Record Mutation

In this lesson, we'll explore different ways of altering the contents of a record.

WE'LL COVER THE FOLLOWING

Mutable Fields

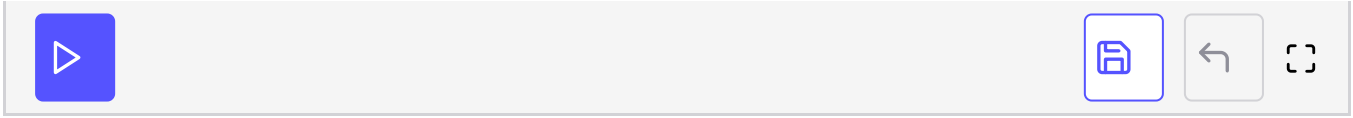
Believe it or not, we can define some record fields as mutable. This means that we can change their values in the future.

To make a field mutable, we must prepend the `mutable` keyword to it.

```
type carType = {
  manufacturer: string,
  mutable model: string,
  mutable year: int
};

let car: carType = {
  manufacturer: "Mercedes",
  model: "McLaren F1",
  year: 2019
};

Js.log(car);
```



The mutable fields can be altered using the `=` operator.

```
type carType = {
  manufacturer: string,
  mutable model: string,
```

```
  mutable year: int
};


let car: carType = {
  manufacturer: "Mercedes",
  model: "McLaren F1",
  year: 2019
};

Js.log(car);

car.model = "Benz";
car.year = 2020;

Js.log(car);
```

The Spread Operator

As we recall, `let` bindings and tuples are immutable, just like records. However, we were able to make new copies of those objects. These immutable copies could have different values.

The same principle holds for records. To make a copy of a record, we need the **spread operator**, `...`. The spread operator makes a deep copy of the original record, after which we can update any field value.

```
type place = {
  name: string,
  population: int
};

let winterfell = {
  name: "Winterfell",
  population: 500000
};
Js.log(winterfell);

let copyOfWinterfell = {
  ...winterfell, /* The spread operator makes a copy of the original record */
  population: 1000000
};
Js.log(copyOfWinterfell);
```

Nested Records

To update a nested record, we must make two new copies; one for the nested record and one for the main record.

Let's take a look at how this is done:

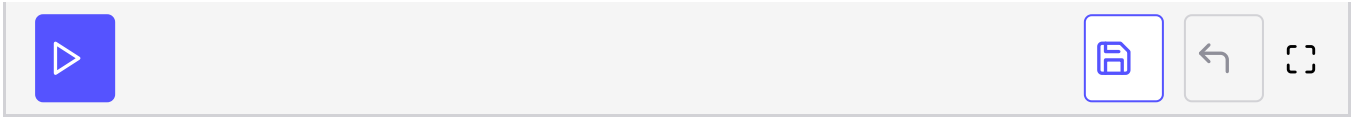
```
type placeDetails = {
    house: string,
    population: int
};
/* The record to be nested */
let placeDetails = {
  house: "Stark",
  population: 500000
};

type place = {
  name: string,
  placeDetails: placeDetails
};

let winterfell = {
  name: "Winterfell",
  placeDetails
};
Js.log(winterfell);

/* Make a new copy of the nested record with the updated population */
let copyOfPlaceDetails = {
  ...placeDetails,
  population: 1000000
};

/* Include it in the new copy of the main record */
let copyOfWinterfell = {
  ...winterfell,
  placeDetails: copyOfPlaceDetails
};
Js.log(copyOfWinterfell);
```



This is all we need to know about records for now. They will be revisited in the coming sections.

The next data structure we'll explore is the **array**. Find out more in the next lesson.