TypeScript

FROM 'CAPABILITY' TO 'EXPERTISE'

Limitations of JavaScript

- Lack of Static Typing
- Missing Compile-Time Error Checking
- Limited IDE Support
- Undefined Behavior and Null References
- Lack of Modularization Features
- Difficulty in Scaling and Maintenance

TypeScript

TYPESCRIPT LETS YOU WRITE JAVASCRIPT THE WAY YOU REALLY WANT TO.

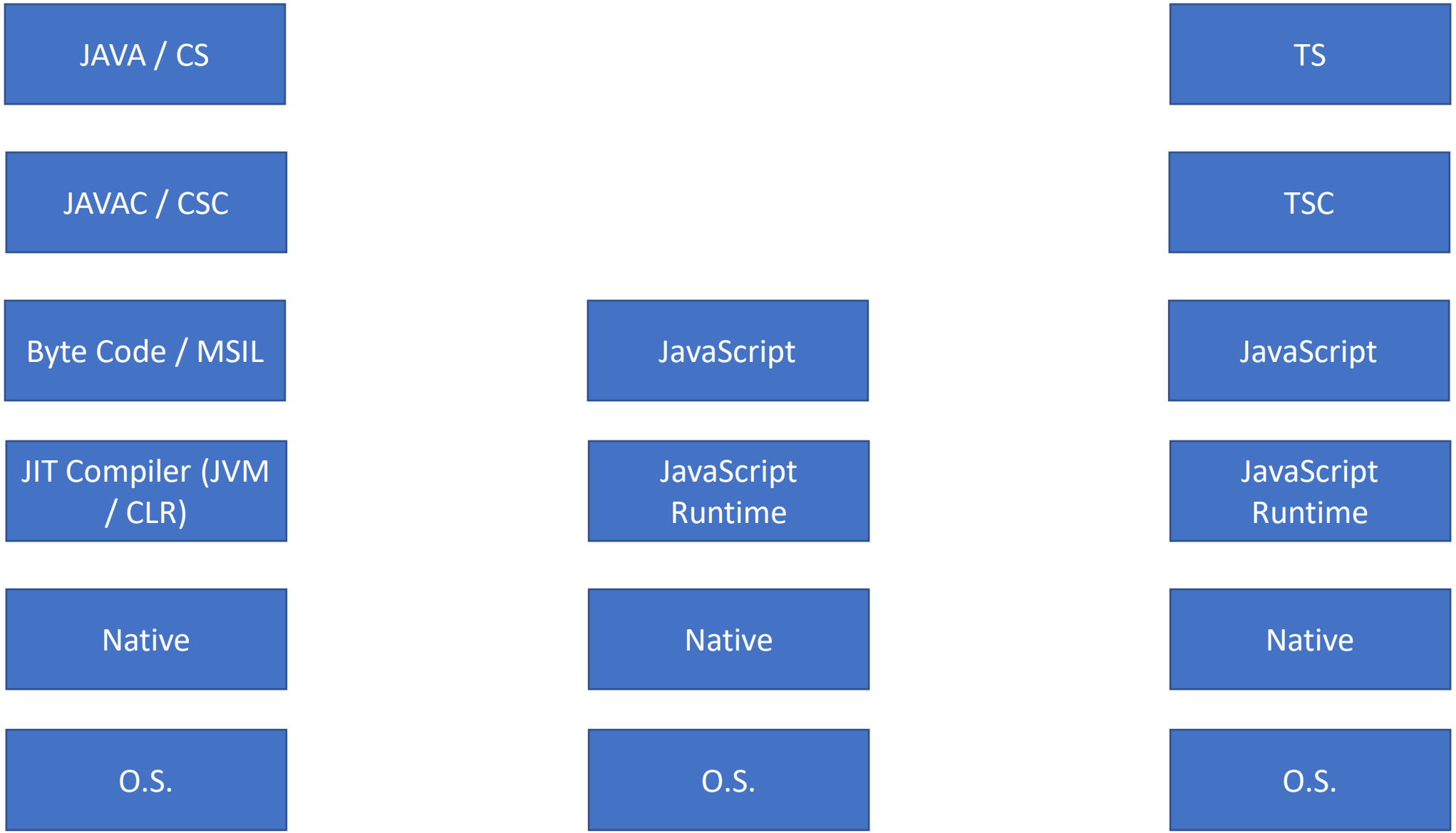
TYPESCRIPT IS A TYPED SUPERSET OF JAVASCRIPT THAT COMPILES TO PLAIN JAVASCRIPT.

SYNTAX BASED ON ECMASCRIPT PROPOSALS.

ANY REGULAR JAVASCRIPT IS VALID TYPESCRIPT CODE

Installation

- Install Node LTS version - https://nodejs.org/en/
- Install / Update Visual Studio Code to Latest Release - https://code.visualstudio.com/
- Extensions of VS Code
  - AutoFileName
  - vscode-icons
  - JavaScript (ES6) code snippets
  - Live Server
- After all Extensions are installed
  - File Menu -> Preferences -> Theme -> File Icon Theme -> Select VSCode Icons

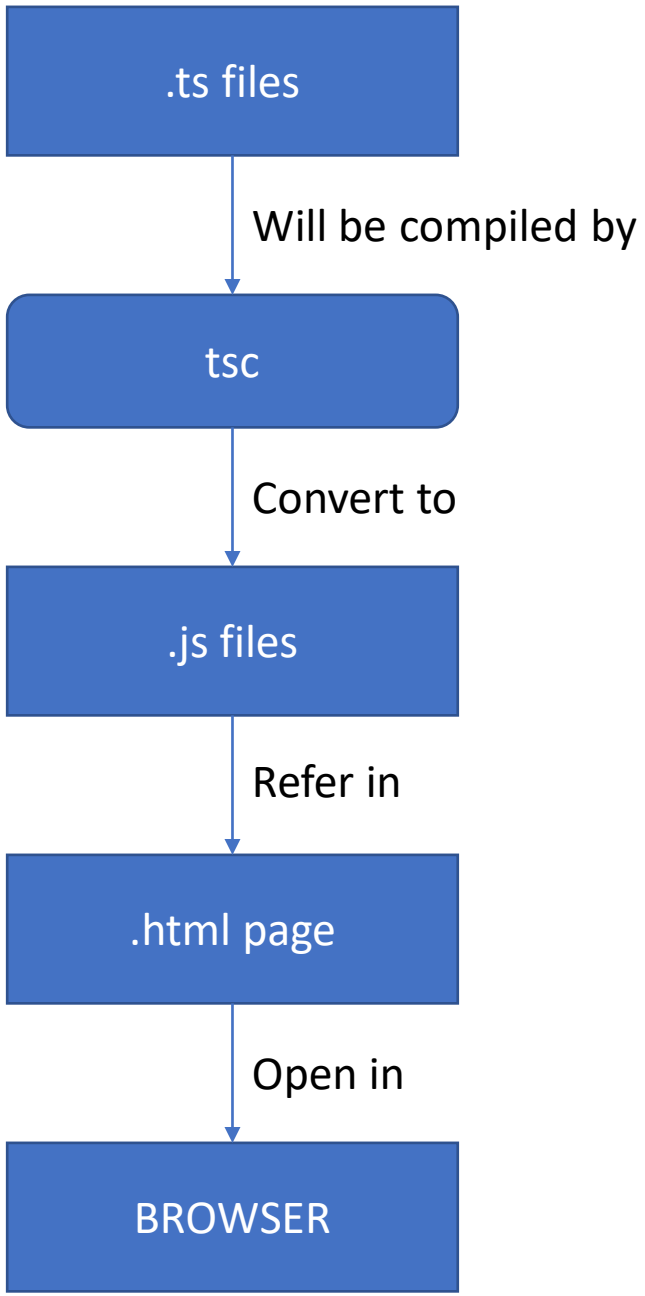

JAVA / CS
JAVAC / CSC
Byte Code / MSIL
JIT Compiler (JVM / CLR)
Native
O.S.
JavaScript
JavaScript Runtime
Native
O.S.
TS
TSC
JavaScript
JavaScript Runtime
Native
O.S.

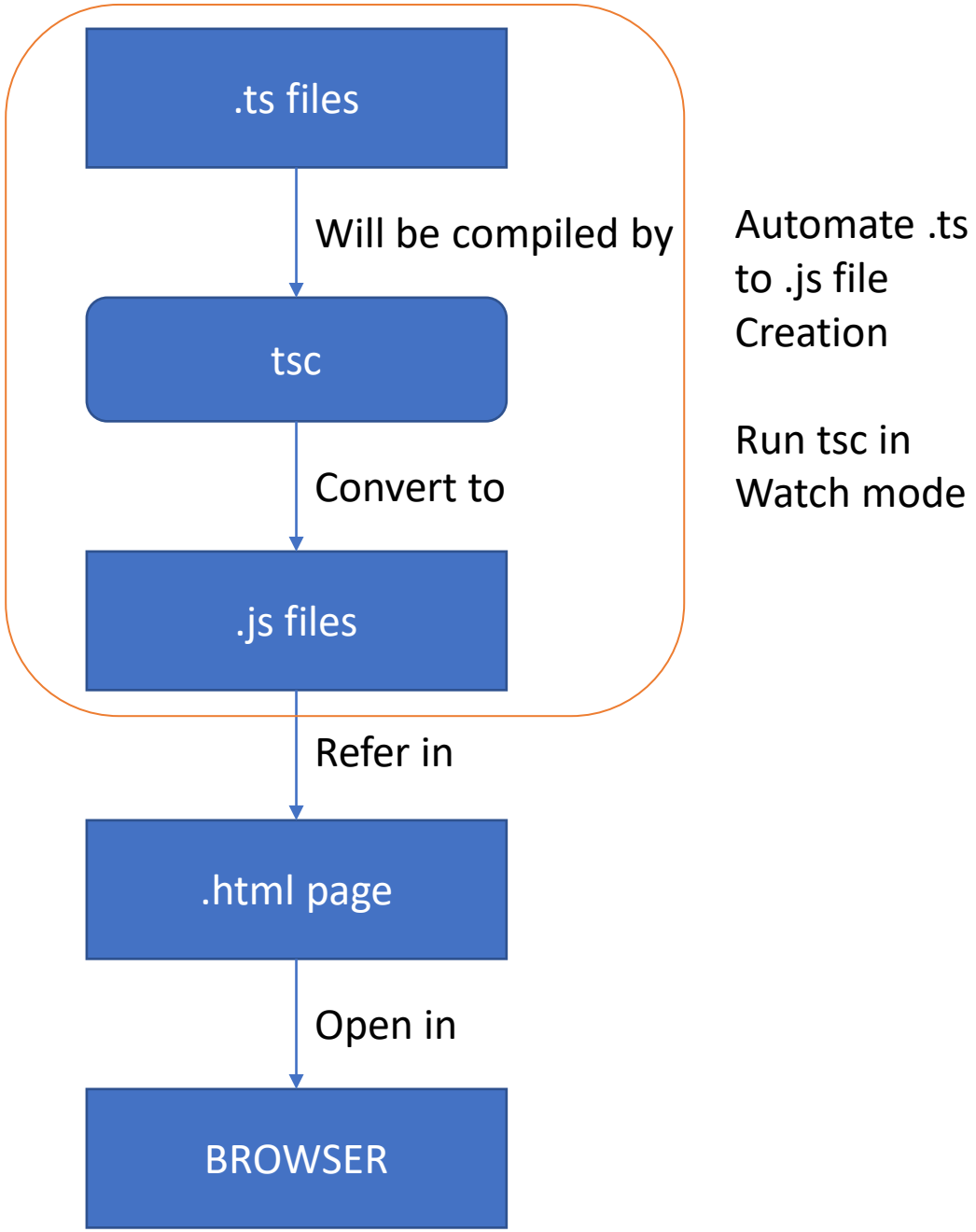
How to install typescript?

- Global Installation
  - npm install -g typescript
    OR
  - npm i -g typescript
  - Check the TypeScript Compiler version installed
    - tsc -v
  - Compile a TypeScript File to JavaScript output
    - tsc <full path of ts file> e.g. tsc demo3.ts
    - tsc <full path of ts file> -t es2015 e.g. tsc demo3.ts -t es2015
- Local Installation
  - npm install --save-dev typescript
    OR
  - npm i -D typescript

Local Installation – New Project

1. Create a new folder
2. Open the folder path in terminal window
3. npm init -y
4. Change the package.json file as required & add the compile script
5. npm i -D typescript@4.8.4
6. Check the TypeScript Compiler version installed
   - npx tsc -v


.ts files
Will be compiled by
tsc
Convert to
.js files
Refer in
.html page
Open in
BROWSER


.ts files
Will be compiled by
tsc
Convert to
.js files
Refer in
.html page
Open in
BROWSER
Automate .ts to .js file Creation
Run tsc in Watch mode

Local Installation – New Project

1. Create a new folder (TSDemos)
2. Open the folder path in terminal window
3. npm init -y
4. Changed the package.json file as required
5. npm i -D typescript@4.8.4
6. Create and configure tsconfig.json
7. Create a file 1_first-demo.ts in root folder
8. To run the compiler in watch mode, use the following command
   - npm run compile

Using the Shared Application

1. Download & extract TSDemos.zip file
2. Open the folder in VS Code
3. Open Terminal window on the folder path and run the following command

   npm install
4. To run the compiler in watch mode, use the following command

   npm run compile

Target - ECMASCRIPT Versions

- ES3
- ES5
- ECMASCRIPT 2015
- ECMASCRIPT 2016
- ECMASCRIPT 2017
- ECMASCRIPT 2018
- ECMASCRIPT 2019
- ECMASCRIPT 2020
- ECMASCRIPT 2021
- ECMASCRIPT 2022
- ECMASCRIPT 2023

Main Goals of TypeScript

- Provide an optional type system for JavaScript.

```
1    var foo = 123;
2    foo = '456'; // Error: cannot assign 'string' to 'number'
3
4    var foo: number = 123;
5    var foo: number = '123'; // Error: cannot assign a 'string' to a 'number'
```

- Provide planned features from future JavaScript editions to current JavaScript engines
- Modular Development

TypeScript Features

- Data Types Supported
- Optional Static Type Annotation
- Classes
- Interface
- Modules
- Arrow Expressions
- Type Assertions
- Ambient Declarations
- Source File Dependencies

Type Inference

- TypeScript tries to infer types
- Four ways to variable declaration -
  - Type and Value in one statement
  - Type but no Value then Value will be undefined
  - Value but on Type then the it will be of Any type but maybe be inferred based on its value.
  - Neither Value nor Type then Type will be Any and Value will be undefined.

```
var message1:string = "Hello World";
var message2:string;
var message3 = "Hello World";
var message4;
```

Data Types

- Any
- Never
- Unknown
- Primitive
  - Number
  - Boolean
  - String
  - Void
  - Null
  - Undefined
- Array
- Tuple
- Enum

Any

- Any is used when it’s impossible to determine the type

```
// When it's impossible to know, there is the "Any" type
var notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

Primitive

- Doesn’t have separate integers and float/double type. These all are floating point values and get the type ‘number’
- boolean - true/false value
- string - both single/double quote can be used
- No separate char type
- void - is used in function type returning nothing
- null and undefined - functions as usual

```
var isDone: boolean = false;
var lines: number = 42;
var name: string = "Hello World";

function bigHorribleAlert(): void {
  alert("I'm a little annoying box!");
}
```

Type Annotations/Checking

- JavaScript

```
var a = 987;
a.trim();
//JavaScript error: TypeError: a.trim is not a function on
```

- TypeScript

```
var a = 987;
a.trim();
//Property 'trim' doesn't exist on 'number'
var a:string = 123
a.trim()
//Cannot convert 'number' to 'string'
```

Type Assertion

- TypeScript's type assertion are purely, you telling the compiler that you know about the types better than it does, and that it should not second guess you.
- Type assertion is a mechanism which tells the compiler about the type of a variable.
- Type assertion is explicitly telling the compiler that we want to treat the entity as a different type.

Type Assertion Example

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

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

Type Guards / Union Types

- Type Guards allow you to narrow down the type of a variable within a conditional block.
- Union types are a powerful way to express a value that can be one of the several types.
- Two or more data types are combined using the pipe symbol (|) to denote a Union Type. In other words, a union type is written as a sequence of types separated by vertical bars.

Functions

- Writing a function in TypeScript is like writing them in JavaScript but with added parameters and return type.
- Note that any JavaScript function is a perfectly valid TypeScript function. However, we can do better by adding type.
- TypeScript function syntax (with two arguments)

```
function functionName(arg1: <arg1Type>,
        arg2: <arg2Type>): <returnType> {
    // Function body...
}
```

Function Parameters

- Function Parameters are required, and you cannot pass extra arguments to a function
- Function Parameters are also type safe, if you don't use any type explicitly
- Adding an optional parameter is super simple, just add ? to the end of the argument name.
- For default argument suffix it with an equal sign and default value (TS compiler will automatically deduce the type for default argument based on provided value).

Lambda Expression

- Implicit return
- No braces for single expression
- Part of ES6 termed as Arrow Functions
- Lexically scoped this
- You don't need to keep typing function
- It lexically captures the meaning of arguments

```
function(arg){
  return arg.toLowerCase();
}

(arg) => arg.toLowerCase();
```

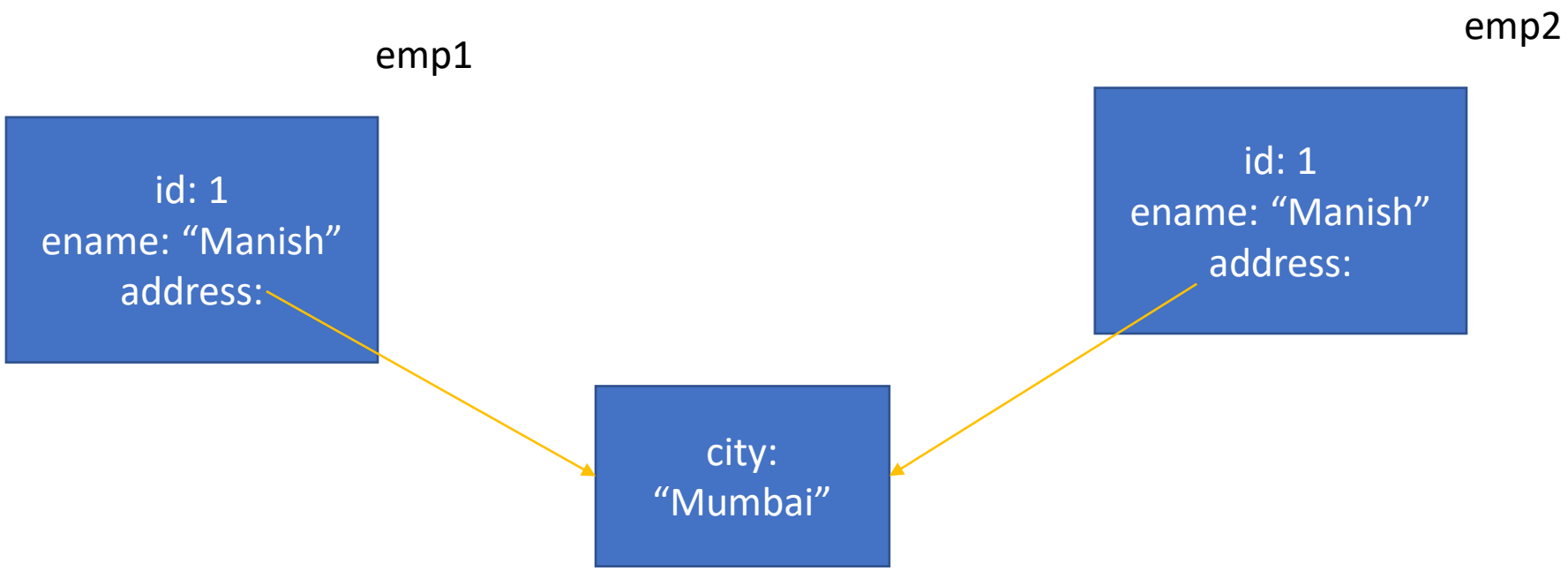
Rest Parameter (...args)

- A rest parameter allows you a function to accept zero or more arguments of the specified type.
- In TypeScript, rest parameters follow these rules:
  - A function has only one rest parameter.
  - The rest parameter appears last in the parameter list.
  - The type of the rest parameter is an array type.
- To declare a rest parameter, you prefix the parameter name with three dots and use the array type as the type annotation:

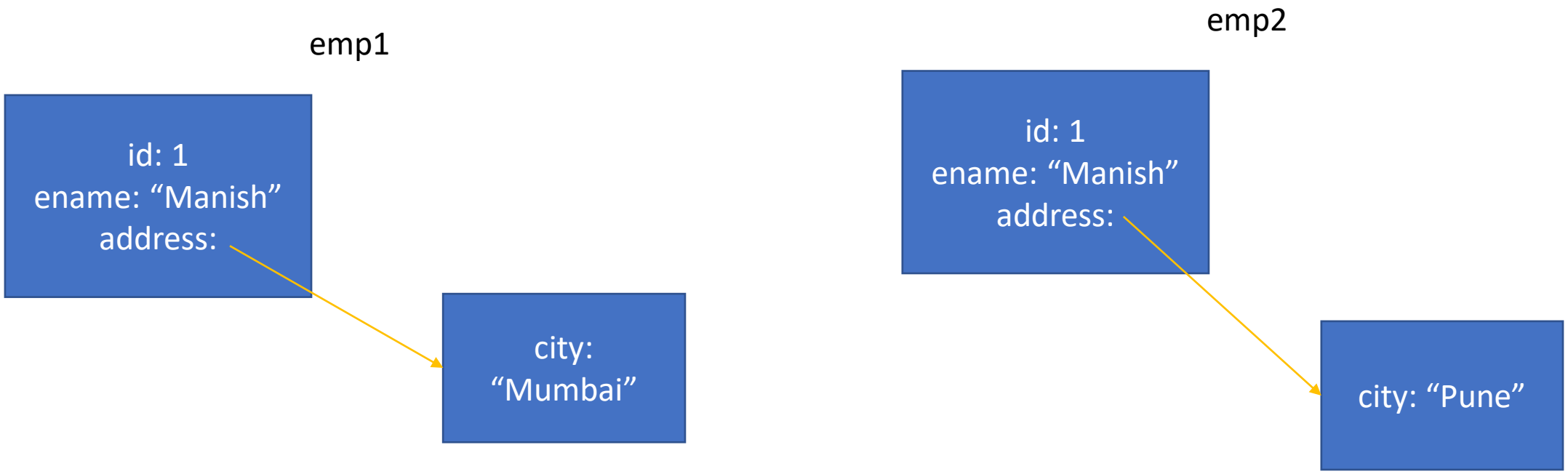
Rest vs Spread

- Rest and spread are two related features in TypeScript that allow developers to work with arrays and objects more easily.
  - Rest Parameters
    - Rest parameters allow you to represent an indefinite number of arguments as an array.
    - You can use the rest parameter syntax (...) to indicate that a function should accept any number of arguments.
  - Spread Syntax
    - Spread syntax allows you to "spread" the elements of an array or object into another array or object.
    - You can use the spread syntax (...) to expand an array into its individual elements, or to merge two or more objects into a new object.

Shallow Copy



Deep Copy


emp1
id: 1
ename: “Manish”
address:
city:
“Mumbai”
emp2
id: 1
ename: “Manish”
address:
city: “Pune”

Arrays

- The use of variables to store values poses the following limitations –
  - Variables are scalar in nature. In other words, a variable declaration can only contain a single at a time. This means that to store n values in a program n variable declarations will be needed. Hence, the use of variables is not feasible when one needs to store a larger collection of values.
  - Variables in a program are allocated memory in random order, thereby making it difficult to retrieve/read the values in the order of their declaration.
- TypeScript introduces the concept of arrays to tackle the same.
- An array is a homogenous collection of values.
- An array is a collection of values of the same data type.
- It is a user defined type.

Features of an Array

- An array declaration allocates sequential memory blocks.
- Arrays are static. This means that an array once initialized cannot be resized.
- Each memory block represents an array element.
- Array elements are identified by a unique integer called as the subscript / index of the element.
- Like variables, arrays too, should be declared before they are used. Use the var keyword to declare an array.
- Array initialization refers to populating the array elements.
- Array element values can be updated or modified but cannot be deleted.

Array

```
var cities:string[] = ["Berlin","Quebec","New York"]
var primes:number[] = [1,3,5,7,11,13]
var bools:boolean[] = [true,false,false,true]

// Alternatively, using the generic array type
var list: Array<number> = [1, 2, 3];
```

Learn for Day 2

- What are Modules in JavaScript?
  - CommonJS
  - Require
  - System
  - AMD
  - UMD
  - ESModule
- Learn about the other compiler options in tsconfig.json file
  - https://www.typescriptlang.org/tsconfig

Tuple

- As we know array consists value of Homogeneous types but sometimes, we need to store a collection of a different type value in a single variable. Then we will go with Tuples.
- Tuples are just like structure in C programming and can also be passed as parameters in a function call.
- To denote a multi-dimensional coordinate system the term used is tuple in abstract mathematics.
- JavaScript doesn’t have tuples as data types, but in typescript Tuple's facility is available.
- Tuple values are individually called items. Tuples are index based. This means that items in a tuple can be accessed using their corresponding numeric index.

Enum

- Enums or enumerations are a new data type supported in TypeScript. Most object-oriented languages like Java and C# use Enums. This is now available in TypeScript too.
- In simple words, Enums allow us to declare a set of named constants i.e., a collection of related values that can be numeric or string values.
- Using Enum, we can make it easier to document intent, or create a set of distinct cases.
- There are three types of Enums:
  - Numeric Enum
  - String Enum
  - Heterogeneous Enum

Enum

- By default, Enums begin numbering their members starting at 0. You can change this by manually setting the value of one its members.

```
// For enumerations:
enum Color {Red, Green, Blue};
var c: Color = Color.Green;
enum Color {Red = 0, Green, Blue};
enum Color {Red = 3, Green, Blue};
```

TypeScript Interface

- Interface is a structure that defines the contract in your application.
- Declared using interface keyword
- Like other TypeScript features its design time features i.e., no extra code would be emitted to resultant JavaScript file
- TypeScript compiler uses interface for type checking. This is also known as "duck typing" or "structural subtyping".
- Errors being shown when interface signature and implementation doesn’t match.

TypeScript Class

- In object-oriented programming languages like Java and C#, classes are the fundamental entities used to create reusable components.
- TypeScript introduced classes to avail the benefit of object-oriented techniques like encapsulation and abstraction.
- The class in TypeScript is compiled to plain JavaScript functions by the TypeScript compiler to work across platforms and browsers.
- The concept of 'Encapsulation' is used to make class members public or private i.e., a class can control the visibility of its members. This is done using access modifiers.
- There are three types of access modifiers in TypeScript:
  - public (Default)
  - private
  - protected.

TypeScript Class

- A class can have the following features
  - Instance methods/members
  - Single constructor with Default/Optional parameter
  - Optional and Required Members in Strict Mode
  - Parameter Members
  - Static methods/members
  - Readonly Members
  - Can implement interfaces
  - Inheritance

Class Example

```
// Classes - members are public by default
class Point {
  // Properties
  x: number;
  constructor(x: number, public y: number = 0) {
    this.x = x;
  }
  // Functions
  dist() { return Math.sqrt(this.x * this.x + this.y * this.y); }
  // Static members
  static origin = new Point(0, 0);
}

var p1 = new Point(10 ,20);
var p2 = new Point(25); //y will be 0

// Inheritance
class Point3D extends Point {
  constructor(x: number, y: number, public z: number = 0) {
    super(x, y); // Explicit call to the super class constructor is mandatory
  }
  // Overwrite
  dist() {
    var d = super.dist();
    return Math.sqrt(d * d + this.z * this.z);
  }
}
```

Static

- The static keyword defines a static method or property for a class, or a class static initialization block.
- Neither static methods nor static properties can be called on instances of the class. Instead, they're called on the class itself.
- Static methods are often utility functions, such as functions to create or clone objects, whereas static properties are useful for caches, fixed-configuration, or any other data you don't need to be replicated across instances.

Use cases for Static

- Utility functions:
  - Static methods can be used to define utility functions that are related to a class but don't require an instance to be created. For example, a Math class could have a static method Math.random() that generates a random number.
- Configuration:
  - Static properties can be used to store configuration values that apply to all instances of a class. For example, a Database class could have a static property Database.maxConnections that specifies the maximum number of connections allowed to the database.
- Singleton pattern:
  - The static keyword can be used to implement the Singleton pattern, where a class has only one instance that is shared across the entire application.
- Factory methods:
  - Static methods can be used as factory methods to create instances of a class. For example, a Person class could have a static method Person.create(name: string, age: number) that creates a new instance of the Person class with the specified name and age.

Readonly

- The readonly keyword is used to indicate that a property or variable should not be modified after it has been initialized.
- When a property is marked as readonly, it can only be set once either in its declaration or in the constructor of the class.
- Similarly, when a variable is marked as readonly, it can only be assigned a value once.
- Using “readonly” can help prevent accidental modifications to properties or variables, making code more robust and less error-prone.

Use cases for Readonly

- Immutable data structures:
  - By marking properties as readonly, you can create immutable data structures that cannot be modified once they are initialized. This can help to prevent bugs that can be caused by unintended modifications to data.
- Public API:
  - When creating a public API for a library or module, marking properties as readonly can help to ensure that users of the API do not accidentally modify values that are not intended to be changed.
- Thread safety:
  - When multiple threads are accessing the same data, marking properties as readonly can help to prevent race conditions and ensure thread safety.
- Performance optimization:
  - Marking properties as readonly can help the TypeScript compiler to optimize code, as it knows that these values will not change at runtime.

Generics

- When writing programs, one of the most important aspects is to build reusable components. This ensures that the program is flexible as well as scalable in the long-term.
- Generics offer a way to create reusable components. Generics provide a way to make components work with any data type and not restrict to one data type.
- So, components can be called or used with a variety of data types, without loosing type safety and intellisense.
- Generic Type can be used with
  - Functions
  - Class
  - Interface

Generic Constraints

- As mentioned earlier, the generic type allows any data type. However, we can restrict it to certain types using constraints.
- A constraint is specified after the generic type in the angle brackets.
- Constraints can be applied using:
  - extends
  - Type Parameters
  - Class Types
  - keyof

Iterators

- Iterables
  - An object is deemed iterable if it has an implementation for the Symbol.iterator property.
  - Some built-in types like Array, Map, Set, String, Int32Array, Uint32Array, etc. have their Symbol.iterator property already implemented.
  - Symbol.iterator function on an object is responsible for returning the list of values to iterate on.
  - for..of, loops over an iterable object, invoking the Symbol.iterator property on the object.

Generators

- In TypeScript, generators provide a new way to work with functions and iterators. Generators provide an easier way to implement iterators.
- Using a generator,
  - you can stop the execution of a function from anywhere inside the function
  - and continue executing code from a halted position
- To create a generator, you need to first define a generator function with function* symbol.
- You can pause the execution of a generator function without executing the whole function body by using yield keyword.
- The yield expression returns a value. However, unlike the return statement, it doesn't terminate the program. That's why you can continue executing code from the last yielded position.

Uses of Generators

- Generators let us write cleaner code while writing asynchronous tasks.
- Generators provide an easier way to implement iterators.
- Generators execute its code only when required.
- Generators are memory efficient.

Decorators

- Decorators provide a way to add both annotations and a meta-programming syntax.
- A Decorator is a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter.
- Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration.
- Decorators are just a clean syntax for wrapping a piece of code with a function

Inheritance

- TypeScript is an object-oriented programming language that supports inheritance.
- Inheritance is a mechanism that allows one class to inherit the properties and methods of another class.
- In TypeScript, inheritance is achieved using the extends keyword.
- When a class extends another class, it inherits all the properties and methods of the base class.
- The subclass can also add its own properties and methods, and override or extend the behavior of the inherited methods.

Mixins

- In JavaScript we can only inherit from a single object. There can be only one **[[Prototype]]** for an object. And a class may extend only one other class.
- A mixin is a class containing methods that can be used by other classes without a need to inherit from it.
- In other words, a mixin provides methods that implement a certain behavior, but we do not use it directly, we use it to add the behavior to other classes.
- The easiest way of implementing a mixin is to create an object with efficient methods, so that one could merge them to a prototype of any class.

Namespaces

- The namespace is used for logical grouping of functionalities. A namespace can include interfaces, classes, functions and variables to support a single or a group of related functionalities.
- A namespace can be created using the namespace keyword followed by the namespace name.
- By default, namespace components cannot be used in other modules or namespaces.
  - You must export each component to make it accessible outside, using the export keyword
- The generated JavaScript code for the namespace uses the IIFE pattern to stop polluting the global scope.

Modules

- A module is just a file. One typescript file is one module. As simple as that.
- A module may contain a class or a library of functions for a specific purpose.
- Modules can load each other and use special directives export and import to interchange functionality, call functions of one module from another one:
  - **export** keyword labels variables and functions that should be accessible from outside the current module.
  - **import** allows the import of functionality from other modules.

Namespace vs Module

Namespace	Module
Must use the namespace keyword and the export keyword to expose namespace components.	Uses the export keyword to expose module functionalities.
Used for logical grouping of functionalities with local scoping.	Used to organize the code in separate files and not pollute the global scope.
To use it, it must be included using triple slash reference syntax e.g. ///<reference path="path to namespace file" />.	Must import it first in order to use it elsewhere.
Compile using the --outFile command.	Compile using the --module command.
Must export functions and classes to be able to access it outside the namespace.	All the exports in a module are accessible outside the module.
Namespaces cannot declare their dependencies.	Modules can declare their dependencies.
No need of module loader. Include the .js file of a namespace using the <script> tag in the HTML page.	Must include the module loader API which was specified at the time of compilation e.g. CommonJS, require.js etc.

Module Loading

- Module loading refers to the process of dynamically loading modules or files at runtime in a web application.
- It enables the retrieval and execution of modules on-demand, rather than loading all modules upfront.
- In modern JavaScript development, web browsers have native support for ECMAScript modules, which can be loaded using the **<script type="module">** tag or dynamically using JavaScript.

Module Bundling

- Module bundling is the process of combining multiple separate modules or files into a single file, usually referred to as a bundle.
- The primary goal of module bundling is to optimize the delivery of web applications by reducing the number of HTTP requests required to load the application and improving performance.

Module Bundlers

- Module bundling, on the other hand, involves the process of combining multiple modules and their dependencies into a single file or a set of files, often referred to as bundles.
- The goal of module bundling is to optimize the size and performance of the application by reducing the number of separate HTTP requests required to load the code.
- The bundler resolves dependencies, applies transformations through loaders (e.g., transpiling, minification), and creates a bundle that can be efficiently delivered to the browser.
- Bundling is commonly used to package JavaScript, CSS, images, and other assets together, reducing the network overhead and improving the performance of the application.

Module Bundlers

- Some of the commonly used Module Bundlers are:
  - Webpack
  - Rollup
  - Parcel
  - Browserify

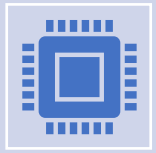
The Purpose of Bundlers

Code Bundling

Combine and optimize multiple files into a single bundle for efficient delivery to the browser.

Code Transformation

Utilize tools like Babel for transpiling modern JavaScript code into a compatible version for broader browser support. Minify code using tools like UglifyJS or Terser to reduce file size and remove unnecessary characters.



Assets Integration

Optimize and compress images with tools like ImageMagick or plugins like imagemin. Integrate fonts by including font files and using CSS @font-face rules. Embed or link other assets (videos, audio files, documents) as appropriate for your project.

Use a Bundler

- Your project involves module bundling, dependency management, and code optimization.
- Code splitting, lazy loading, or other advanced code optimization techniques are required.
- You need to bundle and optimize assets such as JavaScript, CSS, images, and other resources.
- Tree shaking (removing unused code) or transpiling code to a specific target environment is necessary.
- You want to leverage a rich ecosystem of plugins and loaders specific to bundlers.
- You require advanced optimizations for production builds, including minification and compression.

Webpack

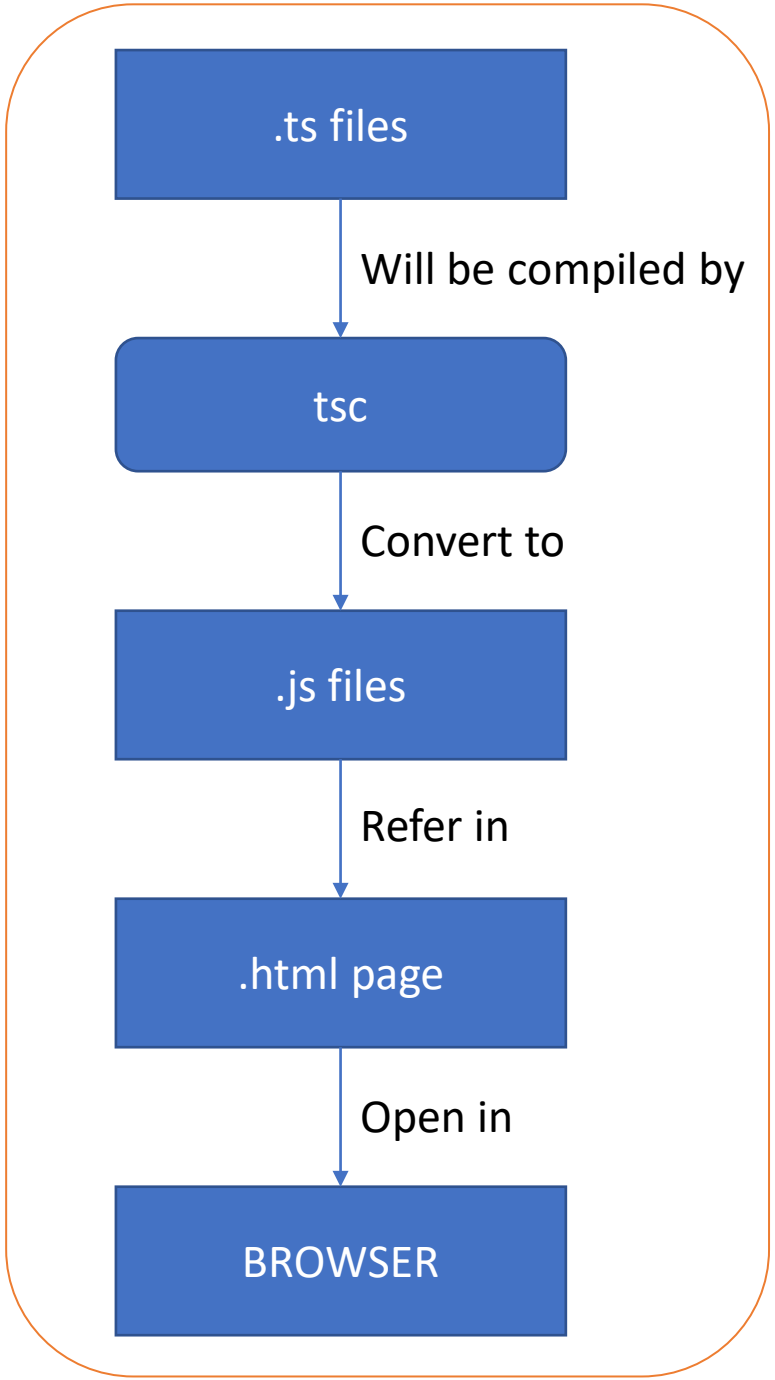
- Webpack is a popular module bundler for JavaScript applications.
- It is widely used in modern web development workflows to manage and bundle various assets, such as JavaScript files, CSS stylesheets, images, and more.
- Webpack takes your project's dependencies, processes them through loaders and plugins, and creates optimized bundles that can be served to the browser.

How Webpack Executes?

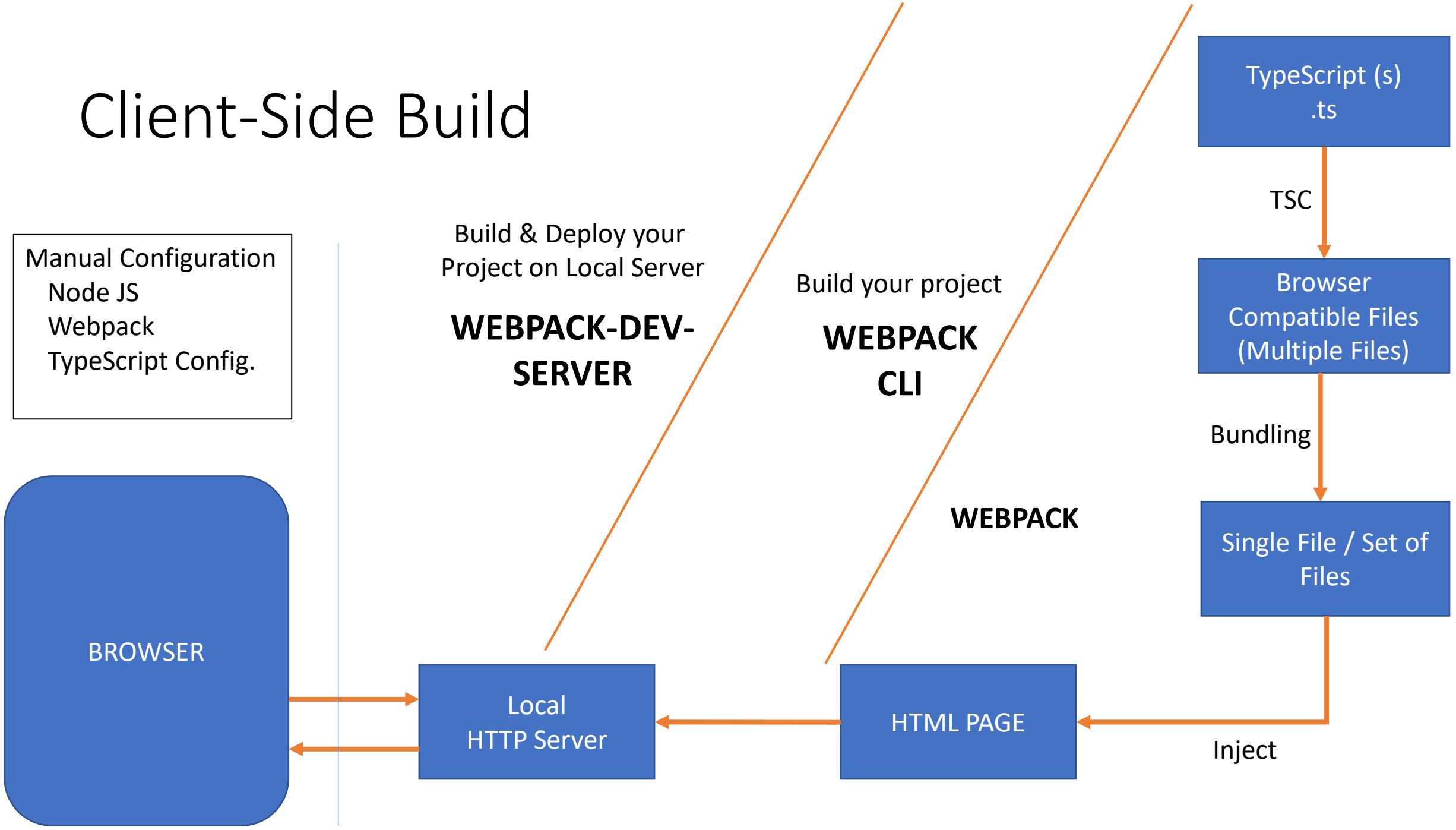
- Webpack executes in a multi-step process that involves analyzing dependencies, transforming code, and generating bundled output.
- The steps involved in Webpack's execution process:
  - Entry Point Resolution
  - Dependency Graph Creation
  - Module Loading and Transformation
  - Code Splitting
  - Module Resolution
  - Bundling and Output Generation
  - Output Files

Configuring Webpack

- The configuration steps for Webpack involve setting up the necessary configuration file to define how Webpack should behave and process your project's source code.
- Here are the main steps for configuring Webpack:
  - Create a Configuration File
  - Entry Points
  - Output Configuration
  - Loaders
  - Plugins
  - Resolve Configuration
  - Optimization Settings
  - Development vs. Production Mode
  - Additional Customizations


.ts files
Will be compiled by
tsc
Convert to
.js files
Refer in
.html page
Open in
BROWSER

Client-Side Build



Steps to run the webpack application

- Download and Extract the zip file.
- Open the extracted folder in Visual Studio Code
- Open the terminal window on the folder path and execute the following command
  - npm install
- After the installation completes, execute the following command
  - npm start– To start the development server
  - npm run build – To create the distributable folder with production build

Contact Me

Manish Sharma | Gmail

Manish Sharma | WhatsApp

Manish Sharma | Facebook

Manish Sharma | LinkedIn