Tuple For Type and Length Arrays

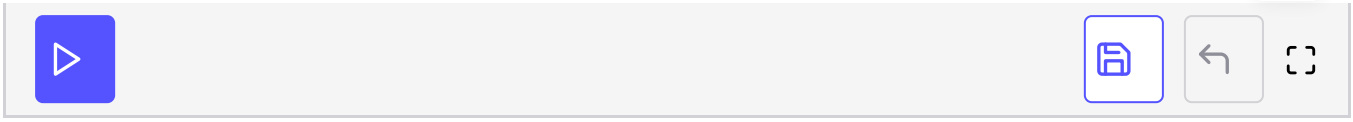
In this lesson, we will discuss about tuples and how TypeScript differentiates them from arrays.

WE'LL COVER THE FOLLOWING

Tuple syntax

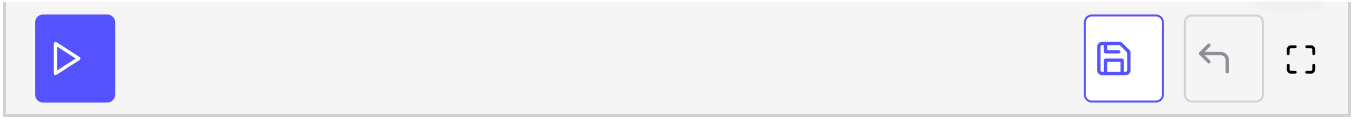
The tuple type is an array of defined elements. To declare a tuple, you use square brackets, as in **line 1**, but instead of specifying a value, you use a type.

```
let numberTuple: [number, number, number];
```



A tuple can handle many different types. The tuple uses the same syntax as for other types.

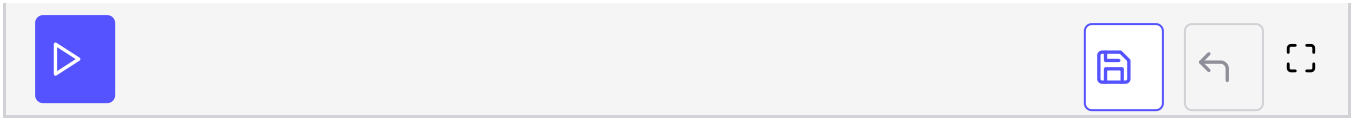
```
let myTupleWithTwoTypes: [number, string];
```



The declaration of a variable and the assignation have no difference. TypeScript checks for type during assignation.

```
let myTuple: [number, string] = [0, "1"];
myTuple = [1, "test"];
```

```
const numberVariable: number = myTuple[0];
const stringVariable: string = myTuple[1];
```

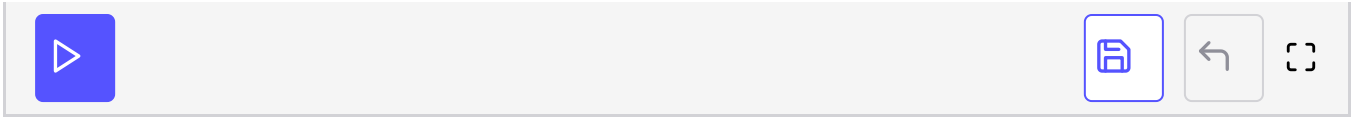


Transpilation Effect

Tuples are converted to a simple array when transpiled to JavaScript. This means that TypeScript lets you assign values to a tuple that are beyond the possible values of the type defined in the square brackets, if the variable defined is beyond the range of either of the types defined in the tuple. For example, if you define a tuple to be a number and string, you can define a third variable that is string or number but not Boolean or an object.

> 📜**Note:** the below code throws an error ✕

```
let myTuple: [number, string];
myTuple = [1, "test"];
myTuple[3] = "one more"; // Won't compile TS >= 2.7
myTuple[4] = 2; // Won't compile TS >= 2.7
myTuple[5] = true; // Won't compile (number|string only)
```

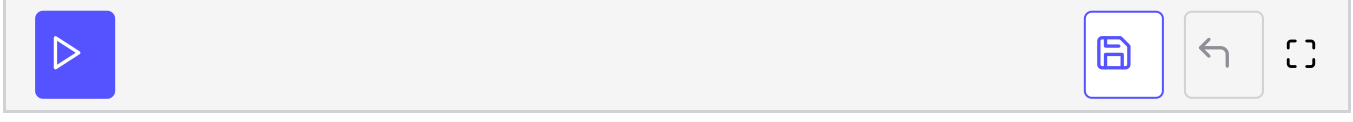


Tuple length validation

With version 2.7, TypeScript made the length property fixed. The notion of length ensures that tuples have the same length and same type at their corresponding positions, as they must in order to be assignable.

> 📜**Note:** the below code throws an error ✕

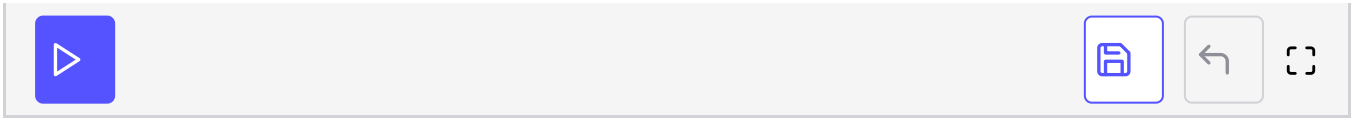
```
let firstTuple: [number, number] = [1, 2];
let secondTuple: [number, number, number] = [3, 4, 5];
secondTuple = firstTuple; // Doesn't compile type mismatch
firstTuple = secondTuple; // Doesn't compile length incompatible
```



Readonly tuple

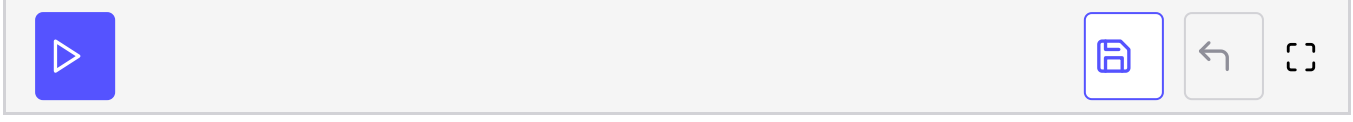
In the array lesson, we demonstrated that it is possible to mark a list to be read-only. Tuples can also benefit from the keyword `readonly`. However, this keyword cannot be used with \generic, similar to `Array<T>`.

```
let firstTuple: [number, number] = [1, 2];
let secondTuple: readonly [number, number,] = [3, 4];

firstTuple[0] = 100;
// secondTuple[0] = 1000; //Error! Read-only Tuple

console.log(firstTuple);
console.log(secondTuple);
```



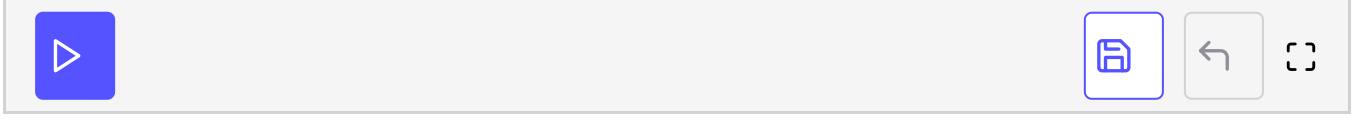
While there is no dedicated `ReadonlyTuple<T>`, it is possible to use `Readonly<T>` on a typed collection. The following code does not compile because the tuple is read-only.

```
let firstTuple: Readonly<[number, number]> = [1, 2];
firstTuple[0] = 100;

console.log(firstTuple);
```



Finally, a read-only tuple can be created by using `as const`. The operation name can be confusing since it uses the constant keyword but it will transform the variable into a typed read-only tuple. You can uncomment line 5 through 7 to see TypeScript errors.

```
let firstTuple: readonly [number, number] = [1, 2];
let secondTuple: Readonly<[number, number]> = [1, 2];
let thirdTuple = [1, 2] as const;

// firstTuple[0] = 0;
// secondTuple[0] = 0;
// thirdTuple[0] = 0;
```



In this lesson, we saw what a tuple is and how it differs from an array. Also, we got introduced to a read-only modification allowing more restrictions to be placed on the data held by tuple.