**Enums**

Our first example of a complex type is also one of the most useful: *enums*. We use enums when we’d like to **enum**erate all the possible values that a variable could have. This is in contrast to most of the other types we have studied. A variable of the string type can have any string as a value; there are infinitely many possible strings, and it would be impossible to list them all. Similarly, a variable of the boolean[] type can have any array of booleans as its value; again, the possibilities are infinite.

enum Direction {  
  North,  
  South,  
  East,  
  West  
}

There are many situations when we might want to limit the possible values of a variable. For example, the code above defines the enum Direction, representing four compass directions: Direction.North, Direction.South, Direction.East, and Direction.West. Any other values, like Direction.Southeast, are not allowed. Check out the example below:

let whichWayToArcticOcean: Direction;  
whichWayToArcticOcean = Direction.North; // No type error.  
whichWayToArcticOcean = Direction.Southeast; // Type error: Southeast is not a valid value for the Direction enum.  
whichWayToArcticOcean = West; // Wrong syntax, we must use Direction.West instead.

As shown above, an enum type can be used in a type annotation like any other type.

Under the hood, TypeScript processes these kinds of enum types using numbers. Enum values are assigned a numerical value according to their listed order. The first value is assigned a number of 0, the second a number of 1, and onwards

For example, if we set whichWayToArticOcean = Direction.North, then whichWayToArticOcean == 0 evaluates to true. Furthermore, we can reassign whichWayToArticOcean to a number value, like whichWayToArticOcean = 2, and it does not raise a type error. This is because Direction.North, Direction.South, Direction.East, and Direction.West are equal to 0, 1, 2, and 3, respectively.

We can change the starting number, writing something like

enum Direction {  
  North = 7,  
  South,  
  East,  
  West  
}

Here, Direction.North, Direction.South, Direction.East, and Direction.West are equal to 7, 8, 9, and 10, respectively.

We can also specify all numbers separately, if needed:

enum Direction {  
  North = 8,  
  South = 2,  
  East = 6,  
  West = 4  
}

(These numbers match up with the keys on the numpad portion of many keyboards.)

Let’s get some practice with TypeScript’s enums.

**Instructions**

**1.**

Potent Rodent is a pet shop with a very limited supply. The entire stock of animals consists of hamsters, rats, chinchillas, and tarantulas. Your job is to improve their order-processing code by making it type safe.

Create an enum named Pet, whose possible values are Pet.Hamster, Pet.Rat, Pet.Chinchilla, and Pet.Tarantula. Please list them in the order given here, and don’t bother assigning their numeric values yourself—let TypeScript do it automatically.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

You may have noticed that the variable petOnSale is not type safe: a wayward coder could assign to it an unacceptable value like petOnSale = 'Ox'!

Create a better, type safe variable petOnSaleTS and assign to it the value Pet.Chinchilla. Make sure to explicitly include the correct type annotation for petOnSaleTS on the same line.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

The array ordersArray is a list of orders to process, in the format [<pet name string>, <number ordered>]. (Why did someone order 50 chinchillas?) This is clearly not type safe by any stretch of the imagination.

Create the type safe version of this array, naming it ordersArrayTS. Pet names should be replaced by corresponding enum values. The number ordered should stay the same.

Don’t do the type annotation just yet, we’ll get to that in a moment.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Now that you have created ordersArrayTS and assigned its value, go back and add the appropriate type annotation. Note that ordersArrayTS is an array of tuples, so the type annotation should reflect that fact.

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

Now that we have a type safe orders array, let’s see what happens when someone tries to order a pet that the shop does not carry. Attempt to add an order for 3 jerboas by adding the code ordersArrayTS.push([Pet.Jerboa, 3]); and clicking the “Run” button.

Then, run tsc in the terminal to make sure TypeScript does not allow this push.

let petOnSale = 'chinchilla';

let ordersArray = [

  ['rat', 2],

  ['chinchilla', 1],

  ['hamster', 2],

  ['chinchilla', 50]

];

// Write your code below:

enum Pet {

Hamster,

Rat,

Chinchilla,

Tarantula

}

let petOnSaleTS:Pet=Pet.Chinchilla;

let ordersArrayTS:[Pet, number][]=[[Pet.Rat,2],[Pet.Chinchilla,1],[Pet.Hamster,2],[Pet.Chinchilla,50]]

;

console.log(Pet.Hamster)

TypeScript Enums (W3schools)

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:

Example

enum CardinalDirections {  
  North,  
  East,  
  South,  
  West  
}  
let currentDirection = CardinalDirections.North;  
// logs 0  
console.log(currentDirection);  
// throws error as 'North' is not a valid enum  
currentDirection = 'North'; // Error: "North" is not assignable to type 'CardinalDirections'.

[Try it Yourself »](https://www.w3schools.com/typescript/trytypescript.php?filename=demo_enums_numeric)

Numeric Enums - Initialized

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

Example

enum CardinalDirections {  
  North = 1,  
  East,  
  South,  
  West  
}  
// logs 1  
console.log(CardinalDirections.North);  
// logs 4  
console.log(CardinalDirections.West);

[Try it Yourself »](https://www.w3schools.com/typescript/trytypescript.php?filename=demo_enums_numeric_init)

Numeric Enums - Fully Initialized

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

Example

enum StatusCodes {  
  NotFound = 404,  
  Success = 200,  
  Accepted = 202,  
  BadRequest = 400  
}  
// logs 404  
console.log(StatusCodes.NotFound);  
// logs 200  
console.log(StatusCodes.Success);

[Try it Yourself »](https://www.w3schools.com/typescript/trytypescript.php?filename=demo_enums_numeric_custom)

String Enums

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

Example

enum CardinalDirections {  
  North = 'North',  
  East = "East",  
  South = "South",  
  West = "West"  
};  
// logs "North"  
console.log(CardinalDirections.North);  
// logs "West"  
console.log(CardinalDirections.West);

[Try it Yourself »](https://www.w3schools.com/typescript/trytypescript.php?filename=demo_enums_strings)

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

**String Enums vs. Numeric Enums**

The enums we have studied so far are referred to as numeric enums, since they are based on numbers. TypeScript also allows us to use enums based on strings, referred to as string enums. They are defined very similarly:

enum DirectionNumber { North, South, East, West }  
enum DirectionString { North = 'NORTH', South = 'SOUTH', East = 'EAST', West = 'WEST' }

With numeric enums, the numbers could be assigned automatically, but with string enums we must write the string explicitly, as shown above. Technically, any string will do: North = 'JabberWocky' is a valid value definition. However, it is much better to use the convention shown here (North = 'NORTH'), where the string value of the enum variable is just the capitalized form of the variable name. This way, error messages and logs will be much more informative.

We recommend to always use string enums because numeric enums allow for some behaviors that can let bugs sneak into our code. For example, numbers can be assigned directly to numeric enum variables:

let whichWayToAntarctica: DirectionNumber;  
whichWayToAntarctica = 1; // Valid TypeScript code.  
whichWayToAntarctica = DirectionNumber.South; // Valid, equivalent to the above line.

Strangely, even assigning arbitrary numbers, as in whichWayToAntarctica = 943205, will not lead to type errors.

String enums are much more strict. With string enums, variables cannot be assigned to strings at all!

let whichWayToAntarctica: DirectionString;  
whichWayToAntarctica = '\ (•◡•) / Arbitrary String \ (•◡•) /'; // Type error!  
whichWayToAntarctica = 'SOUTH'; // STILL a type error!  
whichWayToAntarctica = DirectionString.South; // The only allowable way to do this.

Now, let’s practice.

### Instructions

**1.**

You will now revisit the pet shop example. But this time, you will use string enums instead of numeric enums.

Create a **string** enum named Pet, whose possible values are Pet.Hamster, Pet.Rat, Pet.Chinchilla, and Pet.Tarantula. (List them in this order.) Assign values according to the convention mentioned above, where the values are capitalized forms of the variable name: enum Example { variable = 'VARIABLE'};.

**2.**

Create the type-safe variable petOnSaleTS and assign to it the value Pet.Chinchilla. Make sure to explicitly include the correct type annotation for petOnSaleTS.

Stuck? Get a hint

**3.**

As before, ordersArray is a list of orders to process, in the format [<pet name string>, <number ordered>]. This is not type safe.

Create the type safe version of this array, naming it ordersArrayTS. Pet names should be replaced by corresponding enum values. The number ordered should stay the same.

Include the type annotation for ordersArrayTS as well.

Stuck? Get a hint

**4.**

As discussed above, string enums are very particular about the form their values are written in. Even though you wrote Hamster = 'HAMSTER' in the enum Pet statement, you cannot use 'HAMSTER' as the enum value. You must use Pet.Hamster instead.

Demonstrate this fact by adding the code ordersArrayTS.push(['HAMSTER', 1]); and clicking “Run”. Then, run tsc in the terminal to make sure TypeScript does not allow this.

**Object Types**

It’s time, we can finally discuss object-oriented programming and how it relates to TypeScript! TypeScript’s object types are extremely useful, as they allow us extremely fine-level control over variable types in our programs. They’re also the most common custom types, so we’ll have to understand them if we want to read other people’s programs.

Here’s a type annotation for an object meant to represent a person:

let aPerson: {name: string, age: number};

The type annotation looks like an object literal, but instead of values appearing after properties, we have types. Notice that the variable aPerson has yet to be assigned a value. Trying to assign a value to aPerson that doesn’t have name and age properties of the specified types will lead to a type error:

aPerson = {name: 'Aisle Nevertell', age: "wouldn't you like to know"}; // Type error: age property has the wrong type.  
aPerson = {name: 'Kushim', yearsOld: 5000}; // Type error: no age property.   
aPerson = {name: 'User McCodecad', age: 22}; // Valid code.

Above, in the case of Kushim, the object had properties of the correct types. Still, a type error was thrown because the properties didn’t have the correct names.

TypeScript places no restrictions on the types of an object’s properties. They can be enums, arrays, and even other object types!

let aCompany: {  
  companyName: string,   
  boss: {name: string, age: number},   
  employees: {name: string, age: number}[],   
  employeeOfTheMonth: {name: string, age: number},    
  moneyEarned: number  
};

This is only an introduction to TypeScript’s object types. A thorough description would deserve a lesson of its own (which we will soon arrive at if we keep learning). For now, let’s practice the basics some more.

### Instructions

**1.**

When we write functions, we like to give arguments explanatory names, which help the programmer using the function. For example, the arguments in function sayHappyBirthday(name, age) suggest that sayHappyBirthday('Cody', 10) will be valid input. However, sometimes you will encounter code where data is passed into a function by using an object, as with sayHappyBirthdayWithObject() in the code editor.

The problem is, sayHappyBirthdayWithObject(personObject) is not very explanatory. Here, TypeScript can help us specify the exact properties that personObject should contain for the function to work.

Add a type annotation to the personObject parameter that will ensure that the function will always receive an object with the required properties.

Stuck? Get a hint

**2.**

The code editor defines the variable birthdayBabies without a type annotation. Please add in the correct type annotation yourself.

# **TypeScript Object Types**

TypeScript has a specific syntax for typing objects.

Read more about objects in our [JavaScript Objects chapter](https://www.w3schools.com/js/js_objects.asp).

### Example

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

[Try it Yourself »](https://www.w3schools.com/typescript/trytypescript.php?filename=demo_object_types_ex)

Object types like this can also be written separately, and even be reused, look at [interfaces](https://www.w3schools.com/typescript/typescript_aliases_and_interfaces.php) for more details.

## **Type Inference**

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

### Example

const car = {  
  type: "Toyota",  
};  
car.type = "Ford"; // no error  
car.type = 2; // Error: Type 'number' is not assignable to type 'string'.

[Try it Yourself »](https://www.w3schools.com/typescript/trytypescript.php?filename=demo_object_types_infer)

## **Optional Properties**

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

### 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 } = { // no error  
  type: "Toyota"  
};  
car.mileage = 2000;

[Try it Yourself »](https://www.w3schools.com/typescript/trytypescript.php?filename=demo_object_types_optional)

## **Index Signatures**

Index signatures can be used for objects without a defined list of properties.

### Example

const nameAgeMap: { [index: string]: number } = {};  
nameAgeMap.Jack = 25; // no error  
nameAgeMap.Mark = "Fifty"; // Error: Type 'string' is not assignable to type 'number'.

[Try it Yourself »](https://www.w3schools.com/typescript/trytypescript.php?filename=demo_object_types_undefined_props)

Index signatures like this one can also be expressed with utility types like Record<string, number>.

Learn more about utility types like this in our [TypeScript Utility Types](https://www.w3schools.com/typescript/typescript_utility_types.php) chapter.