Advanced Application Development (CSE-214)

week-2 : Angular - Typescript

Dr. Alper ÖZCAN

Akdeniz University

alper.ozcan@gmail.com

What is Angular ?

- Angular is a framework for building modern single-page applications

Official docs
Tutorials

www.angular.io

What is Single Page Application ?

How is Single-Page application different than Traditional application?

What is Single Page Application ?

- Single-Page Applications (**SPAs**) are web apps that load all the necessary **HTML, CSS, and JavaScript** in the **initial page load**, and then **dynamically update their Document Object Model (DOM)** and **retrieve extra data based on user interactions**.

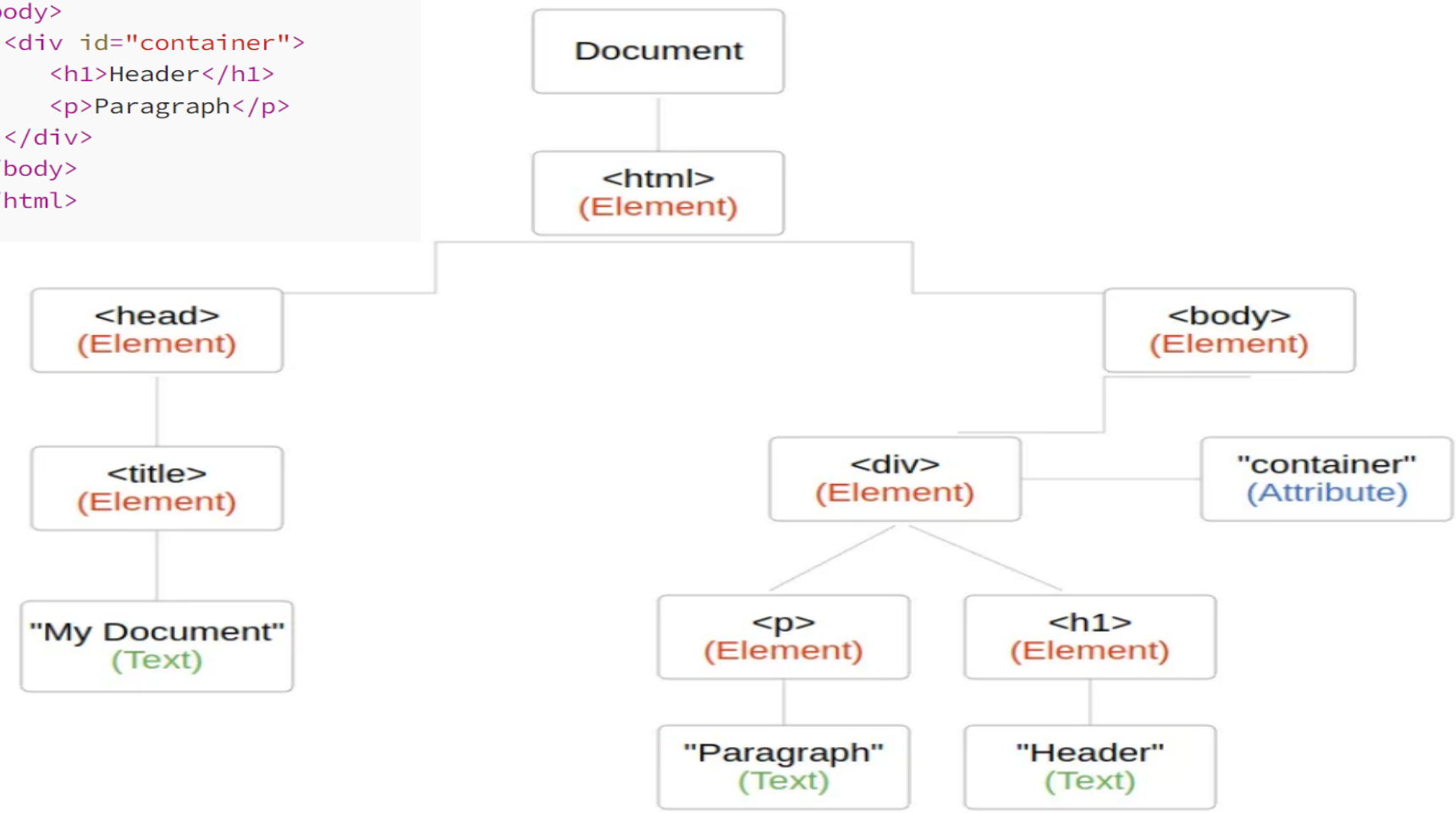
What is Document Object Model (DOM) ?

The DOM tree looks like a **hierarchical structure of nodes**, where **each node** represents an **element, attribute**, or **text** in the document. Here is an example of a **DOM** tree for a simple HTML document:

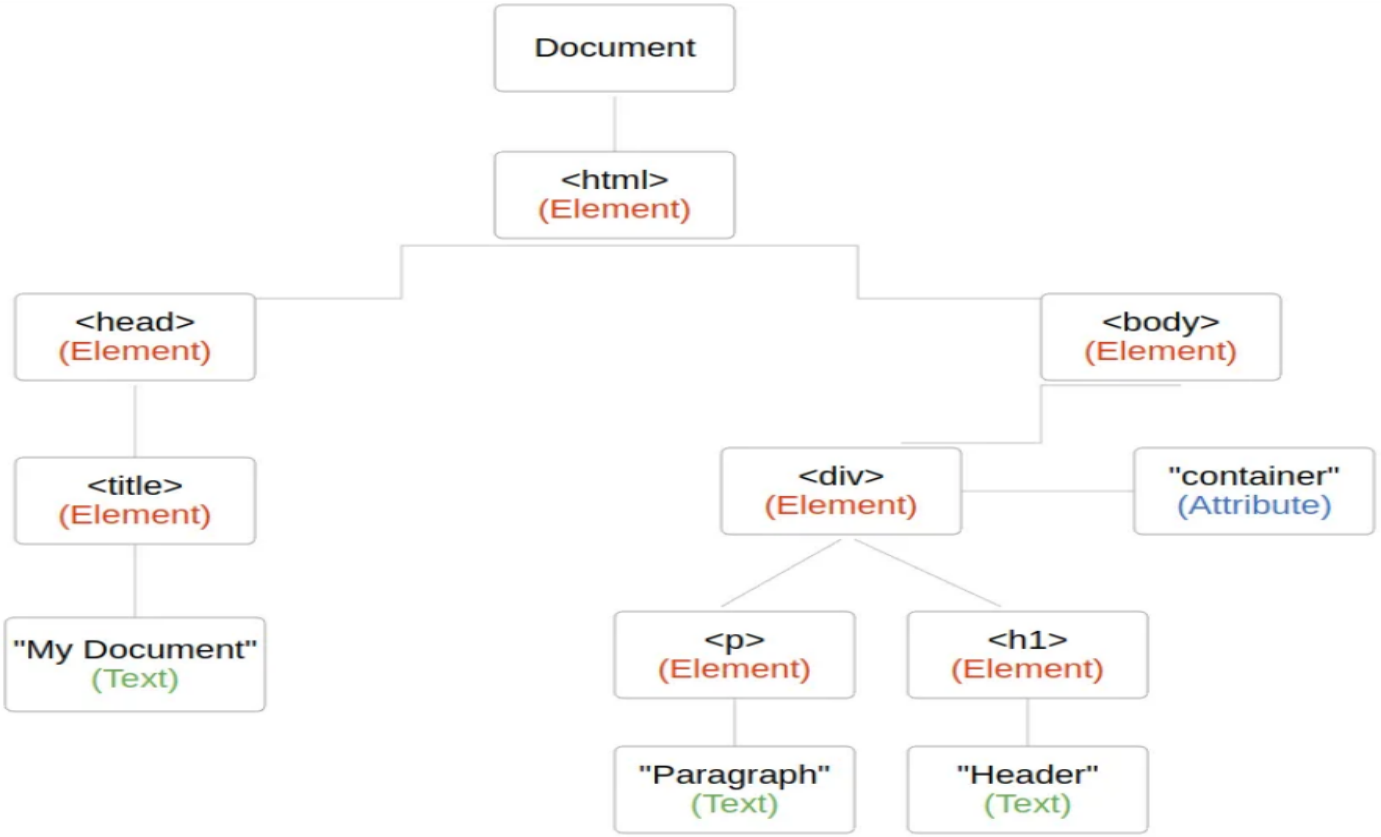
```
<html>
<head>
  <title>My Document</title>
</head>
<body>
  <div id="container">
     <h1>Header</h1>
     <p>Paragraph</p>
  </div>
</body>
</html>
```

What is Document Object Model (DOM) ?

```
<html>
<head>
  <title>My Document</title>
</head>
<body>
  <div id="container">
     <h1>Header</h1>
     <p>Paragraph</p>
  </div>
</body>
</html>
```

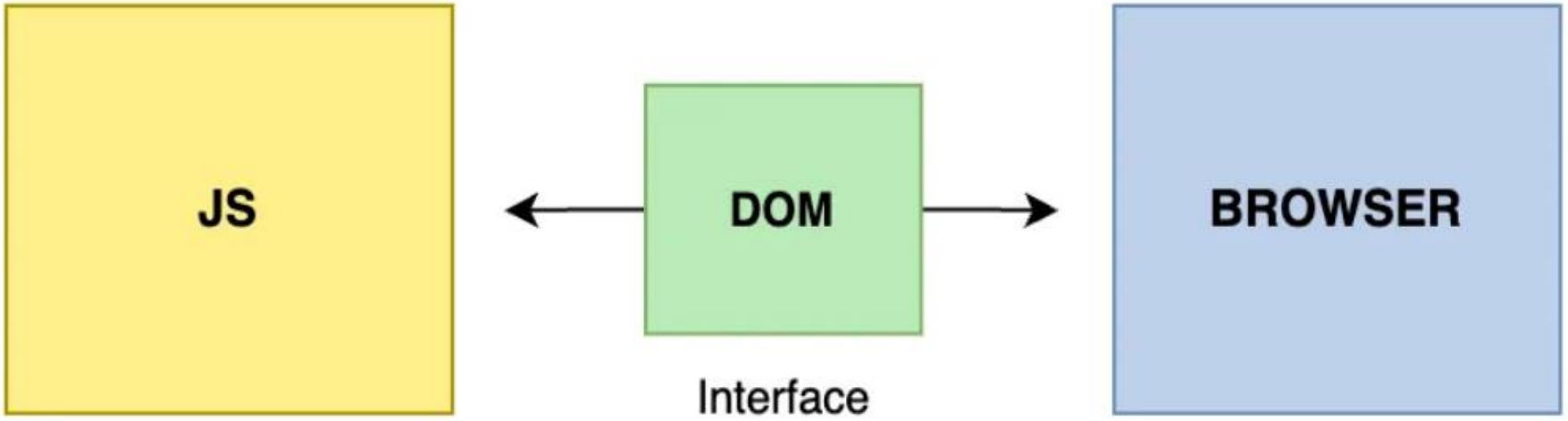


What is Document Object Model (DOM) ?



- **DOM** stands for **Document Object Model**, which is a standard way of **representing** and **manipulating web documents**.

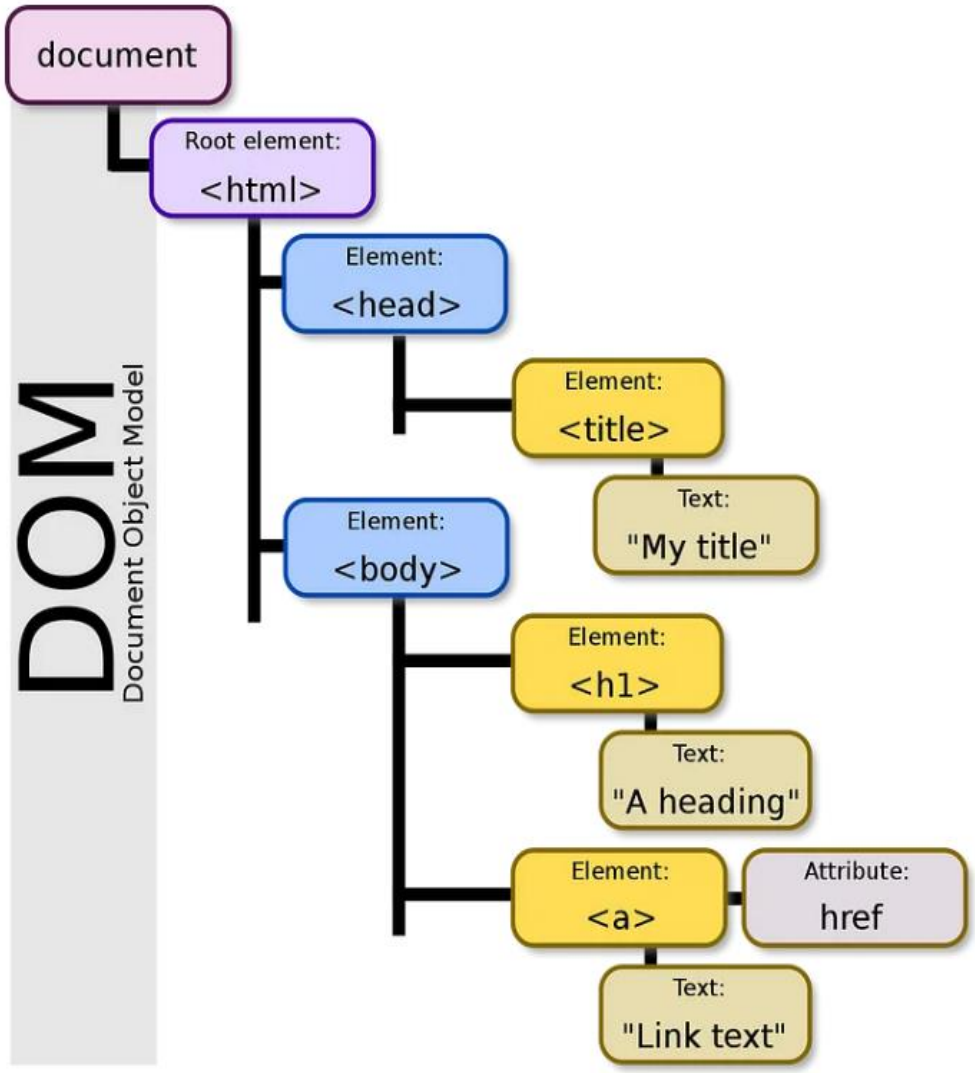
What is Document Object Model (DOM) ?



- The **Document Object Model (DOM)** is an **interface** between all **JavaScript code** and the **browser**, specifically between **HTML documents rendered** in and by the browser.

- **The DOM** represents **the document as a tree of objects**, each with **properties** and **methods** that can be **accessed** and **changed** by **JavaScript**.

What is Document Object Model (DOM) ?

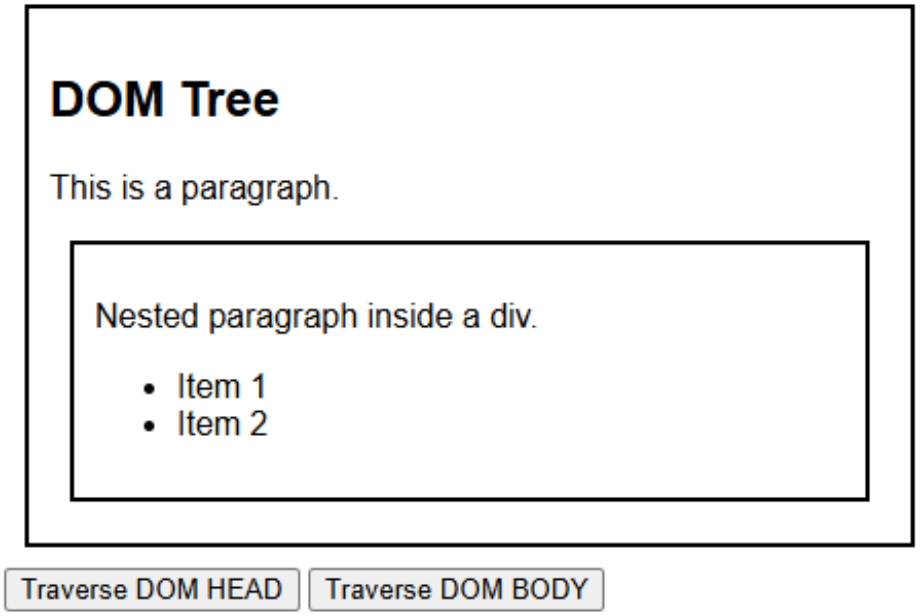


The Document Object Model (DOM) is a programming interface for HTML

It represents the **hierarchical structure of HTML or XML documents as a tree of objects**, where **each object** corresponds to **elements**, **attributes**, and **text** within the HTML.

JavaScript Code for Hierarchical DOM Traversal

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DOM Tree Traversal</title>
    <style>
        body {
            font-family: Arial, sans-serif;
        }
        .container {
            border: 2px solid black;
            padding: 10px;
            margin: 10px;
        }
    </style>
</head>
<body>

    <div id="root" class="container">
        <h2>DOM Tree</h2>
        <p>This is a paragraph.</p>
        <div class="container">
            <p>Nested paragraph inside a div.</p>
            <ul>
                <li>Item 1</li>
                <li>Item 2</li>
            </ul>
        </div>
    </div>

    <button onclick="traverseDOM(document.head, 0)">Traverse DOM HEAD</button>

    <button onclick="traverseDOM(document.body, 0)">Traverse DOM BODY</button>
```

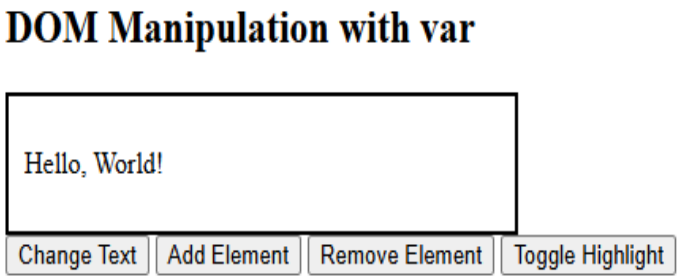


JavaScript Code for Hierarchical DOM Traversal

```
<button onclick="traverseDOM(document.head, 0)">Traverse DOM HEAD</button>

<button onclick="traverseDOM(document.body, 0)">Traverse DOM BODY</button>

<script>
    function traverseDOM(node, depth) {
        // Indent output based on depth level
        // The .repeat(n) method in JavaScript repeats a given string
        //   n times and returns the result as a new string.
        //   "     " (depth * 2 spaces)
        var indent = "  ".repeat(depth * 2);
        console.log(indent + node.nodeName);

        // Loop through children and recursively call traverseDOM
        for (var i = 0; i < node.children.length; i++) {
            traverseDOM(node.children[i], depth + 1);
        }
    }
</script>
```

JavaScript function that **hierarchically traverses** the DOM tree and logs each node's structure: (1) **Starts at document.body** and recursively traverses all child elements. (2) **Indents nodes** based on their depth level in the tree. (3) **Prints the node name** in the console in a hierarchical format.

DOM Manipulation Using **var**




```html
<h2>DOM Manipulation with var</h2>
<div id="container">
    <p id="text">Hello, World!</p>
</div>

<button onclick="changeText()">Change Text</button>
<button onclick="addElement()">Add Element</button>
<button onclick="removeElement()">Remove Element</button>
<button onclick="toggleHighlight()">Toggle Highlight</button>
```

```html
<script>
    function changeText() {
        var textElement = document.getElementById("text");
        textElement.textContent = "Text has been changed!";
    }

    function addElement() {
        var newParagraph = document.createElement("p");
        newParagraph.textContent = "New paragraph added!";
        document.getElementById("container").appendChild(newParagraph);
    }

    function removeElement() {
        var container = document.getElementById("container");
        if (container.lastElementChild.id !== "text") { // Prevent deleting original text
            container.removeChild(container.lastElementChild);
        }
    }

    function toggleHighlight() {
        var textElement = document.getElementById("text");
        textElement.classList.toggle("highlight");
    }
</script>
```

var is function-scoped (NOT block-scoped)

```
function example() {
    if (true) {
        var x = 10;  // Declared inside the function
    }
    console.log(x); // ✅ Works because var is function-scoped (NOT block-scoped)
}
example();
```

✅ Output: `10`

Variables declared with **var** inside a function are accessible throughout the **entire function**, but not outside of it.

var is hoisted but initialized as **undefined**

```
var a;  // Declaration is hoisted to the top
console.log(a); // undefined
a = 5;  // Assignment happens later
console.log(a); // 5
```

JavaScript **hoists (move from lower to higher position)** var declarations, meaning the variable is moved to the top of its function scope, but it is initialized as undefined until assigned a value.

Global Scope Pollution: **var** can leak into the global scope

```
function test() {
    x = 100; // ❌ No "var", so this becomes a global variable!
}
test();
console.log(x); // ✅ Accessible globally (BAD PRACTICE)
```

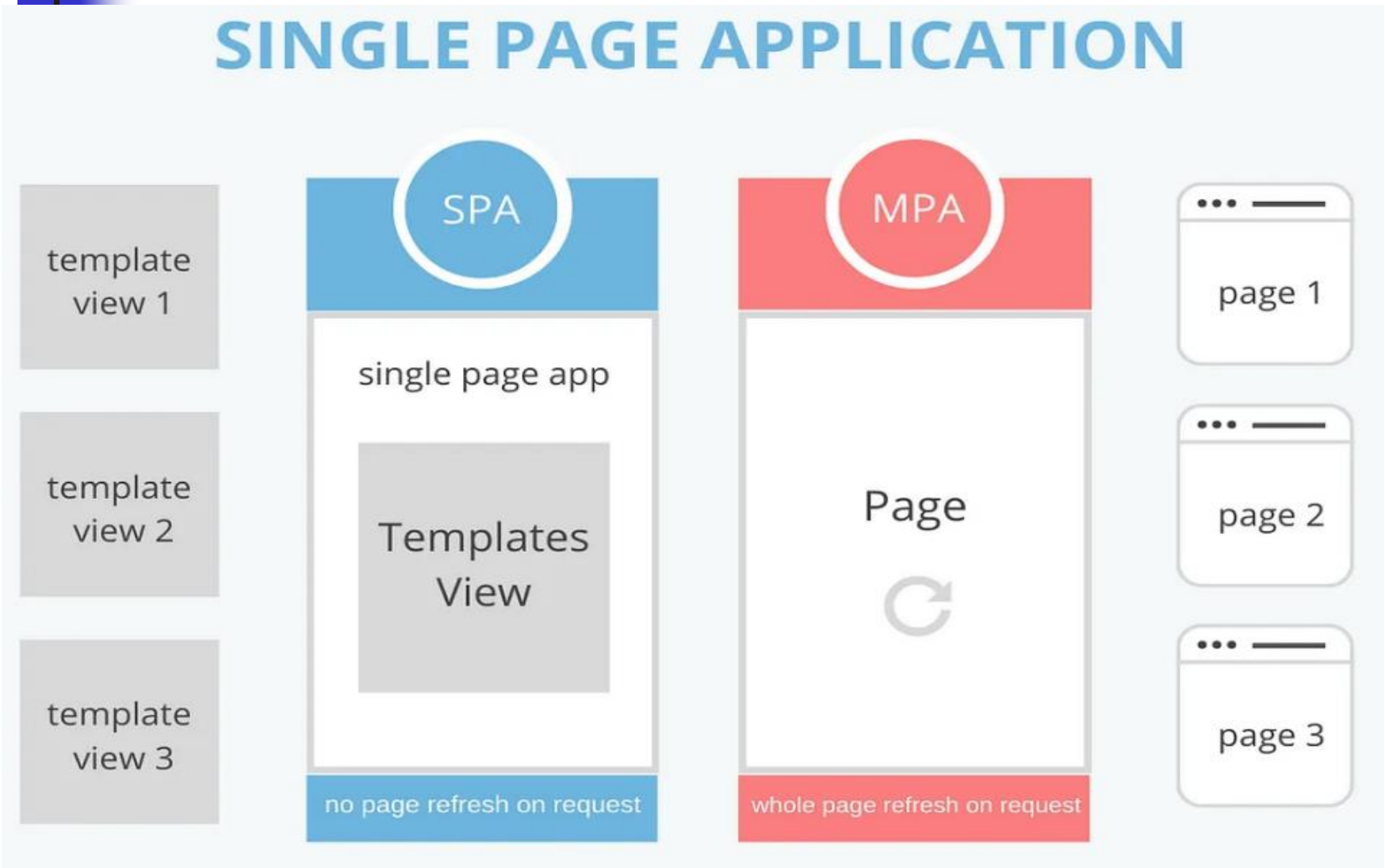
```
function test() {
    var x = 100; // ✅ Now `x` is only inside this function
}
test();
console.log(x); // ❌ ReferenceError: x is not defined
```

If **var** is declared inside a function **without using var inside another function**, it can accidentally become **a global variable**.

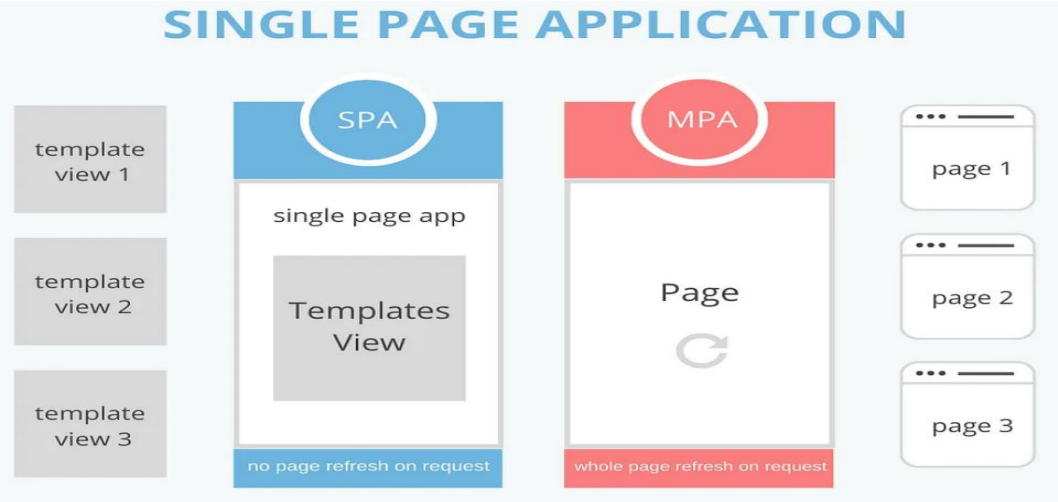
What is Single Page Application ?

- SPAs **differ** from the **traditional approach** of requesting new web documents from the backend servers based on user actions, sometimes called **Server-Side Rendering (SSR)**, which reloads all assets at every user request — although some assets would be cached by the browser.

What is Single Page Application ?



What is Single Page Application ?

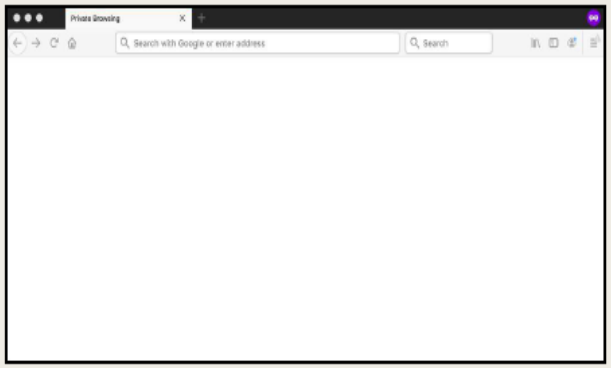


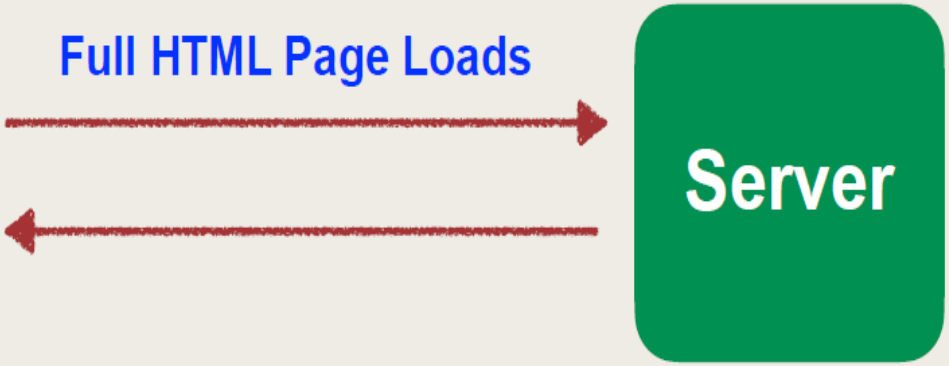
- **Single-page applications (SPA)** frameworks made it easier to build web applications that advanced beyond **JavaScript** and **jQuery**. **Angular** is used to **create entire web applications** in **JavaScript**.

- **Before** the **advent** of **modern JavaScript frameworks**, most websites were implemented as **multi-page applications**. But this makes traditional applications resource-intensive to web servers because **sending entire web pages for every request consumes excessive bandwidth** and **uses CPU time to generate dynamic pages.**

Traditional Application

- Each user action results in a full HTML page load





Traditional Application

- Each user action results in a full HTML page load

- In the **pre-SPA days**, a user would visit a URL in a browser and request one HTML file and all its associated HTML, CSS, and JavaScript files from a web server.
- After some network delay, the user sees the rendered HTML in the browser (client) and starts to interact with it.
- If your **website** is **complex**, **the site browsing experience may appear slow to users**. It will be even **slower** if they have a **poor** or **limited** internet connection.
- To **solve** this problem, many web developers build **their web applications as SPAs**.

Single-Page Application

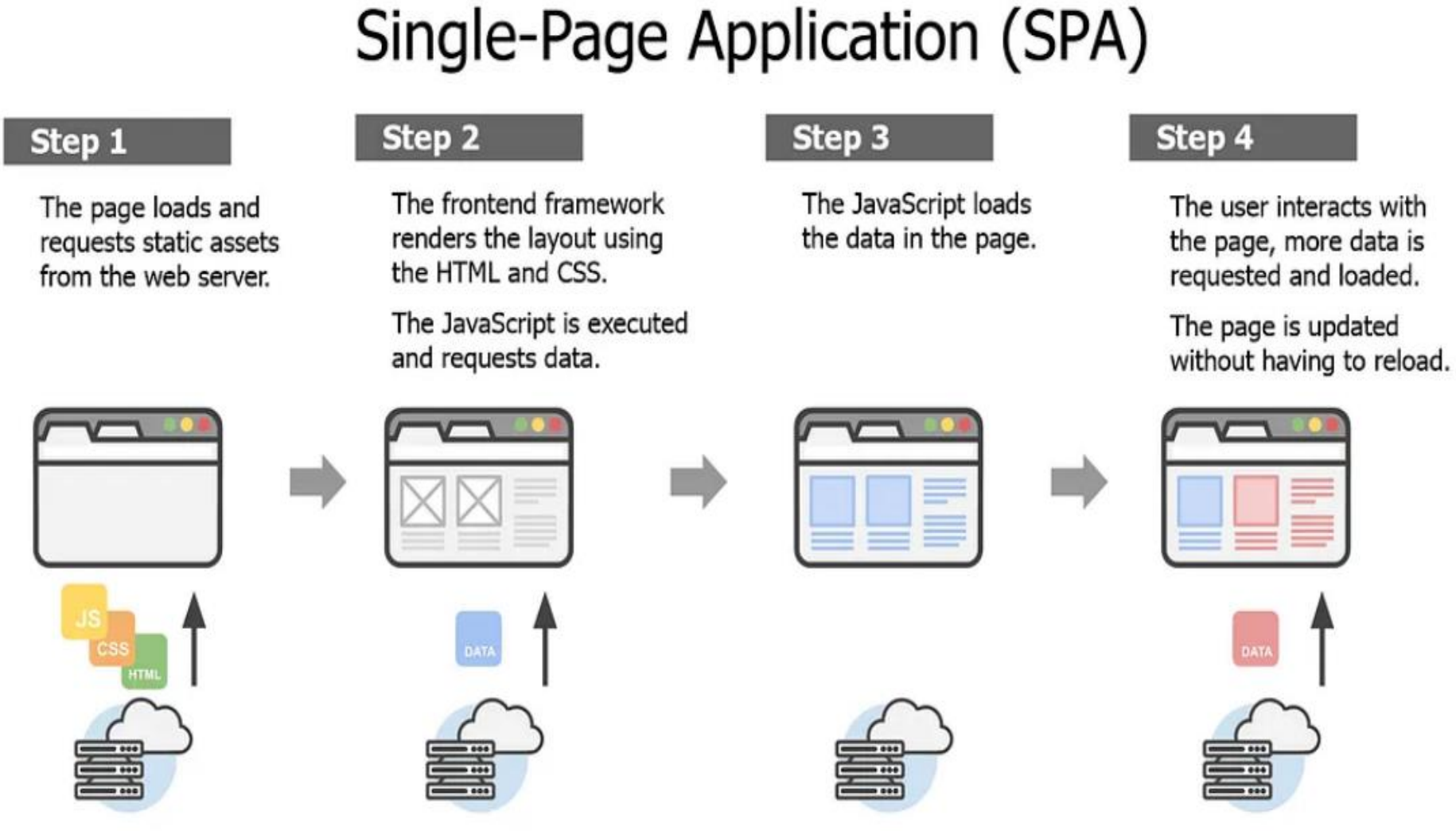
- A web application that is composed of a single page
- Based on user actions, the application page is updated
  - Normally performs a partial update … instead of full page load

Single Page Application

An **SPA allows** the user to **interact** with the website **without** the application **needing** to **download the entire new web pages**.

Instead, it rewrites the current web page as the user interacts with it. The result is a browsing experience that feels faster and more responsive to user input. When the user navigates to the web application in the browser, **the web server returns the necessary resources to run the application**.

Single Page Application



Traditional Application

Server-Side Rendering (SSR)

Step 1

The page loads and requests static assets from the web server.

The data is included in the HTML.

Step 2

The browser has all the information it needs, the page is rendered immediately.

Step 3

The user interacts with the page. This triggers a new request to the server, the page goes blank until it reloads.

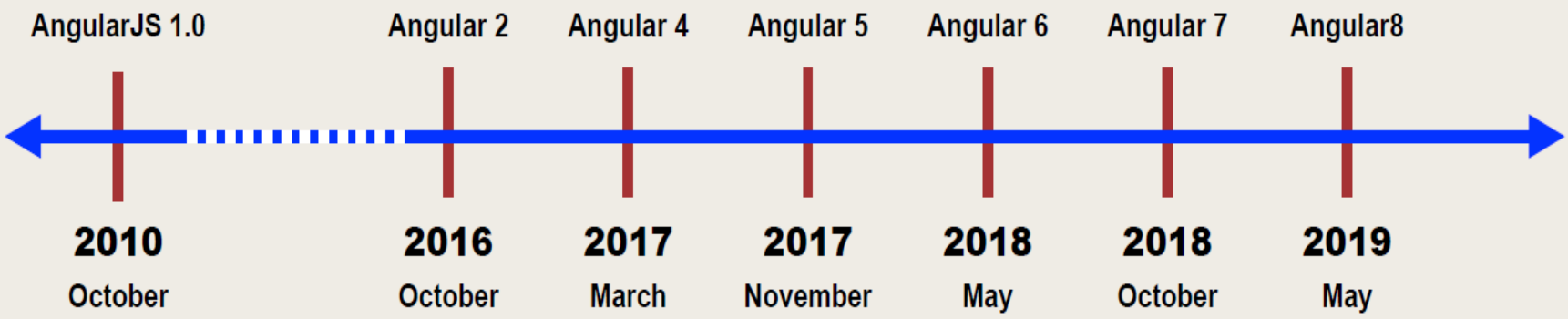
Step 4

The new page loads, showing new content to the user.

Angular Solution

- Angular is a framework for building modern single-page applications

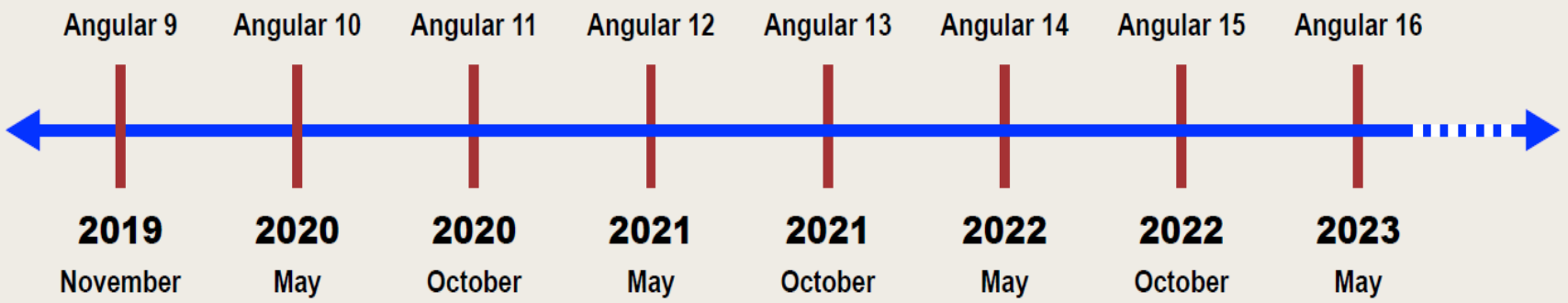
Angular History



Angular 2
was a complete rewrite of
AngularJS

Angular 2 and higher
follow same framework approach
(incremental improvements)

Angular History + Future



Major release every 6 months

Dates are general guidance from Angular team … may shift accordingly

Angular

- **FAQ: Couldn't I just do it myself with JavaScript, jQuery, AJAX etc?**

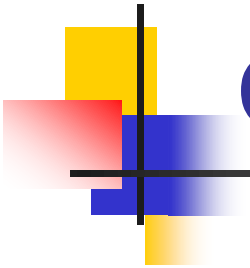
- May work for small basic hobby apps
- For common features such as data binding, you may reinvent the wheel
- Hard to manage and maintain for enterprise real-time applications

- Main reason we have frameworks such **Angular**, **React.js**, **Vue.js**, …

Set up Dev Environment

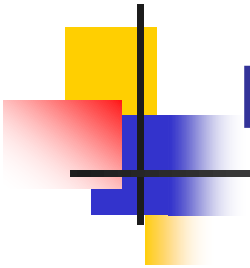
Angular Development Tools

- You can use any text editor or IDE
- Most Angular developers use the **TypeScript** language
  - Superset of JavaScript
  - Strongly-typed language with compile time checking and IDE support
- Command-line tools to compile code and create Angular apps



Command-Line Tools

Tool	Purpose
`node`	For running JavaScript code from command-line
`npm`	Node Package Manager - Download new node packages and features. Similar to Maven
`tsc`	TypeScript Compiler



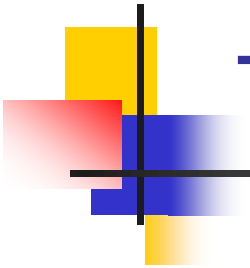
MS Windows - Install Development Tools

- Visual Studio Code

- Node (Node is the runtime environment for executing JavaScript code from the command-line. By using Node, you can create any type of application using JavaScript including server-side / backend applications. We'll use Node to run applications that we develop using TypeScript and Angular)

- Npm (Node Package Manager)

- Tsc (tsc is the TypeScript compiler. We use tsc to compile TypeScript code into JavaScript code. We can install the TypeScript compile using the Node Package Manager(npm))



TypeScript Basics

- Language developed by Microsoft in 2012
  - Free and open-source

www.typescriptlang.org

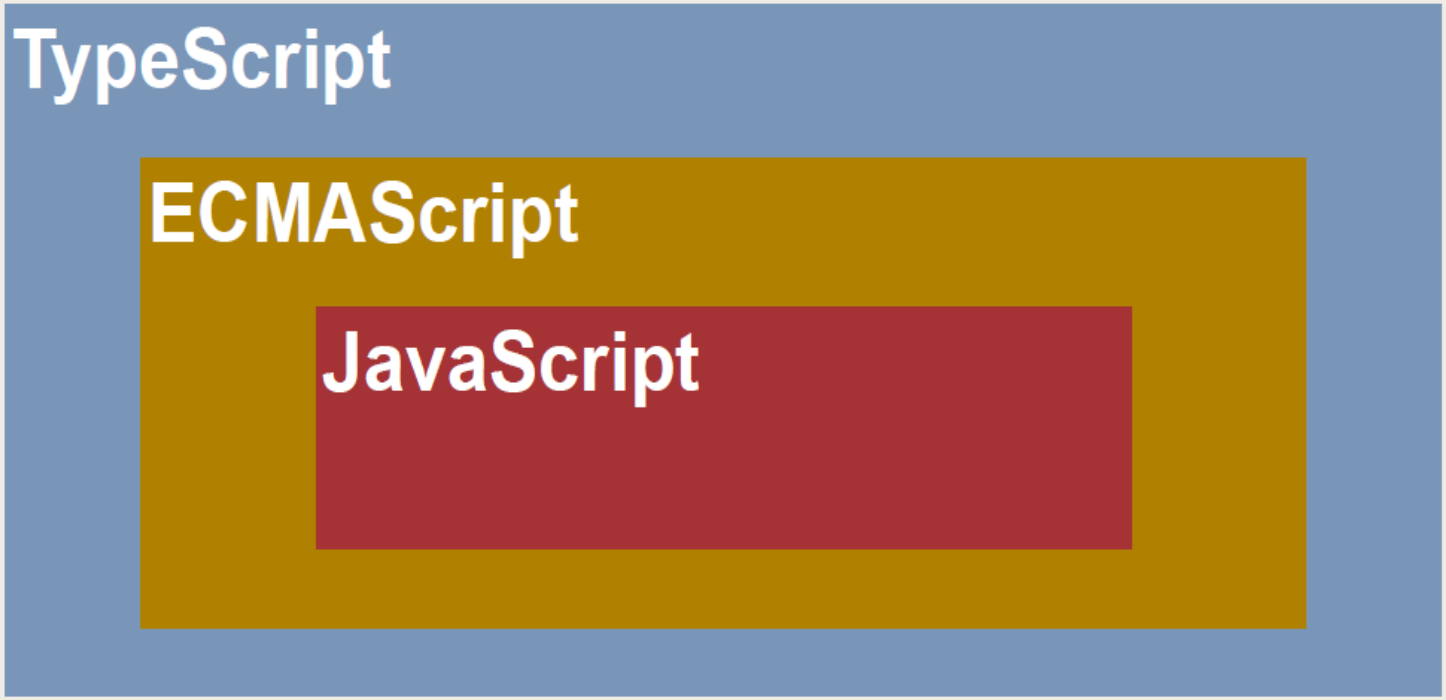
- Provides static typing support to JavaScript
  - Helps with IDE support: code completion and debugging
- Add support for object-oriented programming
  - Classes, objects, inheritance, interfaces, etc …

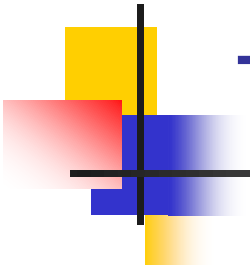
Angular Development

- For Angular development, we can develop using various languages
  - **JavaScript**: extremely popular programming language
  - **ECMAScript:** standardized version of JavaScript (ES6, ES9, …)
  - **TypeScript:** adds optional types to JavaScript
  - *Other languages such as **Dart**, etc …*
- **TypeScript** is the most popular language for Angular development

Angular Development

- TypeScript is a superset of JavaScript and ECMAScript





TypeScript

- **FAQ: Why do most Angular developers use TypeScript?**
- Strongly-typed language with compile time checking and IDE support
- Increased developer productivity and efficiency
- The Angular framework is internally developed using TypeScript
- Docs, online blogs and tutorials use TypeScript for coding examples

Troubleshooting

Permissions Issue with tsc

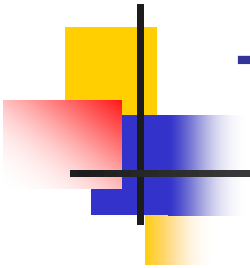
1. If you get the following error when executing tsc command using PowerShell:

```
tsc : File C:\Users\johndoe\AppData\Roaming\npm\tsc.ps1 cannot be loaded because running scripts is disabled on this system

At line:1 char:1
+ tsc sample-datatypes.ts
+ ~~~
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
```

2. You can resolve this issue with the following steps:

   i. Run Visual Studio Code as **Administrator**

   ii. In the Terminal Window of Visual Studio Code, run `Set-ExecutionPolicy RemoteSigned` on PowerShell.



Troubleshooting

Typescript: 'tsc' is not recognized as an internal or external command

1. If you get the following error when executing tsc command:

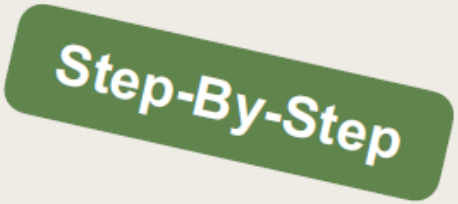
```
Typescript: 'tsc' is not recognized as an internal or external command
```

2. You can resolve this issue with the following:

    i. Add the npm installation folder to your "user variables" AND "environment variables"

TypeScript

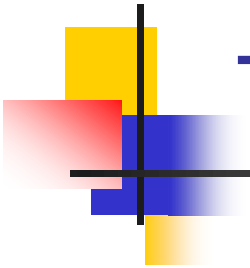
Development Process



1. Create TypeScript code

2. Compile the code

3. Run the code



TypeScript

Step 1: Create the TypeScript code

- TypeScript files have the **.ts** extension

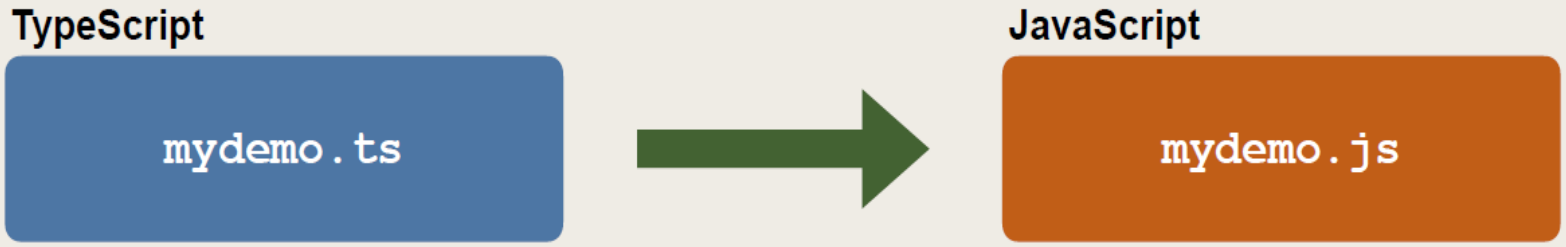
File: mydemo.ts

```
console.log("Hello World!");
```

TypeScript

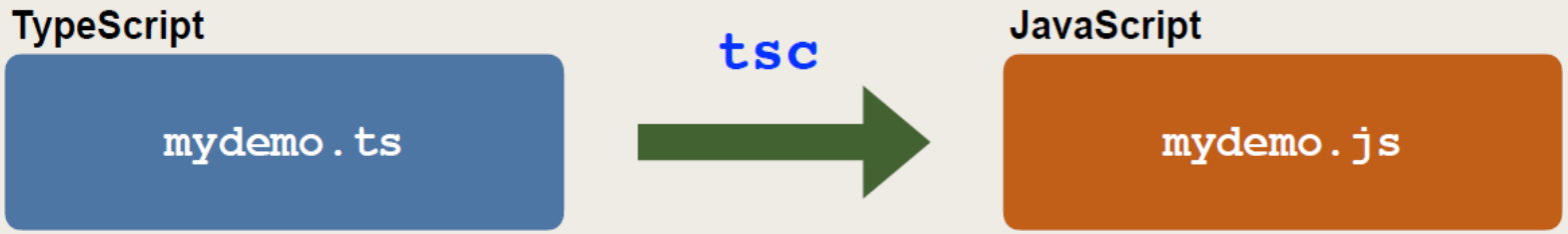
Step 2: Compile the Code

- Web browsers do not understand TypeScript natively
- Have to convert TypeScript code to JavaScript code
- This is known as **"transpiling"**



Step 2: Compile the Code (cont)

- "**Transpiling**" is accomplished with the **`tsc`** command




```
C:\> tsc mydemo.ts
```

Step 3: Run the code

- To run the JavaScript code, we use the **node** command
- Run the generated JavaScript code (`.js` file)

```
C:\> node mydemo.js

Hello World!
```

```
console.log("Hello World!");
```

TypeScript

- The compiler / IDE can find errors earlier at compilation time

```
console.LOGSTUFF("Hello World!");
```

Compile code using: tsc

```
C:\> tsc mydemo.ts

myhello.ts:1:9 - error TS2339: Property 'LOGSTUFF' does not exist on type 'Console'.

 console.LOGSTUFF("Hello World!!");
         ~~~~~~~~

Found 1 error.
```

Compilation error … much better

TypeScript Variables

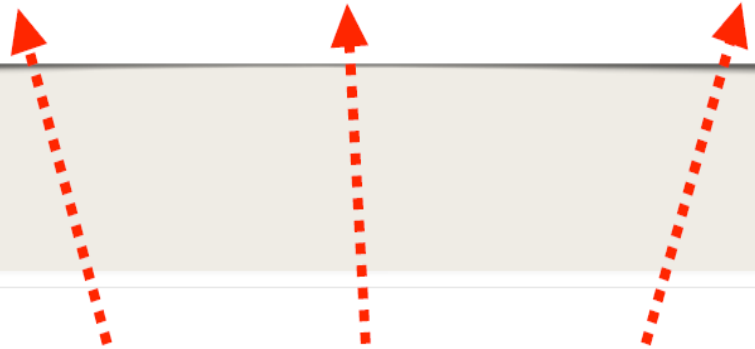
Basic Types

Type	Description
`boolean`	true/false values
`number`	Supports integer and floating point numbers
`string`	Text data. Enclosed in single or double quotes
`any`	Supports "any" datatype assignment
Others ...	See details at www.typescriptlang.org

Define Variables

Syntax

```
let <variableName>: <type> = <initial value>;
```



Example

```
let found: boolean = true;
```

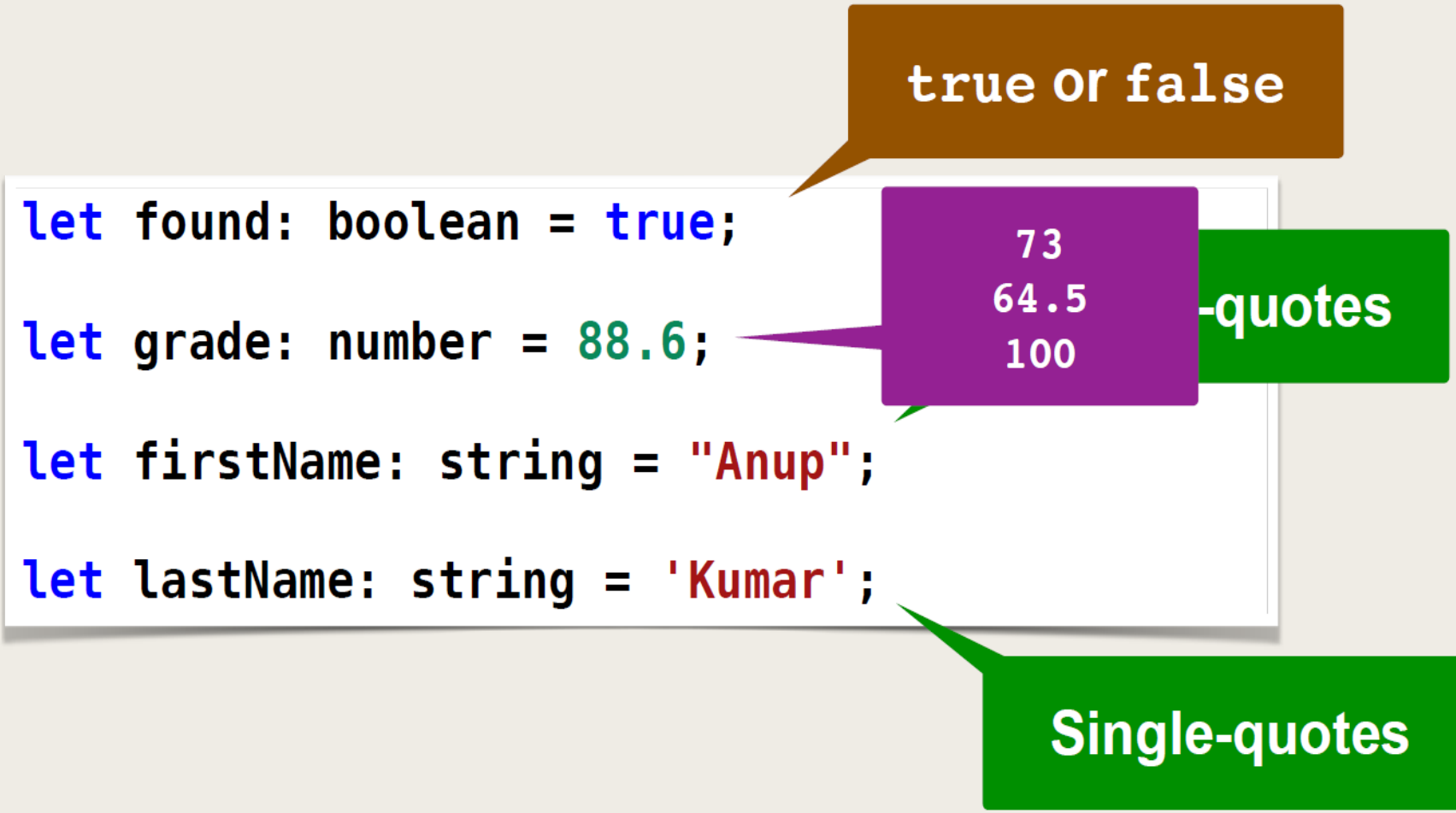
Examples

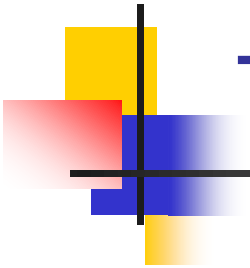
```
let found: boolean = true;

let grade: number = 88.6;

let firstName: string = "Anup";

let lastName: string = 'Kumar';
```

Double-quotes

Single-quotes

Examples


true or false
let found: boolean = true;
let grade: number = 88.6;
73
64.5
100
-quotes
let firstName: string = "Anup";
let lastName: string = 'Kumar';
Single-quotes



TypeScript: "let" keyword

- We are using the new TypeScript **`let`** keyword for variable declarations
  - As opposed to using traditional JavaScript **`var`** keyword
- The JavaScript **`var`** keyword had a number of gotchas and pitfalls
  - Scoping, capturing, shadowing etc
- The new TypeScript **`let`** keyword helps to eliminate those issues

Difference Between **let** and **var** in JavaScript

Feature	`var`	`let`
Scope	Function-scoped ✅	Block-scoped ✅
Hoisting	Yes (hoisted but initialized as `undefined`)	Yes (hoisted but not initialized)
Redeclaration	Allowed	❌ Not allowed
Capturing in Loops	Problematic due to function scope	✅ Safer due to block scope

The **let** and **var** keywords are used to declare variables in JavaScript, but they behave differently in terms of **scoping, hoisting, and redeclaration**

Scope Issue (Function vs. Block Scope)

```
function testVar() {
    if (true) {
        var x = 10; // Declared inside if-block but accessible outside
    }
    console.log(x); // ✅ Outputs: 10
}
testVar();
```

```
function testLet() {
    if (true) {
        let x = 10; // Block-scoped
    }
    console.log(x); // ❌ Error: x is not defined
}
testLet();
```

var is **function-scoped**, meaning it's accessible anywhere inside the function.
let is **block-scoped**, meaning it's only accessible inside the block {} where it was declared.

Hoisting Issue (Undefined Behavior)

```
console.log(a); // ✅ Outputs: undefined (hoisted but uninitialized)
var a = 5;
```

```
console.log(b); // ❌ Error: Cannot access 'b' before initialization
let b = 5;
```

var declarations are **hoisted**, but their values are set to **undefined** until assignment.
let is also hoisted, but accessing it **before initialization** results in an error

Capturing in Loops (var Pitfall)

```
for (var i = 0; i < 3; i++) {
    setTimeout(() => console.log(i), 1000);
}
// ❌ Outputs: 3, 3, 3 (because `var` is function-scoped)
```

```
for (let i = 0; i < 3; i++) {
    setTimeout(() => console.log(i), 1000);
}
// ✅ Outputs: 0, 1, 2 (each iteration gets a new `i`)
```

var does not create a **new variable for each iteration** of a loop.
It captures the last value, causing **unexpected behavior** in **asynchronous code.**
Fix with **let** (Creates a New Variable in Each Iteration)

asynchronous code

```
console.log("Start");

setTimeout(() => {
    console.log("This runs after 2 seconds");
}, 2000);

console.log("End");
```

Output:

```
Start
End
This runs after 2 seconds
```

Asynchronous programming in JavaScript helps execute time-consuming tasks **without blocking** the **main thread**.

setTimeout() is an asynchronous function.

JavaScript does not wait for it and moves to the next line.

TypeScript is Strongly Typed

```
let found: boolean = true;
let grade: number = 88.6;
let firstName: string = "Anup";
let lastName: string = 'Kumar';


// this is okay ... we can assign to different values

found = false;
grade = 99;
firstName = 'Eric';
lastName = 'Noh';
```

This is ok

TypeScript is Strongly Typed

```
let found: boolean = true;
let grade: number = 88.6;
let firstName: string = "Anup";
let lastName: string = 'Kumar';
```

```
// this will generate compilation errors ...

found = 0;
grade = "A";
firstName = false;
lastName = 2099;
```

Type mismatch

Type: any

```
let myData: any = 50.0;

// we can assign different values of any type

myData = false;
myData = 'Eric';
myData = 19;
```

This is ok
But be careful …
you lose type-safety

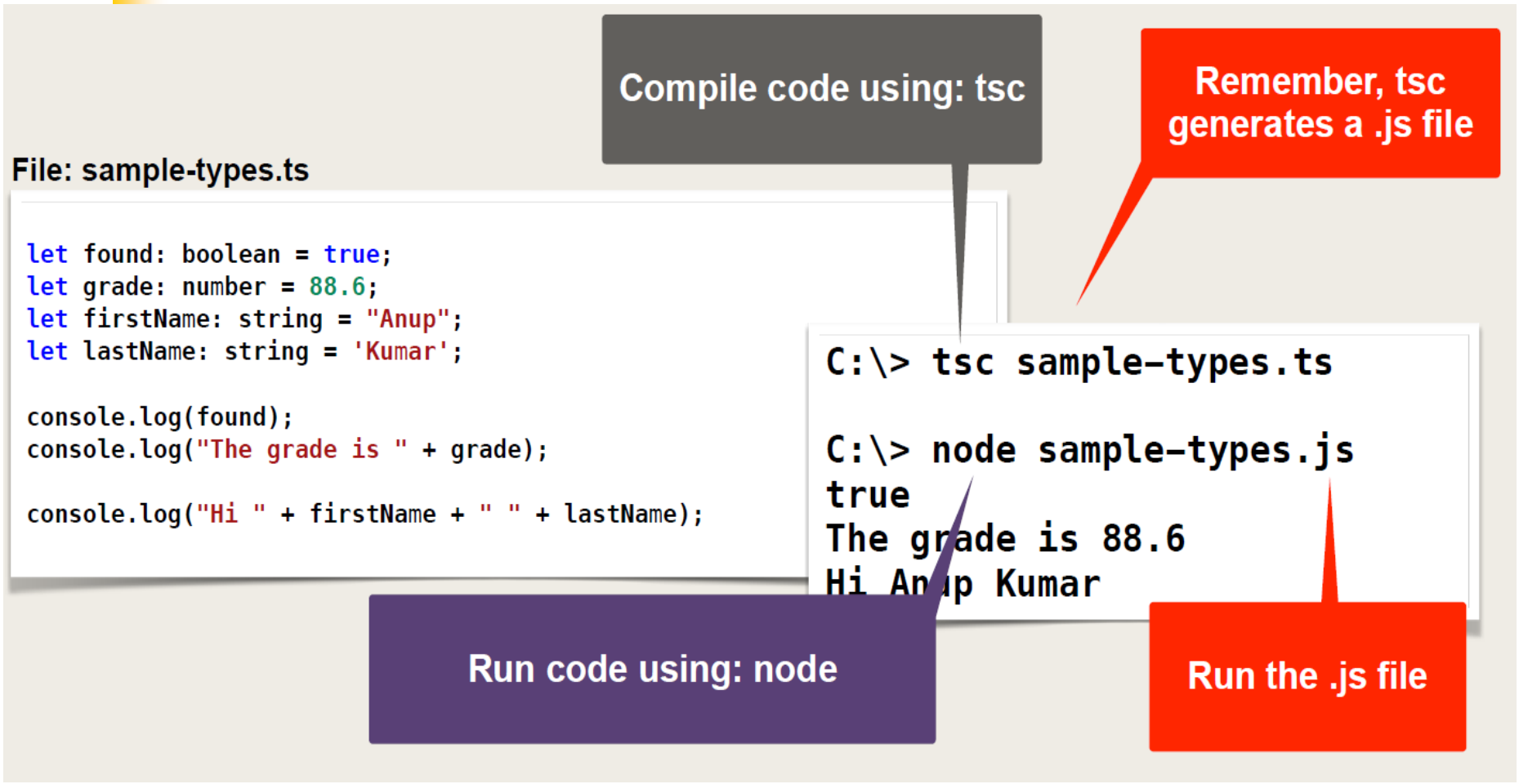
Displaying Output

File: sample-types.ts

```
let found: boolean = true;
let grade: number = 88.6;
let firstName: string = "Anup";
let lastName: string = 'Kumar';

console.log(found);
console.log("The grade is " + grade);
console.log("Hi " + firstName + " " + lastName);
```

```
true
The grade is 88.6
Hi Anup Kumar
```

Run the App


Compile code using: tsc
Remember, tsc generates a .js file
File: sample-types.ts
let found: boolean = true;
let grade: number = 88.6;
let firstName: string = "Anup";
let lastName: string = 'Kumar';
console.log(found);
console.log("The grade is " + grade);
console.log("Hi " + firstName + " " + lastName);
C:\> tsc sample-types.ts
C:\> node sample-types.js
true
The grade is 88.6
Hi Anup Kumar
Run code using: node
Run the .js file

Template Strings

Concatenation could become clunky for long strings

```
let firstName: string = "Anup";
let lastName: string = 'Kumar';

console.log("Hi " + firstName + " " + lastName);
```

Use backticks: `
Reference variables with ${..}

```
console.log(`Hi ${firstName} ${lastName}`);
```

Useful for long strings with a lot of concatenation

For Loops

Compile code using: tsc

File: loops.ts

```
for (let i=0; i < 5; i++) {
    console.log(i);
}
```

```
C:\> tsc loops.ts

C:\> node loops.js
0
1
2
```

Run code using: node

For Loop - Array of numbers

Declare an array

File: reviews.ts

```
let reviews: number[] = [5, 5, 4.5, 1, 3];

for (let i=0; i < reviews.length; i++) {
    console.log(reviews[i]);
}
```

Index into the array

```
C:\> tsc reviews.ts

C:\> node reviews.js
5
5
4.5
1
3
```

Simplified For Loop

File: sports.ts

```
let sportsOne: string[] = ["Golf", "Cricket", "Tennis", "Swimming"];

for (let tempSport of sportsOne) {

    console.log(tempSport);
}
```

Current array element

```
C:\> tsc sports.ts

C:\> node sports.js
Golf
Cricket
Tennis
Swimming
```

Conditionals

File: sports.ts

```
let sportsOne: string[] = ["Golf", "Cricket", "Tenni

for (let tempSport of sportsOne) {

    if (tempSport == "Cricket") {
        console.log(tempSport + " << My Favorite!");
    }
    else {
        console.log(tempSport);
    }
}
```

Conditional

```
C:\> tsc sports.ts

C:\> node sports.js
Golf
Cricket << My Favorite!
Tennis
Swimming
```

Growable Arrays

Arrays in TypeScript are always growable / dynamic

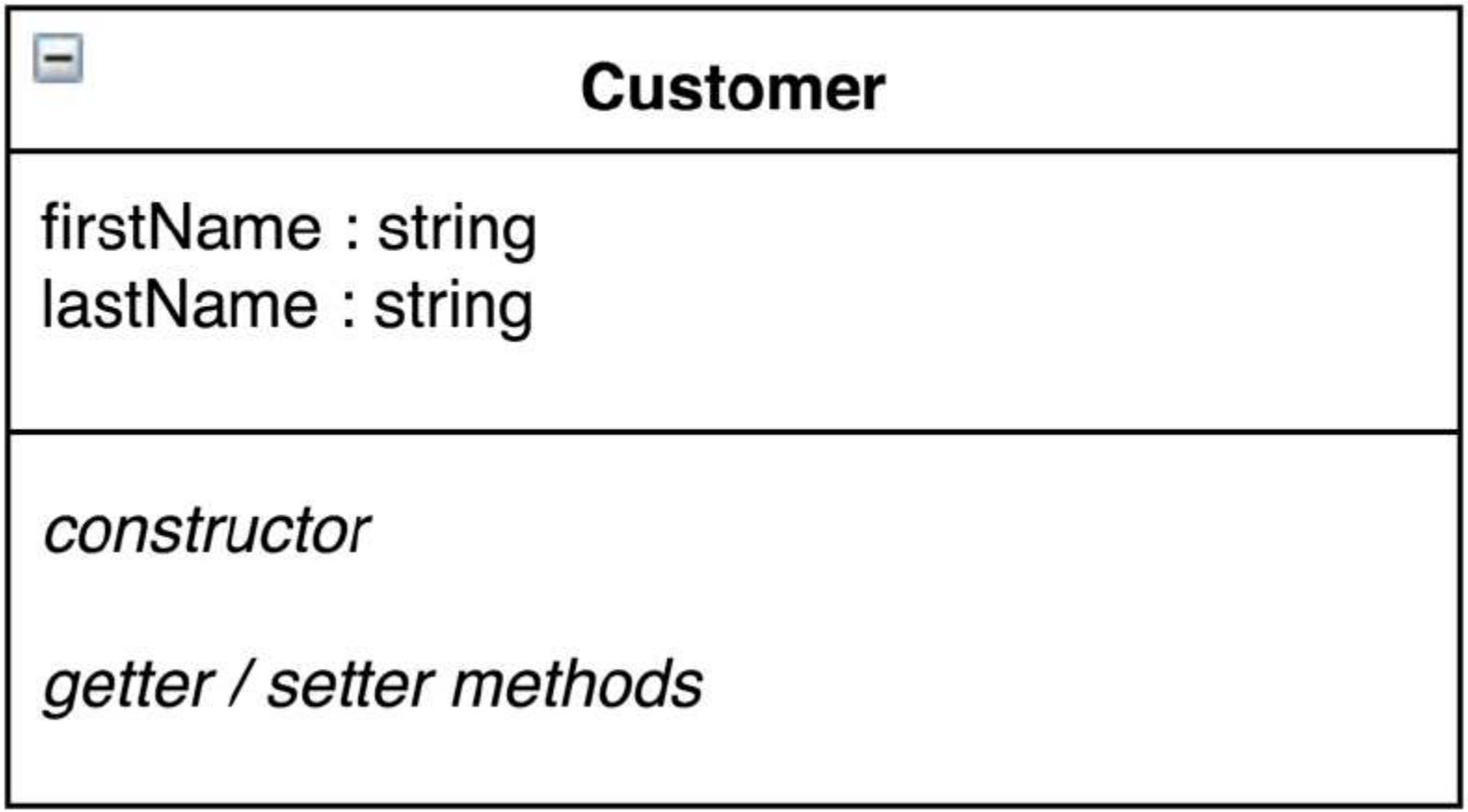
File: growable-arrays.ts

```
let sportsTwo: string[] = ["Golf", "Cricket", "Tennis"];

sportsTwo.push("Baseball");
sportsTwo.push("Futbol");

for (let tempSport of sportsTwo) {
    console.log(tempSport);
}
```

Add elements

```
C:\> tsc growable-arrays.ts

C:\> node growable-arrays.js
Golf
Cricket
Tennis
Baseball
Futbol
```

Creating Classes



Basic Class Structure

File: Customer.ts

```
class Customer {

    // properties

    // constructors

    // getter / setter methods

}
```

Can use any file name: *.ts

mydemo.ts

Properties

File: Customer.ts

```
class Customer {

    // properties
    firstName: string;
    lastName: string;


}
```

Properties are public by default

We'll cover access modifiers shortly

Construct an Instance

File: Customer.ts

```
class Customer {

    // properties
    firstName: string;
    lastName: string;

}

// now let's use it
let myCustomer = new Customer();

myCustomer.firstName = "Martin";
myCustomer.lastName = "Dixon";

console.log(myCustomer.firstName);
console.log(myCustomer.lastName);
```

Construct an instance using the "new" keyword

```
C:\> tsc Customer.ts

C:\> node Customer.js
Martin
Dixon
```

Create a Constructor

File: Customer.ts

```
class Customer {

    firstName: string;
    lastName: string;

    constructor(theFirst: string, theLast: string) {
        this.firstName = theFirst;
        this.lastName = theLast;
    }
}
```

Use the "constructor" keyword

Must use "this" to refer to properties defined in this class

Construct an Instance

File: Customer.ts

```
class Customer {

    firstName: string;
    lastName: string;

    constructor(theFirst: string, theLast: string) {
        this.firstName = theFirst;
        this.lastName = theLast;
    }
}

// now let's use it
let myCustomer = new Customer("Martin", "Dixon");

console.log(myCustomer.firstName);
console.log(myCustomer.lastName);
```

Construct an instance using our new constructor

```
C:\> tsc Customer.ts

C:\> node Customer.js
Martin
Dixon
```

Access Modifiers

Modifier	Definition
`public`	Property is accessible to all classes (default modifier)
`protected`	Property is only accessible in current class and subclasses
`private`	Property is only accessible in current class

Mark the properties as "private"

File: Customer.ts

```
class Customer {

    private firstName: string;
    private lastName: string;

    constructor(theFirst: string, theLast: string) {
        this.firstName = theFirst;
        this.lastName = theLast;
    }
}

// now let'
let myCusto

myCustomer.firstName = "Susan";
myCustomer.lastName = "Public";

console.log(myCustomer.firstName);
console.log(myCustomer.lastName);
```

```
(property) Customer.firstName: string
Property 'firstName' is private and only
accessible within class 'Customer'. ts(2341)
Peek Problem  No quick fixes available
```

Compilation error

Compiling the Code

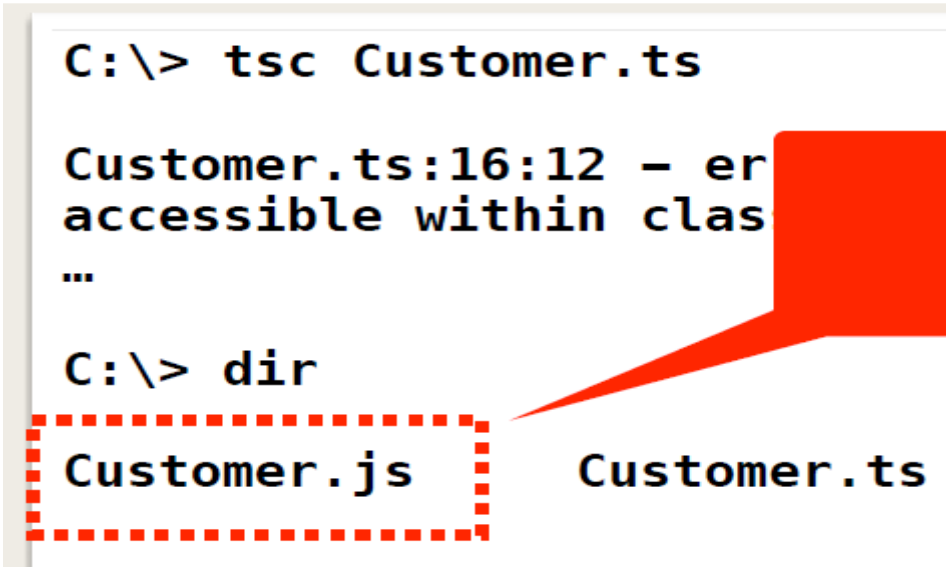
```
C:\> tsc Customer.ts

Customer.ts:16:12 - error TS2341: Property 'firstName' is private and only
accessible within class 'Customer'.

16 myCustomer.firstName = "Susan";
…
…
…
```

```
myCustomer.firstName = "Susan";
myCustomer.lastName = "Public";
```

```
class Customer {

    private firstName: string;
    private lastName: string;
```

Compiling the Code

- Even though there are compilation errors …
- The TypeScript compiler will STILL generates a .js file!



Compiling the Code

Remove previous .js file

```
C:\> del Customer.js

C:\> tsc --noEmitOnError Customer.ts

Customer.ts:16:12 - error TS2341: Property 'firstName' is private and only
accessible within class 'Customer'.
…

C:\> dir

Customer.ts
```

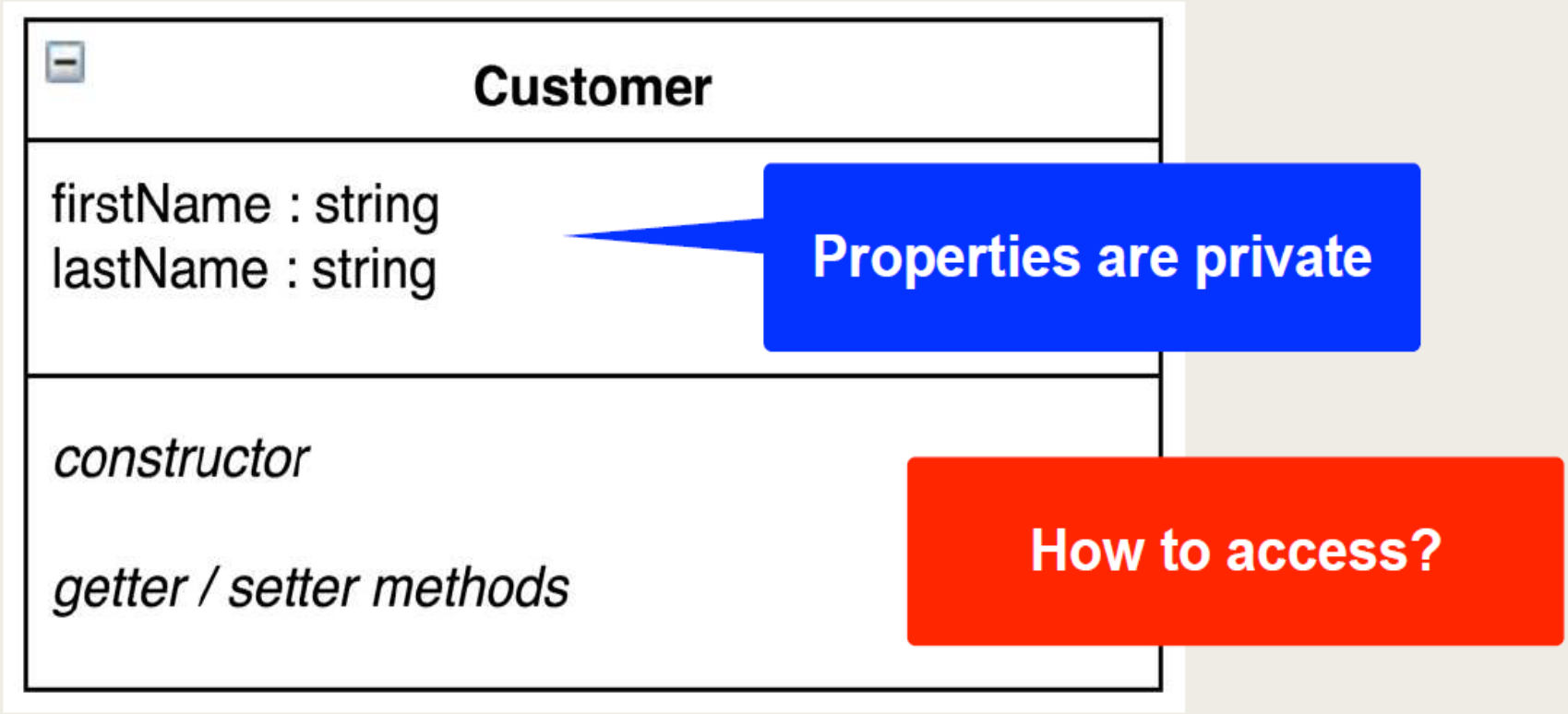
Compiling the Code

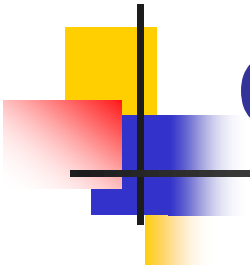
Do not generate .js file
if there is a compilation error

```
C:\> del Customer.js

C:\> tsc --noEmitOnError Customer.ts

Customer.ts:16:12 - error TS2341: Property 'firstName' is private and only
accessible within class 'Customer'.
…

C:\> dir

Customer.ts
```

The .js file was NOT generated!

Class


Customer
firstName : string
lastName : string
Properties are private
constructor
getter / setter methods
How to access?



Getter / Setter Methods

- Since our properties are private, we need a way to access them
- We can create traditional methods as in other OO languages:
  - Define getter/setter methods

Getter / Setter Methods

File: Customer.ts

```
class Custome

    private firstNa e: string;
    private lastNam : string;
    … …

    public getFirstName(): string {
        return this.firstName;
    }
```

Method name

Return type

Getter / Setter Methods

File: Customer.ts

```
class Custome

    private firstNa e: string;
    private lastNam : string;

    public get irstName(): string {
        return this.firstName;
    }

    public setFirstName(theFirst: string): void {
        this.firstName = theFirst;
    }
}
```

Method name

Method name

Return type

Param type

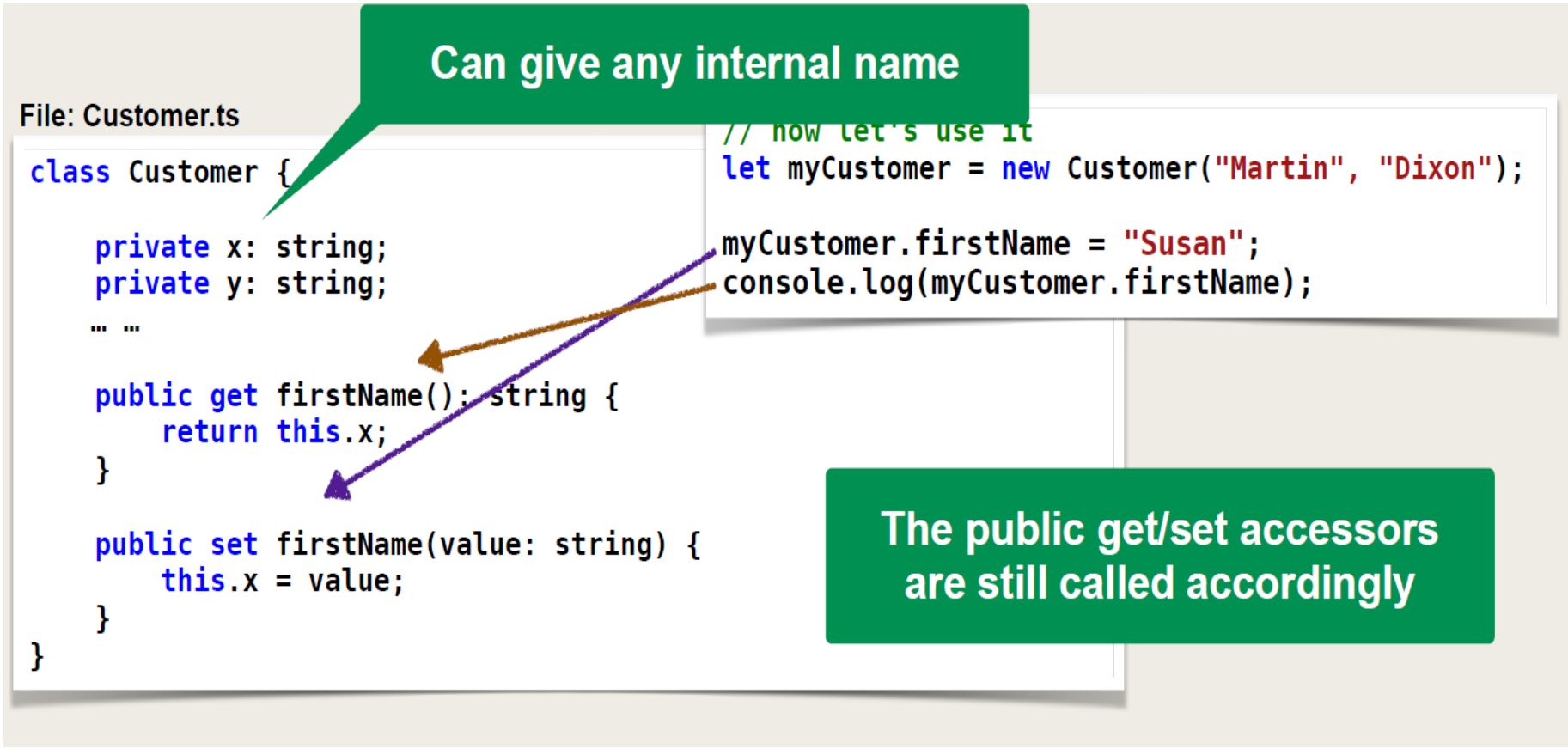
Return type

```
// now let's use it
let myCustomer = new Customer("Martin", "Dixon");

myCustomer.setFirstName("Greg");
console.log(myCustomer.getFirstName());
```

TypeScript: Accessors - Get / Set

- TypeScript also offers an alternate syntax
- Define special: get / set methods
- Known as **Accessors**

TypeScript: Accessors - Get / Set



TypeScript: Accessors - Get / Set

Removed “public” on the accessors.
If no access modifier given, “public” by default

File: Cust

```
class Customer {

    private _firstName: string;
    private _lastName: string;
    … …

    get firstName(): string {
        return this._firstName;
    }

    set firstName(value: string) {
        this._firstName = value;
    }
}
```

Renamed to the original names

Compiler flag

- The get/set accessors feature is only supported in ES5 and higher
- You have to set a compiler flag in order to compile the code

```
C:\> tsc --target ES5 --noEmitOnError Customer.ts
```

Compiler flag

Problem with too many compiler flag

- You may have noticed, that we have a lot of compiler flags
  - Too much stuff to remember … easy to forget
- Wouldn't it be great to set this up in a config file?
- TypeScript has a solution: **tsconfig.json** file

tsconfig.json

- **tsconfig.json** file defines compiler options and project settings
- Place this file in the root of your project directory

File: tsconfig.json

```
{
  "compilerOptions": {
    "noEmitOnError": true,
    "target": "es5"
  }
}
```

tsconfig.json

- You can also generate a template for this file

```
C:\> tsc --init
```

Generates a default tsconfig.json file

- Then edit the **tsconfig.json** accordingly for your project requirements

Compiling your Project

- Once your project has a **tsconfig.json** file, then you can compile with

```
C:\> tsc
```

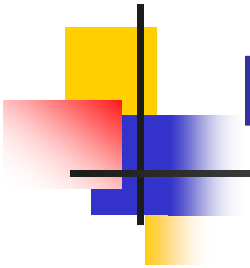
No need to give names of TypeScript files.
By default, will compile all *.ts files

TypeScript: Accessors - Get / Set

File: Customer.ts

```
class Customer {

    private _firstName: string;
    private _lastName: string;
    … …

    get firstName(): string {
        return this._firstName;
    }

    set firstName(value: string) {
        this._firstName = value;
    }
}
```

```
// now let's use it
let myCustomer = new Customer("Martin", "Dixon");

myCustomer.firstName = "Susan";
console.log(myCustomer.firstName);
```



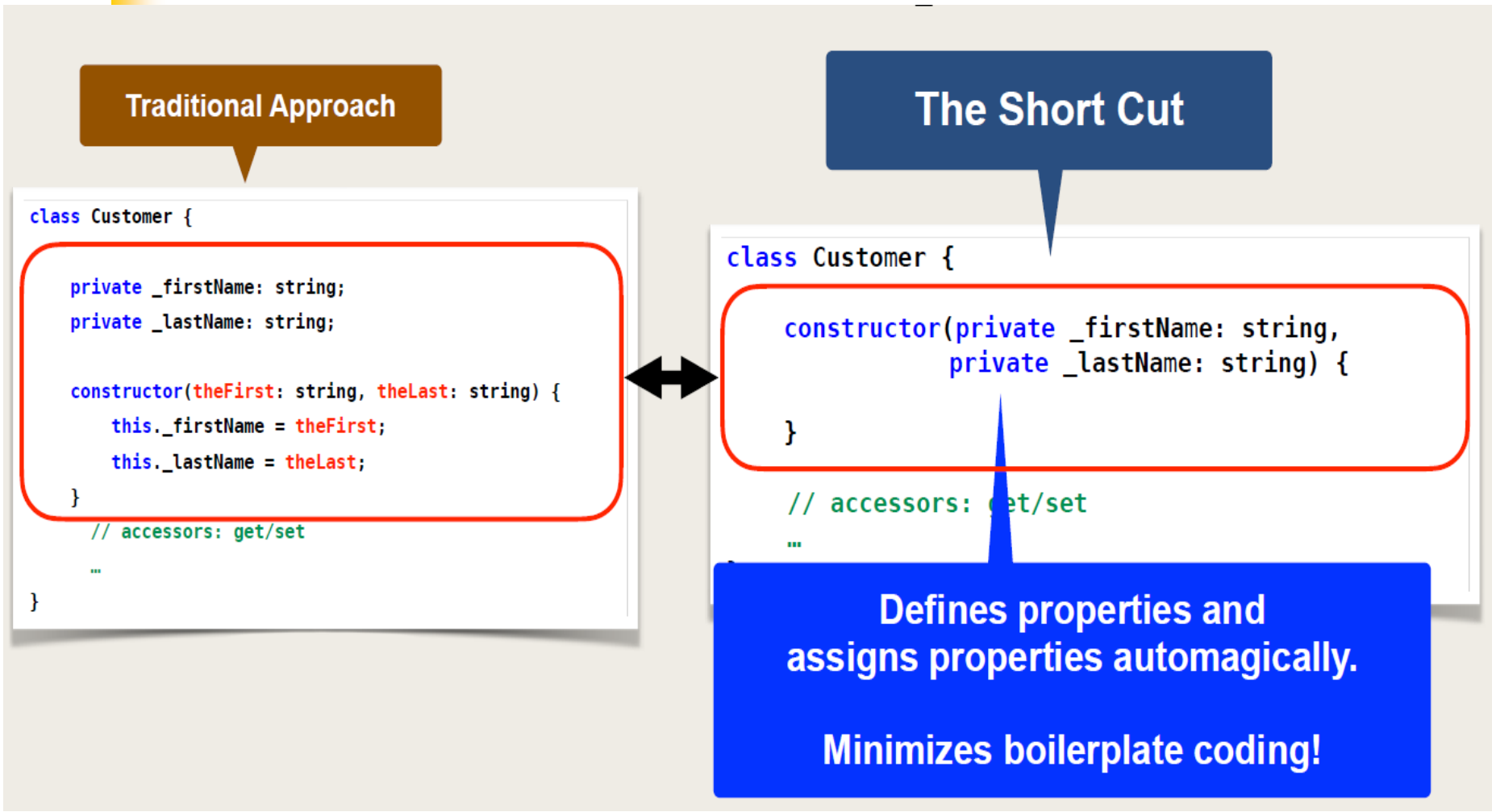
Parameter Properties

- TypeScript offers a short-cut syntax for creating constructors
- Helps to minimize the boilerplate code for constructors

Constructor - Traditional Approach

```
class Customer {

    private _firstName: string;
    private _lastName: string;

    constructor(theFirst: string, theLast: string) {
        this._firstName = theFirst;
        this._lastName = theLast;
    }

    // accessors: get/set
    …
}
```

Constructor - Parameter Properties



Parameter Properties - In Action

File: Customer.ts

```
class Customer {

    constructor(private _firstName: string,
                private _lastName: string) {

    }

    // accessors: get/set
    …
}
```

Defines properties and assigns properties automagically.

Minimizes boilerplate coding!

Parameter Properties - In Action

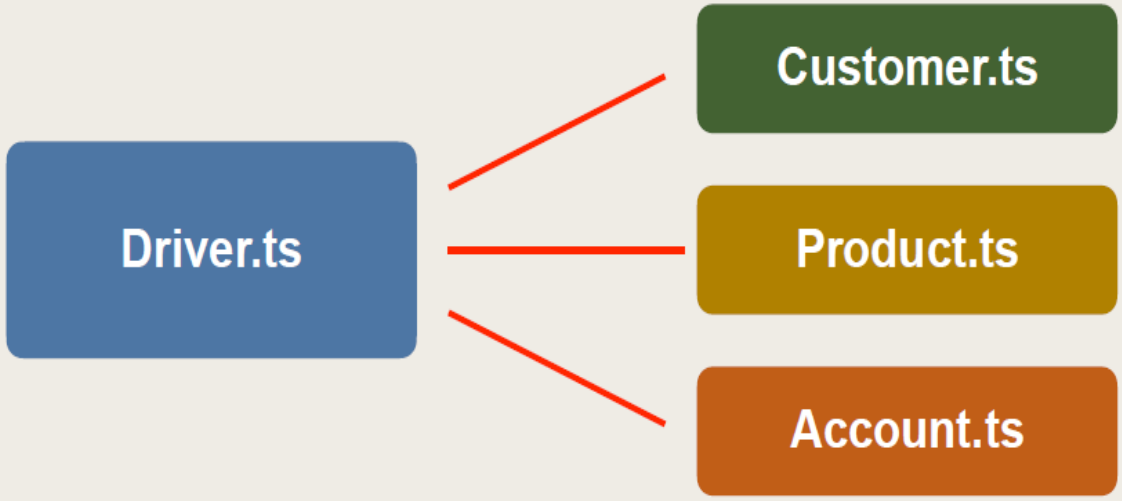
File: Customer.ts

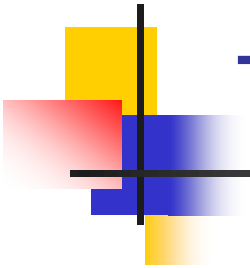
```
class Customer {

    constructor(private _firstName: string,
                private _lastName: string) {

    }

    // accessors: get/set
    ...
}
```

```
// now let's use it
let myCustomer = new Customer("Martin", "Dixon");

myCustomer.firstName = "Susan";
console.log(myCustomer.firstName);
```

Code Organization

- Currently all of our code is in a single file
- For real-time projects, we would like to place code in separate files

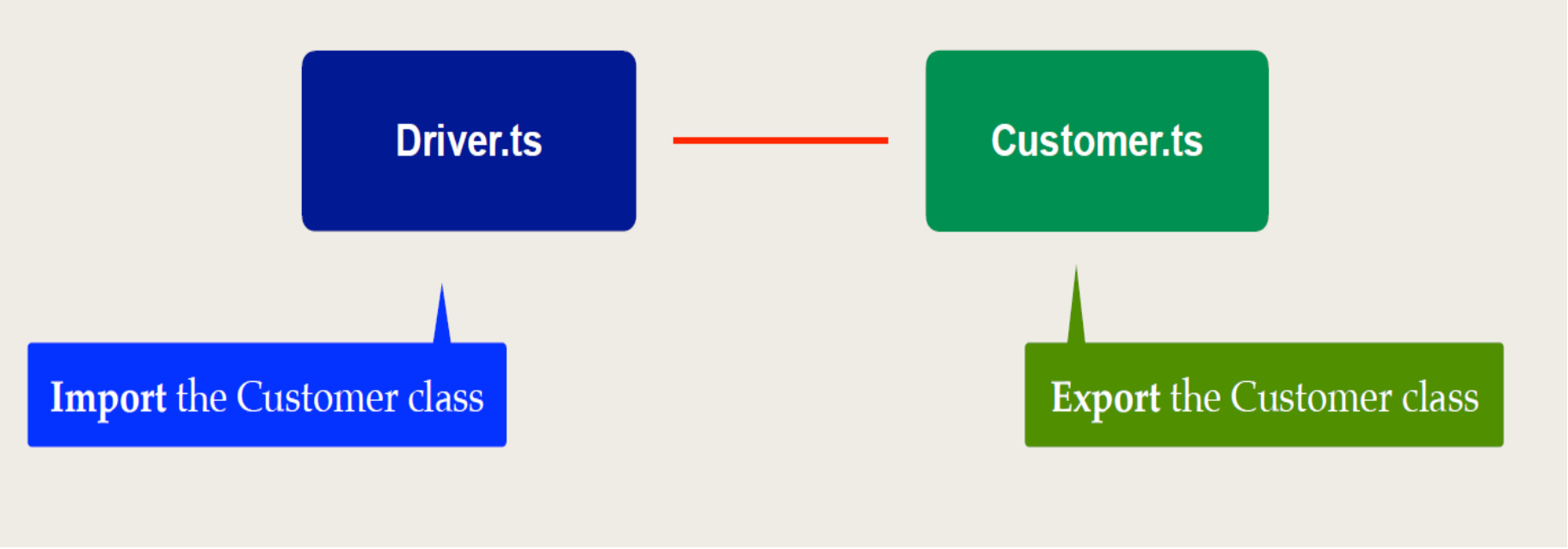




TypeScript Modules

- TypeScript supports the concept of modules
- A module can **export** classes, functions, variables etc
- Another file can **import** classes, functions, variables etc from a module

TypeScript Modules



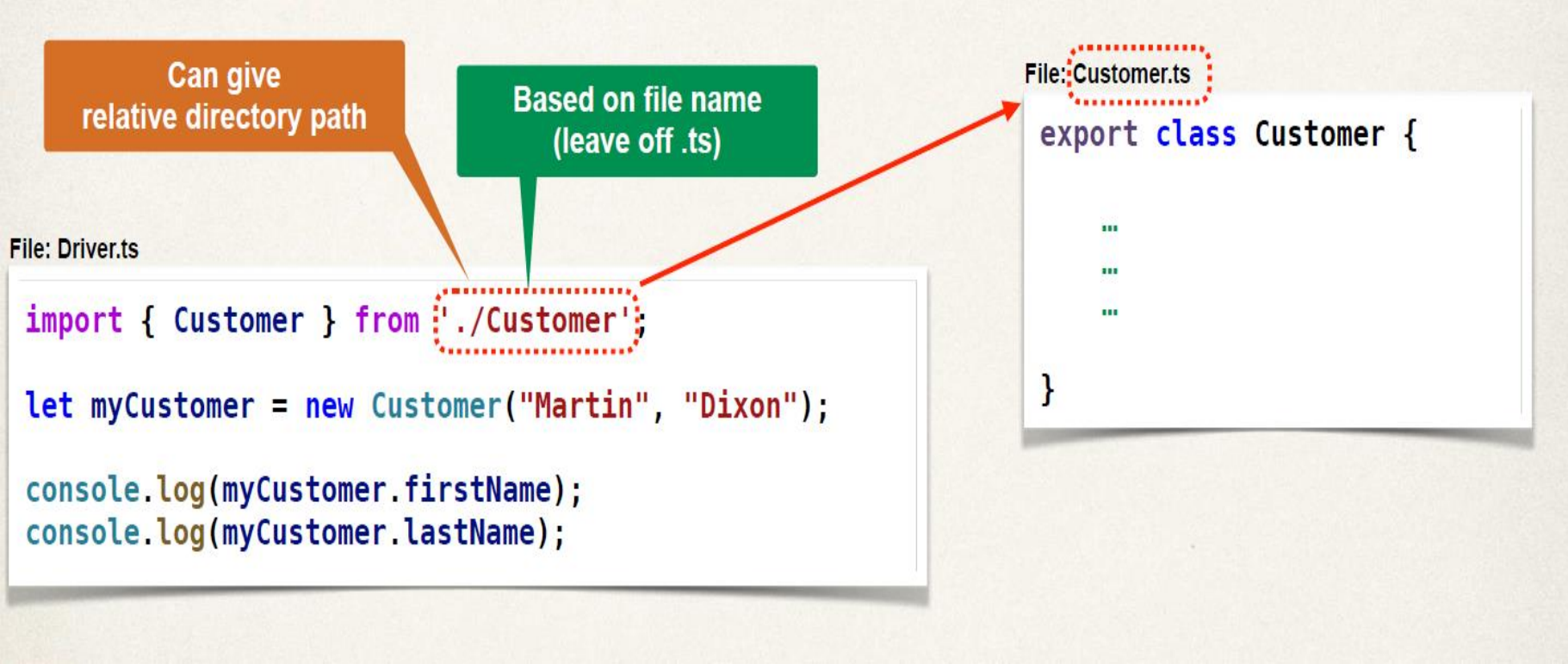
Module Example

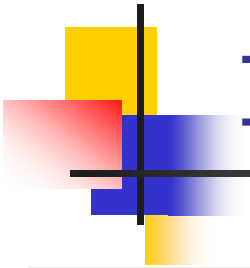
File: Customer.ts

```
export class Customer {

    ...
    ...
    ...

}
```

Export the class

Module Example

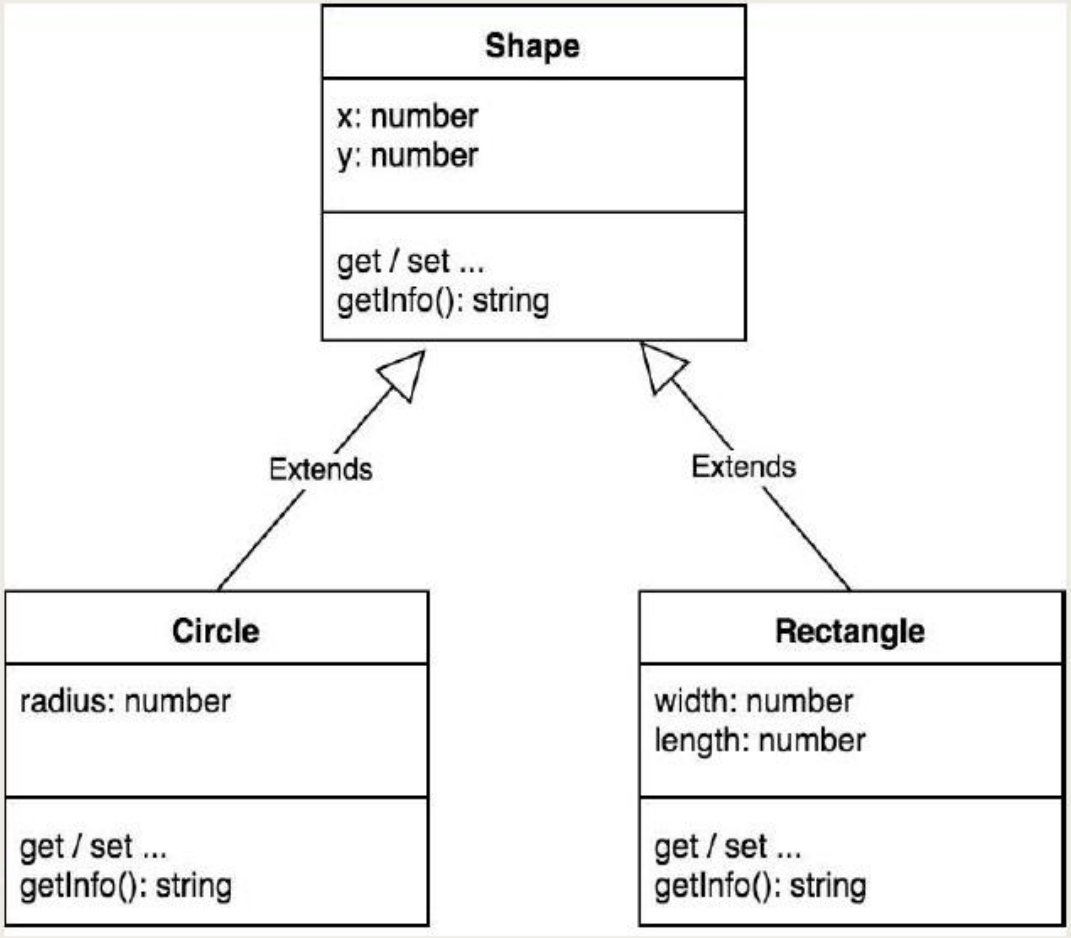




Inheritance

- TypeScript supports the object-oriented concept of inheritance
  - Define common properties and methods in the superclass
  - Subclasses can extend superclasses and add properties and methods
  - Support for abstract classes and overriding
- TypeScript only supports single inheritance
  - However, you can implement multiple interfaces

Inheritance Example



Can override the getInfo() method

Inheritance Example

File: Shape.ts

```
export class Shape {

    constructor(private _x: number, private _y: number) {
    }

    // get/set accessors ...

    getInfo(): string {
        return `x=${this._x}, y=${this._y}`;
    }
}
```

File: Circle.ts

```
import { Shape } from './Shape';

export class Circle extends Shape {

    constructor(theX: number, theY: number,
                private _radius: number) {
        super(theX, theY);
    }

    // get/set accessors ...

    getInfo(): string {
        return super.getInfo() + `, radius=${this._radius}`;
    }
}
```

Override the getInfo() method

Creating a main app

File: Shape.ts

```
export class Shape {
    …
    getInfo() {
        return `x=${this._x}, y=${this._y}`;
    }
}
```

File: Circle.ts

```
import { Shape } from './Shape';

export class Circle extends Shape {
    …
    getInfo() {
        return super.getInfo() + `, radius=${this._radius}`;
    }
}
```

File: Driver.ts

```
import { Shape } from './Shape';
import { Circle } from './Circle';

let myShape = new Shape(10, 15);
console.log(myShape.getInfo());

let myCircle = new Circle(5, 10, 20);
console.log(myCircle.getInfo());
```

```
x=10, y=15
x=5, y=10, radius=20
```

Rectangle

File: Rectangle.ts

```
import { Shape } from './Shape';

export class Rectangle extends Shape {

    constructor(theX: number, theY: number,
                private _width: number, private _length: number) {

        super(theX, theY);
    }

    // get/set accessors ...

    getInfo(): string {
        return super.getInfo() + `, width=${this._width}, length=${this._length}`;
    }
}
```

Creating a main app

File: Driver.ts

```
import { Shape } from './Shape';
import { Circle } from './Circle';
import { Rectangle } from './Rectangle';

let myShape = new Shape(10, 15);
console.log(myShape.getInfo());

let myCircle = new Circle(5, 10, 20);
console.log(myCircle.getInfo());

let myRectangle = new Rectangle(0, 0, 3, 7);
console.log(myRectangle.getInfo());
```

```
x=10, y=15
x=5, y=10, radius=20
x=0, y=0, width=3, length=7
```

Creating an Array of Shapes

File: ArrayDriver.ts

```
… …

let myShape = new Shape(10, 15);
let myCircle = new Circle(5, 10, 20);
let myRectangle = new Rectangle(0, 0, 3, 7);

// declar  array  shapes … initially empty
let theSha   Sh   = [];

// add the s    to the array
theShapes.    pe);
theShape     h(   le);
theShape     ush(my  tangle);

for (let tempShape of theShapes) {
    console.log(tempShape.getInfo());
}
```

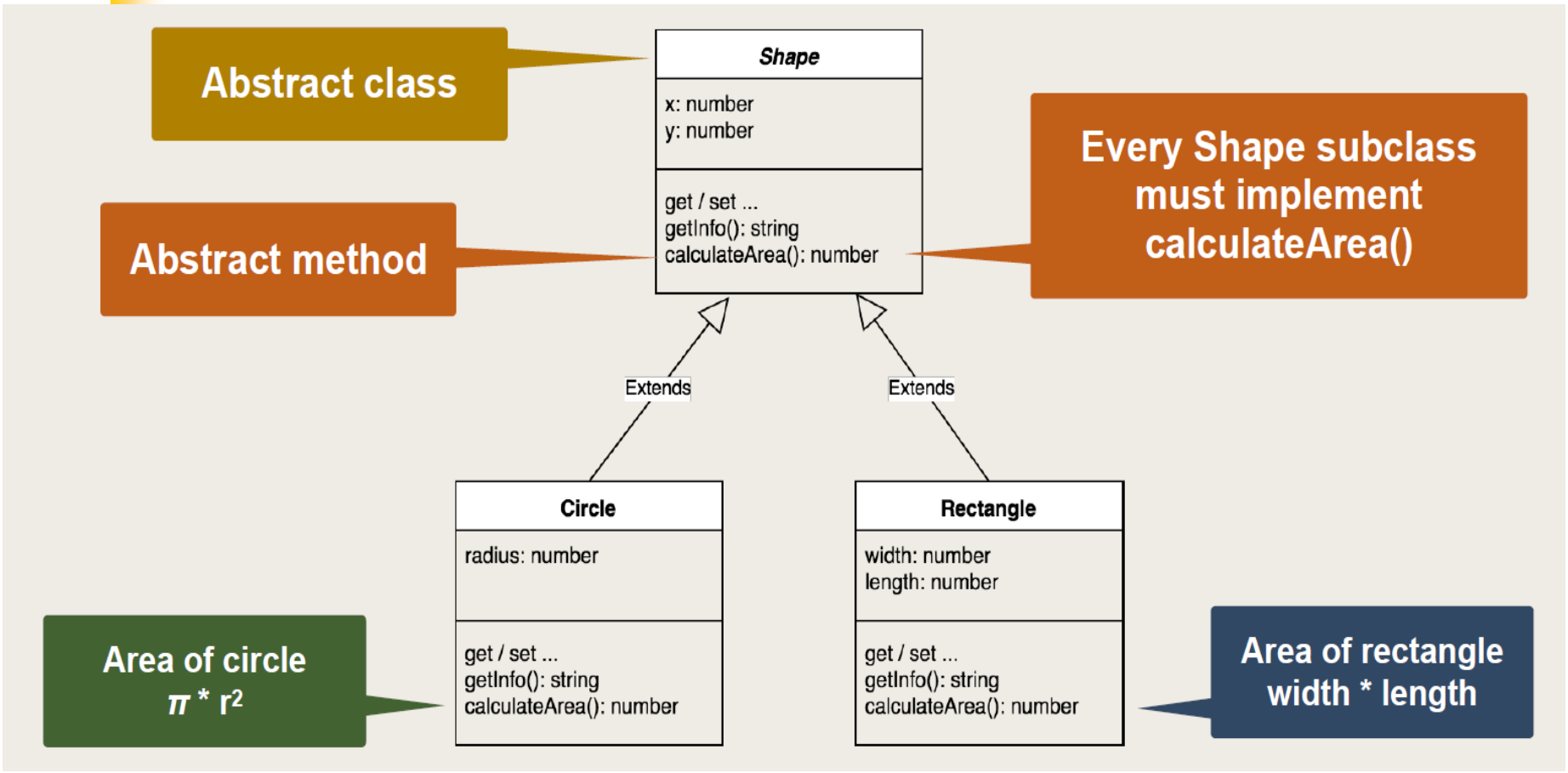
```
// declare an array for shapes
let theShapes: Shape[] = [myShape, myCircle, myRectangle];
```

```
x=10, y=15
x=5, y=10, radius=20
x=0, y=0, width=3, length=7
```

Abstract Class

- An abstract class represents a general concept
  - For example: Shape, Vehicle, Computer, etc …
- Can't create an instance of an abstract class
- Abstract class can also have abstract method(s)
- Abstract method must be implemented by concrete subclasses

Abstract Class Example



Abstract Class Example

Mark the class as abstract

File: Shape.ts

```
export abstract class Shape {

    // previous code ...

    abstract calculateArea(): number;
}
```

Abstract method

Override the calculateArea() method

File: Rectangle.ts

```
import { Shape } from './Shape';

export class Rectangle extends Shape {

    // previous code ...

    calculateArea(): number {
        return this._width * this._length;
    }
}
```

Circle

File: Shape.ts

```
export abstract class Shape {

    // previous code ...

    abstract calculateArea(): number;
}
```

File: Circle.ts

```
import { Shape } from './Shape';

export class Circle extends Shape {

    // previous code ...

    calculateArea(): number {
        return Math.PI * Math.pow(this._radius, 2);
    }
}
```

Override the calculateArea() method

Area of circle
$\pi * r^2$

Creating an Instance

```
let myShape = new Shape(10, 15);
console.log(myShape.getInfo());
```

This will NOT compile since Shape is an abstract class

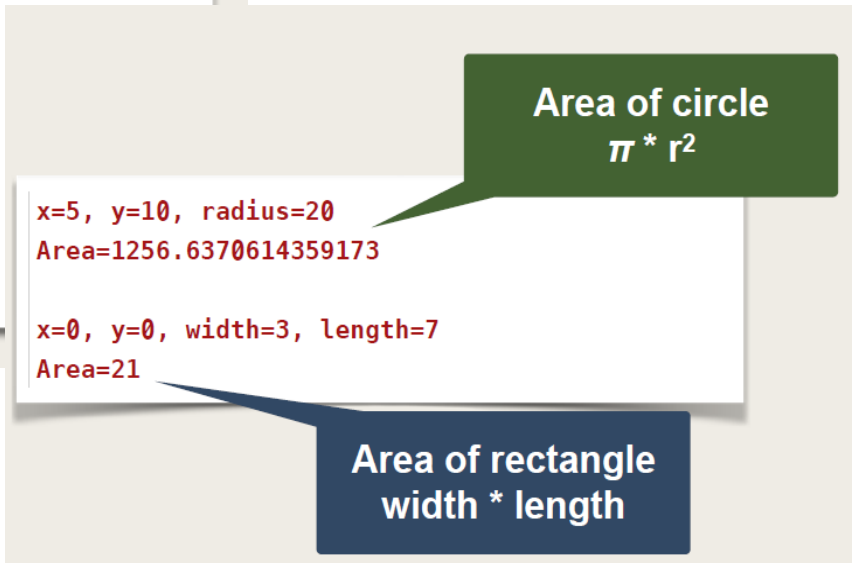
Can't create instance of abstract class directly

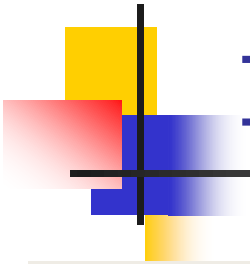
Only concrete subclasses: Circle, Rectangle, …

Creating an Array of Shapes

File: ArrayDriver.ts

```
… …

let myCircle = new Circle(5, 10, 20);
let myRectangle = new Rectangle(0, 0, 3, 7);

// declare an array for shapes … initially empty
let theShapes: Shape[] = [];

// add the shapes to the array
theShapes.push(myCircle);
theShapes.push(myRectangle);
```

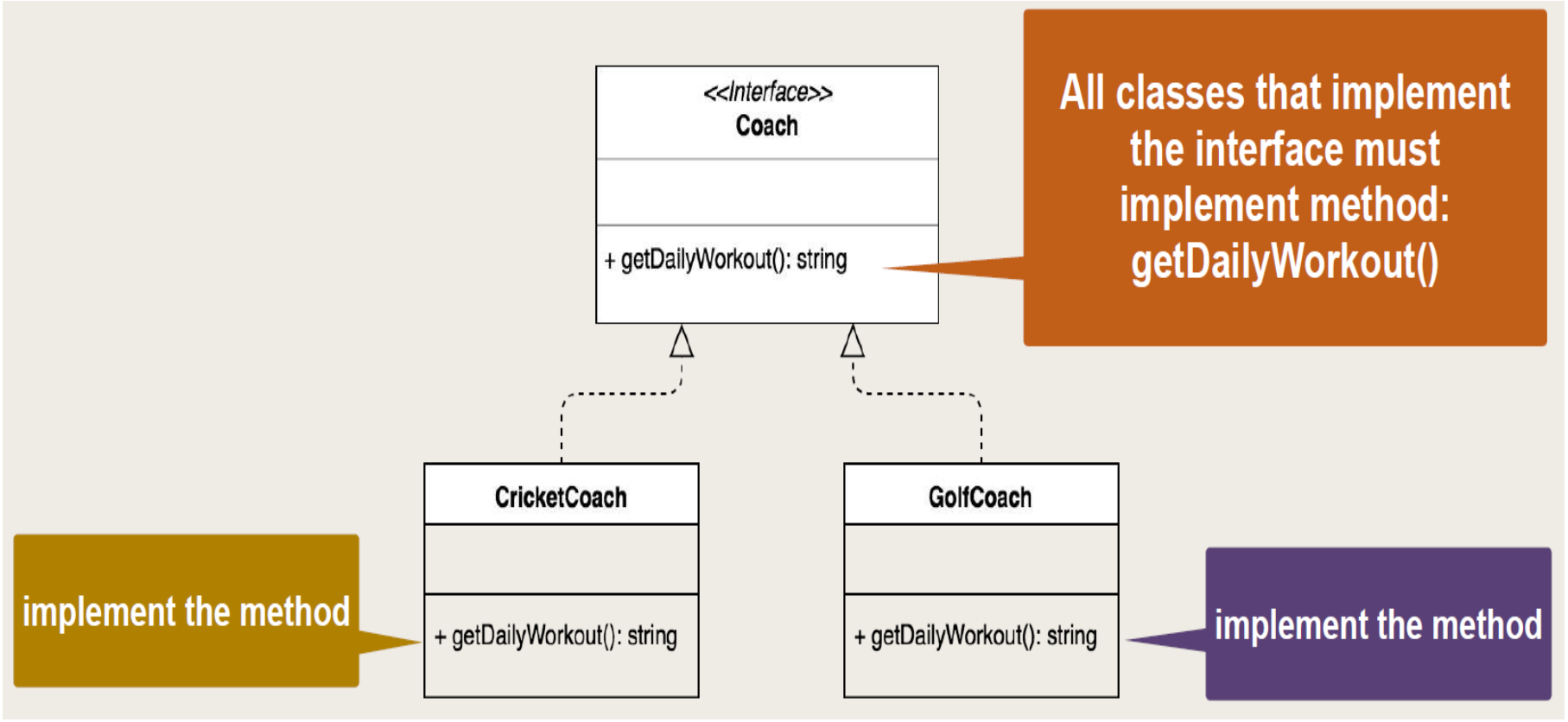




Interfaces

- TypeScript supports interfaces
  - Define an interface with a method contract
  - Classes implement the interface accordingly
  - A class can implement multiple interfaces
- TypeScript can also use interfaces to support contracts with properties
  - For examples with properties, see:
  - `http://www.typescriptlang.org/docs/handbook/interfaces.html`

Interface Example



Interface Example

Define the interface

File: Coach.ts

```
export interface Coach {

    getDailyWorkout(): string;
}
```

Implement the interface

File: CricketCoach.ts

```
import { Coach } from "./Coach";

export class CricketCoach implements Coach {

    getDailyWorkout(): string {
        return "Practice your spin bowling technique.";
    }
}
```

Implement the interface

File: GolfCoach.ts

```
import { Coach } from "./Coach";

export class GolfCoach implements Coach {

    getDailyWorkout(): string {
        return "Hit 100 balls at the golf range.";
    }
}
```

Creating a Main App

File: Driver.ts

```
import { Coach } from "./Coach";
import { CricketCoach } from "./CricketCoach";
import { GolfCoach } from "./GolfCoach";

let myCricketCoach = new CricketCoach();
console.log(myCricketCoach.getDailyWorkout());

let myGolfCoach = new GolfCoach();
console.log(myGolfCoach.getDailyWorkout());
```

```
Practice your spin bowling technique.
Hit 100 balls at the golf range.
```

Creating an Array of Coaches

File: ArrayDriver.ts

```
... ...

let myCricketCoach = new CricketCoach();
let myGolfCoach = new GolfCoach();

// declare an array for coaches ... initially empty
let theCoaches: Coach[] = [];

// add the coaches to the array
theCoaches.push(myCricketCoach);
theCoaches.push(myGolfCoach);

for (let tempCoach of theCoaches) {
    console.log(tempCoach.getDailyWorkout());
}
```

```
Practice your spin bowling technique.
Hit 100 balls at the golf range.
```

interfaces and abstract classes: Key Differences

Feature	Interface	Abstract Class
Definition	Defines a structure (shape) for an object	Can have both implemented and abstract (unimplemented) methods
Implementation	No implementation, only method signatures	Can have method implementations
Inheritance	Can be implemented by multiple classes	Can only be extended (single inheritance)
Usage	Only defines type checking, removed at compile time	Exists in the compiled JavaScript code
Constructors	Cannot have constructors	Can have constructors
Access Modifiers	Cannot have access modifiers (public, private, protected)	Supports access modifiers

In **Angular (TypeScript)**, both **interfaces** and **abstract classes** help define the structure of objects, but they serve different purposes and have key differences.

interfaces and abstract classes: Key Differences

```typescript
interface Vehicle {
  speed: number;
  start(): void;
}

class Car implements Vehicle {
  speed = 100;
  start() {
    console.log("Car started!");
  }
}
```

```typescript
abstract class Vehicle {
  constructor(public speed: number) {}

  abstract start(): void; // Must be implemented in subclasses

  stop() {
    console.log("Vehicle stopped.");
  }
}

class Car extends Vehicle {
  start() {
    console.log("Car started at speed:", this.speed);
  }
}
```

Use an Interface when you just need to define a contract/shape for objects.
Use an Abstract Class when you want to provide a base class with some shared implementation and enforce method overriding.