Set and Dictionary

This lesson shows how to quickly access data without looping a list.

WE'LL COVER THE FOLLOWING

Storing information in a dictionary or a set is a great way to improve your performance when accessing data. In terms of "Big O" notation, dictionaries and sets are best at `O(1)` meaning that the cost is constant regardless of how many items are stored.

> Big O is a standard way to communicate the complexity of an algorithm. O(1) is the most efficient because it is constant regardless of how many elements you are manipulating.

TypeScript has a couple of ways in which you can achieve this quick access.

Index signatures

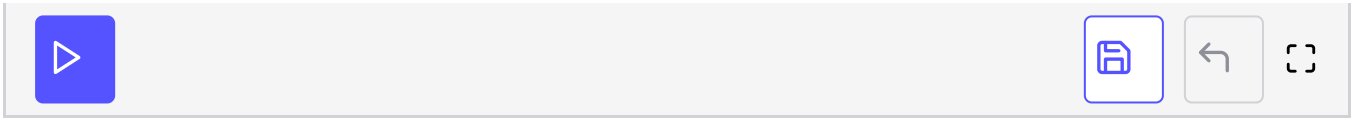
There have already been a couple of lessons concerning index signature, thus I will not reiterate the details. Nevertheless, the simplest way to have a dictionary is by mapping an object as a key to each property as a value.

In the example below, **line 4** is defining the index signature. The **line 8-9** set values to the dictionary. The key to access the value is defined between the square bracket. In that case, the keys are `1` and `10`. Accessing the value at **line 11** is very quick because there is no traversing of a list or complex algorithm involved.

```
interface Person { id: number, name: string };

interface PersonDictionary {
  [id: number]: Person;
}

const dict: PersonDictionary = {
  [1]: { id: 1, name: "First" },
  [10]: { id: 10, name: "Tenth" },
};
console.log(dict[10].name);
```

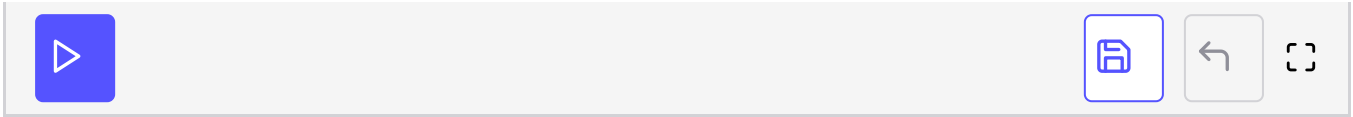


This approach does not scale since we need to build a dictionary for each type. At the moment the dictionary can only handle a `Person` type. A solution to optimize the dictionary for a variety of types is to generalize the dictionary interface with generic.

```
interface Person { id: number, name: string };

interface MyDictionary<T> {
  [id: number]: T;
}

const dict: MyDictionary<Person> = {
  [1]: { id: 1, name: "First" },
  [10]: { id: 10, name: "Tenth" },
};
console.log(dict[10].name);
```



The code above alters the previous example by using a generic type `T` at **line 3** and **line 4**.

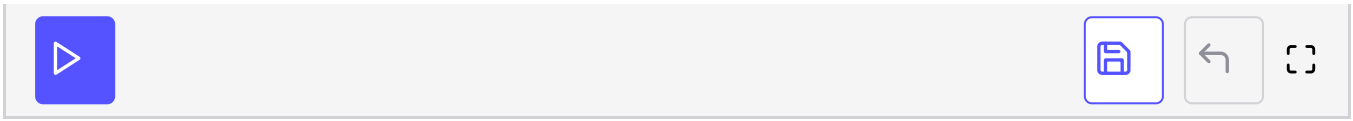
Map

It is possible in TypeScript to have a strongly typed `Map` which is a data structure introduced in the world of JavaScript since EcmaScript 6. To use the `Map` we need to have an instance of the class and then use `set` and `get` to store and access the value.

```
interface Person { id: number, name: string };
let myMap = new Map<number, Person>();
```

```
myMap.set(1, { id: 1, name: "First" });
myMap.set(10, { id: 10, name: "Tenth" });

if (myMap.has(10)) {
    console.log(myMap.get(10)!.name);
}
```



The bang operator

In the `Map` example, we are using the bang operator `!`. This operator is useful when indicating to TypeScript that it shouldn't throw an error. The example was safe because we are looking to see if the value exists with `has`.

```
if (myMap.has(10)) {
    console.log(myMap.get(10)!.name);
}
```

Next, we are accessing the value. Because the `Map` has the object and no `undefined` value can be stored, since we defined the type to be `Map<Person>` we know that it is possible to access the property `name` without getting a *null reference exception*.