TypeScript

Wael Kdouh – Senior Consultant

Presenter Introduction

Wael Kdouh

- Senior Consultant, Premier Developer Practice, Microsoft Services
- Email: wael.kdouh@microsoft.com

Microsoft

Agenda

- Why TypeScript
- What is TypeScript
- What's new in TypeScript 2.0
- What's new in TypeScript 2.1

Blockers to Adopting TypeScript

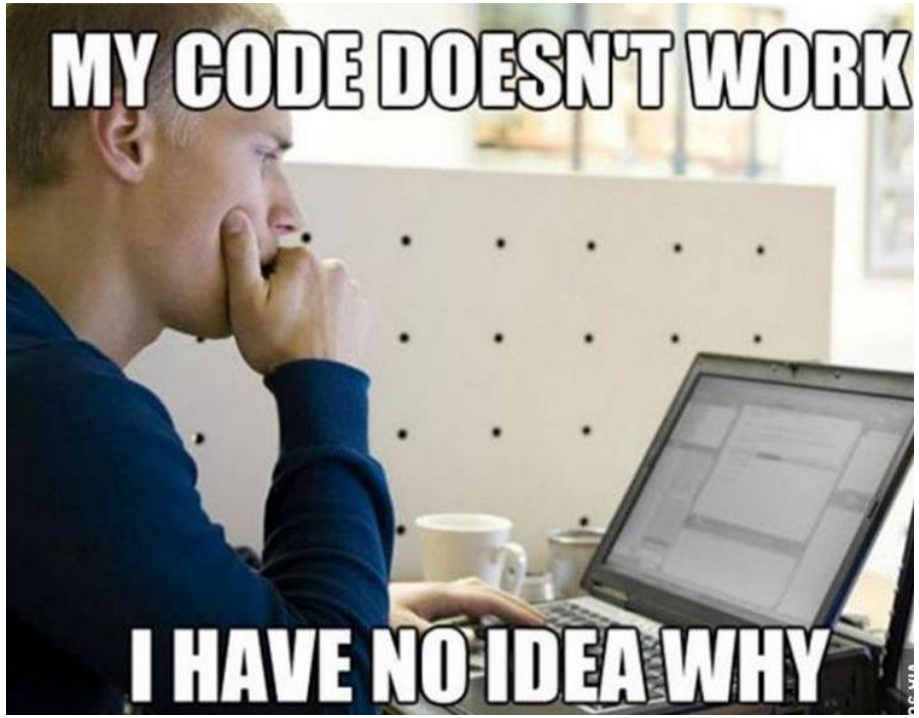
- We find that blockers to adoption of Typescript within large organizations is not individuals but teams

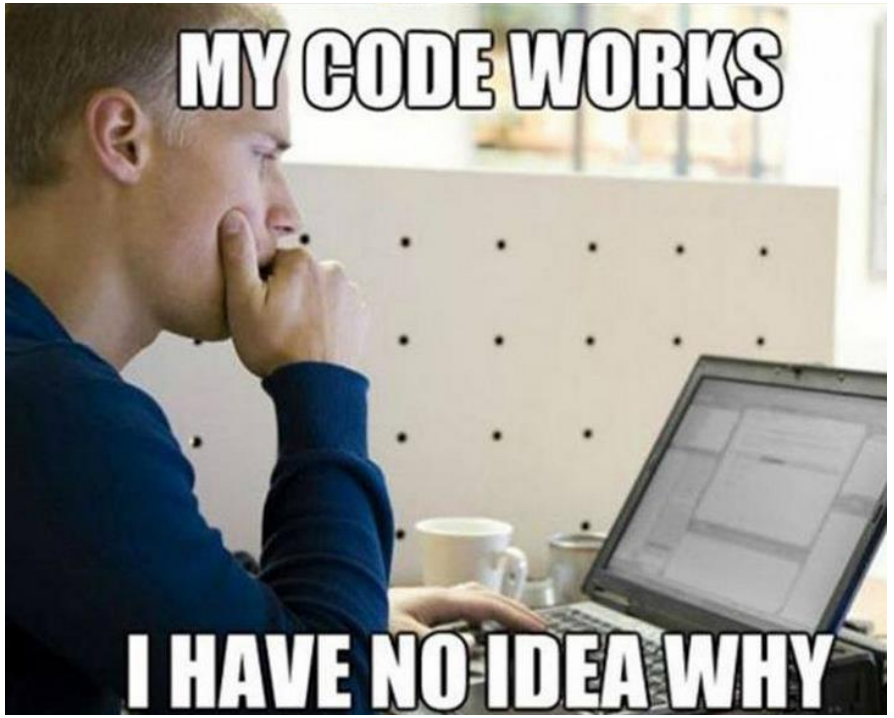
Why Use TypeScript?

Function Spaghetti Code

Ravioli Code
(JavaScript Patterns)

Why Use TypeScript?





JavaScript Dynamic Types

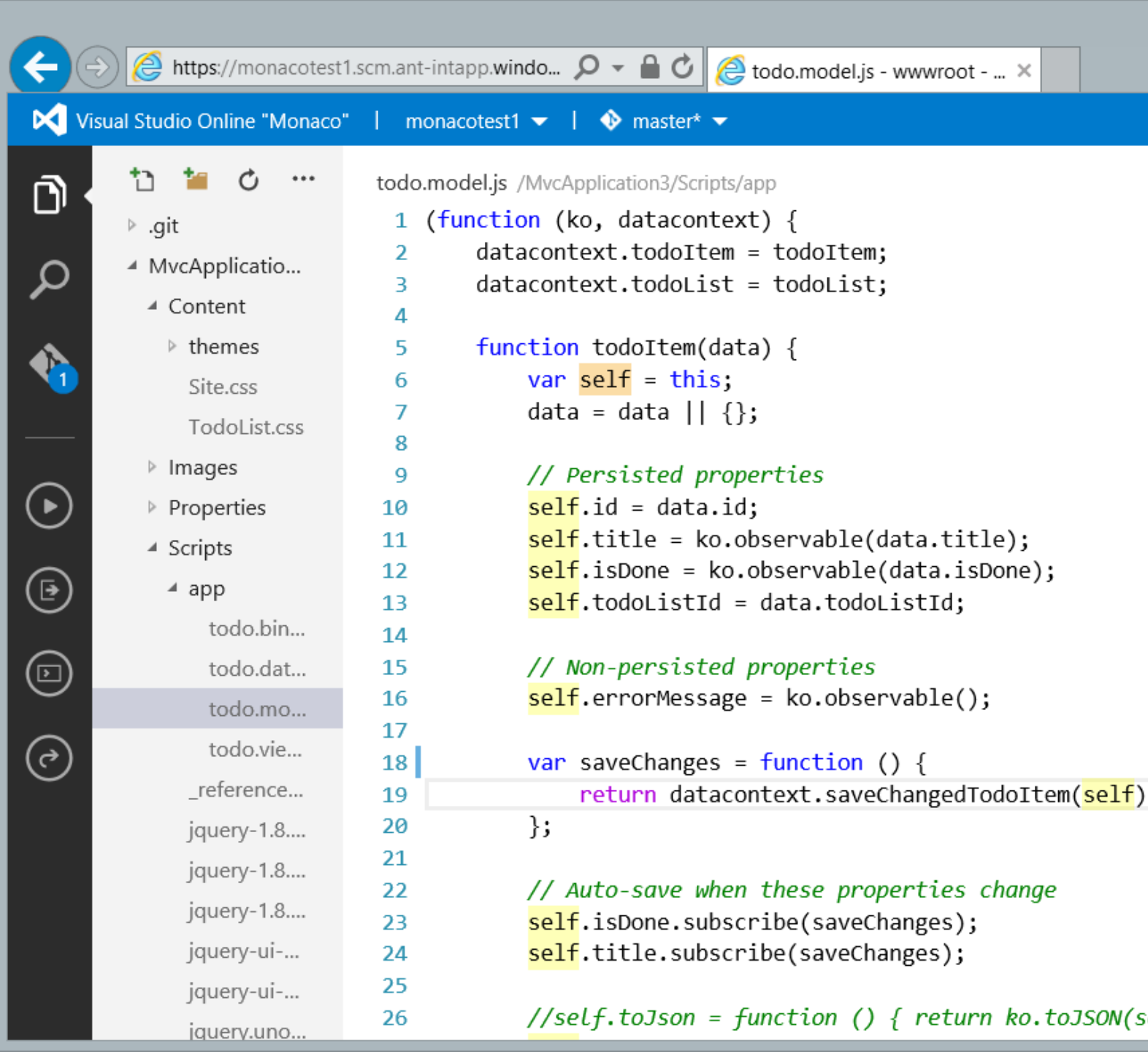
- JavaScript provides a dynamic type system
- The Good:
  - Variables can hold any object
  - Types determined on the fly
  - Implicit type coercion (ex: string to number)
- The Bad:
  - Difficult to ensure proper types are passed without tests
  - Not all developers use ===
  - Enterprise-scale apps can have 1000s of lines of code to maintain

Developers should be able to focus on **creating** amazing things

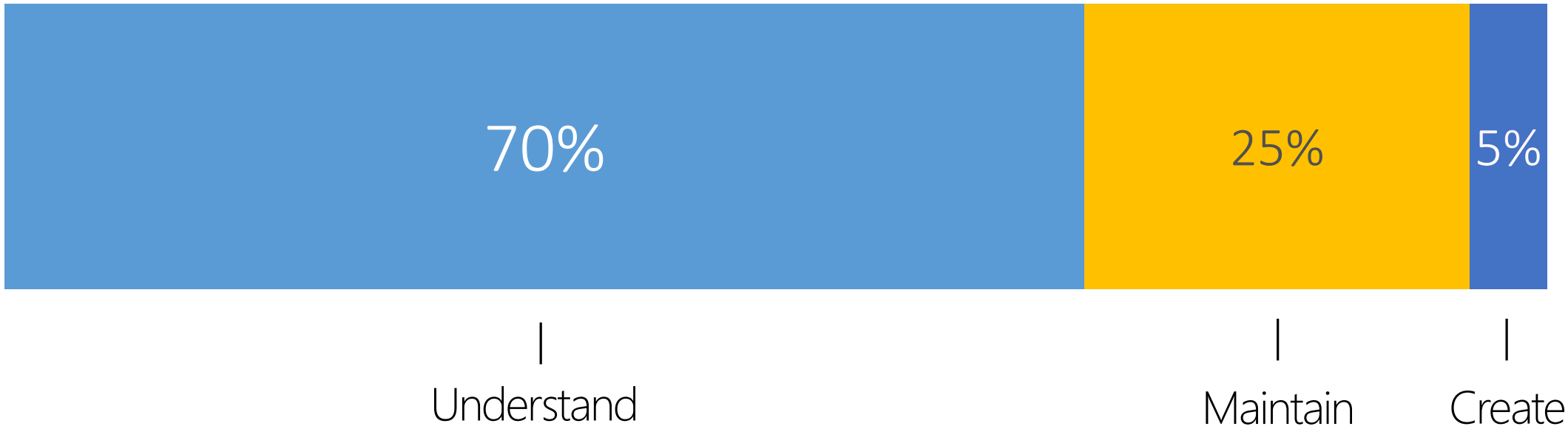
Why Use TypeScript?

Understand Code	Maintain Existing Code	Write New Code
- Read source code - Trace flow of execution - Read library documentation - Find bugs	- Prepare for new features - Fix bugs - Refactor code	- New functions/classes - New files or modules

Monaco


https://monacotest1.scm.ant-intapp.windo...
todo.model.js - wwwroot - ...
Visual Studio Online "Monaco"
monacotest1
master*
.git
MvcApplicatio...
Content
themes
Site.css
TodoList.css
Images
Properties
Scripts
app
todo.bin...
todo.dat...
todo.mo...
todo.vie...
_reference...
jquery-1.8....
jquery-1.8....
jquery-1.8....
jquery-ui-...
jquery-ui-...
jquery.uno...
todo.model.js /MvcApplication3/Scripts/app
(function (ko, datacontext) {
datacontext.todoItem = todoItem;
datacontext.todoList = todoList;
function todoItem(data) {
var self = this;
data = data || {};
// Persisted properties
self.id = data.id;
self.title = ko.observable(data.title);
self.isDone = ko.observable(data.isDone);
self.todoListId = data.todoListId;
// Non-persisted properties
self.errorMessage = ko.observable();
var saveChanges = function () {
return datacontext.saveChangedTodoItem(self)
};
// Auto-save when these properties change
self.isDone.subscribe(saveChanges);
self.title.subscribe(saveChanges);
//self.toJson = function () { return ko.toJSON(s

What If?



What If?

What is TypeScript?

"TypeScript is a typed superset of JavaScript that compiles to plain JavaScript"

~ typescriptlang.org

TypeScript is not a totally separate language from JavaScript. It's actually built on top of JavaScript, that's why it's a superset

Flexible Options

Any Browser

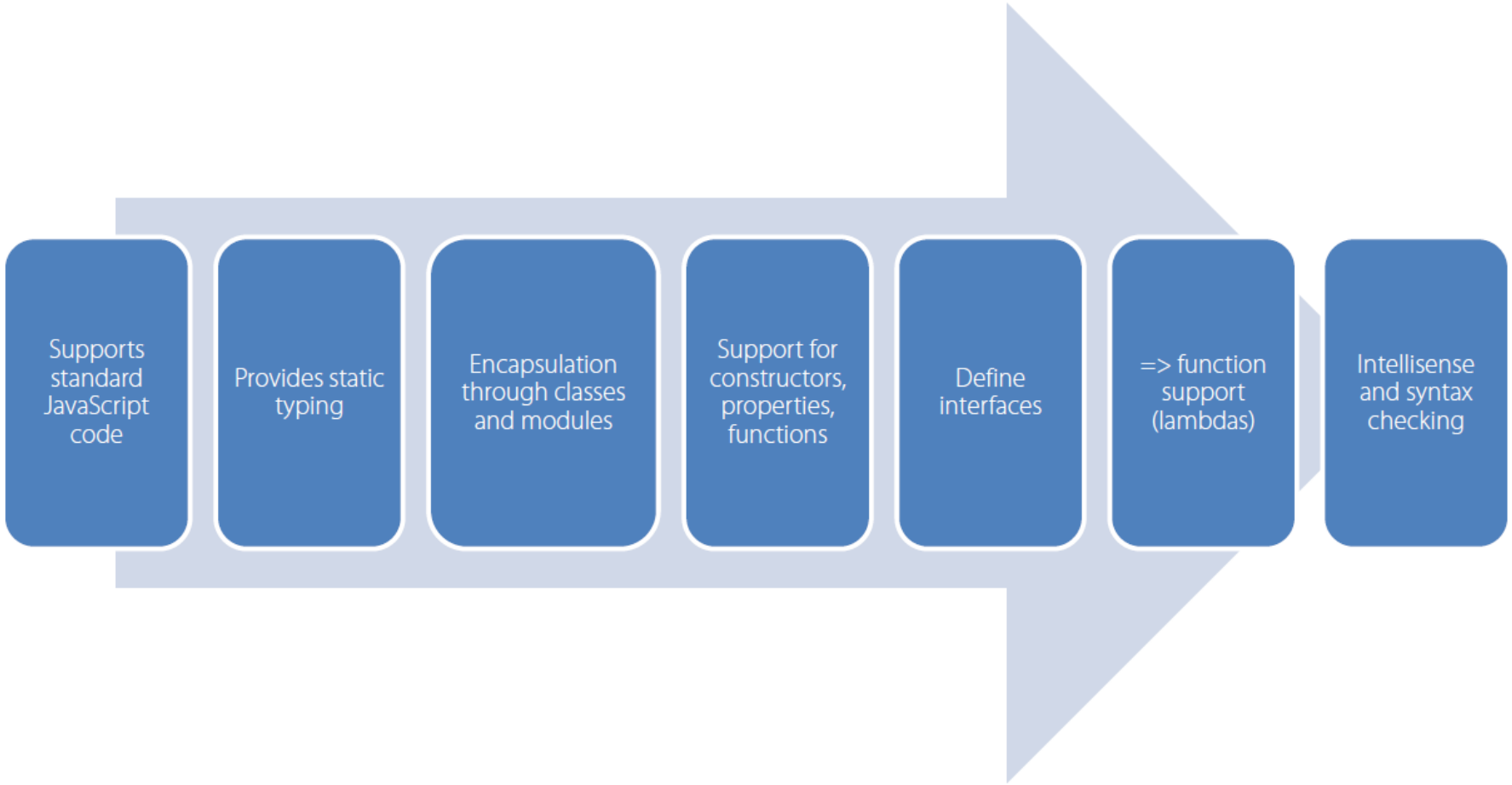
Any Host

Any OS

Open Source

Tool Support

Key TypeScript Features



TypeScript Compiler

Only use the tsc command line tool if you if your tool doesn't support the compilation process directly. For example with Visual Studio when you hit save on your ts file the JavaScript file will be updated in the background

tsc first.ts

How to Find TypeScript Version Under Visual Studio?

- Under the VS command window type **tsc –v**

- Navigate to C:\Program Files (x86)\Microsoft SDKs\TypeScript

Demo
JavaScript to TypeScript

JavaScript to TypeScript at Scale

- Flipping all of the JS files to TS files on large projects (e.g. 2000 files) is not approachable
  - In most cases this leaves you with hundreds of errors that you have to go fix at once

- Solution:
  - A flag called allowjs allows us to feed JavaScript files to TypeScript compiler
    - Allows incremental inclusion of existing js files
    - Basically it makes the conversion process easier
    - It also allows you to transpile ES2015 to ES5 the same way Babel does

Demo
AllowJS flag

Important Keywords and Operations

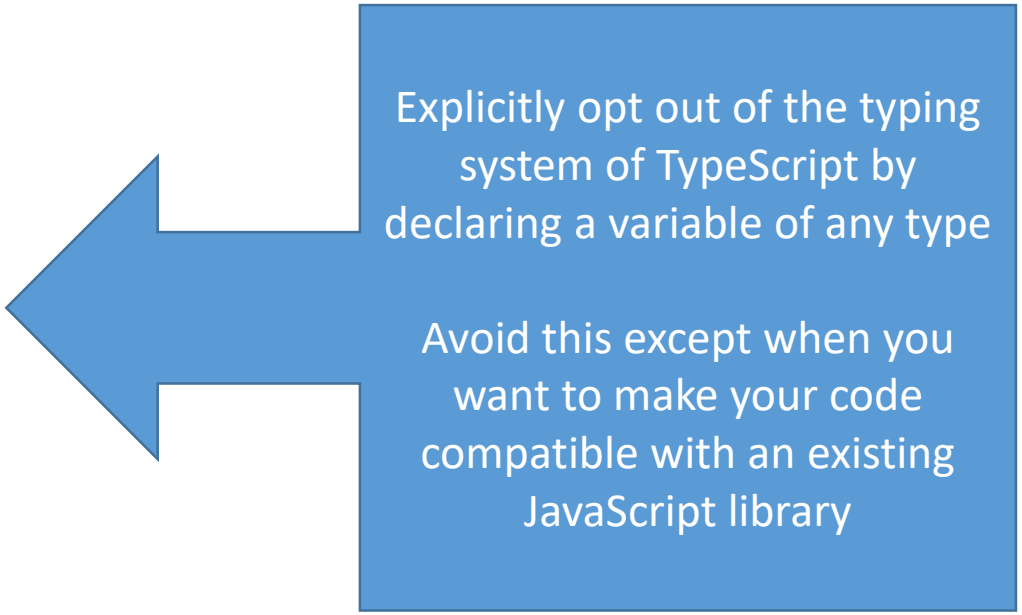
Keyword	Description
class	Container for members such as properties and functions
constructor	Provides initialization functionality in a class
exports	Export a member from a module
extends	Extend a class or interface
implements	Implement an interface
imports	Import a module
interface	Defines a code contract that can be implemented by types
module	Container for classes and other code
public/private	Member visibility modifiers
...	Rest parameter syntax
=>	Arrow syntax used with definitions and functions
<typeName>	< > characters use to cast/convert between types
:	Separator between variable/parameter names and types

Data Types

Built-in	Custom
boolean	enum
string	array
number	interface
	class

Any Type

- let notSure : any =4;
  notSure = “maybe a string instead”;
  notSure = false



Type Assertion

- Lets say for example we have a variable foo

```
let foo = {};
foo.bar = 123; // Error: property 'bar' does not exist on `{}`
foo.bas = 'hello'; // Error: property 'bas' does not exist on `{}`
```

- The code errors are caused by the inferred type of foo which is {} i.e. an object with zero properties. Therefore you are not allowed to add bar or bas to it. You can fix this simply by a type assertion as Foo

```
interface Foo {
    bar: number;
    bas: string;
}
let foo = {} as Foo;
foo.bar = 123;
foo.bas = 'hello';
```

- Doing so would let you take advantage of all the typescript related benefits such as code completion and type checking

Type Assertion

- as string vs. <string>
- There is an ambiguity in the language grammar when using <string> style assertions in JSX:

```
let foo = <string>bar;
</string>
```

- Therefore it is now recommended that you just use `as string` for consistency

```
let foo: any;
let bar = foo as string; // bar is now of type "string"
```

Type Assertion vs. Casting

- The reason why it's not called "type casting" is that casting generally implies some sort of runtime support. However type assertions are purely a compile time construct and a way for you to provide hints to the compiler on how you want your code to be analyzed

Assertion Considered Harmful

- In many cases assertion will allow you to easily migrate legacy code, however you should be careful with your use of assertions

- Take our original code as a sample, the compiler will not protect you from forgetting to actually add the properties you promised:

```
interface Foo {
    bar: number;
    bas: string;
}
let foo = {} as Foo;
// ahhhh .... forgot something?
```

Assertion Considered Harmful

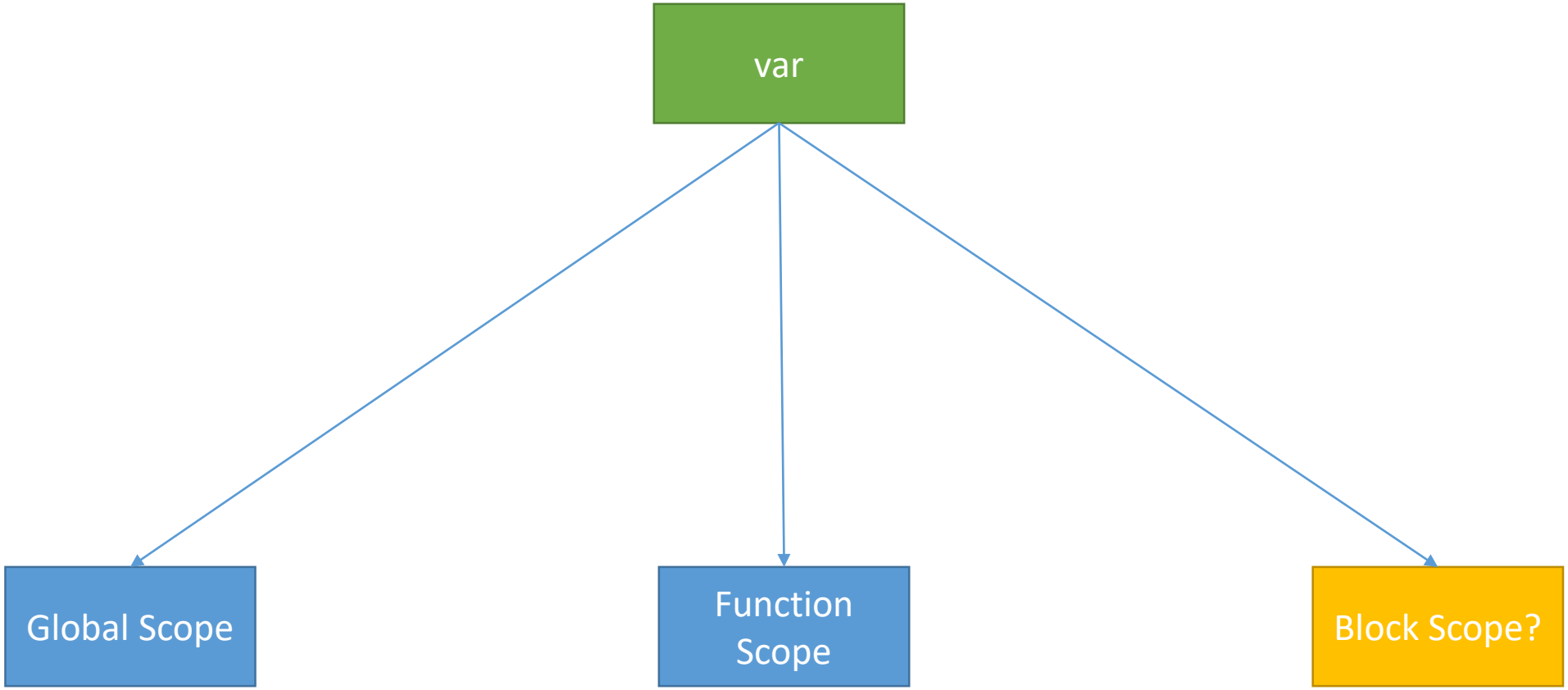
- Also another common thought is using an assertion as a means of providing autocomplete:

```
interface Foo {
    bar: number;
    bas: string;
}
let foo = <Foo>{
    // the compiler will provide autocomplete for properties of Foo
    // But it is easy for the developer to forget adding all the properties
    // Also this code is likely to break if Foo gets refactored (e.g. a new
       property added)
};
```

Start using “let” Everywhere

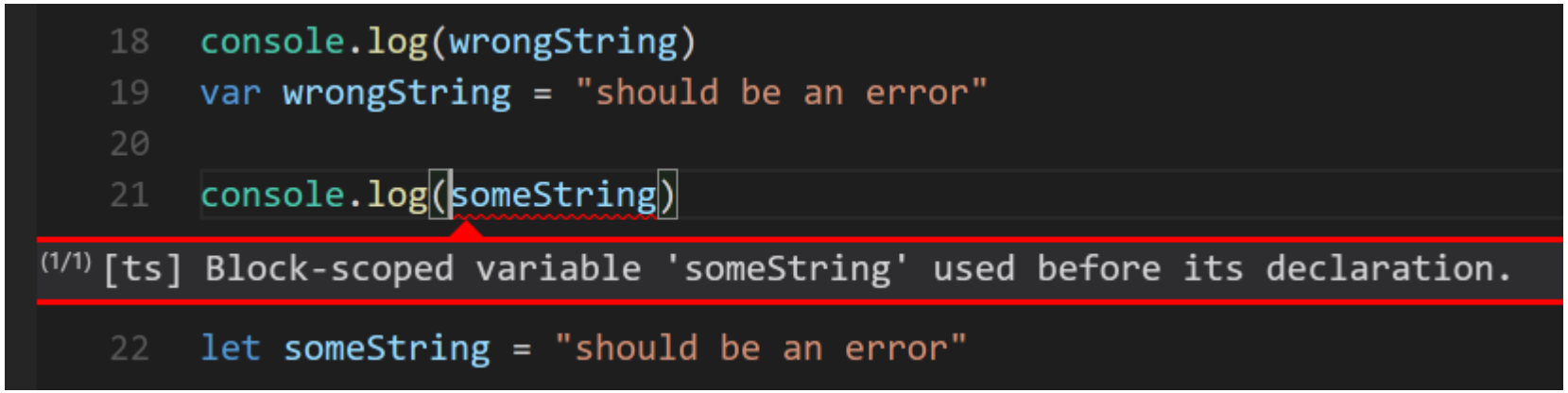
- Used to define a variable

- What about var?



Start using “let” Everywhere

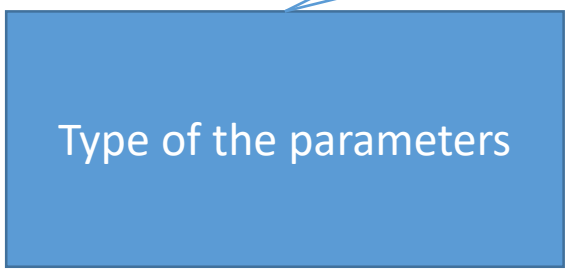
- It also helps us avoid issues like these:



Demo
let

Adding Type Annotations To Functions

```
function wrongWay(value1, value2) {
    return "This is the old world."
}

function rightWay(score:number, message?:string):string {
    return "welcome to the world of TypeScript."
}
```



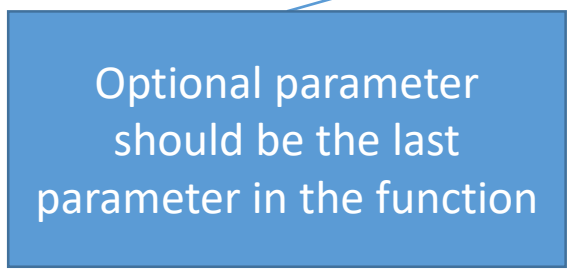
Adding Type Annotations To Functions

```
function wrondWay(value1, value2) {
    return "This is the old world."
}

function rightWay(score:number, message?:string):string {
    return "welcome to the world of TypeScript."
}
```

Specify that this parameter is optional. By default all parameters are required unlike JavaScript where by default they are optional which can cause some issues

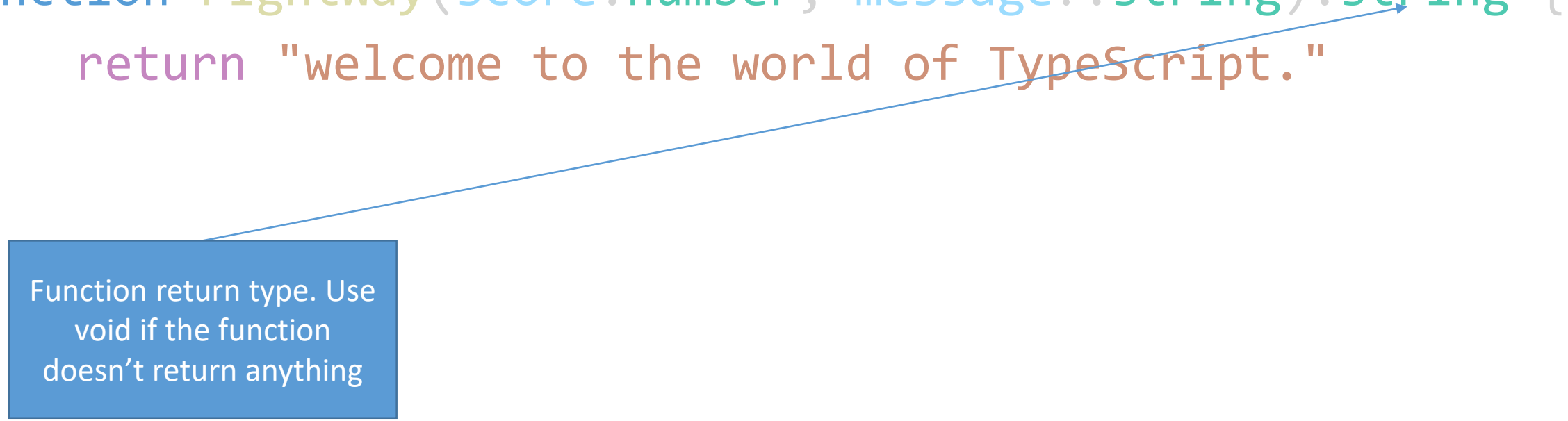
Adding Type Annotations To Functions

```
function wrondWay(value1, value2) {
    return "This is the old world."
}

function rightWay(score:number, message?:string):string {
    return "welcome to the world of TypeScript."
}
```



Adding Type Annotations To Functions

```
function wrondWay(value1, value2) {
    return "This is the old world."
}

function rightWay(score:number, message?:string):string {
    return "welcome to the world of TypeScript."
}
```



Adding Type Annotations To Functions

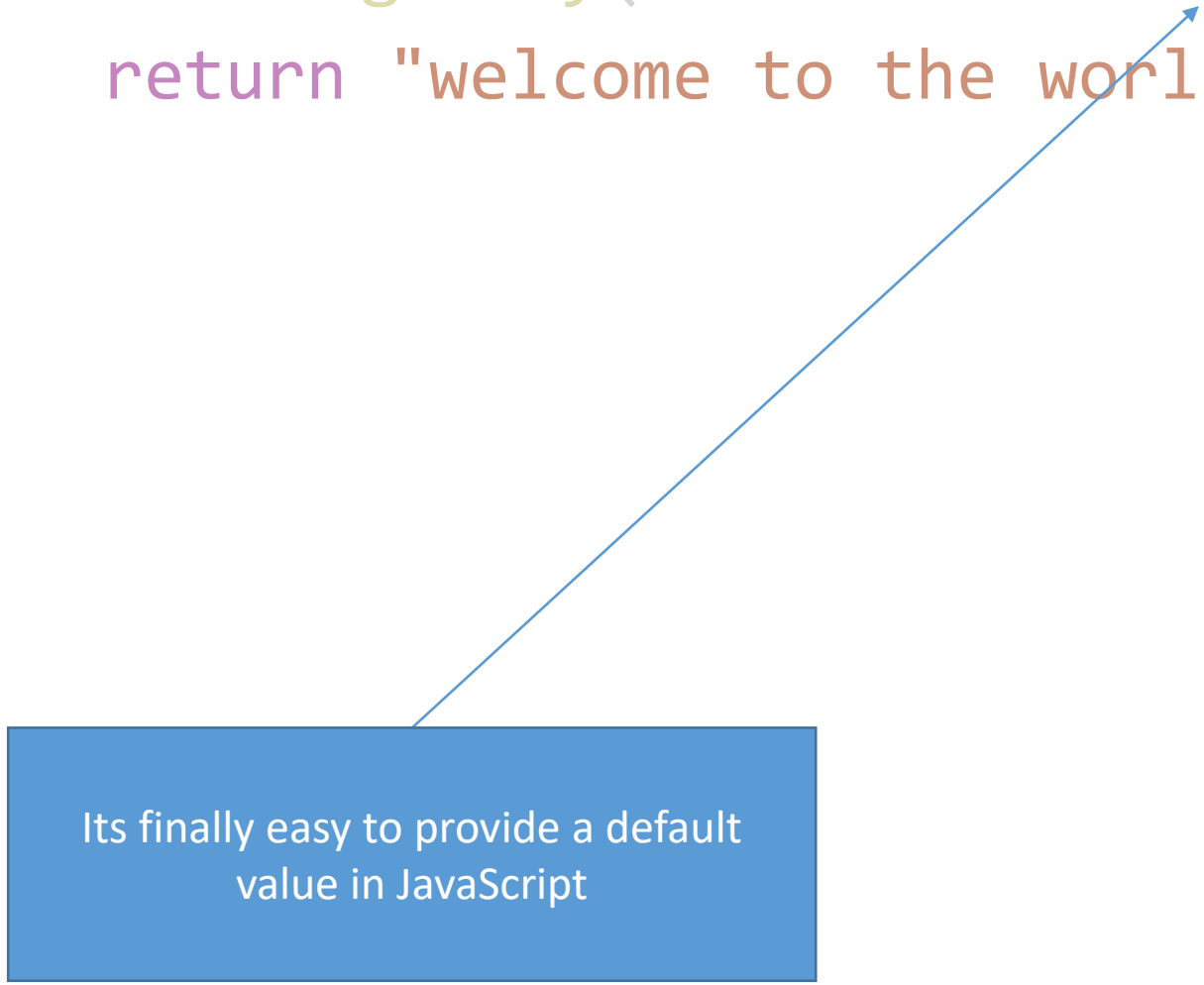
```
function wrondWay(value1, value2) {
    return "This is the old world."
}

function rightWay(score:number, message?:string):string {
    return "welcome to the world of TypeScript."
}
```

You can activate the - -noImplicitAny flag to prevent this from compiling

Demo
-- noImplicitAny

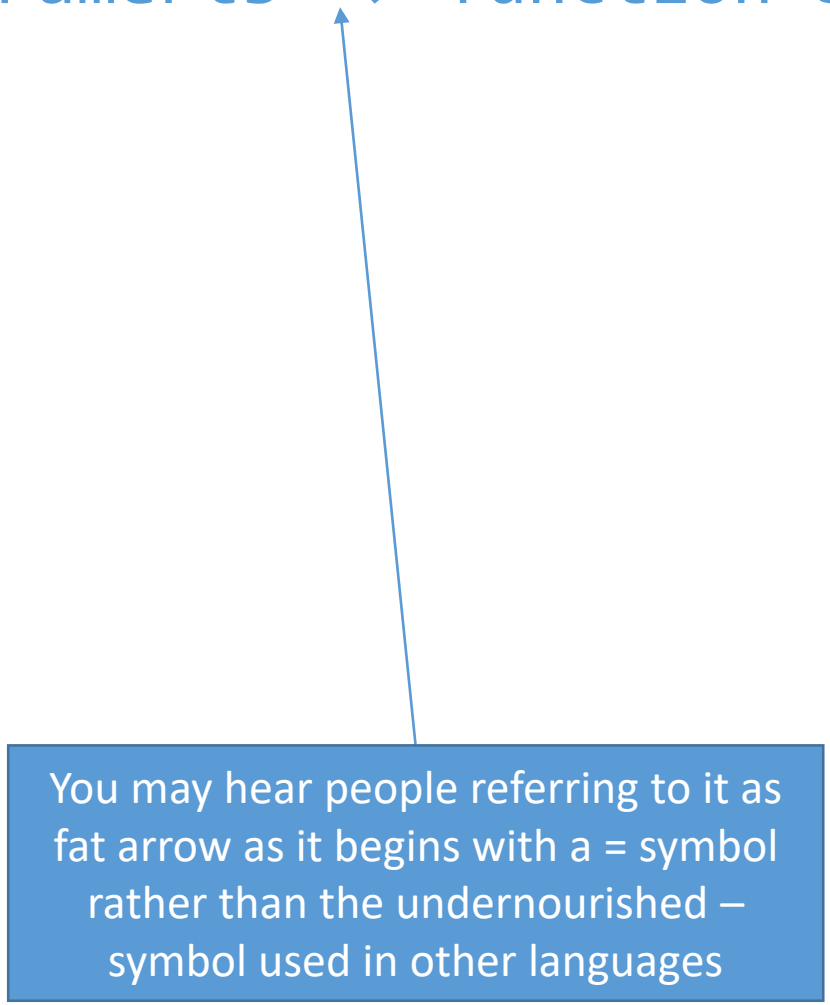
Default-Initialized Parameters

```
function rightWay(score:number=42, message?:string):string {
    return "welcome to the world of TypeScript."
}
```



Arrow Functions

```
paramerts => function body
```



Arrow Functions

```
paramerts => function body


let squareIt = x => x*x;


let addIt = (x,y) => x+y; // parentheses required for more
                          // than one parameter
let greeting = () => console.log("no parameters passed");
```

Arrow Functions

- Proper use case of Arrow functions

```
let scores: number[] = [70, 125, 85, 110];
let highScores: number[];
highScores = scores.filter((element, index, array) => {
    if (element > 100) {
        return true;
    }
});
```

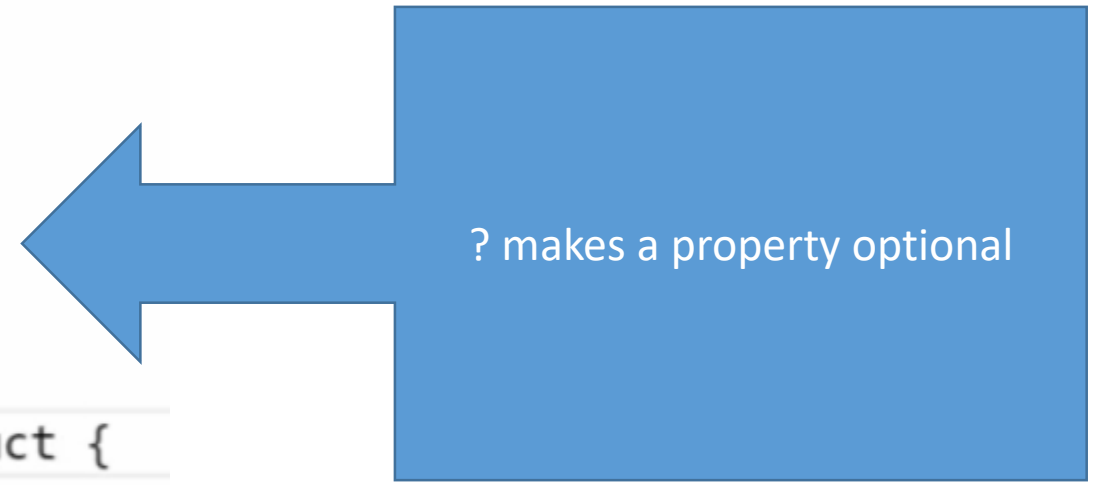
Creating Custom Types In TypeScript

Creating Custom Types In TypeScript

- Creating your own custom types In TypeScript is all about two TypeScript language features:
  - Interfaces
  - Classes

Interfaces

```
interface Product {
    name: string
    price: number
    category?: ProductCategory
}

class CocaCola implements Product {
    name = "Coca-Cola"
    price = 2.30
    category = new SodaCategory()
}
```



Interfaces

```
interface Product {
    name: string
    price: number
    category?: ProductCategory
}

class CocaCola implements Product {
    name = "Coca-Cola"
    price = 2.30
    category = new SodaCategory()
}
```

```
class productFactory {
    static GetProduct(): Product {
        let random = Math.floor(Math.random() * 11);
        switch(random) {
            case 0: return new CocaCola()
            case 1: return new Fanta()
            case 2: return new Sprite()
            case 3: return new Peanuts()
            case 4: return new Cashews()
            case 5: return new Plain()
            case 6: return new Cheddar()
            case 7: return new Mints()
            case 8: return new Gummies()
            case 9: return new Hersey()
            case 10: return new MilkyWay()
        }
    }
}
```

Interfaces

```
interface Employee {
    name:string;
    age:number;
}

interface Manager extends Employee {
    department: string;
    numberOfEmployees: number;
    scheduleMeeting: (topic: string) => void;
}
```

Methods don’t provide any implementation, just a signature

Structural Type System

- Typescript uses what is calls a structural type system uses the structure of objects to determine their compatibility

```
interface Employee {
    name:string;
    age:number
}

let manager = {
    name:"Wael",
    age: 42,
    occupation:"Magician"
}


let newEmployee:Employee = manager;
```

This object has three properties, but since it has all of the required properties of the Employee interface it can be used anywhere the Employee interface is expected although it was not explicitly declare that it represents an Employee

Classes

```
class VendingMachine {
    private paid = 0
    acceptCoin = (coin: Quarter): void => {
        this.paid = this.paid + coin.value
    }
}
```

```
var VendingMachine = (function () {
    function VendingMachine() {
        var _this = this;
        this.paid = 0;
        this.acceptCoin = function (coin) {
            _this.paid = _this.paid + coin;
        };
    }
    return VendingMachine;
}());
```

Constructors

- Special types of function that gets executed when a new instance is created

```
class Person {
    constructor() {
        console.log('Creating a new person')
    }
}

class Employee extends Person {
    readonly myReadonlyProperty: string;
    constructor(value: string) {
        super();
        this.myReadonlyProperty = value;
    }
}
```

Needs to call super when the child class has a constructor and extends a parent class

Can only be initialized when they are declared or inside a constructor

Type Inference

```
class Quarter {
    value: number = .25;
    getImageUrl (): string {
        this.value = "blah";
        return "img/Quarter.png";
    }
}
```

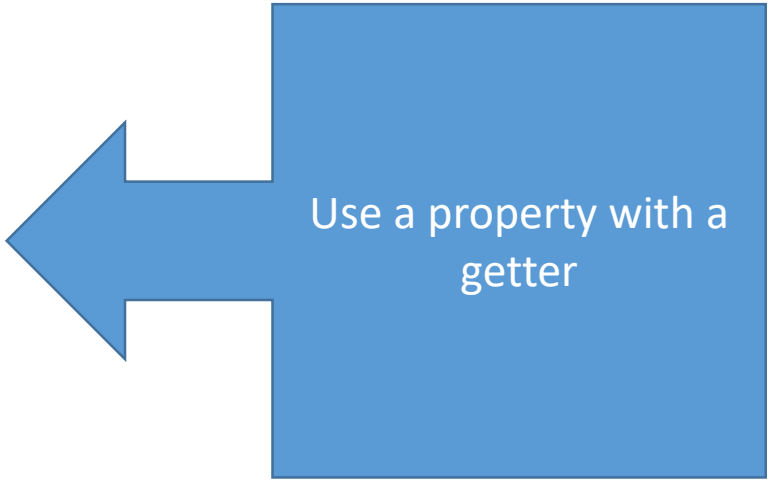
Explicit typing prevents the assignment of a string to a number

```
class Quarter {
    value = .25;
    getIm  IDBCursorWithValue interface IDBCursorWithValue var IDBCursorWithV
        this.value = "blah";
        return "img/Quarter.png";
    }
}
```

You will still get an error even if you do not specify the type explicitly

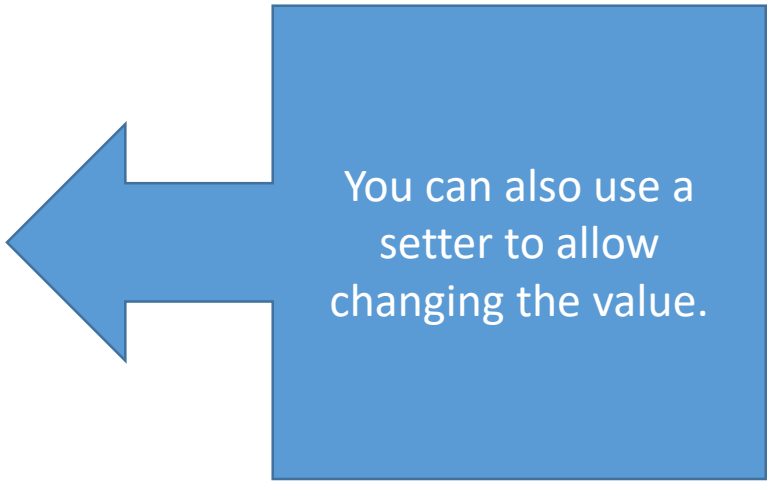
Properties

```
class Quarter {
    private value = .25;
    get Value() {
        return this.value;
    }
    getImageUrl () {
        return "img/Quarter.png";
    }
}

var coin = new Quarter();
var value = coin.Value;
```



Properties

```
    private value = .25;
    get Value() {
        return this.value;
    }
    set Value(newValue: number) {
        this.value = newValue;
    }
    getImageUrl () {
        return "img/Quarter.png";
    }
}
```



Access Modifiers

```
class Quarter {
    value = .25;
    getImageUrl () {
        return "img/Quarter.png";
    }
}

var coin = new Quarter();
coin.
```

```
getImageUrl (method) Quarter.getImageUrl(): string
value
```

All members in Typescript classes are public by default

What if we don't want to give access to "value"?

Access Modifiers

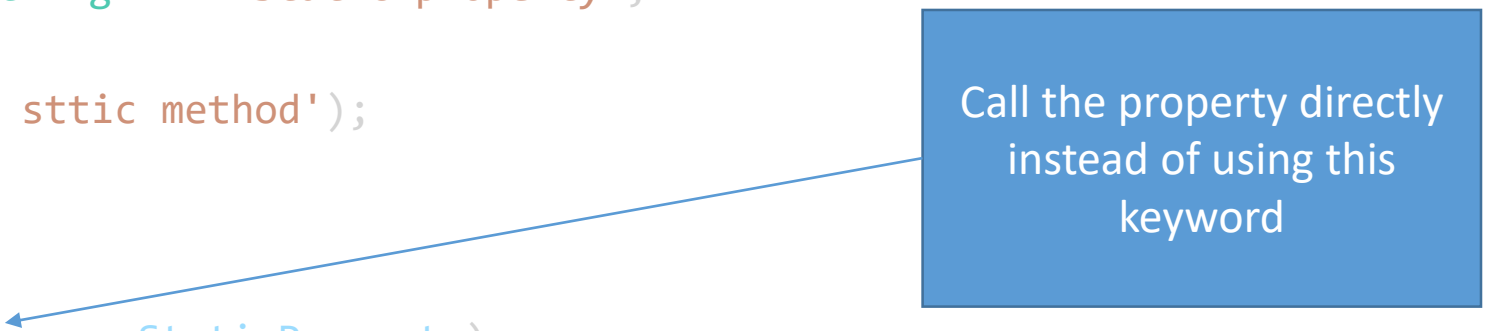
```
class Quarter {
    private value = .25;
    getImageUrl () {
        return "img/Quarter.png";
    }
}

var coin = new Quarter();
coin.value = .5;
```

Setting private makes it inaccessible outside the class.

But what if you want to allow read only access?

Static and Instance Members

```
class MyStaticClass {
    static myStaticProperty: string = "A static property";
    static myStaticMethod(){
        console.log('This is a sttic method');
    }

    nonStaticMethod() {
        console.log(MyStaticClass.myStaticProperty);
    }
}

MyStaticClass.myStaticProperty = "This works"
```



Modules

- What are modules needed for?
  - Encapsulation
  - Reusability
  - Create higher-level abstractions

- TypeScript modules adopted the ES2015 module system

Modules

```
export class Person {
    constructor() {
        console.log('Creating a new person')
    }
}

export class Employee extends Person {
    readonly myReadonlyProperty: string;
    constructor(value: string) {
        super();
        this.myReadonlyProperty = value;
    }
}
```

Use the export keyword to export

```
import {Employee as APerson,Person} from "./file";
```

Use the import keyword to import

Use an Alias for the imported class

Modules

```
export default class Person {
    constructor() {
        console.log('Creating a new person')
    }
}

export class Employee extends Person {
    readonly myReadonlyProperty: string;
    constructor(value: string) {
        super();
        this.myReadonlyProperty = value;
    }
}
```

Default keyword being used

```
import {Employee as APerson} from "./file";
import Person from "./file";
```

Curley Braces not required anymore

Modules

```
class Person {
    constructor() {
        console.log('Creating a new person')
    }
}

class Employee extends Person {
    readonly myReadonlyProperty: string;
    constructor(value: string) {
        super();
        this.myReadonlyProperty = value;
    }
}
```

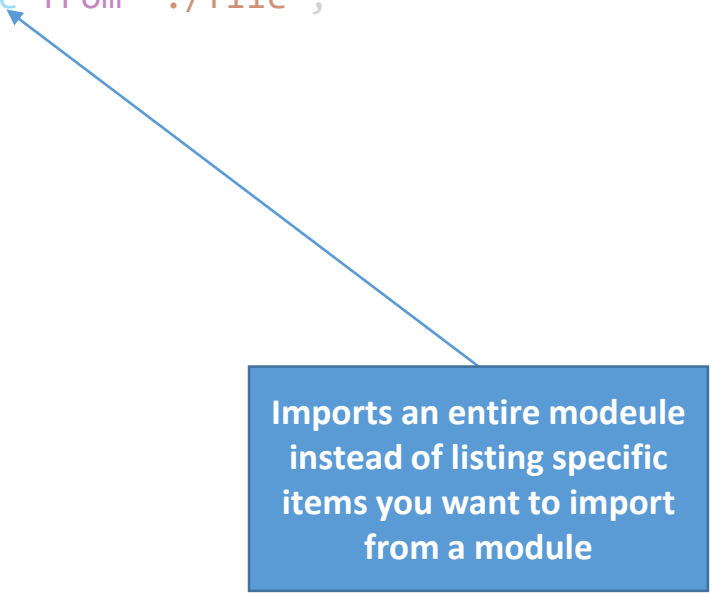
```
import {Person,APerson} from "./file";
```


```
export {Person, Employee as APerson}
```

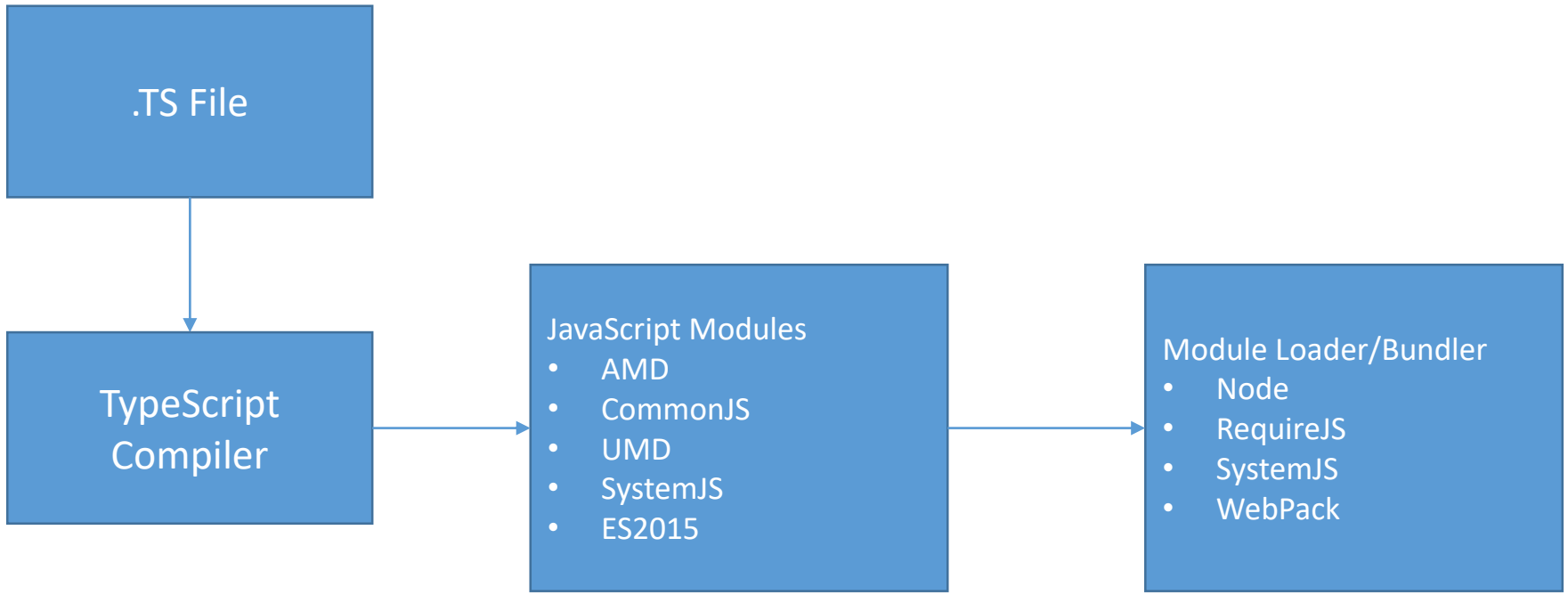
Modules

```
class Person {
    constructor() {
        console.log('Creating a new person')
    }
}

class Employee extends Person {
    readonly myReadonlyProperty: string;
    constructor(value: string) {
        super();
        this.myReadonlyProperty = value;
    }
}



export {Person, Employee as APerson}
```

```
import * as People from "./file";
```



Modules


.TS File
TypeScript Compiler
JavaScript Modules
• AMD
• CommonJS
• UMD
• SystemJS
• ES2015
Module Loader/Bundler
• Node
• RequireJS
• SystemJS
• WebPack

Relative vs. Non-relative Imports

- In order to understand how Typescript resolves the location of the modules you need to understand the difference between relative imports and non-relative imports

Relative Imports

- Direct the compiler to a specific location on the file system where the module can be found

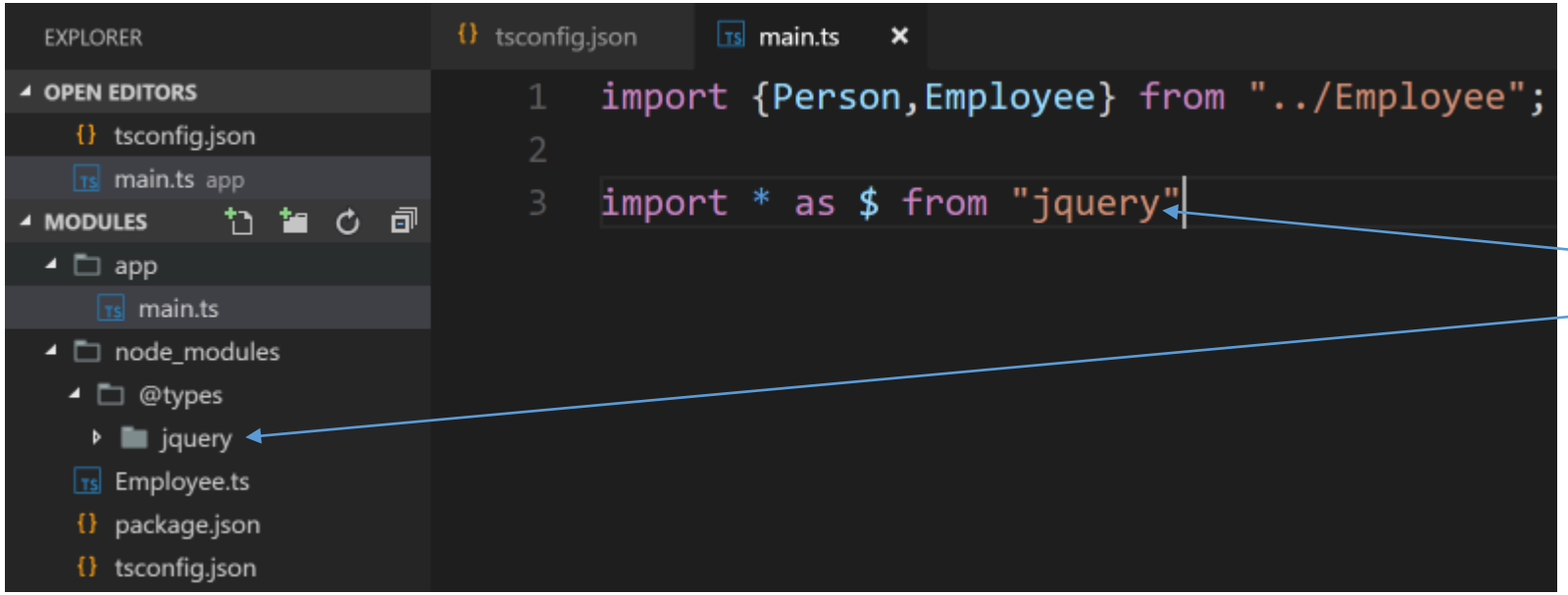
- All relative references start with /,./,or ../

- Use relative imports when referring to your own modules

Non-Relative Imports

- Similar to relative imports but they don't include any reference to a directory structure for the module

- Use non-relative imports when referring to third party modules



No references to paths are included

Module Resolution Strategies

- Steps to resolve a module
  - Step 1: Check if relative or non-relative reference
  - Step 2: Attempt to locate the module using the configured Module Resolution Strategy

- Module Resolution Strategy is set using the --moduleResolution flag

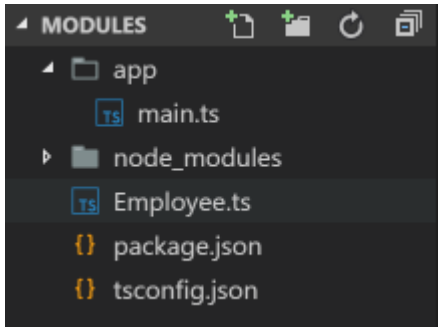
- The values could be Classic | Node

Module Resolution Strategies

Classic	Node
Default when emitting AMD, System, or ES2015	Default when emitting CommonJs or UMD modules
Simple module resolution	Closely mirrors Node module resolution
Less Configurable	More Configurable
This used to be TypeScript's default resolution strategy. Nowadays, this strategy is mainly present for backward compatibility	Use Node moving forward

Resolving Classic Relative Imports

```
// File Location: /modules/app
import {Person,Employee} from "../Employee";
```



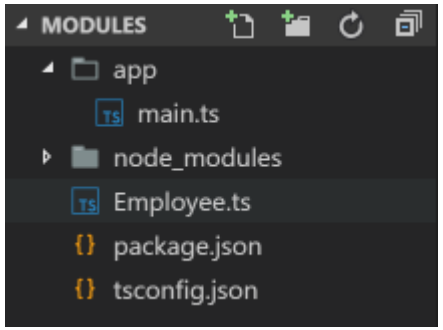
- Looks under

  `/modules/Employee.ts`

  `/modules/Employee.d.ts`

Resolving Classic Non-Relative Imports

```
// File Location: /modules/app
import {Person,Employee} from "Employee";
```



- Looks under

  `/modules/app/Employee.ts`

  `/modules/app/Employee.d.ts`

- However if the file is not found at the same level it will go up one level on the directory structure and search for the same two files there

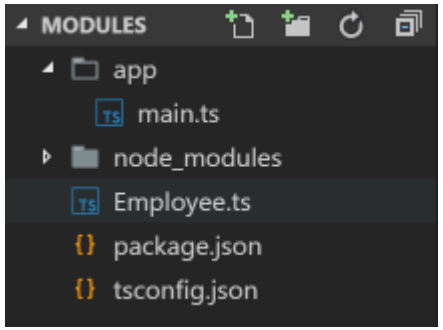
  `/modules/Employee.ts`

  `/modules/Employee.d.ts`

- It will continue searching up the directory structure until it runs out of directories

Resolving Node Relative Imports

```
// File Location: /modules/app
import {Person,Employee} from "../Employee";
```

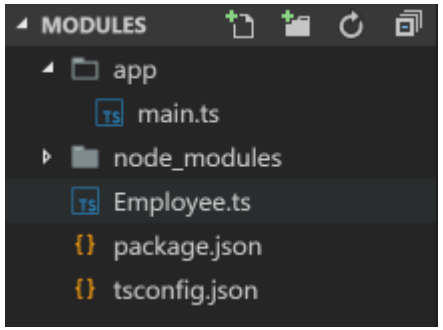


- Looks under
  - /modules/Employee.ts
  - /modules/Employee.d.tsx
  - /modules/Employee.d.ts

javaScript equivalent of jsx files used for frameworks like react

Resolving Node Non-Relative Imports

```
// File Location: /modules/app
import {Person,Employee} from "Employee";
```



- Looks under

  `/modules/app/Employee.ts(Employee.tsx,Employee.d.ts)`

- However if the file is not found at the same level it will go up one level on the directory structure and search for the same two files there

  `/modules/Employee.ts(Employee.tsx,Employee.d.ts)`

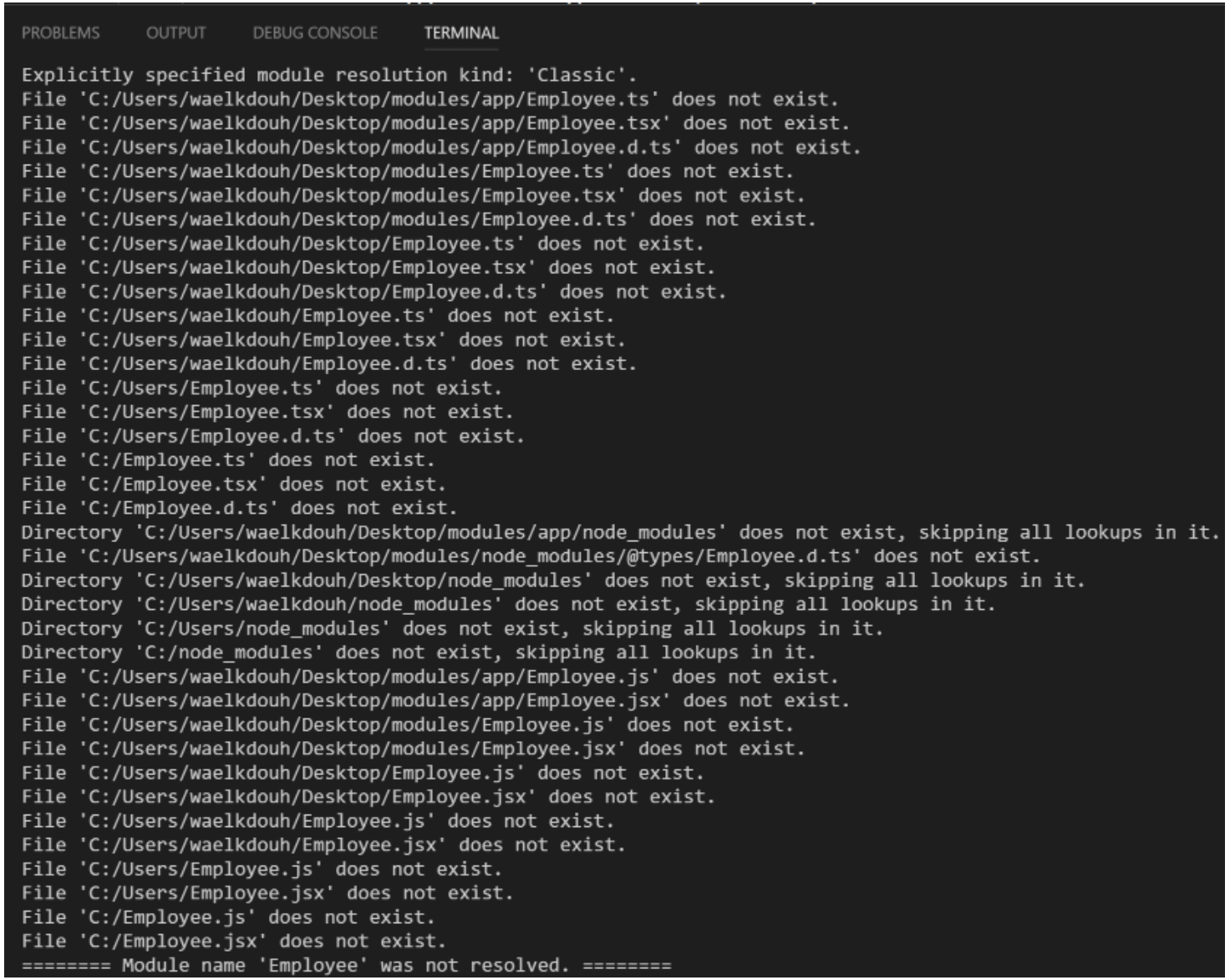
- It will continue searching up the directory structure until it runs out of directories

- Finally it will look under

  `/modules/node_modules/@types/Employee.d.ts`

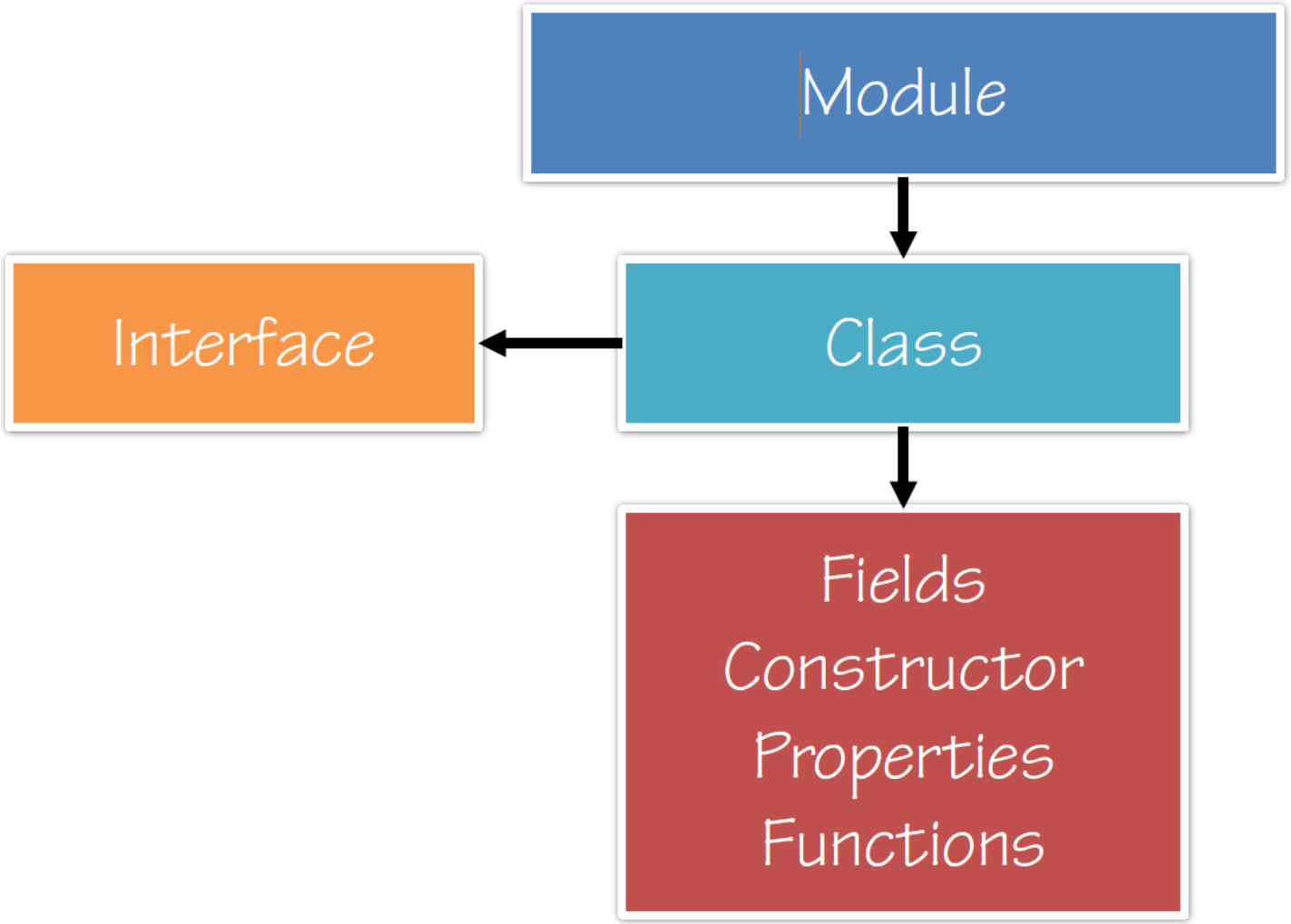
Module Resolution Tip

Use the --traceResolution flag to enable tracing of the name resolution process



Demo
Modules

Code Hierarchy



TypeScript 2

Packed with Features

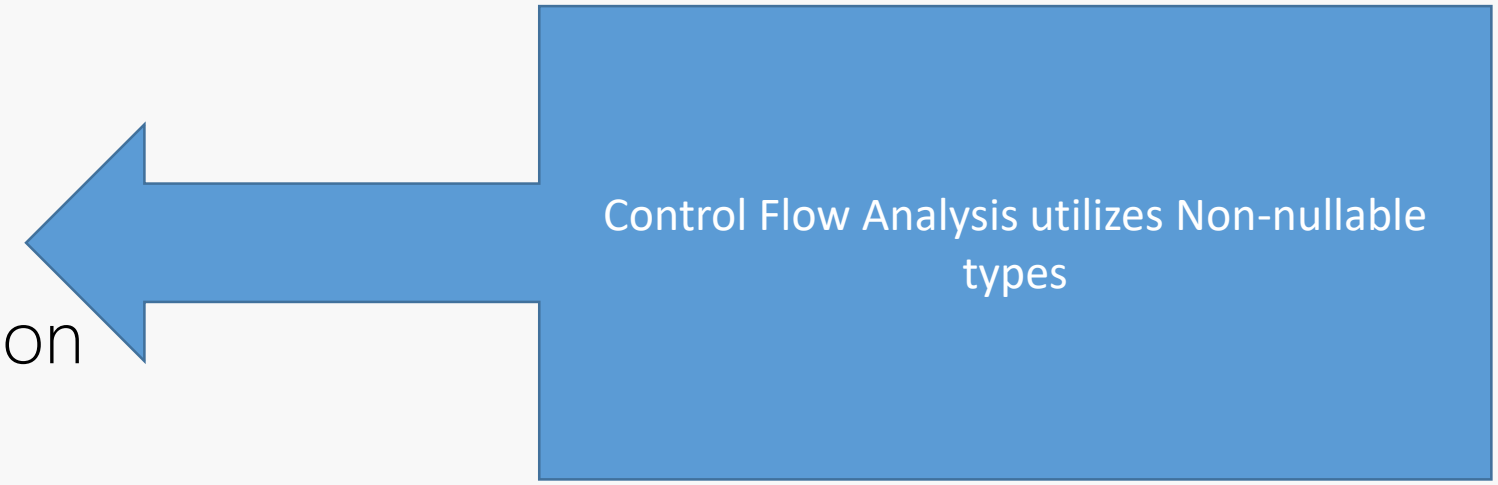
- Non-nullable types
- Control flow analysis
- Discriminated Union types
- Never types
- Read-only properties
- This types for functions
- Glob support in tsconfig
- New module resolution
- Quick ambient modules
- @types .d.ts acquisition
- UMD module definitions
- Optional class properties
- Private constructors
- So much more…

Packed with Features

- Non-nullable types
- Control flow analysis
- Discriminated Union types
- Never types
- Read-only properties
- This types for functions
- Glob support in tsconfig
- New module resolution
- Quick ambient modules
- @types .d.ts acquisition
- UMD module definitions
- Optional class properties
- Private constructors
- So much more…

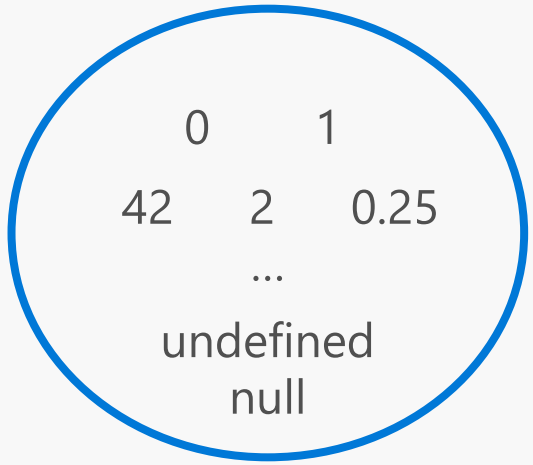
Packed with Features

- Non-nullable types
- Control flow analysis
- @types .d.ts acquisition

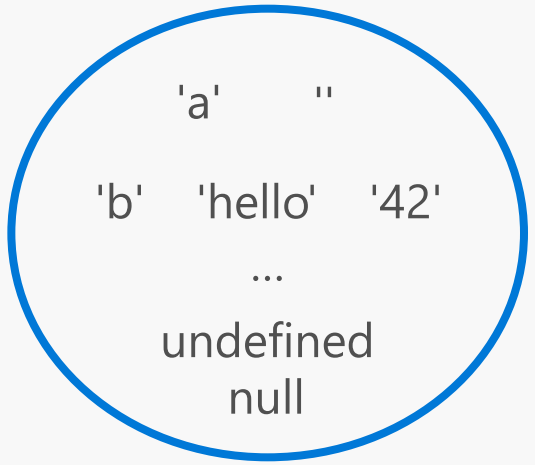


Nullable types

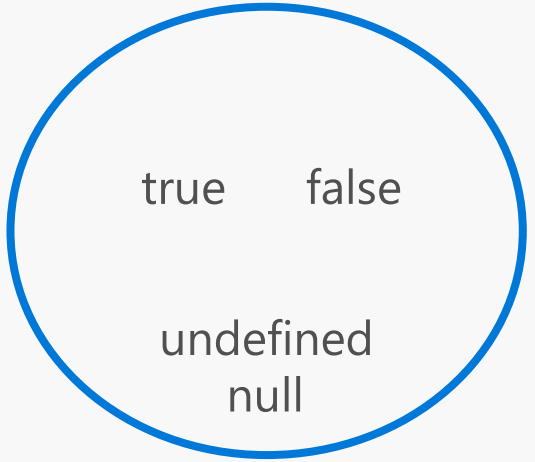
number



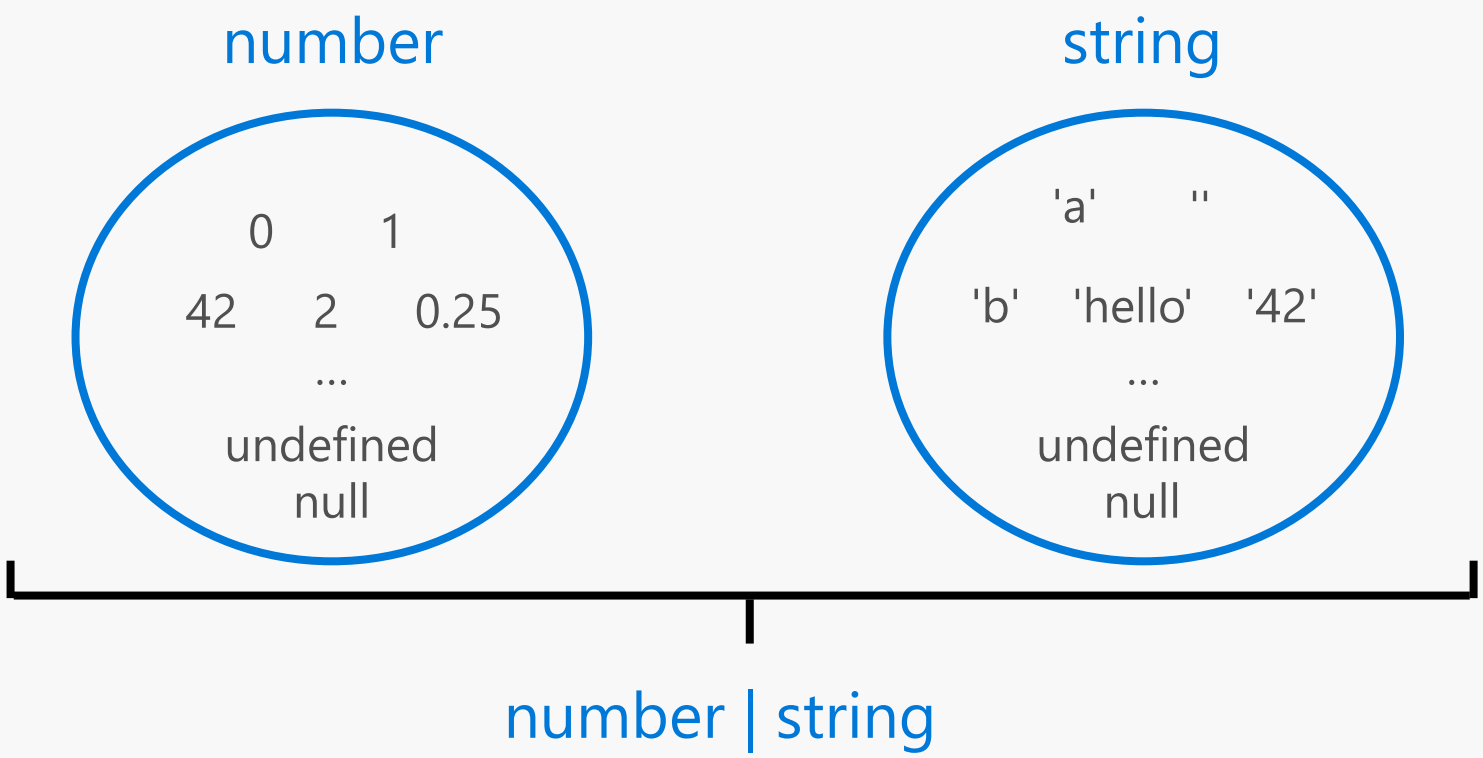
string



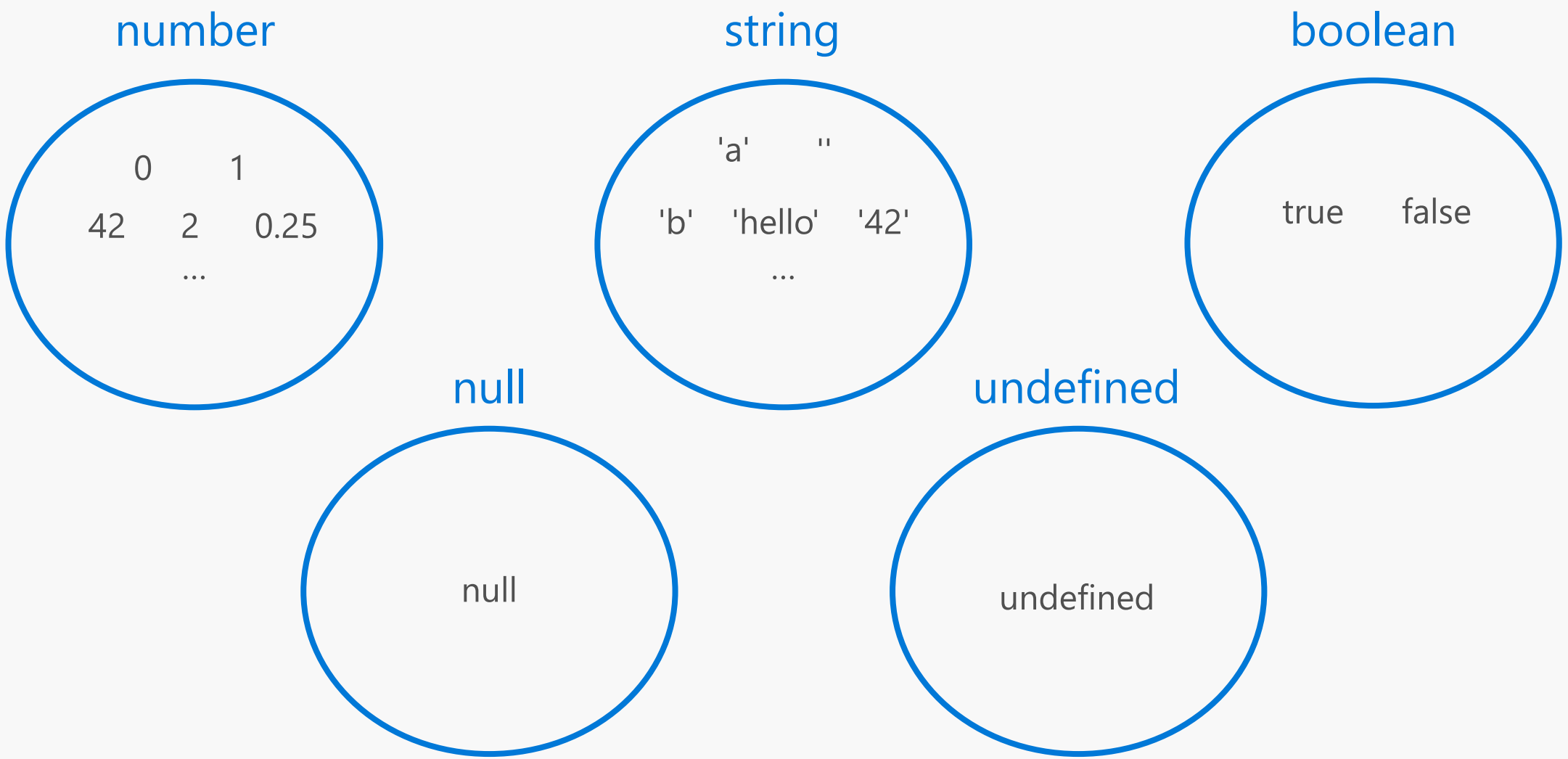
boolean



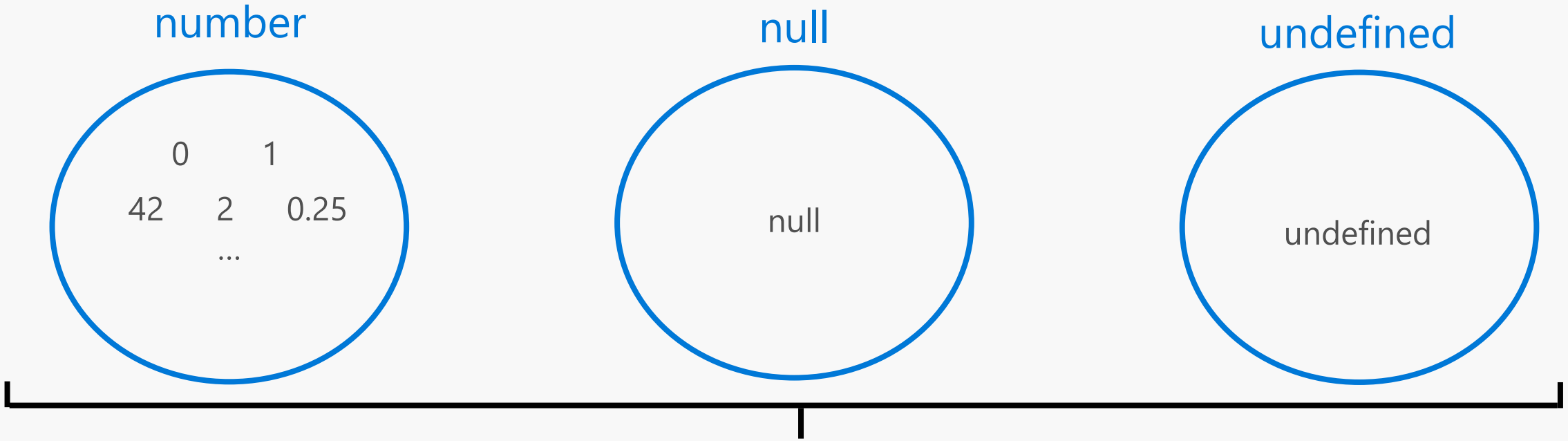
Nullable types



Non-Nullable types



Non-Nullable types

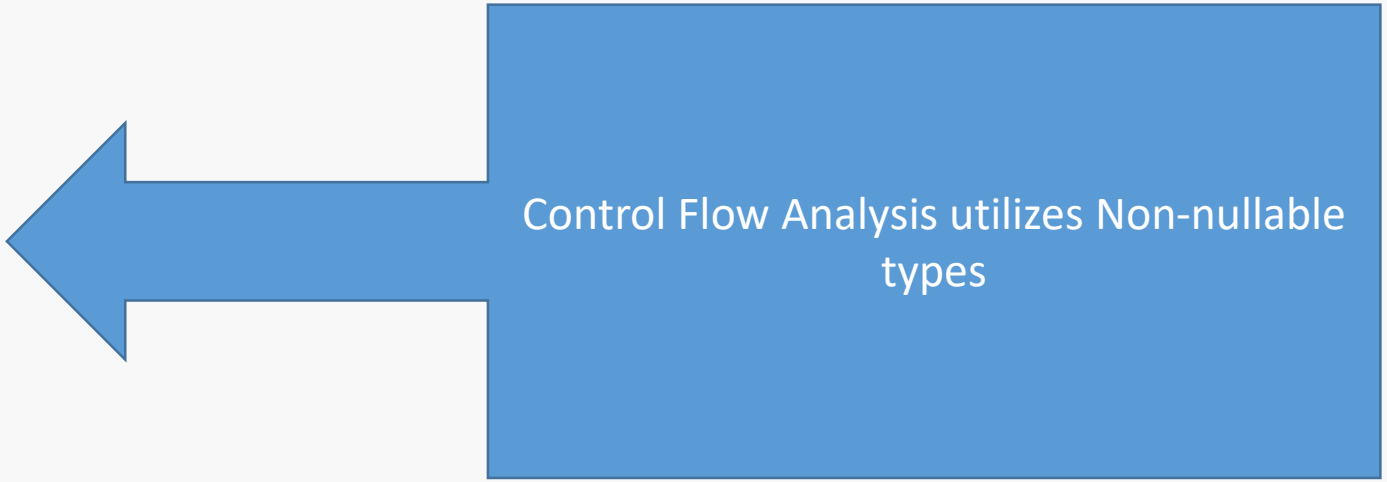


No need to do a falsy check to make sure that its not null or undefined

Demo
Non-Nullable Types & Control Flow Analysis

Packed with Features

- Non-nullable types
- Control flow analysis
- @types .d.ts acquisition



Type Declaration Files (.d.ts)

- Often referred to as type definition files or type libraries
- They are just wrappers for existing JavaScript libraries
- The goal of a type declaration file it to declare types Variables, Functions, etc. that match the intended use of those items

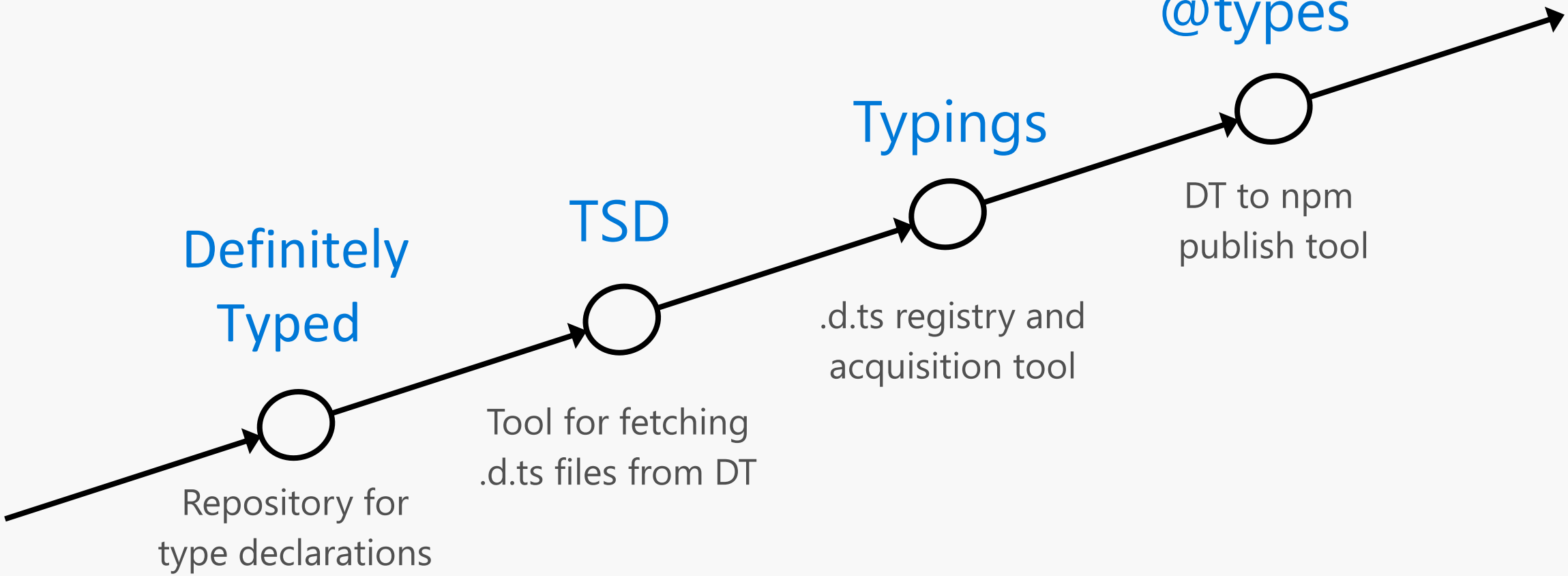
Type Declaration Files (.d.ts)

- This allows the TypeScript compiler to make sure you are using the library correctly. This could include referencing the correct properties on an object or passing the expected number and types of parameters to a function

- These are common runtime problems when working with JavaScript libraries directly, but compiling with declaration files means you can find those problems at compile time

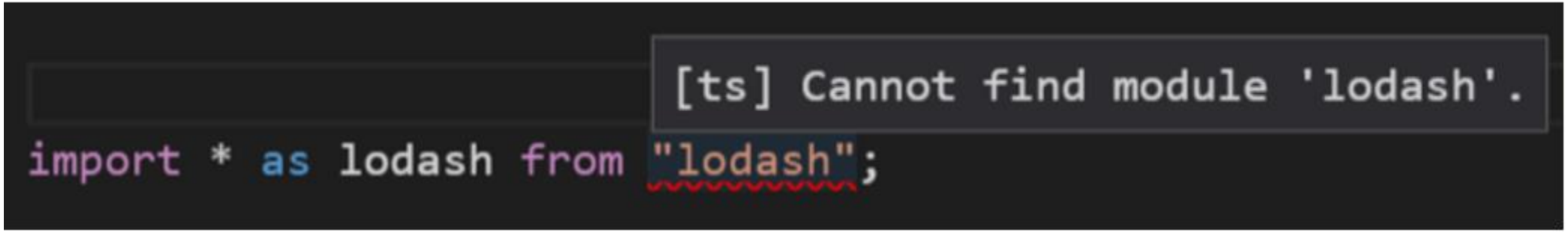
- Declaration files don't replace the JavaScript libraries, but the are just a development time tool to assist the compiler

Evolution of Type Acquisition



Evolution of Type Acquisition

- It gets better



- "But I already have that package installed!" you might say

- The problem is that TypeScript didn't trust the import since it couldn't find any declaration files for lodash. The fix is pretty simple:
  - npm install --save @types/lodash

- Starting with TypeScript 2.1 so long as you have a package installed, you can use it

Demo
Easier Imports

TypeScript 2.1

Async/Await

- Planned to be included as a feature under ES2016/ES2017
- TypeScript has supported the async/await keywords since version 1.7, which came out in November of 2015
- The compiler transformed asynchronous functions to generator functions using yield. This meant that you couldn't target ES3 or ES5 because generators were only introduced in ES2015
- Starting with TypeScript 2.1 downlevel async functions is now supported. This means that you can start using async/await and target ES3/ES5 without using any other tools

References

- Typescript
- Typescript MSD Blog
- Typescript Github Page
- Typescript Road Ahead
- Modules