Welcome

Introduction to TypeScript

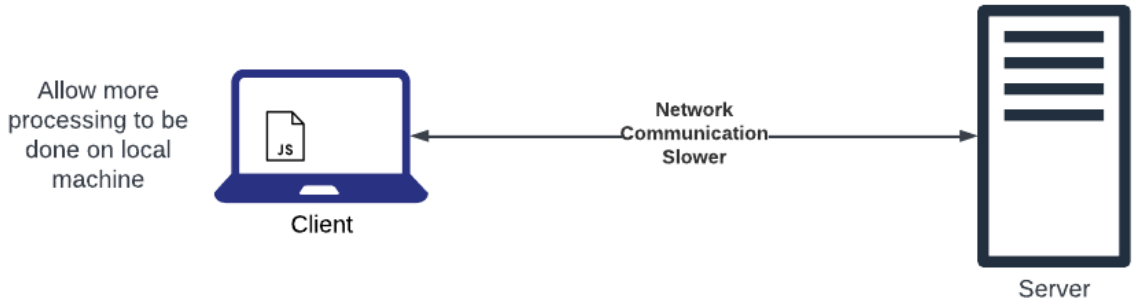
Why study this subject?

- Coding for the Web is a “staple” in modern software engineering
- JavaScript is one of the primary languages used to build Web apps, and it’s very flexible
- Sometimes that flexibility (and almost no constraint) can make building Web apps more difficult
- Utilizing a language like TypeScript can help bring proper balance between flexibility and order, ultimately helping to increase productivity

What is TypeScript?

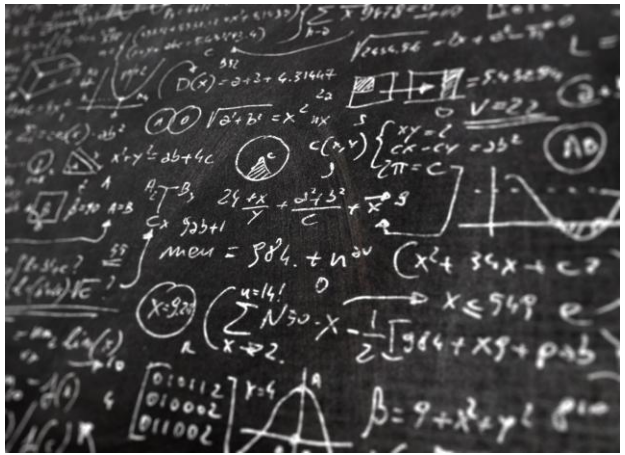
“Vanilla” JavaScript

- First released around 1995
- Initially created to fill need for client-side scripting language (since online speeds were slow)
- Integrated into every modern browser available today



“Vanilla” JavaScript

- Dynamically-typed (sometimes aka “weakly-typed” or “untyped”)
- Means that syntax rules and context associated to a type of data are not enforced
- Can translate to hard to track down errors in our code



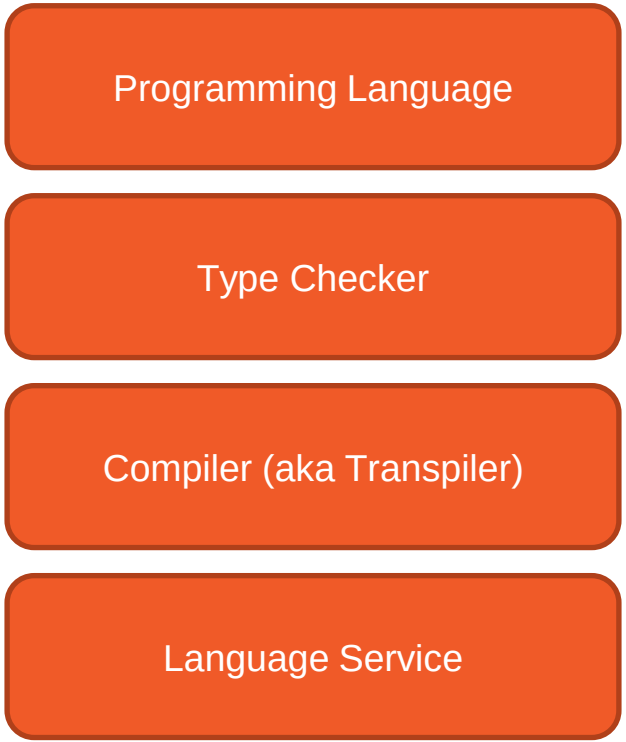
“Vanilla” JavaScript

- Very little (if any) enforcement of structure in your code
- In some cases, if JavaScript can’t understand the code as written, it will simply “take a best guess”
- Incredibly flexible but error-prone

Enter TypeScript!

- Programming Language
- Type Checker
- Compiler (aka Transpiler)
- Language Service

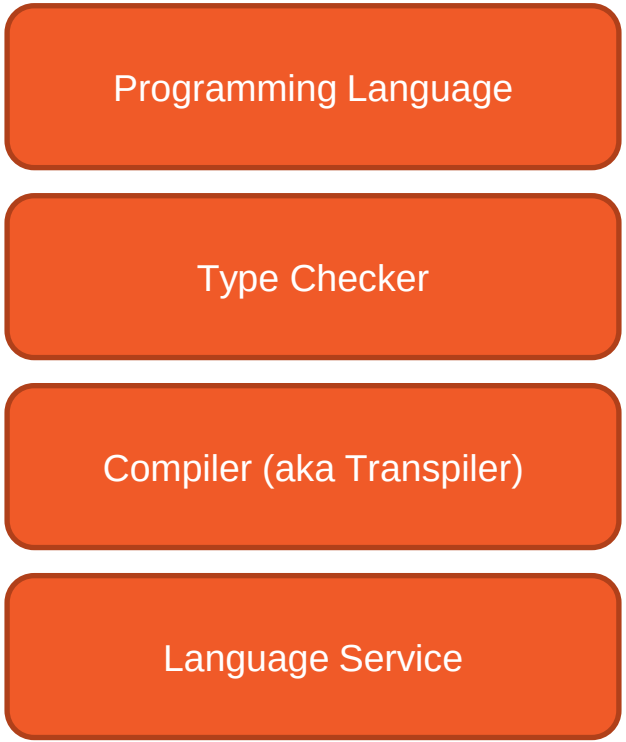
Enter TypeScript!



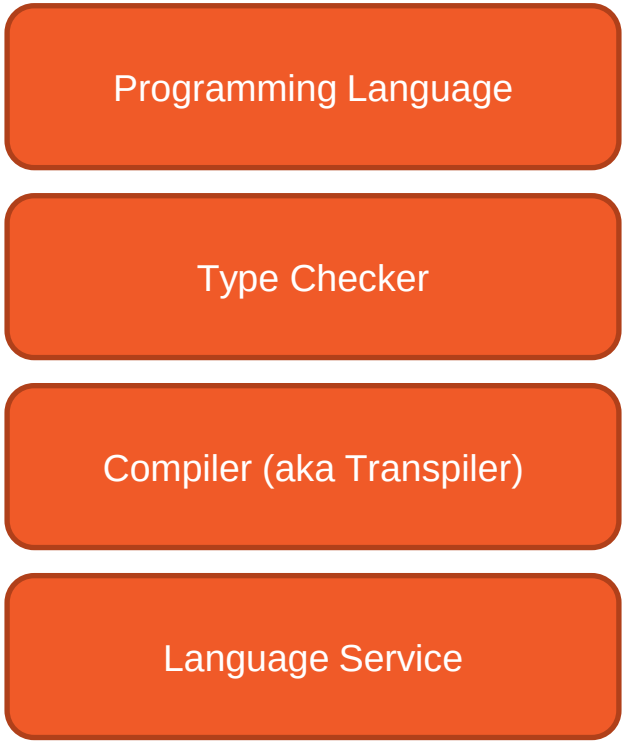
All existing JavaScript syntax
+ new TypeScript-specific
syntax

Enter TypeScript!

Allows association of strong typing to code enabling the enforcement of rules around syntax and structure

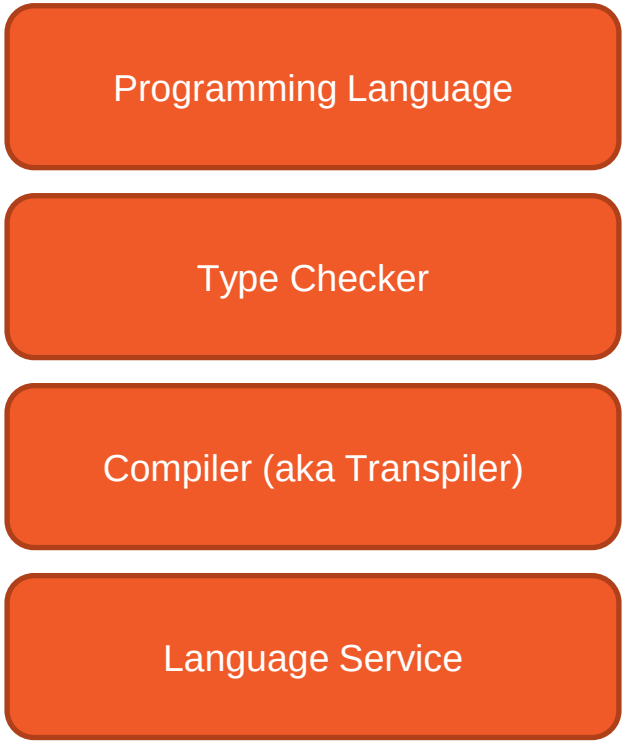


Enter TypeScript!



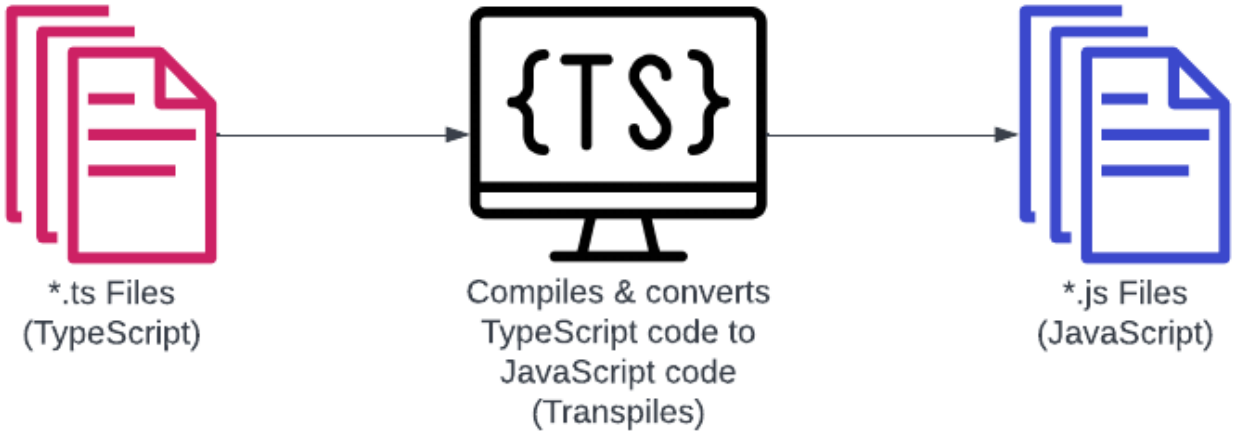
Runs build time checks of syntax and structure, raises errors if invalid, and converts the TypeScript code to JavaScript

Enter TypeScript!



Component that can be linked into dev tools to provide intellisense, syntax hints, and linting

Enter TypeScript!



Up and Running with TypeScript

Installing TypeScript

Install

```
npm i -g typescript
```

Verify

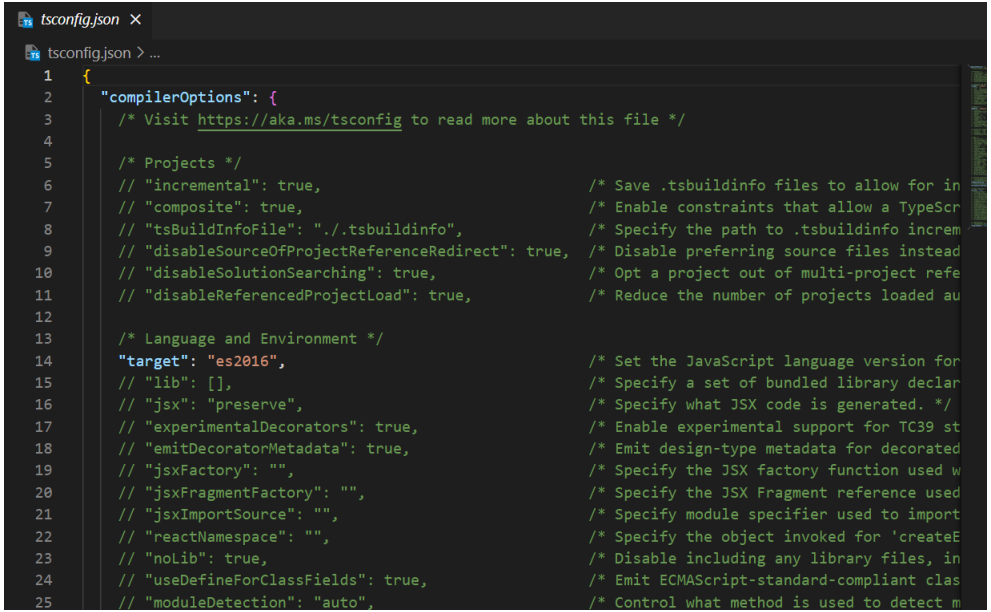
```
tsc --version
```

Installing TypeScript

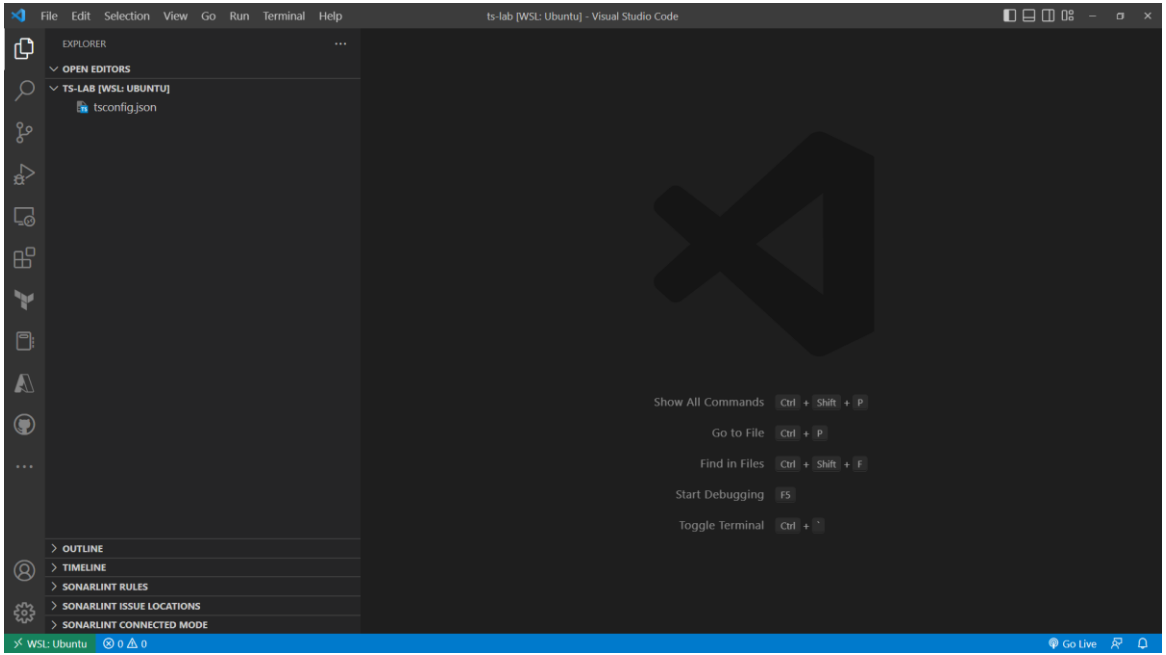
Navigate to folder and run

```
tsc --init
```

Generates & initializes config



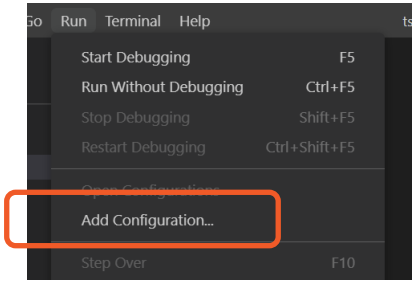
Visual Studio Code

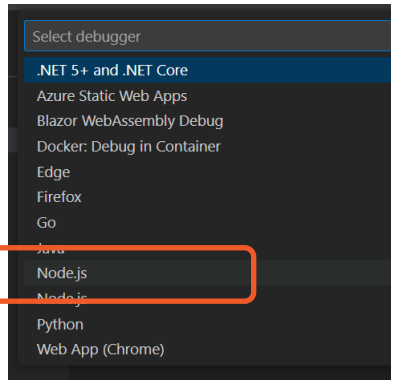


Cross-platform, open-source IDE that natively supports TypeScript (in fact, it was written in TypeScript - https://github.com/microsoft/vscode)

Debugging TypeScript in Visual Studio Code

Click VS Code menu Run | Add Configuration…

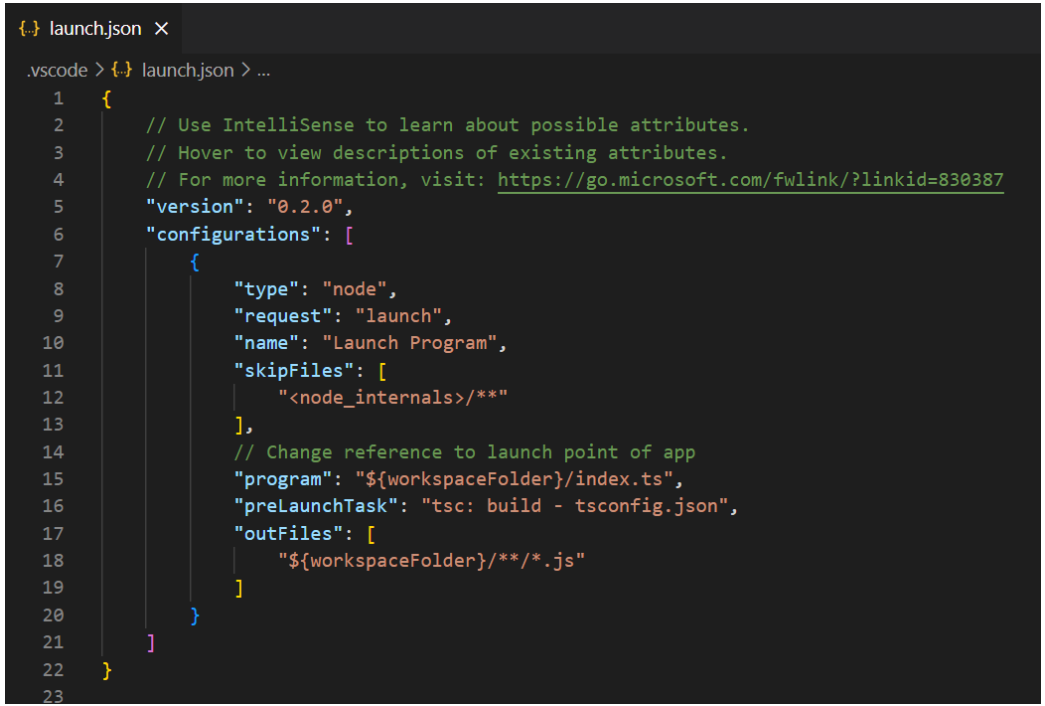




Choose Node.js as the debugger

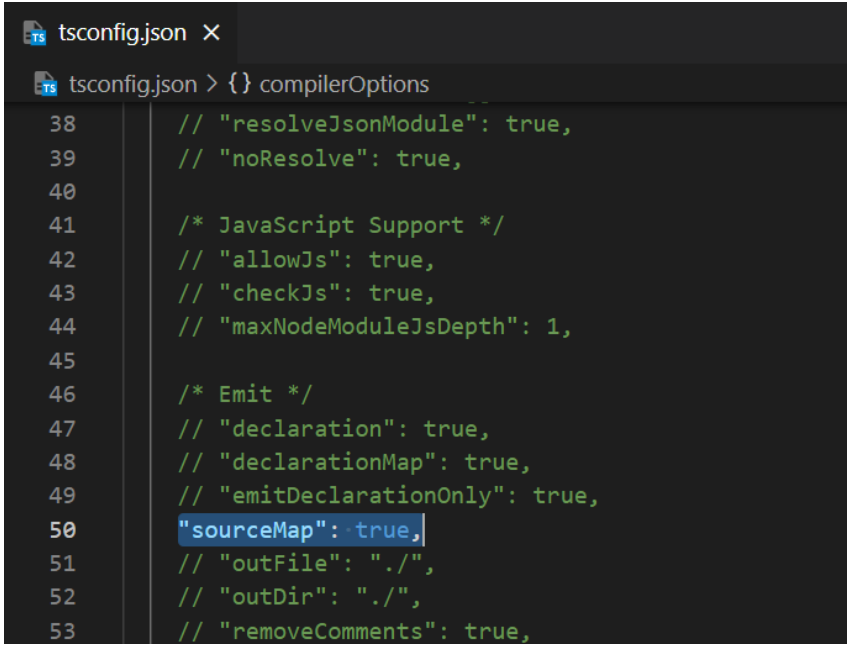
Debugging TypeScript in Visual Studio Code

Replace generated .vscode/launch.json with



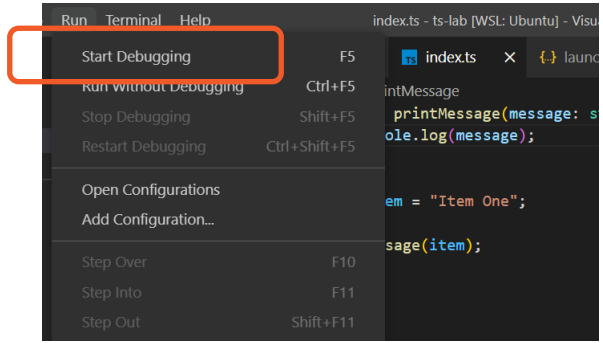
Debugging TypeScript in Visual Studio Code

Uncomment or add a “sourceMap”: true setting in your tsconfig.json file



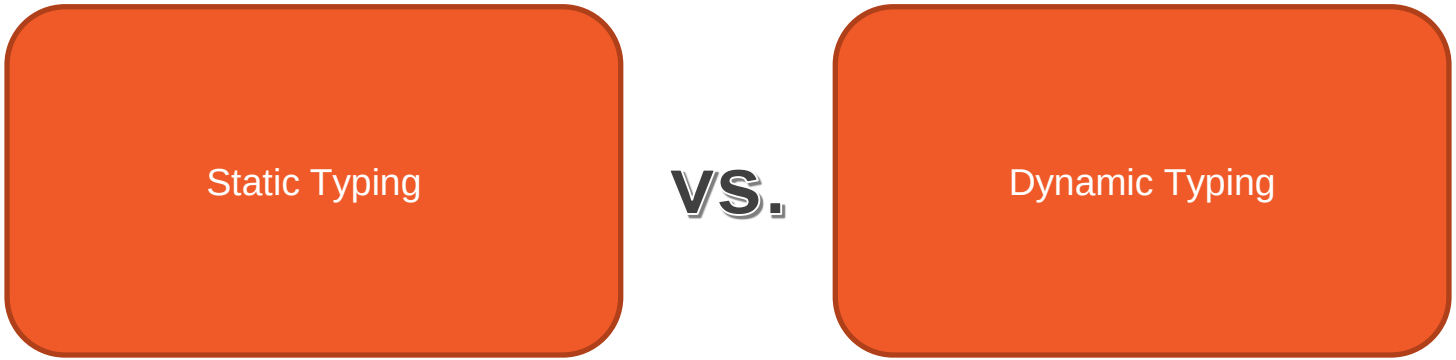
Debugging TypeScript in Visual Studio Code

You can then set a breakpoint in your code and click VS Code menu Run | Start Debugging



The “Type” in TypeScript

Static vs. Dynamic Typing



Type checks executed at compile time (prior to execution)

Type checks executed at runtime

Structural Typing

- TypeScript uses structural typing – types are matched based upon “shape”
- Values that match the shape of a type can be used as that type even if not explicitly defined as such

Structural Typing

index.ts > ...

```ts
type Animal = {
    species: string;
}

type Character = {
    name: string;
}

const poohBear = {
    species: "bear",
    name: "Winnie the Pooh",
};

function printSpecies(subject: Animal): void {
    console.log(subject.species);
}

function printName(subject: Character): void {
    console.log(subject.name);
}

function printBoth(subject: Animal & Character): void {
    console.log(`${subject.name} is a ${subject.species}`);
}

printSpecies(poohBear);
printName(poohBear);
printBoth(poohBear);

```

Structural Typing

In the resulting JavaScript, types are “erased”

Type System in TypeScript

OOTB (Out-of-the-Box) Types in TypeScript

- null
- undefined
- boolean
- string
- number
- bigint
- any

Specifying Types

- Type can be explicitly defined on variable declarations
- If not defined, TypeScript will infer the type

```
const var1 = "Hello, World!";
const var2 = 42.9;

console.log(typeof(var1) === "string");
console.log(typeof(var2) === "number");
console.log(typeof(var1) === "boolean");
```

```
true
true
false
```

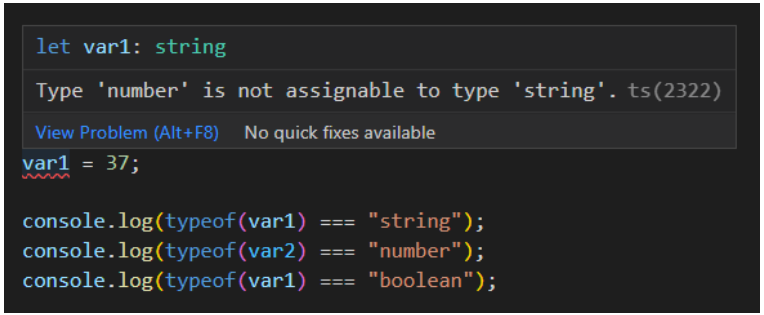
Assignability

TypeScript verifies what type of data can be assigned to a variable based on its declared type – if unable to complete assignment, results in an “assignability error”

```
let var1 = "Hello, World!";
const var2 = 42.9;

var1 = 37;

console.log(typeof(var1) === "string");
console.log(typeof(var2) === "number");
console.log(typeof(var1) === "boolean");
```



Evolving Any

The Any type can literally hold “any” type of value – if type not defined at variable declaration, any will be the default. Assignments of different values will cause the variable to take on the value’s type – evolving through the assignments as applied.

```
let var1;

var1 = "Hello, World!";
console.log(var1);
console.log(typeof(var1));
console.log(var1.toUpperCase());

var1 = 42.934233472888;
console.log(var1);
console.log(typeof(var1));
console.log(var1.toFixed(7));
// console.log(var1.toUpperCase()); // Not allowed
```

```
Hello, World!
string
HELLO, WORLD!
42.934233472888
number
42.9342335
```

Unions & Literals

Union Types

Defines an “either/or” designation for data type. Possible type options are separated using the | character. Often used with null & undefined types for a specific target data type which can also be null or undefined.

```
    let favoriteIceCream: string | undefined
let favoriteIceCream = Math.random() > 0.5
    ? undefined :
    "Chocolate";
console.log(favoriteIceCream);
```

TypeScript can automatically infer the union type (or we can define it explicitly).

```
let favoriteIceCream: string | undefined = Math.random() > 0.5
    ? undefined :
    "Chocolate";
console.log(favoriteIceCream);
```

Assignment Narrowing

Occurs when a variable is declared with a union type and is assigned an initial value. Variable will take on the exclusive type used for the assignment.

```
let meaningOfLife: number | string;

meaningOfLife = "Life, Libery, & the Pursuit of Happiness";
console.log(meaningOfLife.toUpperCase());
console.log(typeof meaningOfLife);

// meaningOfLife.toFixed(2);
```

Literal Types

Represent more specific versions of the primitive types – indicative of TypeScript's commitment to typing! Union types can include primitives & literals.

```
let meaningOfLife: 42 | string;

meaningOfLife = 42;
console.log(typeof meaningOfLife);
meaningOfLife = "Life, Libery, & the Pursuit of Happiness";
console.log(typeof meaningOfLife);

// meaningOfLife = 89;
```

Strict Null Checking

On of the options in tsconfig.json (“strictNullChecks”) which can be true or false. If disabled, silently adds “ | null | undefined” automatically to the types in your code. Best practice suggests enabling to take full advantage of TypeScript’s typing capabilities.

```
// "noImplicitAny": true,
"strictNullChecks": false,
// "strictFunctionTypes": true,
```

```
let favoriteIceCream = Math.random() > 0.5
    ? undefined :
    "Chocolate";

console.log(favoriteIceCream.toLowerCase());
```

```
TypeError: Cannot read properties of undefined (reading 'toLowerCase')
    at Object.<anonymous> (/home/sysadmin/source/repos/ts-lab/unions/index.js:4:30)
    at Module._compile (node:internal/modules/cjs/loader:1159:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1213:10)
    at Module.load (node:internal/modules/cjs/loader:1037:32)
    at Module._load (node:internal/modules/cjs/loader:878:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:23:47
```

or

```
chocolate
```

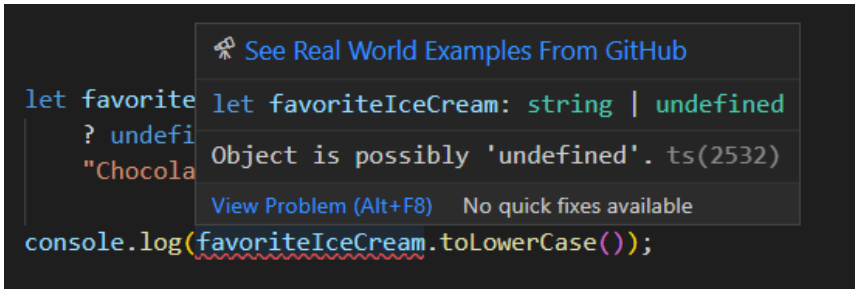
Not caught until runtime

Strict Null Checking

```
// "noImplicitAny": true,
"strictNullChecks": true,
// "strictFunctionTypes": true,
```

```
let favoriteIceCream = Math.random() > 0.5
    ? undefined :
    "Chocolate";

console.log(favoriteIceCream.toLowerCase());
```



Caught at compile time

Strict Null Checking

```
let favoriteIceCream = Math.random() > 0.5
    ? undefined :
    "Chocolate";

console.log(favoriteIceCream?.toLowerCase());
```

? protects against calling a function on an undefined reference

Objects in TypeScript

Aliased Object Types

Can use a type alias that “maps” to an object definition (a set of property names and associated types). The type alias can then be used like any other type in your code.

```
type Movie = {
    title: string;
    director: string;
    yearReleased: number;
}

let theGodfather: Movie;

theGodfather = {
    title: "The Godfather",
    director: "Francis Ford Coppola",
    yearReleased: 1972
};

console.log(theGodfather);
console.log(theGodfather.title);
console.log(theGodfather["director"]);
console.log(theGodfather.yearReleased);
```

```
{
  title: 'The Godfather',
  director: 'Francis Ford Coppola',
  yearReleased: 1972
}
The Godfather
Francis Ford Coppola
1972
```

Nested Object Types

Object types can be nested to create more complex hierarchical relationships.

```ts
type Movie = {
    title: string;
    director: {
        firstName: string;
        middleName: string;
        lastName: string;
    };
    yearReleased: number;
}

let theGodfather: Movie;

theGodfather = {
    title: "The Godfather",
    director: {
        firstName: "Francis",
        middleName: "Ford",
        lastName: "Coppola"
    },
    yearReleased: 1972
};

console.log(theGodfather);
console.log(theGodfather.title);
console.log(theGodfather["director"]);
console.log(`${theGodfather.director.firstName} ${theGodfather.director.middleName} ${theGodfather.director.lastName}`)
console.log(theGodfather.yearReleased);
```

```
{
  title: 'The Godfather',
  director: { firstName: 'Francis', middleName: 'Ford', lastName: 'Coppola' },
  yearReleased: 1972
}
The Godfather
{ firstName: 'Francis', middleName: 'Ford', lastName: 'Coppola' }
Francis Ford Coppola
1972
```

Nested Object Types

Nested structures can include other layers of aliased type as well.

```
type Director = {
    firstName: string;
    middleName: string;
    lastName: string;
}

type Movie = {
    title: string;
    director: Director;
    yearReleased: number;
}

let theGodfather: Movie;

theGodfather = {
    title: "The Godfather",
    director: {
        firstName: "Francis",
        middleName: "Ford",
        lastName: "Coppola"
    },
    yearReleased: 1972
};

console.log(theGodfather);
console.log(theGodfather.title);
console.log(theGodfather["director"]);
console.log(`${theGodfather.director.firstName} ${theGodfather.director.middleName} ${theGodfather.director.lastName}`)
console.log(theGodfather.yearReleased);
```

Optional Properties

Objects can designate properties as optional by appending a ? to the property name.

```
type Director = {
    firstName: string;
    middleName?: string;
    lastName: string;
}

type Movie = {
    title: string;
    director: Director;
    yearReleased: number;
}

let theGodfather: Movie;

theGodfather = {
    title: "The Godfather",
    director: {
        firstName: "Francis",
        lastName: "Coppola"
    },
    yearReleased: 1972
};

console.log(theGodfather);
console.log(theGodfather.title);
console.log(theGodfather["director"]);
console.log(`${theGodfather.director.firstName} ${theGodfather.director.middleName ? " " + theGodfather.director.middleName : ""}${theGodfather.director.lastName}`)
console.log(theGodfather.yearReleased);
```

Functions in TypeScript

Function Parameters

When defining functions, you can include 1 or more required parameters (function inputs).

```
function formatName(firstName: string, lastName: string) {
    console.log(`${lastName}, ${firstName}`);
}

formatName("Darth", "Vader");
formatName("Vito", "Corleone");
// formatName("Cher");
// formatName("Francis", "Ford", "Coppola");
```

Function Parameters

When defining functions, parameters can be made optional by appending a ? to the end of the parameter name. Optional parameters must appear last in the function signature.

```
function formatName(firstName: string, lastName: string, middleName?: string) {
    if (middleName) {
        console.log(`${lastName}, ${firstName} ${middleName}`);
    } else {
        console.log(`${lastName}, ${firstName}`);
    }
}

formatName("Darth", "Vader");
formatName("Vito", "Corleone");
// formatName("Cher");
formatName("Francis", "Coppola", "Ford");
```

Function Parameters

You can also assign default values to function parameters as part of the function signature. You can define types for the default parameters – however, TypeScript is able to infer parameter type from the value assigned.

```
function formatName(firstName: string, lastName: string, middleName = "<empty>") {
    console.log(`${lastName}, ${firstName} ${middleName}`);
}

formatName("Darth", "Vader");
formatName("Vito", "Corleone");
// formatName("Cher");
formatName("Francis", "Coppola", "Ford");
```

Function Parameters

For some parameters, you may want to leverage the *spread* operator (…) to indicate that the function can take 0 or more arguments as inputs. Parameter type will be an array of another type to represent a potential collection of inputs of that type.

```
function formatName(...names: string[]) {
        console.log("Here are the names provided:");
        console.log(names.join(" "));
}

formatName("Darth", "Vader");
formatName("Vito", "Corleone");
formatName("Cher");
formatName("Francis", "Coppola", "Ford");
formatName();
```

Function Return Types

TypeScript can infer the return type for a function from the type(s) of data returned from it. This includes inferring a union type if multiple types are returned.

```
function firstFunction() {
    return "Simple String";
}
          function secondFunction(): false | 4.33
function secondFunction() {
    return Math.random() > 0.5 ?
        false : 4.33;
}

const result1 = firstFunction();
console.log(typeof result1);
const result2 = secondFunction();
console.log(typeof result2);
```

Function Return Types

You can also explicitly define the return type for a given function.

```
function firstFunction(): string {
    return "Simple String";
}

function secondFunction(): boolean | number {
    return Math.random() > 0.5 ?
        false : 4.33;
}

const result1 = firstFunction();
console.log(typeof result1);
const result2 = secondFunction();
console.log(typeof result2);
```

Function Types

As in JavaScript, TypeScript allows you to reference functions as variables and pass them as values. Those values will have a specific type (the function type or defined signature). Useful in cases where you want to “inject” different strategies at runtime all possessing a common shape.

```
const names = [
    { firstName: "Darth", lastName: "Vader" },
    { firstName: "Vito", lastName: "Corleone" },
    { firstName: "Francis", middleName: "Ford", lastName: "Corleone" },
];

function displayNames(nameStrategy: (firstName: string, lastName: string, middleName?: string) => void) {
    for (let counter = 0; counter < names.length; counter++) {
        nameStrategy(names[counter].firstName, names[counter].lastName, names[counter].middleName);
    }
}

function formatName(firstName: string, lastName: string, middleName?: string) {
    if (middleName) {
        console.log(`${lastName}, ${firstName} ${middleName}`);
    } else {
        console.log(`${lastName}, ${firstName}`);
    }
}

function formatNameSpecial(firstName: string, lastName: string, middleName?: string) {
    if (middleName) {
        console.log(`${firstName.toUpperCase()} ${middleName.toUpperCase()} ${lastName.toUpperCase()}`);
    } else {
        console.log(`${firstName.toUpperCase()} ${lastName.toUpperCase()}`);
    }
}

displayNames(formatName);
displayNames(formatNameSpecial);
```

Arrays in TypeScript

Array Types

Arrays in TypeScript function similarly in syntax to arrays in JavaScript. The difference is really in the typing – TypeScript expects an array to have a defined type representing the types of values that will be stored in it. This type can be inferred from the values the array is initialized with and it can include a union type.

```
type Director = {
    firstName: string;
    middleName?: string;
    lastName: string;
}

type Movie = {
    title: string;
    director: Director;
    yearReleased: number;
}

const movies: Movie[] = [
    { title: "The Godfather", director: { firstName: "Francis", middleName: "Ford", lastName: "Coppola" }, yearReleased: 1972 },
    { title: "Star Wars", director: { firstName: "George", lastName: "Lucas" }, yearReleased: 1977 },
]

movies.push({ title: "Aliens", director: { firstName: "James", lastName: "Cameron" }, yearReleased: 1986});

console.log(movies[movies.length - 1]);
for (let counter = 0; counter < movies.length; counter++) {
    console.log(movies[counter]);
}
```

The Spread Operator (…)

The spread operator (…) can be used to combine arrays or add the elements of one array to another. We've already seen the spread operator used with function parameters to designate that the parameter accepts 0 or more of the type for that parameter (uses an array).

```
type Director = {
    firstName: string;
    middleName?: string;
    lastName: string;
}

type Movie = {
    title: string;
    director: Director;
    yearReleased: number;
}

const movieSetOne: Movie[] = [
    { title: "The Godfather", director: { firstName: "Francis", middleName: "Ford", lastName: "Coppola" }, yearReleased: 1972 },
    { title: "Star Wars", director: { firstName: "George", lastName: "Lucas" }, yearReleased: 1977 },
]

const movieSetTwo: Movie[] = [
    { title: "Aliens", director: { firstName: "James", lastName: "Cameron" }, yearReleased: 1986 },
]

const movieSetThree: Movie[] = [
    { title: "Spider-Man: No Way Home", director: { firstName: "Jon", lastName: "Watts" }, yearReleased: 2021 },
]

let movies = [...movieSetOne, ...movieSetTwo];
movies.push(...movieSetThree);

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

Interfaces in TypeScript

Interfaces

When discussing objects, we looked at type aliases. Separately, TypeScript includes the concept of an *interface* as well. Syntax and functionality between a type alias and an interface will look similar.

```
interface Director {
    firstName: string;
    middleName?: string;
    lastName: string;
}

interface Movie {
    title: string;
    director: Director;
    yearReleased: number;
}

let theGodfather: Movie;

theGodfather = {
    title: "The Godfather",
    director: {
        firstName: "Francis",
        lastName: "Coppola"
    },
    yearReleased: 1972
};

console.log(theGodfather);
console.log(theGodfather.title);
console.log(theGodfather["director"]);
console.log(`${theGodfather.director.firstName} ${theGodfather.director.middleName ? " " + theGodfather.director.middleName : ""}${theGodfather.director.lastName}`);
console.log(theGodfather.yearReleased);
```

Interfaces

There are some differences and reasons for preferring an interface over a type alias. Some of those reasons include:

Interfaces can be “merged” together to augment object shape

Interfaces work well with classes (which we’ll talk about next) and can be used to enforce the structure of class declarations

Generally, work faster with TypeScript’s type checker

Can result is easier to read and interpret error messages (clearer detail)

Interfaces

Interfaces support optional properties and read-only properties.

```
interface Director {
    readonly id: number;
    firstName: string;
    middleName?: string;
    lastName: string;
}

const ff: Director = {
    id: 1,
    firstName: "Francis",
    middleName: "Ford",
    lastName: "Coppola",
};

const gl: Director = {
    id: 2,
    firstName: "George",
    lastName: "Lucas",
}

console.log(ff);
console.log(gl);
gl.firstName = "GDog";
// gl.id = 3;
```

Functions & Methods

There are 2 ways to define functions in an interface – using method syntax or property syntax. Function references also support optional designations as well.

```
interface Director {
    readonly id: number;
    firstName: string;
    middleName?: string;
    lastName: string;
    getFullNamePs?: () => string;
    getFullNameMs(): string;
}

function getFullName(firstName: string, lastName: string, middleName?: string): string {
    return `${firstName}${middleName ? " " + middleName : ""} ${lastName}`;
}

const ff: Director = {
    id: 1,
    firstName: "Francis",
    middleName: "Ford",
    lastName: "Coppola",
    getFullNamePs: () => {
        return getFullName(ff.firstName, ff.lastName, ff.middleName);
    },
    getFullNameMs() {
        return getFullName(ff.firstName, ff.lastName, ff.middleName);
    },
};

const gl: Director = {
    id: 2,
    firstName: "George",
    lastName: "Lucas",
    getFullNameMs() {
        return getFullName(gl.firstName, gl.lastName);
    },
}

console.log(ff);
console.log(ff.getFullNamePs ? ff.getFullNamePs() : "Not Provided");
console.log(ff.getFullNameMs());
console.log(gl.getFullNamePs ? gl.getFullNamePs() : "Not Provided");
console.log(gl.getFullNameMs());
```

Classes in TypeScript

Class Definition & Usage

```typescript
interface Learner {
    name: string;
    study(hours: number): void;
}

class Student implements Learner {
    name: string;

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

    study(hours: number) {
        for (let i = 0; i < hours; i++) {
            console.log(`${this.name} is studying`);
        }
    }
}
```

```typescript
class RegularJoe implements Learner {
    name: string;

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

    study(hours: number) {
        for (let i = 0; i < hours * 2; i++) {
            console.log(`${this.name} is really studying`);
        }
    }
}

function processLearner(learner: Learner, numHours: number) {
    learner.study(numHours);
}

processLearner(new Student("Bob Roberts"), 5);
processLearner(new RegularJoe("Joe Schmoe"), 5);
```

Classes can implement multiple interfaces as well

Extending Classes

You can create an inheritance hierarchy by extending one class from another. The parent class's properties and functions are available to the child class (depending on visibility – more to come in a sec). You can also override the parent class's constructor and functions.

```
class Employee {
    name: string;
    badgeNumber: string;

    constructor(badgeNumber: string, name: string) {
        this.badgeNumber = badgeNumber;
        this.name = name;
    }

    calculatePay(payRate: number, numHours: number) {
        return payRate * numHours;
    }
}

class SalariedEmployee extends Employee {
    weeklyBonus: number;

    constructor(badgeNumber: string, name: string, weeklyBonus: number) {
        super(badgeNumber, name);
        this.weeklyBonus = weeklyBonus;
    }

    calculatePay(payRate: number): number {
        return (payRate / 52) + this.weeklyBonus;
    }

    toString(): string {
        return `${this.name} (${this.badgeNumber}) with weekly bonus of ${this.weeklyBonus}`;
    }
}

const salEmployee = new SalariedEmployee("00001", "Joe Schmoe", 100);
console.log(salEmployee.toString());
console.log(`Weekly pay is ${salEmployee.calculatePay(50000).toFixed(2)}`);
```

Generics in TypeScript

Generics

- Generics in TypeScript allow the definition of code that still uses static typing but offers flexibility in the type that will be used
- *Type parameters* act as placeholders for a specific type
- When an instance of the generic construct is created, a *type argument* (specific type) can be substituted for the placeholder, and the instance takes on that type for its lifetime
- Generics are supported in type aliases, functions, interfaces, classes, and class members
- Type parameters can be set to a default type, and they can be constrained, ensuring that the type substituted meets a set of defined criteria

Generics

```
class SimpleDictionary<Key, Value = Key> {
    key: Key;
    value: Value;

    constructor(key: Key, value: Value) {
        this.key = key;
        this.value = value;
    }

    getValue(key: Key): Value | undefined {
        return this.key === key
            ? this.value
            : undefined;
    }
}

const games: SimpleDictionary<number, string>[] = [
    new SimpleDictionary(1000, "Monopoly"),
    new SimpleDictionary(2000, "Checkers"),
    new SimpleDictionary(3000, "Chess"),
    new SimpleDictionary(4000, "Splendor"),
]

const states: SimpleDictionary<string>[] = [
    new SimpleDictionary("OH", "Ohio"),
    new SimpleDictionary("CA", "California"),
    new SimpleDictionary("NY", "New York"),
]
```

```
function findAThing<Key, Value>(key: Key, collection: SimpleDictionary<Key, Value>[]): Value | undefined {
    for (let c = 0; c < collection.length; c++) {
        const val = collection[c].getValue(key as Key);
        if (val) {
            return val;
        }
    }
    return undefined;
}

console.log(findAThing(1000, games));
console.log(findAThing(4000, games));
console.log(findAThing(5000, games));

console.log(findAThing("OH", states));
console.log(findAThing("CA", states));
console.log(findAThing("MI", states));
```

Implementing Generic Interface

```
interface Faculty<Subject> {
    name: string;
    subject: Subject;
}

class TeachingPosition implements Faculty<string> {
    name: string;
    subject: string;
    isTenured: boolean;

    constructor(name: string, subject: string, isTenured: boolean) {
        this.name = name;
        this.subject = subject;
        this.isTenured = isTenured;
    }

    toString(): string {
        return `${this.name} teaches ${this.subject} and ${this.isTenured ? "is tenured" : "is not yet tenured"}`;
    }
}

const teachingJob1 = new TeachingPosition("Joe Schmoe", "Applied Mathematics", true);
const teachingJob2 = new TeachingPosition("Bob Roberts", "Basket Weaving", false);

console.log(teachingJob1.toString())
console.log(teachingJob2.toString())
```

Thank you!

If you have additional questions, please reach out to me at:
asanders@gamuttechnologysvcs.com

DevelopIntelligence
A PLURALSIGHT COMPANY