



Welcome

Introduction to TypeScript

 **Develop**Intelligence

A PLURALSIGHT COMPANY

Hello

HELLO
my name is

Allen Sanders
with DevelopIntelligence,
a Pluralsight Company.

About me...



- 25+ years in the industry
- 20+ years in teaching
- Certified Cloud architect
- Passionate about learning
- Also, passionate about Reese's Cups!



Prerequisites

This course assumes you:

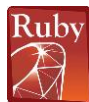
- Have experience writing software using JavaScript
- Have working knowledge of npm and Node.js
- Some degree of experience with or knowledge of Object-Oriented Programming (OOP)



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 actually 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

We teach over 400 technology topics



You experience our impact on a daily basis!





My pledge to you

I will...

- Make this interactive
- Ask you questions
- Ensure everyone can speak
- Use an on-screen timer



Objectives

At the end of this course you will be able to:

- Leverage core TypeScript concepts to write web-browser and server applications
- Understand the similarities and differences between the TypeScript and JavaScript types and typing systems
- Think in terms of structural typing
- Use TypeScript features to write Classical Object-Oriented Code



How we're going to work together

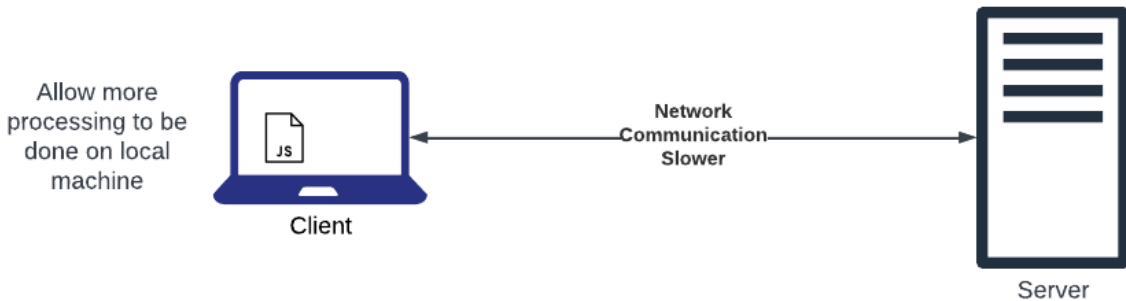
- Lab work will take place on your workstations using configuration identified in the tech setup for the class
- There will be some concepts summarized via slides
- Significant focus of the class will be on hands on/lab work – we'll use that lab work to reinforce key topics
- I welcome being interrupted – if you need more info, or clarification, or anything else, just break in and ask. I am here to help you.

Getting Started

What is TypeScript?

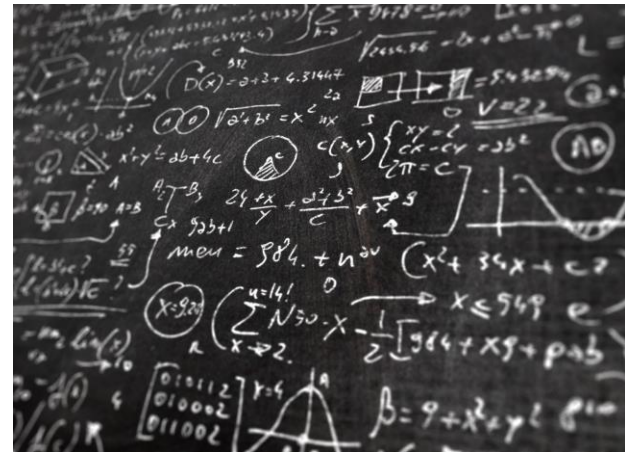
“Vanilla” JavaScript

- 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!

Programming Language

All existing JavaScript syntax
+ new TypeScript-specific
syntax

Type Checker

Compiler (aka Transpiler)

Language Service

Enter TypeScript!

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

Programming Language

Type Checker

Compiler (aka Transpiler)

Language Service

Enter TypeScript!

Programming Language

Type Checker

Compiler (aka Transpiler)

Language Service

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

Enter TypeScript!

Programming Language

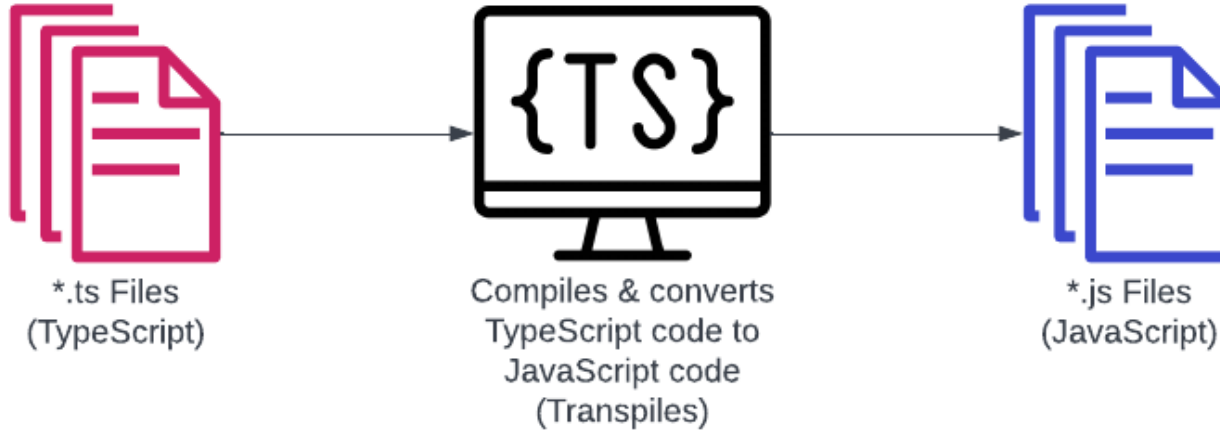
Type Checker

Compiler (aka Transpiler)

Language Service

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

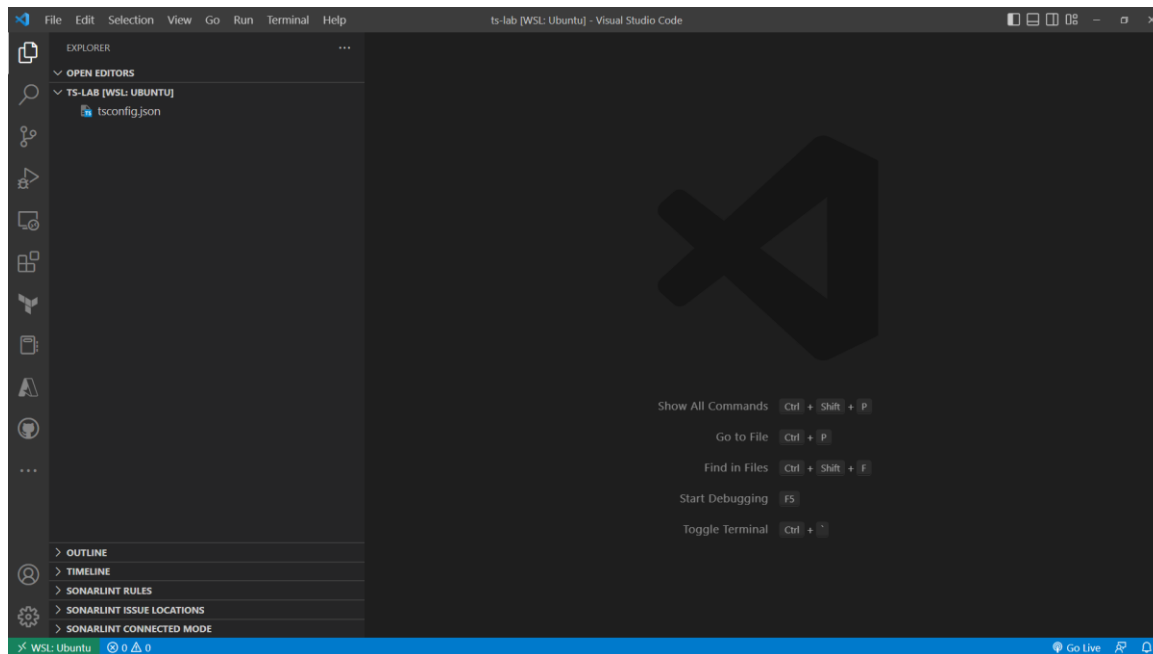
```
tsc --init
```



Generates &
initializes config

```
tsconfig.json X
tsconfig.json > ...
1  {
2    "compilerOptions": {
3      /* Visit https://aka.ms/tsconfig to read more about this file */
4
5      /* Projects */
6      // "incremental": true,           /* Save .tsbuildinfo files to allow for in
7      // "composite": true,           /* Enable constraints that allow a TypeScr
8      // "tsBuildInfoFile": "./.tsbuildinfo", /* Specify the path to .tsbuildinfo increm
9      // "disableSourceOfProjectReferenceRedirect": true, /* Disable preferring source files instead
10     // "disableSolutionSearching": true, /* Opt a project out of multi-project refe
11     // "disableReferencedProjectLoad": true, /* Reduce the number of projects loaded au
12
13     /* Language and Environment */
14     "target": "es2016",               /* Set the JavaScript language version for
15     // "lib": [],                     /* Specify a set of bundled library declar
16     // "jsx": "preserve",             /* Specify what JSX code is generated. */
17     // "experimentalDecorators": true, /* Enable experimental support for TC39 st
18     // "emitDecoratorMetadata": true, /* Emit design-type metadata for decorated
19     // "jsxFactory": "",              /* Specify the JSX factory function used w
20     // "jsxFragmentFactory": "",      /* Specify the JSX Fragment reference used
21     // "jsxImportSource": "",         /* Specify module specifier used to import
22     // "reactNamespace": "",          /* Specify the object invoked for 'createE
23     // "noLib": true,                 /* Disable including any library files, in
24     // "useDefineForClassFields": true, /* Emit ECMAScript-standard-compliant clas
25     // "moduleDetection": "auto",     /* Control what method is used to detect m
```

Visual Studio Code



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

Linting with ESLint

Install

```
npm install eslint --save-dev
```

Local to
project
(execute in
project root)

From
project root
run

```
./node_modules/.bin/eslint --init
```

This will prompt you for different options used to configure a
.eslintrc.json file

Linting with ESLint

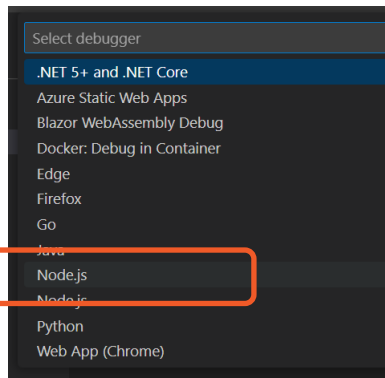
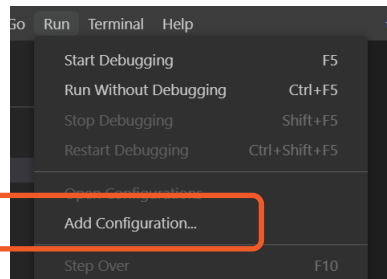
The generated
.eslintrc.json can
be modified to
configure rules,
language
options, etc.

```
.eslintrc.json X
.eslintrc.json > ...
1  {
2    "env": {
3      "browser": true,
4      "es2021": true
5    },
6    "extends": [
7      "eslint:recommended",
8      "plugin:react/recommended",
9      "plugin:@typescript-eslint/recommended"
10   ],
11   "overrides": [
12   ],
13   "parser": "@typescript-eslint/parser",
14   "parserOptions": {
15     "ecmaVersion": "latest",
16     "sourceType": "module"
17   },
18   "plugins": [
19     "react",
20     "@typescript-eslint"
21   ],
22   "rules": {
23   }
24 }
25
```

See <https://eslint.org/docs/latest/user-guide/configuring/> for additional detail on available options

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

```
{.} launch.json X
.vscode > {.} launch.json > ...
1  {
2      // Use IntelliSense to learn about possible attributes.
3      // Hover to view descriptions of existing attributes.
4      // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
5      "version": "0.2.0",
6      "configurations": [
7          {
8              "type": "node",
9              "request": "launch",
10             "name": "Launch Program",
11             "skipFiles": [
12                 "<node_internals>/**"
13             ],
14             // Change reference to launch point of app
15             "program": "${workspaceFolder}/index.ts",
16             "preLaunchTask": "tsc: build - tsconfig.json",
17             "outFiles": [
18                 "${workspaceFolder}/**/*.js"
19             ]
20         }
21     ]
22 }
23
```

Debugging TypeScript in Visual Studio Code

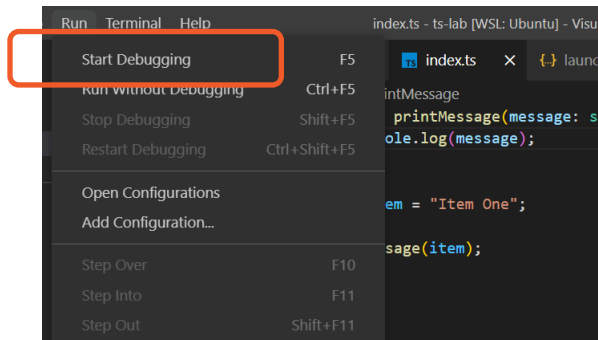
Uncomment or add a
"sourceMap": true
setting in your
tsconfig.json file



```
tsconfig.json x
tsconfig.json > {} compilerOptions
38 // "resolveJsonModule": true,
39 // "noResolve": true,
40
41 /* JavaScript Support */
42 // "allowJs": true,
43 // "checkJs": true,
44 // "maxNodeModuleJsDepth": 1,
45
46 /* Emit */
47 // "declaration": true,
48 // "declarationMap": true,
49 // "emitDeclarationOnly": true,
50 "sourceMap": true,
51 // "outFile": "./",
52 // "outDir": "./",
53 // "removeComments": true,
```

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

Static Typing

Type checks executed at
compile time (prior to
execution)

VS.

Dynamic Typing

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 > ...
1  type Animal = {
2    species: string;
3  }
4
5  type Character = {
6    name: string;
7  }
8
9  const poohBear = {
10    species: "bear",
11    name: "Winnie the Pooh",
12  };
13
14  function printSpecies(subject: Animal): void {
15    console.log(subject.species);
16  }
17
18  function printName(subject: Character): void {
19    console.log(subject.name);
20  }
21
22  function printBoth(subject: Animal & Character): void {
23    console.log(`${subject.name} is a ${subject.species}`);
24  }
25
26  printSpecies(poohBear);
27  printName(poohBear);
28  printBoth(poohBear);
29
```

Structural Typing

In the resulting
JavaScript, types
are “erased”

```
JS index.js 1 X
JS index.js > ...
1  var poohBear = {
2    species: "bear",
3    name: "Winnie the Pooh"
4  };
5  function printSpecies(subject) {
6    console.log(subject.species);
7  }
8  function printName(subject) {
9    console.log(subject.name);
10 }
11 function printBoth(subject) {
12   console.log(`${subject.name} is a ${subject.species}`);
13 }
14 printSpecies(poohBear);
15 printName(poohBear);
16 printBoth(poohBear);
17
```

LAB 01

From JavaScript to TypeScript

Execute lab 01 available in GitHub at <https://github.com/KernelGamut32/intro-to-typescript-public/tree/main/labs/lab01>.

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");
```

```
let var1: string  
Type 'number' is not assignable to type 'string'. ts(2322)  
View Problem \(Alt+F8\) No quick fixes available  
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
```

LAB 02

The Type System

Execute lab 02 available in GitHub at <https://github.com/KernelGamut32/intro-to-typescript-public/tree/main/labs/lab02>.

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, Liberty, & 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, Liberty, & the Pursuit of Happiness";
console.log(typeof meaningOfLife);

// meaningOfLife = 89;
```

Strict Null Checking

One 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,  
// "strictBindCallApply": 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,  
// "strictBindCallApply": true
```

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

```
let favoriteIceCream: string | undefined  
  ? undefined  
  "Chocolate";  
  
console.log(favoriteIceCream.toLowerCase());
```

See Real World Examples From GitHub

let favoriteIceCream: string | undefined

Object is possibly 'undefined'. ts(2532)

View Problem (Alt+F8) No quick fixes available

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

LAB 03

Unions & Literals

Execute lab 03 available in GitHub at <https://github.com/KernelGamut32/intro-to-typescript-public/tree/main/labs/lab03>.

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.

```
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);
```

Discriminated Unions

Includes a property on the object (e.g., “type”) which clearly indicates the shape of the underlying object. Can be used in conditional checks to branch along different routes (as needed).

```
type Vegetable = {  
  name: string;  
  nutritionalValue: number;  
  type: "nutrition";  
};  
  
type IceCream = {  
  name: string;  
  flavor: string;  
  type: "fun";  
};  
  
type Food = Vegetable | IceCream;  
  
const food: Food = Math.random() > 0.5  
  ? { name: "Broccoli", nutritionalValue: 35.5, type: "nutrition" }  
  : { name: "Happiness", flavor: "Chocolate", type: "fun" };  
  
if (food.type === "nutrition") {  
  console.log(`Booooo! It's ${food.name} with a nutritional value of ${food.nutritionalValue}`);  
} else {  
  console.log(`Yay!!! I'm having ${food.name} which is ${food.flavor} ice cream!`);  
}
```


Intersection Types

Allow for the declaration of a new type which combines multiple other types (more of an “AND” rather than an “OR” with unions).

```
type ShortPoem = { author: string } & (  
  | { kigo: string; type: "haiku"; }  
  | { meter: number; type: "villanelle"; }  
);
```

```
const morningGlory: ShortPoem = {  
  author: "Fukuda Chiyo-ni",  
  kigo: "Morning Glory",  
  type: "haiku",  
};
```

```
const oneArt: ShortPoem = {  
  author: "Elizabeth Bishop",  
  meter: 12,  
  type: "villanelle",  
};
```

```
console.log(morningGlory);  
console.log(oneArt);
```

```
{ author: 'Fukuda Chiyo-ni', kigo: 'Morning Glory', type: 'haiku' }  
{ author: 'Elizabeth Bishop', meter: 12, type: 'villanelle' }
```

LAB 04

Unions & Literals

Execute lab 04 available in GitHub at <https://github.com/KernelGamut32/intro-to-typescript-public/tree/main/labs/lab04>.

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  
{  
    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 ?  
        true : 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);
```

Void and Never Return Types

A function that is expected to execute to completion but does not return a value can be marked with the void return type. For functions that are not expected to return at all (e.g., raising exceptions), they can be marked with the never return type.

```
function noReturnValue(message: string): void {  
    console.log(message);  
}  
  
function neverReturns(message: string): never {  
    throw new Error(message);  
}  
  
noReturnValue("Hello, World!");  
neverReturns("Goodbye!!");
```

LAB 05

Functions

Execute lab 05 available in GitHub at <https://github.com/KernelGamut32/intro-to-typescript-public/tree/main/labs/lab05>.

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]);
}
```

Tuples

Represents an array in TypeScript of a fixed size with a known type at each index of the array. Sometimes used as another way to represent an object or to accommodate assignment of multiple values together.

```
let movieDetail: [string, string, number];

movieDetail = ["The Godfather", "Francis Ford Coppola", 1972];

console.log(movieDetail);

console.log("The best movie is...");
let [title, director, yearReleased] = Math.random() > 0.5 ?
  ["The Godfather", "Francis Ford Coppola", 1972] :
  ["Star Wars", "George Lucas", 1977];

console.log(`${title} released in ${yearReleased} and directed by ${director}`);
```


Tuples as Spread Parameters

Tuples can be passed to as a spread argument to a function – the individual values captured in the tuple will be “mapped” to function parameters (if matching).

```
function displayMovie(name: string, director: [string, string, string?], yearReleased: number) {  
  console.log(`${name}, directed by ${director[0]}${director[2] ? " " + director[2] : ""} ${director[1]} was released in ${yearReleased}`);  
}  
  
const movies: [string, [string, string, string?], number][] = [  
  ["The Godfather", ["Francis", "Coppola", "Ford"], 1972],  
  ["Star Wars", ["George", "Lucas"], 1977],  
]  
  
movies.forEach(movie => displayMovie(...movie));
```

```
The Godfather, directed by Francis Ford Coppola was released in 1972  
Star Wars, directed by George Lucas was released in 1977
```

LAB 06

Arrays

Execute lab 06 available in GitHub at <https://github.com/KernelGamut32/intro-to-typescript-public/tree/main/labs/lab06>.

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());
```

Call Signatures

Interfaces can declare call signatures. Only values that can be called the way the signature is defined can be assigned to a variable of the interface type. Allows description of function that can have other properties in addition to the function logic.

```
interface FunctionWithCount {  
  count: number;  
  numCalls: number;  
  (): void;  
}  
  
let hasCallCount: FunctionWithCount;  
  
function keepsTrackOfCalls() {  
  for (let c = 0; c < keepsTrackOfCalls.numCalls; c++) {  
    keepsTrackOfCalls.count++;  
    console.log(`I've been called ${keepsTrackOfCalls.count} times!`);  
  }  
}  
  
keepsTrackOfCalls.count = 0;  
keepsTrackOfCalls.numCalls = 5;  
  
hasCallCount = keepsTrackOfCalls;  
hasCallCount();
```


Index Signatures

Index signatures can be used in an interface to create something like a key-value pair. The key is assigned a type associated to a value of a different (or same) type.

```
interface Product {
  name: string;
  description: string;
}

interface ProductsById {
  [i: string]: Product;
}

const products: ProductsById = {
  ABC123: { name: "Widget", description: "Best widget on the planet" },
  DEF456: { name: "Whatchamacallit", description: "Handy, dandy gadget" },
}

console.log(products.ABC123)
console.log(typeof products.ABC123);
console.log(products.DEF456)
console.log(typeof products.DEF456);
console.log(products.GHI789);
console.log(typeof products.GHI789);
```

LAB 07

Interfaces

Execute lab 07 available in GitHub at <https://github.com/KernelGamut32/intro-to-typescript-public/tree/main/labs/lab07>.

Classes in TypeScript

Class Definition & Usage

```
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`);
    }
  }
}
```

```
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)}`);
```

Miscellaneous

- You can create abstract classes using *abstract class MyClass { }* – not able to directly instantiate, instead intended for extension
- You can declare a class member to be static (belonging to class rather than instance)

```
class MyClass {  
    static statValue = "static";  
    ...  
}  
...  
console.log(MyClass.statValue);
```

- You can control visibility of class members using access modifiers applied to them:
 - *public* – default; available to any and all
 - *protected* – available to class where defined and its subclasses
 - *private* – available only to class where defined

LAB 08

Classes

Execute lab 08 available in GitHub at <https://github.com/KernelGamut32/intro-to-typescript-public/tree/main/labs/lab08>.

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())
```

LAB 09

Generics

Execute lab 09 available in GitHub at <https://github.com/KernelGamut32/intro-to-typescript-public/tree/main/labs/lab09>.

Declaration Files



Declaration Files

- Declaration files allow for the definition of type shapes apart from the type's implementation
- Files end with `.d.ts` extension – can be shared as a standalone “typings” package or distributed as part of an npm package
- Cannot include any runtime code that can be transpiled to JavaScript
- Can be imported like other TypeScript files
- Creates an “ambient context” – type declarations only, no values or implementation



Type Declarations

- Created using the *declare* keyword
- The declare keyword most commonly appears in declaration files but can also be used in modules or scripts
- In modules/scripts, declare can be used to declare global variables for use in either the local file only (or in other files) depending on how defined



Global Values

- TypeScript files with no import or export statements are treated as scripts
- When a TypeScript file includes imports/exports, it becomes a module
- In a script, globally declared types, variables, functions, etc. are available for global use



Global Values

- In a TypeScript module (includes imports/exports), sometimes defining items in global scope can be helpful
- The *declare global* syntax can be used to define a block in a module that should be accessible globally
- Other items defined outside of this block will be locally-scoped to the module



Exposing Package Types

- Typical package distribution for a package written in TypeScript will include the transpiled *.js outputs and the TypeScript declaration files
- You can use a *--declaration* command-line flag when running *tsc* to generate the *.d.ts files automatically
- For projects/packages that are meant to be distributed via npm, you should add a *types* field in the package's *package.json* file referencing the root declaration file

Supporting Tools

- *DefinitelyTyped* (a GitHub repo available at <https://github.com/DefinitelyTyped/DefinitelyTyped>) provides a repository for community-authored definitions
- DefinitelyTyped packages are published on npm under the `@types` scope (same name as the package the types are being provided for)
- <https://www.typescriptlang.org/dt/search?search=> can also be used to find type declarations for packages (either bundled or catalogued separate in DefinitelyTyped)

LAB 10

Declaration Files

Execute lab 10 available in GitHub at <https://github.com/KernelGamut32/intro-to-typescript-public/tree/main/labs/lab10>.



Thank you!

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