Devknus

TYPESCRIPT BOOK

Prepared By:
Kamal Nayan Upadhyay

Devknus

TYPESCRIPT BOOK

Prepared By:

Kamal Nayan Upadhyay

Content

S.No.	Title
1.	TypeScript Introduction
2.	TypeScript Setup
3.	TypeScript Simple Types
4.	TypeScript Special Types
5.	TypeScript Arrays
6.	TypeScript Tuples
7.	TypeScript Object Types
8.	TypeScript Enums
9.	TypeScript Type Aliases and Interfaces
10.	TypeScript Union Types
11.	TypeScript Functions
12.	TypeScript Casting
13.	TypeScript Basic Generics
14.	TypeScript Utility Types
15.	TypeScript Keyof
16.	TypeScript Null & Undefined

1. TYPESCRIPT INTRODUCTION

DEVKNUS

Summary: in this tutorial, you'll understand what TypeScript is and its advantages over vanilla JavaScript.

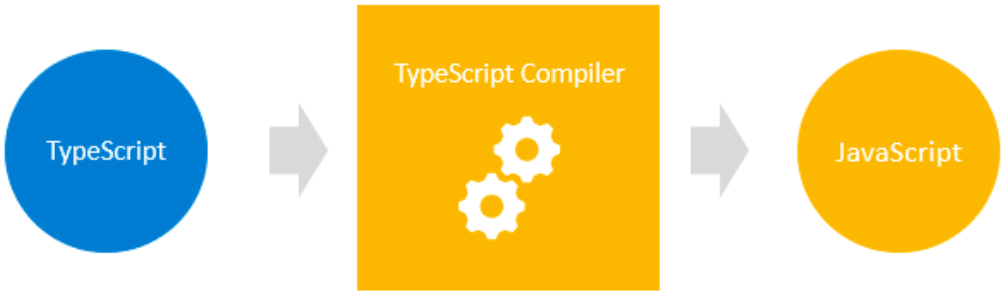
Introduction to TypeScript

TypeScript is a **superset** of **JavaScript**.

TypeScript builds on top of **JavaScript**. First, you **write** the **TypeScript code**. Then, you **compile** the **TypeScript code** into **plain JavaScript code** using a **TypeScript compiler**.

Once you have the **plain JavaScript code**, you can deploy it to any **environment** that **JavaScript runs**.

TypeScript **files** use the **.ts** extension rather than the **.js** extension of **JavaScript** files.



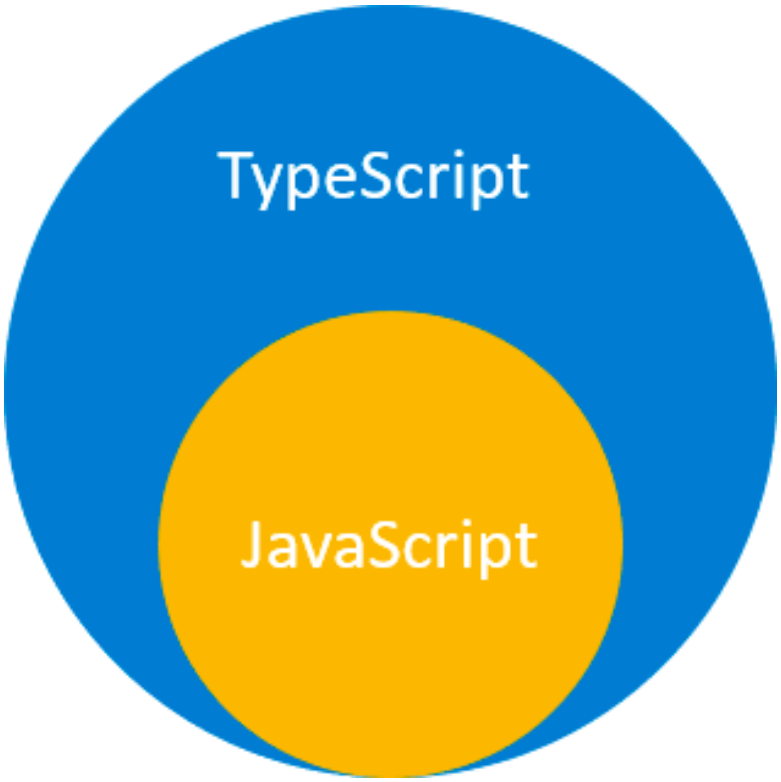
TypeScript uses the **JavaScript syntaxes** and **adds** additional **syntaxes** for **supporting Types**.

If you have a **JavaScript program** that doesn't have **any syntax errors**, it is also a **TypeScript program**. It means that all **JavaScript programs** are **TypeScript programs**. This is very **helpful** if you're **migrating** an existing **JavaScript** codebase to **TypeScript**.

1. TYPESCRIPT INTRODUCTION

DEVKNUS

The following **diagram** shows the **relationship** between **TypeScript** and **JavaScript**:



Why TypeScript

The main goals of TypeScript are:

- Introduce **optional** types to **JavaScript**.
- Implement **planned** features of **future JavaScript**, a.k.a. ECMAScript Next or ES Next to the current JavaScript.

1. TYPESCRIPT INTRODUCTION

DEVKNUS

---

1) TypeScript improves your **productivity** while helping avoid **bugs**

Types increase **productivity** by helping you avoid many **mistakes**. By using **types**, you can catch **bugs** at the **compile time** instead of having them occur at **runtime**.

For example, the following function **adds** two **numbers x** and **y**:

```
function add(x, y) {
   return x + y;
}
```

If you get the **values** from **HTML** input **elements** and **pass** them into the **function**, you may get an **unexpected** result:

```
let result = add(input1.value, input2.value);
console.log(result); // result of concatenating strings
// Output 1020
```

For example, if users entered **10** and **20**, the **add()** function would **return 1020**, instead of **30**.

The reason is that **input1.value** and **input2.value** are **strings**, not **numbers**. When you use the **operator +** to **add** two **strings**, it **concatenates** them into a **single string**.

1. TYPESCRIPT INTRODUCTION

DEVKNUS

When you use **TypeScript** to **specify** the **type** for the **parameters** like this **explicitly**:

```
function add(x: number, y: number) {
   return x + y;
}
```

In this **function**, we **added** the **number** types to the **parameters**. The function **add()** will **accept** only **numbers**, not any other **values**.

When you invoke the **function** as follows:

```
// function
let result = add(input1.value, input2.value);
```

… the TypeScript compiler will issue an error if you compile the TypeScript code into JavaScript. Hence, you can prevent the error from happening at runtime.

1. TYPESCRIPT INTRODUCTION

DEVKNUS

---

2) TypeScript brings the **future JavaScript** to **today**

TypeScript supports the upcoming features planned in the ES Next for the current JavaScript engines. It means you can use the new JavaScript features before web browsers (or other environments) fully support them.

Every year, TC39 releases several new features for ECMAScript, which is the standard of JavaScript. The feature proposals typically go through five stages:

- Stage 0: Strawperson
- Stage 1: Proposal
- Stage 2: Draft
- Stage 3: Candidate
- Stage 4: Finished

And **TypeScript** generally supports **features** that are in **stage 3**.

2. TYPESCRIPT SETUP

DEVKNUS

Summary: in this tutorial, you'll learn how to set up a TypeScript development environment.

The following tools you need to set up to **start** with **TypeScript**:

- **Node.js** – Node.js is the environment in which you will run the TypeScript compiler. Note that you don't need to know node.js.
- **TypeScript compiler** – a Node.js module that compiles TypeScript into JavaScript. If you use JavaScript for node.js, you can install the ts-node module. It is a TypeScript execution and REPL for node.js
- **Visual Studio Code** or **VS Code** – is a code editor that supports TypeScript. VS Code is highly recommended. However, you can use your favorite editor.

Install TypeScript compiler

To install the TypeScript compiler, you launch the Terminal on macOS or Linux and Command Prompt on Windows and type the following command:

```
// TypeScript
npm install -g typescript
```

After the installation, you can type the following command to check the current version of the TypeScript compiler:

```
// ts version
tsc --v
```

2. TYPESCRIPT SETUP

DEVKNUS

It should **return** the **version** like this:

```
// ts version
Version 4.0.2
```

Note that your **version** is probably **newer** than this **version**.

If you're on **Windows** and got the following **error**:

```
'tsc' is not recognized as an internal or external command,
operable program or batch file.
```

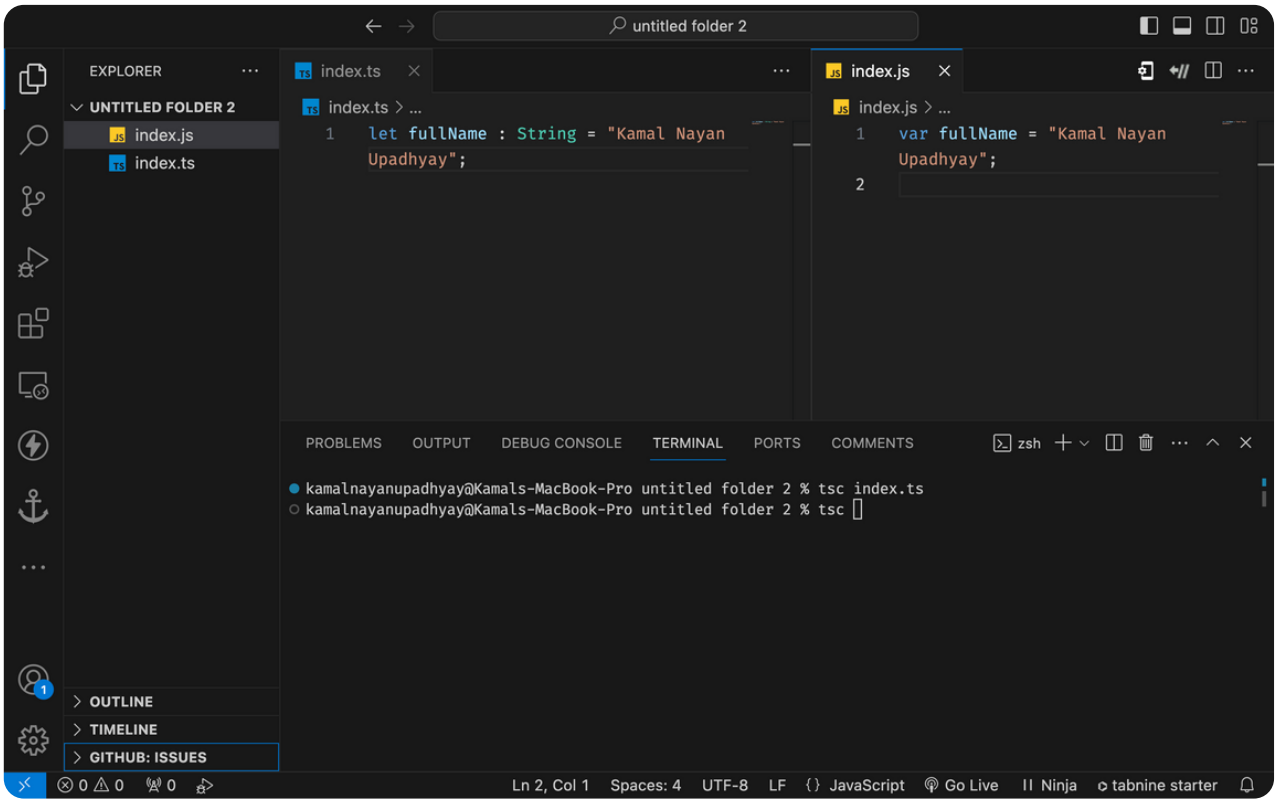
… then you should add the following path **C:\Users\<user>\AppData\Roaming\npm** to the PATH **variable**. Notice that you should change the **<user>** to your **Windows user**.

To install the **ts-node** module globally, you run the following command from the Terminal on macOS and Linux or Command Prompt on Windows:

```
// ts-node
npm install -g ts-node
```

2. TYPESCRIPT SETUP

DEVKNUS



3. TYPESCRIPT SIMPLE TYPES

DEVKNUS

TypeScript supports some **simple** types (primitives) you may know.

```
✅ There are three main primitives in JavaScript and TypeScript.

👉 boolean - true or false values
👉 number - whole numbers and floating point values
👉 string - text values like "TypeScript Rocks"

✅ There are also 2 less common primitives used in later versions of Javascript
   and TypeScript.

👉 bigint - whole numbers and floating point values, but allows larger negative
            and positive numbers than the number type.
👉 symbol - symbol are used to create a globally unique identifier.
```

Type Assignment

When creating a variable, there are two main ways TypeScript assigns a type:

1. Explicit
2. Implicit

In **both** examples below **firstName** is of type **string**

1. Explicit Type

Explicit - writing out the type:

```
// Explicit
let firstName: string = "Kamal Nayan";
```

Explicit type **assignment** are **easier** to **read** and more **intentional**.

3. TYPESCRIPT SIMPLE TYPES

DEVKNUS

TypeScript supports some **simple** types (primitives) you may know.

```
✅ There are three main primitives in JavaScript and TypeScript.

👉 boolean - true or false values
👉 number - whole numbers and floating point values
👉 string - text values like "TypeScript Rocks"

✅ There are also 2 less common primitives used in later versions of Javascript
   and TypeScript.

👉 bigint - whole numbers and floating point values, but allows larger negative
            and positive numbers than the number type.
👉 symbol - symbol are used to create a globally unique identifier.
```

Type Assignment

When creating a variable, there are two main ways TypeScript assigns a type:

1. Explicit
2. Implicit

In **both** examples below **firstName** is of type **string**

1. Explicit Type

Explicit - writing out the type:

```
// Explicit
let firstName: string = "Kamal Nayan";
```

Explicit type **assignment** are **easier** to **read** and more **intentional**.

3. TYPESCRIPT SIMPLE TYPES

DEVKNUS

2. Implicit Type

Implicit - TypeScript will "**guess**" the type, based on the **assigned value**:

```
// Implicit
let firstName: string = "Kamal Nayan";
```

Note: Having TypeScript "**guess**" the **type** of a value is called **infer**.

Implicit assignment forces **TypeScript** to **infer** the **value**.

Implicit type **assignment** are **shorter**, **faster** to **type**, and **often** used when **developing** and **testing**.

Error In Type Assignment

TypeScript will throw an error if data types do not match.

```
// Explicit Type
let firstName: string = "Kamal Nayan"; // type string
firstName = 33; // attempts to re-assign the value to a different type
```

Implicit type **assignment** would have made **firstName** less **noticeable** as a **string**, but **both** will **throw** an **error**:

```
// Implicit Type
let firstName = "Kamal Nayan"; // inferred to type string
firstName = 33; // attempts to re-assign the value to a different type
```

JavaScript will not **throw** an **error** for **mismatched types**.

4. TYPESCRIPT SPECIAL TYPES

DEVKNUS

TypeScript has **special types** that may not **refer** to any **specific** type of **data**.

Type: any

any is a **type** that disables type checking and effectively allows all types to be used.

The example below does not use **any** and will throw an error:

```
// Example without any
let u = true;
u = "string"; // Error: Type 'string' is not assignable to type 'boolean'.
Math.round(u); // Error: Argument of type 'boolean' is not assignable to
parameter of type 'number'.
```

Setting **any** to the special type **any** disables type checking:

```
// Example with any
let v: any = true;
v = "string"; // no error as it can be "any" type
Math.round(v); // no error as it can be "any" type
```

any can be a useful way to get past errors since it disables type checking, but TypeScript will not be able provide type safety, and tools which rely on type data, such as auto completion, will not work. Remember, it should be avoided at "any" cost...

4. TYPESCRIPT SPECIAL TYPES

DEVKNUS

Type: unknown

unknown is a similar, but safer alternative to **any**.

TypeScript will prevent **unknown** types from being used, as shown in the below example:

```
// Type: unknown
let val: unknown;
console.log(val); // undefined
val = true;
console.log(val); // true
val = 7;
console.log(val); // 7
```

Compare the example above to the previous example, with **any**.

unknown is best used when you don't know the type of data being typed. To add a type later, you'll need to cast it.

Casting is when we use the "as" keyword to say property or variable is of the casted type.

Type: never

never effectively throws an error whenever it is defined.

```
// Type: never
let x: never = true; // Error: Type 'boolean' is not assignable to type 'never'.
```

never is rarely used, especially by itself, its primary use is in advanced generics.

4. TYPESCRIPT SPECIAL TYPES

DEVKNUS

Type: undefined & null

undefined and **null** are types that refer to the JavaScript primitives **undefined** and **null** respectively.

```
let y: undefined = undefined;
console.log(typeof y); // undefined

let z: null = null;
console.log(typeof z); // object
```

These types don't have much use unless **strictNullChecks** is enabled in the **tsconfig.json** file.

5. TYPESCRIPT ARRAYS

DEVKNUS

TypeScript has a specific syntax for typing **arrays**.

```
const names: string[] = [];
names.push("Kamal Nayan"); // no error
// names.push(3); // Error: Argument of type 'number' is not assignable to
parameter of type 'string'.
```

Readonly

The readonly keyword can prevent arrays from being changed.

```
const names: readonly string[] = ["Dylan"];

names.push("Jack"); // Error: Property 'push' does not exist on type 'readonly
string[]'.

// try removing the readonly modifier and see if it works?
console.log(names);
```

Type Inference

TypeScript can infer the type of an array if it has values.

```
const numbers = [1, 2, 3]; // inferred to type number[]

numbers.push(4); // no error

// comment line below out to see the successful assignment
numbers.push("2"); // Error: Argument of type 'string' is not assignable to
parameter of type 'number'.

console.log(numbers);

let head: number = numbers[0]; // no error

console.log(head);
```

6. TYPESCRIPT TUPLES

DEVKNUS

Typed Arrays

A tuple is a typed array with a pre-defined length and types for each index.

Tuples are great because they allow each element in the array to be a known type of value.

To define a tuple, specify the type of each element in the array:

```
// define our tuple
let ourTuple: [number, boolean, string];

// initialize correctly
ourTuple = [5, false, 'Coding God was here'];

console.log(ourTuple); // [ 5, false, 'Coding God was here' ]
```

As you can see we have a number, boolean and a string. But what happens if we try to set them in the wrong order:

```
// define our tuple
let ourTuple: [number, boolean, string];

// initialize incorrectly throws an error
ourTuple = [false, 'Coding God was mistaken', 5];

console.log(ourTuple); // error
```

Even though we have a boolean, string, and number the order matters in our tuple and will throw an error.

6. TYPESCRIPT TUPLES

DEVKNUS

Readonly Tuple

A good practice is to make your **tuple readonly**.

Tuples only have strongly defined types for the initial values:

```
// define our tuple
let ourTuple: [number, boolean, string];

// initialize correctly
ourTuple = [5, false, 'Coding God was here'];

// We have no type safety in our tuple for indexes 3+
ourTuple.push('Something new and wrong');

console.log(ourTuple); // [ 5, false, 'Coding God was here', 'Something new and
wrong' ]
```

You see the new valueTuples only have strongly defined types for the initial values:

```
// define our tuple
let ourTuple: [number, boolean, string];

// initialize correctly
ourTuple = [5, false, 'Coding God was here'];

// We have no type safety in our tuple for indexes 3+
ourTuple.push('Something new and wrong');

// instead use our readonly tuple
const ourReadonlyTuple: readonly [number, boolean, string] = [5, true, 'The Real
Coding God'];

// throws error as it is readonly.
ourReadonlyTuple.push('Coding God took a day off'); // prog.ts(4,18): error
TS2339: Property 'push' does not exist on type 'readonly [number, boolean,
string]'.
```

6. TYPESCRIPT TUPLES

DEVKNUS

If you have ever used **React** before you have **worked** with **tuples** more than likely.
useState returns a **tuple** of the value and a **setter function**.
const **[firstName, setFirstName]** = useState('**Kamal**') is a common example.
Because of the structure we know our first value in our list will be a certain value type in this case a string and the second value a function.

Named Tuples

Named tuples allow us to provide context for our values at each index.

```
// Named Tuples
const graph: [x: number, y: number] = [55.2, 41.3];
```

Named tuples provide more context for what our index values represent.

Destructuring Tuples

Since tuples are arrays we can also destructure them.

```
const graph: [number, number] = [55.2, 41.3];
const [x, y] = graph;
```

7. TYPESCRIPT OBJECT TYPES

DEVKNUS

TypeScript has a specific syntax for typing objects.

```
const car: { type: string, model: string, year: number } = {
  type: "Toyota",
  model: "Corolla",
  year: 2009
};

console.log(car);
```

Object types like this can also be written separately, and even be reused, look at interfaces for more details.

Type Inference

TypeScript can infer the types of properties based on their values.

```
const car = {
  type: "Toyota",
};

car.type = "Ford"; // no error

car.type = 2; // Error: Type 'number' is not assignable to type 'string'.

console.log(car);
```

Optional Properties

Optional properties are properties that don't have to be defined in the object definition.

7. TYPESCRIPT OBJECT TYPES

DEVKNUS

```
// Example without an optional property
const car: { type: string, mileage: number } = { // Error: Property 'mileage' is
missing in type '{ type: string; }' but required in type '{ type: string;
mileage: number; }'.
  type: "Toyota",
};
car.mileage = 2000;
```

```
// Example with an optional property
const car: { type: string, mileage?: number } = {
  type: "Toyota"
};

car.mileage = 2000;

console.log(car); // { type: 'Toyota', mileage: 2000 }
```

8. TYPESCRIPT ENUMS

DEVKNUS

An **enum** is a special "class" that represents a group of constants (unchangeable variables).

Enums come in two **flavors string** and **numeric**. Lets start with **numeric**.

Numeric Enums – Default

By default, **enums** will initialize the first value to **0** and add **1** to each additional value:

```
enum CardinalDirections {
  North,
  East,
  South,
  West
};

let currentDirection = CardinalDirections.North;

// North is the first value so it logs '0'
console.log(currentDirection);

// throws error when commented in as 'North' is not a valid enum
// currentDirection = 'North';
// Error: "North" is not assignable to type 'CardinalDirections'.
```

Numeric Enums – Initialized

You can set the value of the first numeric enum and have it auto increment from that:

8. TYPESCRIPT ENUMS

DEVKNUS

```
enum CardinalDirections {
  North,
  East,
  South,
  West
};

let currentDirection = CardinalDirections.North;

// North is the first value so it logs '0'
console.log(currentDirection);

// throws error when commented in as 'North' is not a valid enum
// currentDirection = 'North';
// Error: "North" is not assignable to type 'CardinalDirections'.
```

Numeric Enums – Fully Initialized

You can assign unique number values for each enum value. Then the values will not incremented automatically:

```
enum StatusCodes {
  NotFound = 404,
  Success = 200,
  Accepted = 202,
  BadRequest = 400
};

// logs 404
console.log(StatusCodes.NotFound);

// logs 200
console.log(StatusCodes.Success);
```

8. TYPESCRIPT ENUMS

DEVKNUS

String Enums

Enums can also contain **strings**. This is more common than numeric enums, because of their readability and intent.

```
enum CardinalDirections {
  North = 'North',
  East = "East",
  South = "South",
  West = "West"
};

// logs "North"
console.log(CardinalDirections.North);

// logs "West"
console.log(CardinalDirections.West);
```

Technically, you can mix and match string and numeric enum values, but it is recommended not to do so.

9. TYPESCRIPT TYPE ALIASES AND INTERFACES

DEVKNUS

TypeScript allows types to be defined separately from the variables that use them.

Aliases and Interfaces allows types to be easily shared between different variables/objects.

Type Aliases

Type Aliases allow defining types with a custom name (an Alias).

Type Aliases can be used for primitives like **string** or more complex types such as **objects** and **arrays**:

```
// Try creating a new Car using the alias provided
type CarYear = number;
type CarType = string;
type CarModel = string;
type Car = {
  year: CarYear,
  type: CarType,
  model: CarModel
};

const carYear: CarYear = 2001
const carType: CarType = "Toyota"
const carModel: CarModel = "Corolla"
const car: Car = {
  year: carYear,
  type: carType,
  model: carModel
};

console.log(car); // { year: 2001, type: 'Toyota', model: 'Corolla' }
```

9. TYPESCRIPT TYPE ALIASES AND INTERFACES

DEVKNUS

Interfaces

Interfaces are similar to type aliases, except they only apply to object types.

```
interface Rectangle {
  height: number,
  width: number
};

const rectangle: Rectangle = {
  height: 20,
  width: 10
};

console.log(rectangle); // { height: 20, width: 10 }
```

Extending Interfaces

Interfaces can extend each other's definition.

Extending an interface means you are **creating** a new **interface** with the same properties as the **original**, plus something new.

```
interface Rectangle {
  height: number,
  width: number
}

interface ColoredRectangle extends Rectangle {
  color: string
}

const coloredRectangle: ColoredRectangle = {
  height: 20,
  width: 10,
  color: "red"
};

console.log(coloredRectangle); // { height: 20, width: 10, color: 'red' }
```

10. TYPESCRIPT UNION TYPES

DEVKNUS

Union types are used when a value can be more than a single type.

Such as when a property would be **string** or **number**.

Union | (OR)

Using the | we are saying our parameter is a **string** or **number**:

```
function printStatusCode(code: string | number) {
  console.log(`My status code is ${code}.`)
}

printStatusCode(404); // My status code is 404.
printStatusCode('404'); // My status code is 404.
```

Union Type Errors

Note: you need to know what your type is when union types are being used to avoid type errors:

```
function printStatusCode(code: string | number) {
  console.log(`My status code is ${code.toUpperCase()}.`) // error: Property
'toUpperCase' does not exist on type 'string | number'. Property 'toUpperCase'
does not exist on type 'number'
}
```

In our example we are having an issue invoking **toUpperCase()** as its a **string** method and **number** doesn't have access to it.

11. TYPESCRIPT FUNCTIONS

DEVKNUS

TypeScript has a specific syntax for typing function parameters and return values.

Return Type

The type of the value returned by the function can be explicitly defined.

```
// the `: number` here specifies that this function returns a number
function getTime(): number {
  return new Date().getTime();
}

console.log(getTime()); // 1648464745471
```

If no return type is defined, TypeScript will attempt to infer it through the types of the variables or expressions returned.

Void Return Type

The type **void** can be used to indicate a function doesn't return any value.

```
function printHello(): void {
  console.log('Hello!');
}

printHello(); // Hello
```

Parameters

Function parameters are typed with a similar syntax as variable declarations.

```
function multiply(a: number, b: number) {
  return a * b;
}

console.log(multiply(2,5)) // 10
```

11. TYPESCRIPT FUNCTIONS

DEVKNUS

If no parameter type is defined, TypeScript will default to using **any**, unless additional type information is available as shown in the Default Parameters and Type Alias sections below.

Optional Parameters

By default TypeScript will assume all parameters are required, but they can be explicitly marked as optional.

```
// the `?` operator here marks parameter `c` as optional
function add(a: number, b: number, c?: number) {
  return a + b + (c || 0);
}

console.log(add(2,5)) // 7
console.log(add(2,5,1)) // 8
```

Default Parameters

For parameters with default values, the default value goes after the type annotation:

```
function pow(value: number, exponent: number = 10) {
  return value ** exponent;
}

console.log(pow(10)); // 10000000000
```

Named Parameters

Typing named parameters follows the same pattern as typing normal parameters.

11. TYPESCRIPT FUNCTIONS

DEVKNUS

```
function divide({ dividend, divisor }: { dividend: number, divisor: number }) {
  return dividend / divisor;
}

console.log(divide({dividend: 10, divisor: 2})); // 5
```

Rest Parameters

Rest parameters can be typed like normal parameters, but the type must be an array as rest parameters are always arrays.

```
function add(a: number, b: number, ...rest: number[]) {
  return a + b + rest.reduce((p, c) => p + c, 0);
}

console.log(add(10,10,10,10,10)); // 50
```

Type Alias

Function types can be specified separately from functions with type aliases.

These types are written similarly to arrow functions,.

```
type Negate = (value: number) => number;

// in this function, the parameter `value` automatically gets assigned the type
`number` from the type `Negate`
const negateFunction: Negate = (value) => value * -1;

console.log(negateFunction(10)); // -10
```

12. TYPESCRIPT CASTING

DEVKNUS

There are times when working with types where it's necessary to override the type of a variable, such as when incorrect types are provided by a library.

Casting is the process of overriding a type.

Casting with as

A straightforward way to cast a variable is using the **as** keyword, which will directly change the type of the given variable.

```
let x: unknown = 'hello';

console.log((x as string).length); // 5
```

Casting doesn't actually change the type of the data within the variable, for example the following code will not work as expected since the variable x is still holds a number.

```
let x: unknown = 4;
console.log((x as string).length); // prints undefined since numbers don't have a
length
```

TypeScript will still attempt to typecheck casts to prevent casts that don't seem correct, for example the following will throw a type error since TypeScript knows casting a string to a number doesn't makes sense without converting the data:

```
console.log((4 as string).length); // Error: Conversion of type 'number' to type
'string' may be a mistake because neither type sufficiently overlaps with the
other. If this was intentional, convert the expression to 'unknown' first.
```

The Force casting section below covers how to override this.

12. TYPESCRIPT CASTING

DEVKNUS

Casting with <>

Using <> works the same as casting with as.

```
let x: unknown = 'hello';

console.log((<string>x).length); // 5
```

This type of casting will not work with TSX, such as when working on React files.

Force casting

To override type errors that TypeScript may throw when casting, first cast to unknown, then to the target type.

```
let x = 'hello';

console.log(((x as unknown) as number).length); // x is not actually a number so this will return undefined
```

13. TYPESCRIPT BASIC GENERICS

DEVKNUS

Generics allow creating 'type variables' which can be used to create classes, functions & type aliases that don't need to explicitly define the types that they use.

Generics makes it easier to write reusable code.

Functions

Generics with functions help make more generalized methods which more accurately represent the types used and returned.

```
function createPair<S, T>(v1: S, v2: T): [S, T] {
  return [v1, v2];
}

console.log(createPair<string, number>('hello', 42)); // ['hello', 42]
```

TypeScript can also infer the type of the generic parameter from the function parameters.

Type Aliases

Generics in type aliases allow creating types that are more reusable.

```
type Wrapped<T> = { value: T };

const wrappedValue: Wrapped<number> = { value: 10 };
```

This also works with interfaces with the following syntax: **interface Wrapped<T> {**

14. TYPESCRIPT UTILITY TYPES

DEVKNUS

TypeScript comes with a large number of types that can help with some common type manipulation, usually referred to as utility types.

This chapter covers the most popular utility types.

Partial

Partial changes all the properties in an object to be optional.

```
interface Point {
  x: number;
  y: number;
}

let pointPart: Partial<Point> = {}; // `Partial` allows x and y to be optional
pointPart.x = 10;

console.log(pointPart); // { x: 10 }
```

Required

Required changes all the properties in an object to be required.

```
interface Car {
  make: string;
  model: string;
  mileage?: number;
}

let myCar: Required<Car> = {
  make: 'Ford',
  model: 'Focus',
  mileage: 12000 // `Required` forces mileage to be defined
};

console.log(myCar); // { make: 'Ford', model: 'Focus', mileage: 12000 }
```

14. TYPESCRIPT UTILITY TYPES

DEVKNUS

Record

Record is a shortcut to defining an object type with a specific key type and value type.

```
const nameAgeMap: Record<string, number> = {
  'Alice': 21,
  'Bob': 25
};

console.log(nameAgeMap); // { Alice: 21, Bob: 25 }
```

Omit

Omit removes keys from an object type.

```
interface Person {
  name: string;
  age: number;
  location?: string;
}

const bob: Omit<Person, 'age' | 'location'> = {
  name: 'Bob'
  // `Omit` has removed age and location from the type and they can't be defined
here
};

console.log(bob); // { name: 'Bob' }
```

14. TYPESCRIPT UTILITY TYPES

DEVKNUS

Pick

Pick removes all but the specified keys from an object type.

```
interface Person {
  name: string;
  age: number;
  location?: string;
}

const bob: Pick<Person, 'name'> = {
  name: 'Bob'
  // `Pick` has only kept name, so age and location were removed from the type
and they can't be defined here
};

console.log(bob); // { name: 'Bob' }
```

Exclude

Exclude removes types from a union.

```
type Primitive = string | number | boolean;

const value: Exclude<Primitive, string> = true;
// a string cannot be used here since Exclude removed it from the type.

console.log(typeof value); // boolean
```

ReturnType

ReturnType extracts the return type of a function type.

14. TYPESCRIPT UTILITY TYPES

DEVKNUS

```ts
type PointGenerator = () => { x: number; y: number; };

const point: ReturnType<PointGenerator> = {
  x: 10,
  y: 20
};
```

Parameters

Parameters extracts the parameter types of a function type as an array.

```ts
type PointPrinter = (p: { x: number; y: number; }) => void;

const point: Parameters<PointPrinter>[0] = {
  x: 10,
  y: 20
};
```

15. TYPESCRIPT KEYOF

DEVKNUS

keyof is a keyword in TypeScript which is used to extract the key type from an object type.

keyof with explicit keys

When used on an object type with explicit keys, keyof creates a union type with those keys.

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

// `keyof Person` here creates a union type of "name" and "age", other strings
will not be allowed
function printPersonProperty(person: Person, property: keyof Person) {
  console.log(`Printing person property ${property}: "${person[property]}"`);
}

let person = {
  name: "Max",
  age: 27
};

printPersonProperty(person, "name"); // Printing person property name: "Max"s
```

keyof with index signatures

keyof can also be used with index signatures to extract the index type.

```
type StringMap = { [key: string]: unknown };
// `keyof StringMap` resolves to `string` here
function createStringPair(property: keyof StringMap, value: string): StringMap {
  return { [property]: value };
}
```

16. TYPESCRIPT NULL & UNDEFINED

DEVKNUS

TypeScript has a powerful system to deal with **null** or **undefined** values.

By default **null** and **undefined** handling is disabled, and can be enabled by setting **strictNullChecks** to true.
The rest of this page applies for when **strictNullChecks** is enabled.

Types

null and **undefined** are primitive types and can be used like other types, such as string.

```
let value: string | undefined | null = null;
console.log(typeof value); // object


value = 'hello';
console.log(typeof value); // string


value = undefined;
console.log(typeof value); // undefined
```

When **strictNullChecks** is enabled, TypeScript requires values to be set unless undefined is explicitly added to the type.

Optional Chaining

Optional Chaining is a JavaScript feature that works well with TypeScript's null handling. It allows accessing properties on an object, that may or may not exist, with a compact syntax. It can be used with the **?**. operator when accessing properties.

16. TYPESCRIPT NULL & UNDEFINED

DEVKNUS

```ts
interface House {
  sqft: number;
  yard?: {
    sqft: number;
  };
}

function printYardSize(house: House) {
  const yardSize = house.yard?.sqft;

  if (yardSize === undefined) {
    console.log('No yard');
  } else {
    console.log(`Yard is ${yardSize} sqft`);
  }
}

let home: House = {
  sqft: 500
};

printYardSize(home); // Prints 'No yard'
```

Nullish Coalescence

Nullish Coalescence is another JavaScript feature that also works well with TypeScript's null handling. It allows writing expressions that have a fallback specifically when dealing with null or undefined. This is useful when other falsy values can occur in the expression but are still valid. It can be used with the **??** operator in an expression, similar to using the && operator.

16. TYPESCRIPT NULL & UNDEFINED

DEVKNUS

```
function printMileage(mileage: number | null | undefined) {
  console.log(`Mileage: ${mileage ?? 'Not Available'}`);
}

printMileage(null); // Prints 'Mileage: Not Available'

printMileage(0); // Prints 'Mileage: 0'
```

Null Assertion

TypeScript's inference system isn't perfect, there are times when it makes sense to ignore a value's possibility of being null or undefined. An easy way to do this is to use casting, but TypeScript also provides the **!** operator as a convenient shortcut.

```
function getValue(): string | undefined {
  return 'hello';
}

let value = getValue();
console.log('value length: ' + value!.length); // value length: 5
```

Just like casting, this can be unsafe and should be used with care.