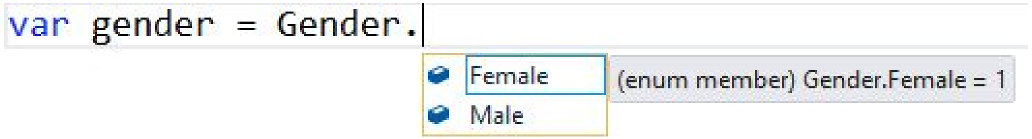
```
//Assigning a type to function parameters
function add(msg: string, x: number, y: number) {
    console.log(msg + (x + y));
}

add('Total = ', 3, 2);
```

Union Types

```
//values can be a number or a number[]
var values : number | number[];

values = [5, 5, 5, 5]; //array
values = 50; //number
```

Enums

```
enum Gender { Male, Female };
```



```
var gender = Gender.
```

Female
Male

(enum member) Gender.Female = 1

Const Enums

```
const enum Gender { Male, Female };

var gender = Gender.Female;
```

Compiles to:
var gender = 1;

Checkpoint

TypeScript supports strongly-typed variables

Parameter types can be assigned a type

Union types can minimize the number of function overloads

Const Enums reduce the amount of generated code

TypeScript Functions

TypeScript Functions

- Functions can be defined several different ways:
  - Named functions
  - Anonymous functions/methods
  - Lambda functions
  - Class functions

Named Function

```
function displayOutput(msg: string) {
    content.innerHTML = msg;
}
```

Anonymous Function with Type Inference

```
var add = function (x: number, y: number) : number {
    return x + y;
}
```

Anonymous Function without Type Inference

Input Types

Output Type

```
var add: (x: number, y: number) => number =
    function (x: number, y: number): number {
        return x + y;
    };
```

Lambda Functions

Lambda Operator

```
var $ = (id) => document.getElementById(id);

var $ = function(id) {
    return document.getElementById(id);
};
```

Optional, Default and Rest Parameters

Optional Parameters

Function parameters are required by default:

```
function buildAddress(address1: string, address2: string, city: string) {
    //all parameters must be passed
}
```

Optional parameters are defined using the ? character:

```
function buildAddress(address1: string, city: string, address2?: string) {
    //address2 parameter is optional
}

buildAddress('1234 Central', 'Seattle'); //address2 not passed
```

Optional Parameter

Optional parameters must be placed after all required parameters

Default Parameters

Optional but provide a "default" value if the parameter isn't passed:

```
function buildAddressDefault(address1: string, city: string, address2 = 'N/A')
{
  //address2 parameter will default to N/A if not passed
}

buildAddress('1234 Central', 'Seattle'); //address2 not passed
```

Default Parameter

Must be placed after all required parameters

Rest Parameters

Allows the "rest of the parameters" to be passed as an array using … syntax:

Rest Parameter

```
function buildAddress(city: string, ...restOfAddress: string[]){
    //city + an array of string parameters can be passed
}

buildAddress(city, address, address2); //address & address2 are "rest"
parameters
```

Must be placed after all required parameters

Lambdas and Using "this"

JavaScript's "this" keyword can be tricky to use

Changes context depending on the caller

"this" is start (a button)

```
this.start.addEventListener('click', this.updateTimer);
```

TypeScript lambdas capture "this"

"this" is captured

```
this.start.addEventListener('click',() => this.updateTimer());
```

Checkpoint

Functions can be defined multiple ways

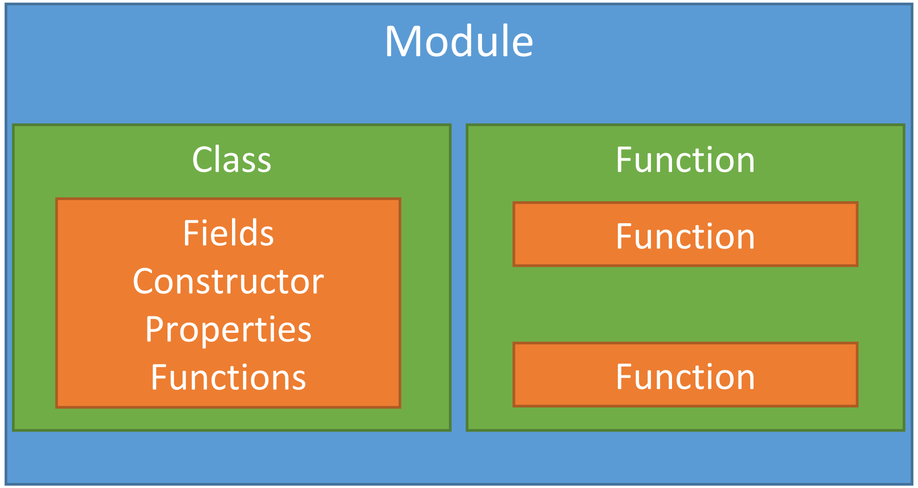
Parameters can be optional, default or rest

Lambdas provide short-cut functionality

Working with "this" can be simplified by using lambdas

Classes

TypeScript Code Organization



Class Example

Class

Property

Constructor

Function

```
class Greeter {
    element: HTMLElement;

    constructor(element: HTMLElement) {
        this.element = element;
    }

    greet(msg: string ) {
        this.element.innerHTML = msg;
    }
}
```

Converting Classes to ES5 Compliant Code

TypeScript

```
class Greeter {
    element: HTMLElement;

    constructor(element: HTMLElement) {
        this.element = element;
    }

    greet(msg: string ) {
        this.element.innerHTML = msg;
    }
}
```

JavaScript

```
var Greeter = (function () {
    function Greeter(element) {
        this.element = element;
    }
    Greeter.prototype.greet = function (msg) {
        this.element.innerHTML = msg;
    };
    return Greeter;
})();
```

The Constructor and Properties

```
class Greeter {
    element: HTMLElement;

    constructor(element: HTMLElement) {
        this.element = element;
    }

    greet(msg: string ) {
        this.element.innerHTML = msg;
    }
}


var greeter = new Greeter(el);
```

Constructor called when class is initialized

Stores parameter value in a property

Invoke Constructor

Auto-Generating Properties

Because "private" is used the property will be auto-generated.

```
class Greeter {

    constructor(private element: HTMLElement) { }

    greet(msg: string ) {
        this.element.innerHTML = msg;
    }
}
```

Defining Properties

Defined using **get** and **set** keywords:

```
class Account {

    _balance: number = 0;

    get balance() {
        return this._balance;
    }

    set balance(val: number) {
        this._balance = val;
    }

}
```

Public and Private Modifiers

Class members are **public** by default:

```
class Account {
    _balance: number = 0;
}
```

Public by default

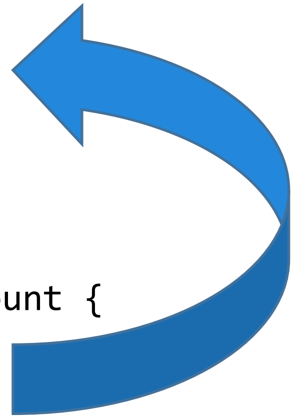
Members can be marked as **private**:

```
class Account {
    private _balance: number = 0;
}
```

Only accessible from within the class

Class Inheritance

```
class Account {
    private _title: string;
    constructor(title: string) {
        this._title = title;
    }
}

class CheckingAccount extends Account {
    constructor(title: string) {
        super(title);
    }
}
```



Checkpoint

Classes encapsulate members

Members include fields, constructors, properties, functions

TypeScript supports class extension

The super keyword can be used to call into a base class

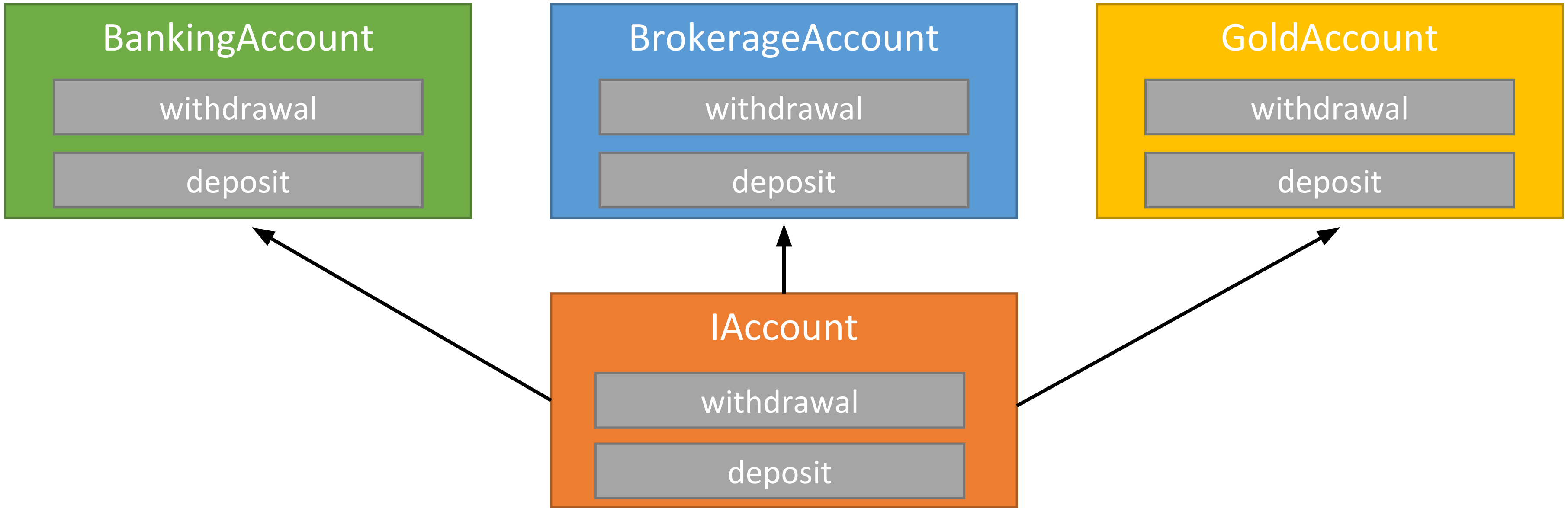
Interfaces

An interface is a "code contract"

Drive Consistency across classes
Clarify function parameter and return types
Create custom function and array types
Define type definition files for libraries and frameworks

The Need for Interfaces: Scenario 1

Classes can all implement same interface

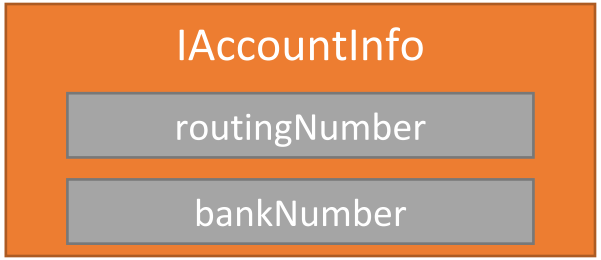


The Need for Interfaces: Scenario 2

```
class BankingAccount {

    get accountInfo() {
        return {
            routingNumber: Constants.ROUTING_NUMBER,
            bankNumber: Constants.BANK_NUMBER
        }
    }
}

var acct = new BankingAccount();
var info = acct.accountInfo();
```

What type is the **info** variable?

Using an Interface as a Type



```
class BankingAccount {

    get accountInfo() : IAccountInfo {
        return {
            routingNumber: Constants.ROUTING_NUMBER,
            bankNumber: Constants.BANK_NUMBER
        }
    }
var acct = new BankingAccount();
var info: IAccountInfo = acct.accountInfo();
```

The type of **info** is clear now

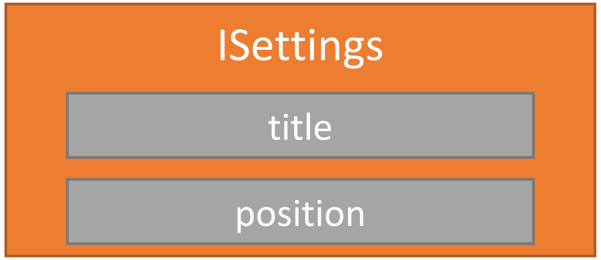
The Need for Interfaces: Scenario 3

```
class MyObject {
    _settings;

    constructor(settings) {
        this._settings = settings;
    }
}
```

What properties does **settings** have?

Using an Interface as a Parameter Type

```
class MyObject {
    _settings: ISettings;

    constructor(settings: ISettings) {
        this._settings = settings;
    }
}
```



Defining an Interface with Members

Defining an Interface

```
interface IMessage {
    greeting: string;
}


interface IGreet {
    greet(msg: IMessage): void;
}
```

Implementing an Interface

```
class Greeter implements IGreet {
    element: HTMLElement;

    constructor(element: HTMLElement) {
        this.element = element;
    }

    greet(msg: IMessage) {
        this.element.innerHTML = msg.greeting;
    }
}
```

Implementing an Interface: Example 1

```
class Greeter implements IGreet {
    element: HTMLElement;

    constructor(element: HTMLElement) {
        this.element = element;
    }

    greet(msg: IMessage) {
        this.element.innerHTML = msg.greeting;
    }
}
```

```
interface IGreet {
    greet(msg: IMessage): void;
}


interface IMessage {
    greeting: string;
}
```

Defining Optional Properties

```
interface IAccount extends IDepositWithdrawal {
    accountInfo: IAccountInfo;
    balance : number;
    title: string;
    internalId?: number;
}
```

Optional Property

Creating Custom Array and Function Types

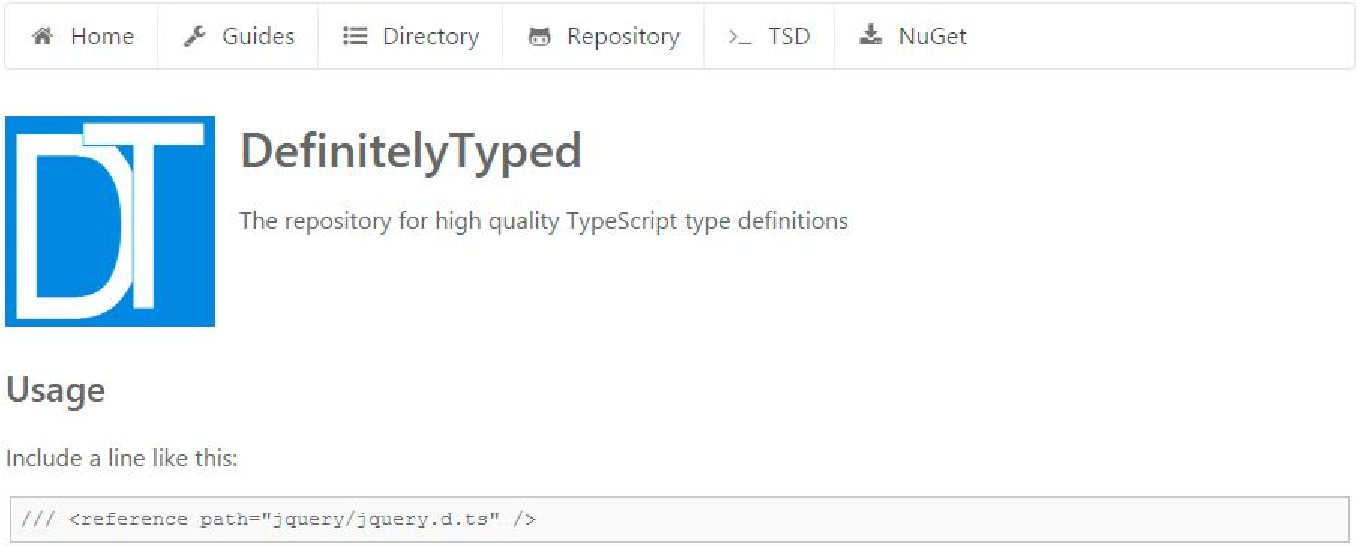
Function Types

Custom Function Type

```
interface SearchFunc {
    (source: string, subStr: string): boolean;
}

var mySearch: SearchFunc = function (source: string, subStr: string)
{
    var result = source.search(subStr);
    return (result !== -1);
}
```

Interfaces and Type Definition Files

Interfaces can Describe External Scripts



DefinitelyTyped

The repository for high quality TypeScript type definitions

Usage

Include a line like this:

```
/// <reference path="jquery/jquery.d.ts" />
```

http://definitelytyped.org

Checkpoint

Interfaces are Code Contracts

Interfaces can extend other interfaces

Classes can implement one or more interfaces

Interfaces play a key role in Type Definition Files

Generics

A generic is a "code template" that relies on type variables:

<T>

Generics Features

Provide reusable code templates

Provide more flexibility when working with types

Compile-time only checks

Can be used in many scenarios (classes, functions, etc.)

Can minimize the use of "any"

The Need for Generics

```
class ListOfNumbers {
    _items: number[] = [];

    add(item: number) {
        this._items.push(item);
    }

    getItems(): number[] {
        return this._items;
    }
}
```

```
class ListOfString {
    _items: string[] = [];

    add(item: string) {
        this._items.push(item);
    }

    getItems(): string[] {
        return this._items;
    }
}
```

The Answer is Generics

```
class List<T> {
    _items: T[] = [];

    add(item: T) {
        this._items.push(item);
    }

    getItems(): T[] {
        return this._items;
    }
}

var nameList = new List<string>();
```

```
class List {
    _items: string[] = [];

    add(item: string) {
        this._items.push(item);
    }

    getItems(): string[] {
        return this._items;
    }
}
```

Creating a Generic Function

Generics type variable

```
function processData<T>(data: T) {
    //process the data here
}

processData<number>(504);
```

Providing the type

$\longrightarrow$

```
function processData(data: number)
{
    //process the data here
}
```

Using Generics with an Interface

```
interface IAccountInfo<TRouteNumber, TBankNumber> {
    routingNumber: TRouteNumber;
    bankNumber: TBankNumber;
}


class BankingAccount implements IAccount{

    get accountInfo() : IAccountInfo<string, number> {
        return {
            routingNumber: Constants.ROUTING_NUMBER,
            bankNumber: Constants.BANK_NUMBER
        }
    }

}
```

Generic Constraints

Generic Constraints

T is constrained

```
class List<T extends IAccount> {
    _items: T[] = [];

    add(item: T) {
        this._items.push(item);
    }

    getItems(): T[] {
        return this._items;
    }
}
```

```
interface IAccount extends
IDepositWithdrawal {
    accountInfo: IAccountInfo;
    balance : number;
    title: string;
    internalId?: number;
}
```

Checkpoint

Generics are "code templates"

Generic templates rely on type variables: <T>

Generics templates are reusable

Generics provide more flexibility with types

Namespaces

Key Module Features

Organize Code

Pull Code out of the Global Scope

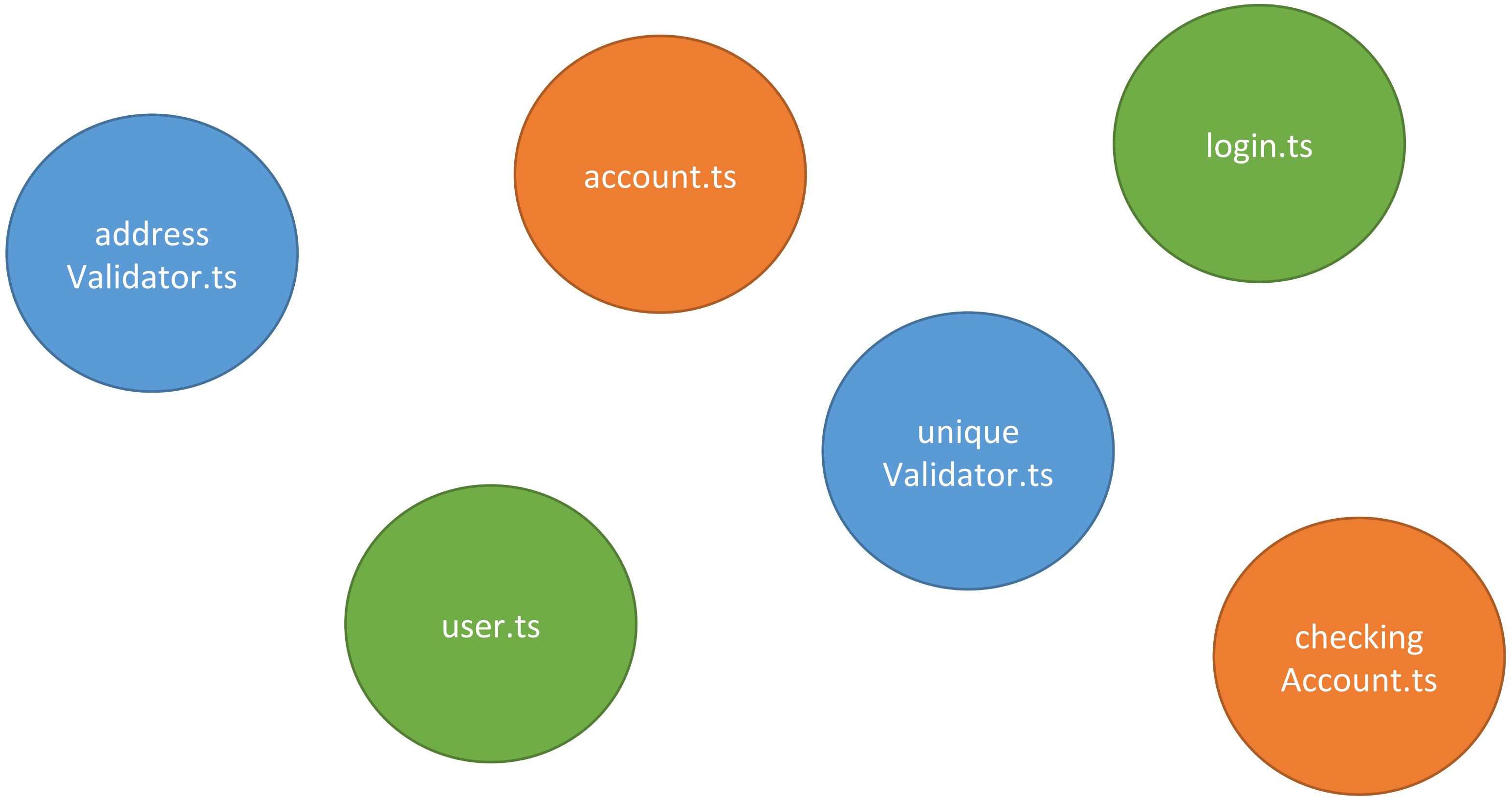
Enhance Code Reusability

TypeScript Module Types

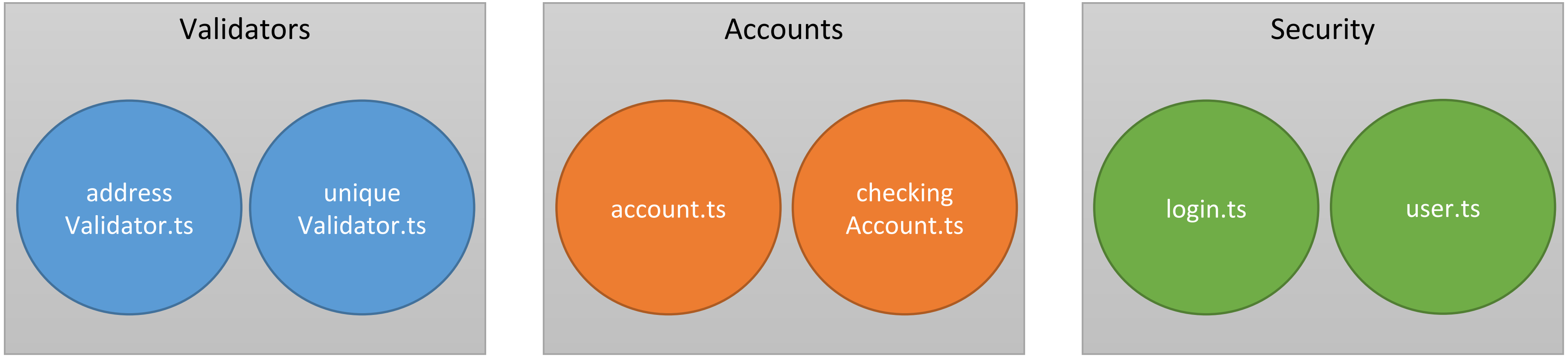
Namespaces
(code organization)

Modules
(CommonJS or AMD loading)

Why do we Need Namespaces?



Organizing Code with Namespaces



Avoiding Global Scope

Added to the "global" scope

```
class MyGlobalClass {
    constructor() {
        console.log('In MyGlobalClass constructor');
    }
}
```

MyGlobalClass a member of the window object

```
window['MyGlobalClass']
```

Creating and Using Namespaces

Creating a Namespace

```
namespace ModuleWithExport {

    export class Hello {
        constructor() {
            console.log('Hello ');
            console.log('Calling into World class constructor ' +
                        'from ModuleWithExport.Hello.');
            var world = new World();
        }
    }

    class World {
        constructor() {
            console.log('World');
        }
    }
}
```

Accessible outside of the namespace

Only accessible within the namespace

Referencing a Namespace Member

```
namespace ModuleWithExport {

    export class Hello { … }

}
```

Reference namespace

```
var hello = new ModuleWithExport.Hello();
```

Checkpoint

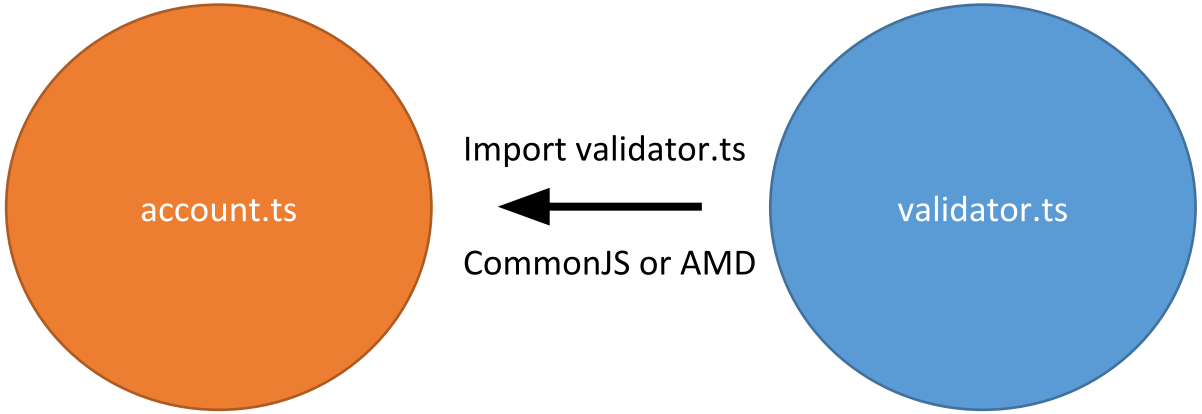
Namespaces encapsulate and organize code

Pull objects out of the global scope

TypeScript also supports External Modules
(more on this later!)

Modules

The Need for Modules



TypeScript Module Types

Namespace
(code organization)

Modules
(CommonJS or AMD loading)

Key Module Features

Useful in large applications

Load files dynamically

Manage dependency chains

Use CommonJS, AMD, ES6

What is CommonJS?

Node.js applications use CommonJS to require/import modules:

```
var app = require('express');
```

What is AMD?

- AMD = Asynchronous Module Definition

- require.js is a popular AMD library

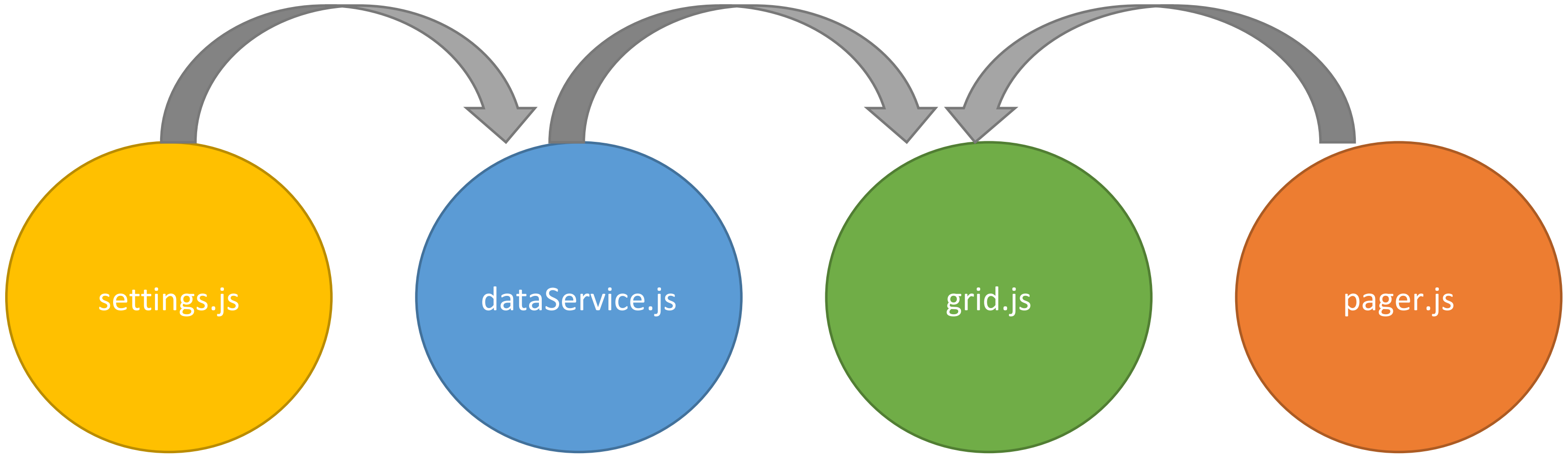
The Need for Modules

Why do we Need Modules?

```
<script src="scripts/jquery.js"></script>
<script src="scripts/bizrules.js"></script>
<script src="scripts/dataservice.js"></script>
<script src="scripts/grid.js"></script>
<script src="scripts/pager.js"></script>
<script src="scripts/settings.js"></script>
```

Are the scripts loaded in the proper order?

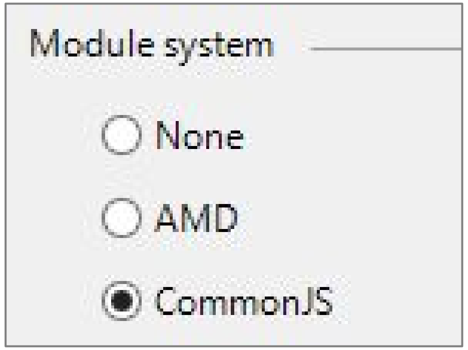
Dependency Chains and Modules



Creating and Using CommonJS/Node.js Modules

Creating CommonJS JavaScript Modules

CommonJS modules can be created by the TypeScript compiler in Visual Studio or using the command-line:



Module system

- None
- AMD
- [✓ CommonJS]

```
tsc --module commonjs myfile.ts
```

CommonJS Module Flow (Node.js)

server.ts

```
import http = require('http');
import msg = require('./lib/message');

http.createServer(function (req, res) {});
```

message.ts

```
export class Message {
    getText() : string {
        return 'Hello from the Message Module!';
    }
}
```

ES6 Modules in TypeScript

ES6 Modules in TypeScript

- TypeScript 1.5+ supports ES6 module syntax:

```
import * as Math from "my/math";
import { add, subtract } from "my/math";


// math.ts
export function add(x, y) { return x + y }
export function subtract(x, y) { return x – y }
export default function multiply(x, y) { return x * y }
```

Checkpoint

TypeScript supports AMD, CommonJS and ES6 modules

Modules allow dependency chains to be simplified

The export keyword is used with modules

Thanks for Coming!

http://codewithdan.me/ts-workshop

TypeScript Fundamentals

by Dan Wahlin and John Papa

Pluralsight

http://jpapa.me/danandjohnfun

Onsite Training

Looking for onsite training on TypeScript, JavaScript, Node.js, ASP.NET Core, Angular 2 and other Web technologies?

Contact us at training@codewithdan.com!