TypeScript Tuples

- 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:

Syntax

```
let tuple_name = [val1, val2, val3, ...val n];
```

Example

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

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

// define and initialize tuple simultaneously

let arrTuple: [number, string, number, string] = [501, "welcome", 505,
"Mohan"];
```

What happens if we try to set them in the wrong order:

Example

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

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

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

Readonly Tuple

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

Example

```
// define 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');
```

Note: Tuples allow adding individual values, through push method (if the tuple is not declared read-only). Moreover, it allows adding values without following the tuple structure declared (data types and order declared is not considered). This is because tuple is not a grouped array

Example

```
// 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

ourTuple.push('Something new and wrong');
console.log(ourTuple);
```

Named Tuples

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

Example

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

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

Operations on Tuples

1. Push Operation

The push() method adds elements to the tuple.

```
var employee: [number, string] = [1, "Steve"];
employee.push(2, "Bill");
console.log(employee);
```

Output: [1, 'Steve', 2, 'Bill']

This type of declaration is allowed in tuples because we are adding number and string values to the tuple and they are valid for the **employee** tuple.

Example 2:

```
let empTuple: [string, number, string] = ["Vivek Singh", 22,
"Honesty"];
console.log("Items: " + empTuple);
```

```
empTuple.push(10001);
console.log("Length of Tuple Items after push: " +
empTuple.length);
console.log("Items: " + empTuple);
```

Output

```
Items: Vivek Singh, 22, Honesty Length of Tuple Items after push: 4 Items: Vivek Singh, 22, Honesty, 10001
```

2. Pop Operation

The pop() method removes the last element from the tuple.

Example

```
let empTuple: [string, number, string, number] = ["Mohit Singh",
25, "Developer", 10001];
console.log("Items: " + empTuple);
empTuple.pop();
console.log("Length of Tuple Items after pop: " +
empTuple.length);
console.log("Items: " + empTuple);
```

Output

```
Items: Mohit Singh, 25, Developer, 10001
Length of Tuple Items after pop: 3
Items: Mohit Singh, 25, Developer
```

Update or Modify Tuple Elements

You can modify tuple elements using their indices and the assignment operator.

Example

```
let empTuple: [string, number, string] = ["Ganesh Singh", 25,
"TCS"];
empTuple[1] = 60;
console.log("Name of the Employee is: " + empTuple[0]);
console.log("Age of the Employee is: " + empTuple[1]);
console.log(empTuple[0] + " is working in " + empTuple[2]);
```

Output

```
Name of the Employee is: Ganesh Singh
Age of the Employee is: 60
Ganesh Singh is working in TCS
```

Clear Tuple Fields

You can clear the fields of a tuple by assigning it an empty tuple.

Example

```
let empTuple: [string, number, string] = ["Rohit Sharma", 25,
"JavaTpoint"];
empTuple = [] as [string, number, string];
console.log(empTuple);
```

Output: []

Passing Tuples to Functions

Tuples can be passed as parameters to functions.

Example

```
let empTuple: [string, number, string] = ["Developer", 101, "Abhishek"];

function display(tuple_values: any[]) {
  for (let i = 0; i < tuple_values.length; i++) {
    console.log(tuple_values[i]);
  }
}

display(empTuple);
```

Output

```
Developer
101
Abhishek
```

Destructuring Tuples

Since tuples are arrays, we can also destructure them.

Example

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

Destructuring

To illustrate destructuring, we'll make a sandwich. Do you take everything out of the refrigerator to make your sandwich? No, you only take out the items you would like to use on your sandwich.

Destructuring is exactly the same. We may have an array or object that we are working with, but we only need some of the items contained in these.

Destructuring makes it easy to extract only what is needed.

Destructuring Arrays

Here is the old way of assigning array items to a variable:

Example

Before:

```
const vehicles = ['mustang', 'f-150', 'expedition'];
// old way
const car = vehicles[0];
const truck = vehicles[1];
const suv = vehicles[2];
```

New way of assigning array items to a variable:(De-structuring)

Example

With destructuring:

```
const vehicles = ['mustang', 'f-150', 'expedition'];
const [car, truck, suv] = vehicles;
```

When destructuring arrays, the order that variables are declared is important.

If we only want the car and suv we can simply leave out the truck but keep the comma:

```
const vehicles = ['mustang', 'f-150', 'expedition'];
const [car,, suv] = vehicles;
```

Destructuring comes in handy when a function returns an array:

Example

```
function calculate(a, b) {
  const add = a + b;
  const subtract = a - b;
  const multiply = a * b;
  const divide = a / b;
  return [add, subtract, multiply, divide];
}
const [add, subtract, multiply, divide] = calculate(4, 7);
```

De-structuring Objects

Old way of using an object inside a function:

Example: Before / Old Way

```
const vehicleOne = {
  brand: 'Ford',
  model: 'Mustang',
  type: 'car',
  year: 2021,
  color: 'red'
}
myVehicle(vehicleOne);
// old way
function myVehicle(vehicle) {
  const message = 'My ' + vehicle.type + ' is a ' + vehicle.color + ' '
+ vehicle.brand + ' ' + vehicle.model + '.';
}
```

New way of using an object inside a function:

Example: With destructuring:

```
const vehicleOne = {
  brand: 'Ford',
  model: 'Mustang',
  type: 'car',
  year: 2021,
  color: 'red'
}
myVehicle(vehicleOne);
function myVehicle({type, color, brand, model}) {
```

```
  const message = 'My ' + type + ' is a ' + color + ' ' + brand + ' ' +
model + '.';
}
```

Notice that the object properties do not have to be declared in a specific order.

We can even de-structure deeply nested objects by referencing the nested object then using a colon and curly braces to again destructure the items needed from the nested object:

Example

```
const vehicleOne = {
  brand: 'Ford',
  model: 'Mustang',
  type: 'car',
  year: 2021,
  color: 'red',
  registration: {
    city: 'Houston',
    state: 'Texas',
    country: 'USA'
  }
}
myVehicle(vehicleOne)
function myVehicle({ model, registration: { state } }) {
  const message = 'My ' + model + ' is registered in ' + state + '.';
}
```