Const Assertion for Literal Values

In this lesson, we will see how to define a lilteral value with the "as const" keyword.

WE'LL COVER THE FOLLOWING

With TypeScript version 3.4, released the option to define literals with `as const`. Const assertion is useful when creating an immutable variable.

`const` vs `as const`

At first, `as const` may seem redundant because it is possible to declare a variable with `const` and make the value be unchangeable (**line 1**).

However, `const` and `as const` differ. With `as const` (**line 3**) the declaration is done with `let`, allowing the value to be changed. But, only to the literal type. Here is an example:

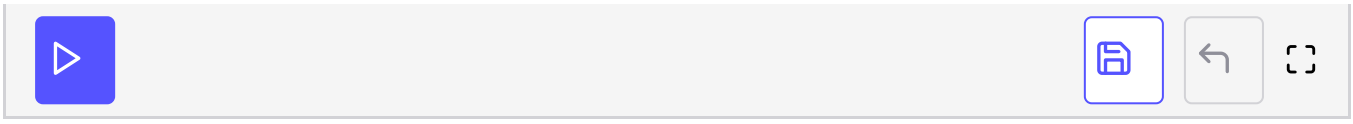
```
const v1 = 10;
// v1 = 10; // Does not compile
let v2 = 10 as const;
v2 = 10;
```

Array as `const`

It is also convenient to have an array that is read-only. Using `const` creates an object that cannot be re-assigned, but can change in terms of values.

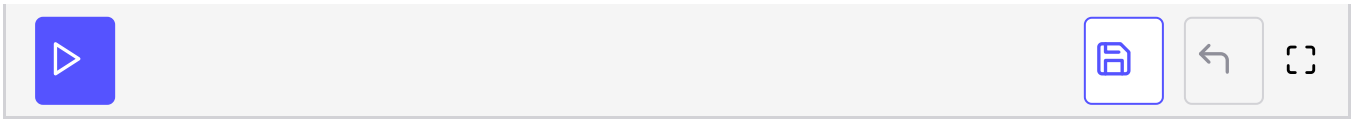
```
// Const
const myArr1 = [1, 2, 3];
myArr1.push(4);
console.log(myArr1);

// myArr1 = []; // Doest not compile because const
```



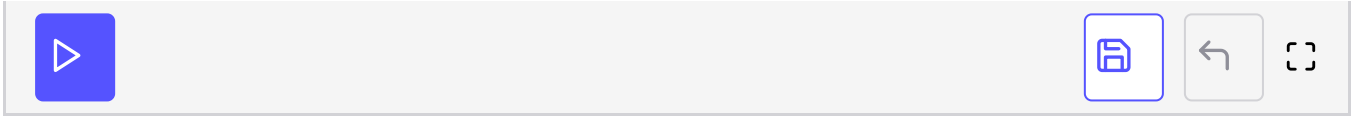
In contrast, an array with `as const` will have immutable values. Declaring with `let` and using the `const` assertion does not allow you to `push` or alter the collection. You can try by uncommenting **line 2**. TypeScript will not allow to transpile. Intellisense does not provide the possibility, neither is possible to compile code that tries to change the value.

```
let myArr2 = [1, 2, 3] as const;
// myArr2.push(4); // Does not provide the auto-complete, neither compile.
```



In the case of the array, the `myArr2` of the previous example, at **line 1** is of type `readonly [1, 2, 3]`. Thus, it is possible to write the previous example in a more convoluted way. In the following code snippet, at **line 1** you can see the long syntax. It is the same as the previous example **line 1**.

```
let myArr3: readonly [1, 2, 3] = [1, 2, 3];
// Or:
let myArr4: readonly [1, 2, 3];
myArr4 = [1, 2, 3];
// But not:
// myArr4 = [1, 2];
```

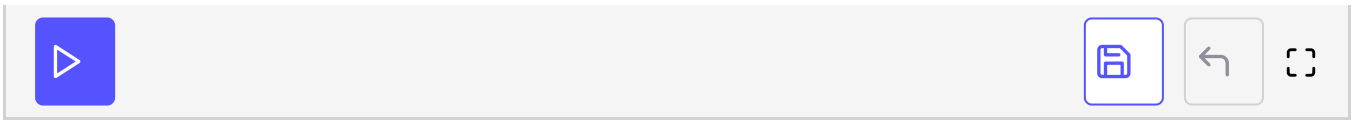


Note that all values may be set *during declaration* or later, but *must be complete* since the type is the full definition after read-only. For example, the previous code block does not allow to define only `[1, 2]`, see **line 6** because the declaration specifies `[1, 2, 3]`.

Objects with `as const`

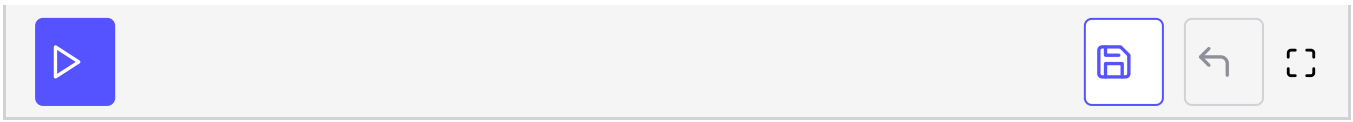
Similar to an array, it is possible to use const assertions to have an immutable object. Changing the type of an object or trying to add a new member with the index signature will both result in a compilation error.

```
let immutable1 = { id: "1" } as const;
// immutable1.id = 2; // Does not compile
// immutable1["newprop"] = 2; // Does not compile
console.log(immutable1);
```



Similar to the array, TypeScript converts the object's properties with `readonly`.

```
let immutable2: {
    readonly id: number
} = { id: 1 };
console.log(immutable2);
```

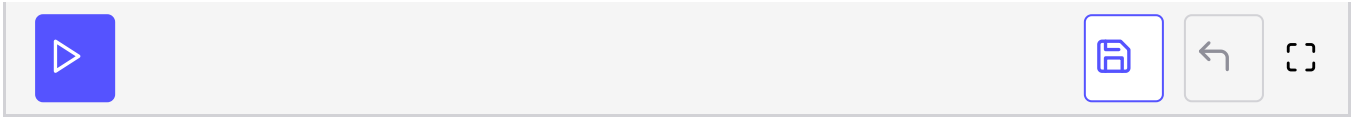


Nested objects are automatically set with `readonly` at each level making TypeScript and `as const` a quick way to ensure immutability at design time without having to handle recursive and complex data structures.

```
let person = {
  id: 1,
  name: {
    first: "Patrick",
    last: "Desjardins",
    middleName: null
  },
  location: {
    country: "USA",
    state: "CA"
  },
  relatives: [
    {
      id: 2,
      name: {
        first: "Person2",
        last: "Person22",
        middle: "Mid"
      }
    }
  ]
```

```
} as const;

// Doest not compile:
// person.relatives.push({ id: 2, name: { first: "New", last: "New", middle: "" } });
// person.id = 4;
```



It is also possible to be more accurate and select only a portion of an object to use `as const`. For example, we can specify only the location to be immutable.

```
let person = {
  id: 1,
  name: {
    first: "Patrick",
    last: "Desjardins",
    middleName: null
  },
  location: {
    country: "USA",
    state: "CA"
  } as const,
  relatives: [
    {
      id: 2,
      name: {
        first: "Person2",
        last: "Person22",
        middle: "Mid"
      }
    }
  ]
};

person.relatives.push({ id: 2, name: { first: "New", last: "New", middle: "" } });
person.id = 4;
```