Nested Records

This lesson will teach us how to use records within records.

WE'LL COVER THE FOLLOWING

Type Nesting

In order to create nested records, we must first create nested record types. Once the record's type structure allows nesting, we'll have no problem creating nested records.

So, let's work on our `wizardInfo` example from the previous lesson. We'll be defining the `schoolInfo` type which contains the attributes `school` and `house`:

```
type schoolInfo = {
  school: string,
  house: string
};

/* Nesting schoolInfo into the wizardInfo type */
type wizardInfo = {
  name: string,
  age: int,
  schoolAndHouse: schoolInfo
};
```

Creating a Sub-Record

Now, we need to create a record of the `schoolInfo` type so that we can use it as a sub-record in our `wizard` record.

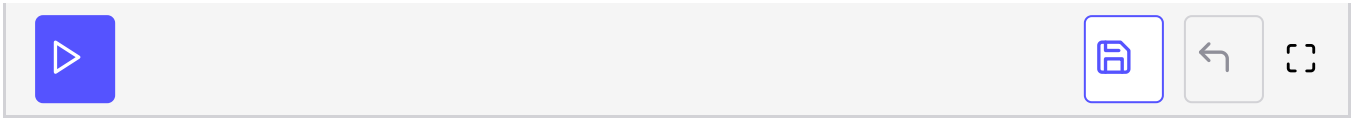
```
type schoolInfo = {
  school: string,

  house: string
};

/* Nesting schoolInfo into the wizardInfo type */
type wizardInfo = {
  name: string,
  age: int,
  schoolAndHouse: schoolInfo
}

/* An instance of the schoolInfo type */
let wizardSchool: schoolInfo = {
  school: "Hogwarts",
  house: "Gryffindor"
};
```

Nesting the Sub-Record

For the final step, we simply need to include the sub-record in our main record. We named our record `wizard`'; therefore, we'll recreate it using the `harrySchool` record.

```
type schoolInfo = {
  school: string,
  house: string
};

/* Nesting schoolInfo into the wizardInfo type */
type wizardInfo = {
  name: string,
  age: int,
  schoolAndHouse: schoolInfo
}

/* An instance of the schoolInfo type */
let wizardSchool: schoolInfo = {
  school: "Hogwarts",
  house: "Gryffindor"
};

/* A nested record */
let wizard: wizardInfo = {
  name: "Harry",
  schoolAndHouse: wizardSchool,
  age: 14
};

/* Accessing the complete wizard record */
Js.log(wizard);

/* Accessing the house field */
let {schoolAndHouse: {house}} = wizard;
Js.log(house);
```

```
/* Another way of accessing the house */
let house = wizard.schoolAndHouse.house;
Js.log(house);
```



As we can see, the `wizard` record looks more concise and cleaner now. We do not have to define all of the fields within the records itself, which is a big help when writing complex programs.

Punning

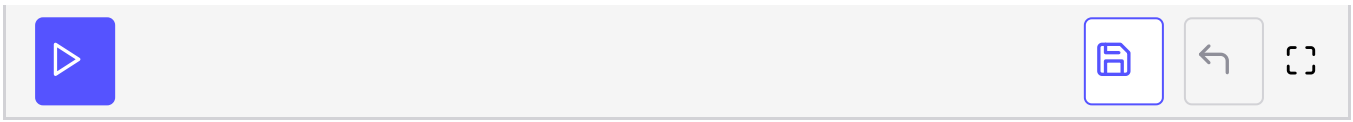
Punning is a technique which can make our code even simpler. Through punning, we can omit the type of our nested record when it is being included in the main record.

This can only be done if the type and the field have the **same name**. In the `wizard` example, the `schoolAndHouse` field is of the type `schoolInfo`.

The record used here is called `wizardInfo`. Let's rename all of these entities to `schoolInfo`, which is the same name as the type.

```
type schoolInfo = {
  school: string,
  house: string
};

/* The record will be created first so that
it can be used in the
next record type */
let schoolInfo: schoolInfo = {
  school: "Hogwarts",
  house: "Gryffindor"
};

type wizardInfo = {
  name: string,
  age: int,
  schoolInfo: schoolInfo
};

/* A nested record */
let wizard: wizardInfo = {
  name: "Harry",
  schoolInfo,
  age: 14
};

/* Accessing the complete wizard record */
Js.log(wizard);
```

```
/* Accessing the house field */
let {schoolInfo: {house}} = wizard;

Js.log(house);

/* Another way of accessing the house */
let house = wizard.schoolInfo.house;
Js.log(house);
```



The benefit of punning is shown in **line 23**, where we do not need to specify the type of the `schoolInfo` record again.

A strict rule of punning is that it only works on a record if that record contains **more than one field**.

That ends our discussion on nested records. We could always nest records further and further, but we will have to take the same steps that we've seen above.

In the next lesson, we'll explore the different ways of modifying records.