Pick

This lesson explains the Pick mapped type.

WE'LL COVER THE FOLLOWING

Description of `Pick`

The `Pick` mapped type is a new addition that comes with TypeScript and allows you to select a subset of type's properties in order to create a dynamic type.

If a type has five members and you need only two of them, then there are many possible patterns. Without a mapped type, you have two options. The first one is to create two different types.

```
// An interface that defines fields for an Animal
interface Animal {
  age: number;
  numberOfLegs: number;
  canSwim: boolean;
  runningSpeed: number;
  name: string;
}

// An interface that defines fields for a Fish
interface Fish {
  age: number;
  name: string;
```

```
}
```

The inheritance problem

The problem with that approach is the duplication of two members (`age` and `name`). The second option is to use inheritance.

```
// An Animal has all fields from Fish
interface Animal extends Fish{
  numberOfLegs: number;
  canSwim: boolean;
  runningSpeed: number;
}

// Fish schema
interface Fish {
  age: number;
  name: string;
}
```

Inheritance that does not make sense

Improvement of the inheritance solution

Inheritance works well, but it needs to be built smartly which was not the case in the previous block of code. This has been done on purpose to illustrate that an `Animal` is not a `Fish` but technically it helps us to not have the members repeated. A better inheritance approach for the same problem would be a better division of the entities' schema.

```
interface Animal {
  age: number;
  name: string;
}

interface Fish extends Animal {
  maximumDeepness: number;
}

interface Felin extends Animal {
  numberOfLegs: number;
  canSwim: boolean;
  runningSpeed: number;
}
```

Inheritance structured in a better way

`Pick` for a dynamic type creation

The inheritance approach is *the* approach while building your model. It is clear to understand, natural to a newcomer in the codebase to have a sense of how an entity is structured and reusable when adding a new entity that can extend the existing one.

However, if you need to dynamically create several types that are a subset of an existing type, the `Pick` mapped type is a handy tool to use. A word of caution: it is easy to abuse `Pick` by creating a gigantic general type and then picking members of it. The maintainability of your code will take a hit. It is suggested to use this only in a scenario where the selection of a member is dynamically required.

One maintainability limitation is that if the property is renamed, the string will remain intact causing transpilation to fail. Another issue is that it is harder to understand what something is by listing out key attributes than by simply naming it. It is easier to understand the concept of `Fish` than the concept of something that has `age`, `name` and `maximumDeepness`.

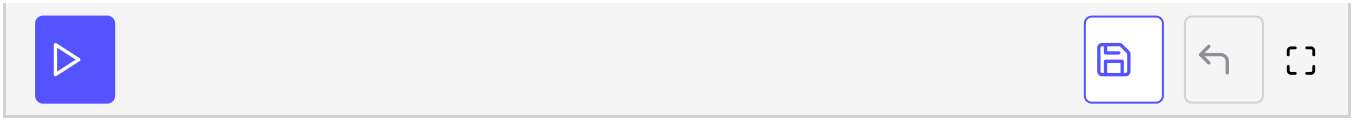
```
interface Animal {
  age: number;
  name: string;

  maximumDeepness: number;

  numberOfLegs: number;
  canSwim: boolean;
  runningSpeed: number;
}

function buyAFish(fishEntity: Pick<Animal, "age" | "name" | "maximumDeepness">) {
  console.log(fishEntity);
}

buyAFish({
  age: 1,
  name: "Clown Fish",
  maximumDeepness: 10
})
```



Getting a Fish without defining the Fish interface

`Pick` usage with `keyOf`

The code remains safe because it uses `keyOf` under the hood which limits the strings to the members' name of the type given in the first parameter. The `keyOf` is explained in detail in a later lesson. For the moment, let's see `keyOf` as a way to extract field names from a type. For example, using `keyOf Animal` would extract all the six fields of `Animal` and ensure that only these strings can be used.

You can try by writing something like "food" and TypeScript will not compile, because "food" does not exist in the `Animal` entity. As you can see, because it is taking strings, it can be dynamically adjusted.

A few people advocate avoiding the proliferation of interface and inheritance. TypeScript does not have an opinion, but provides a way to avoid interface by building a type with the picked-up property.

The following code demonstrates how the type `Fish` can be made without duplicating members from `Animal` while avoiding creating an `Interface`. Instead of *extending*, the code is *picking*.

```
// Interface with all Animal fields
interface Animal {
    age: number;
    name: string;

    maximumDeepness: number;

    numberOfLegs: number;
    canSwim: boolean;
    runningSpeed: number;
}

// Fish type built upon Animal
type Fish = Pick<Animal, "age" | "name" | "maximumDeepness">;

function buyAFish(fishEntity: Fish) {
    console.log(fishEntity);
}

buyAFish({
    age: 1,
    name: "Clown Fish",
    maximumDeepness: 10,
});
```

Defining the Fish type without duplicating fields' name

A better solution with `Pick`

The `Pick` mapped type is a quick way to extract a portion of an existing type. It is important to keep in mind that the passed in object's type is picked up, but the object itself is not changed.

In the following code, a complete `Animal` is passed down. However, TypeScript compiles fine, because it respects the return type requiring the three members. But, the output will contain every field. The reason is that TypeScript block in design type to the appropriate three members picked in the `Pick` but does not perform runtime manipulation on the object.

A more robust code would create a new object by selecting the three values. Commenting out **line 14** demonstrates that you can build an object but not with other members than the three in the return type of the function, thus TypeScript is guiding the developer to avoid a misstep.

Comment **line 13**, and adjust **line 14** by removing the member not specified in the return type (`otherStuff`). You will see that the console prints the three expected members.

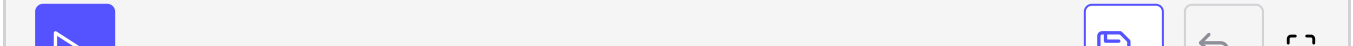
```
interface Animal {
    age: number;
    name: string;

    maximumDeepness: number;

    numberOfLegs: number;
    canSwim: boolean;
    runningSpeed: number;
}

function transformAnAnimationToAFish(fishEntity: Animal): Pick<Animal, "age" | "name" | "maxi
    return fishEntity;
    // return { age: 1, name: "name", maximumDeepness: 123, otherStuff: "no too fast" };
}

console.log(
    transformAnAnimationToAFish({
        age: 1,
        name: "Clown Fish",
        maximumDeepness: 10,
        numberOfLegs: 0,
        canSwim: true,
        runningSpeed: 0,
    })
);
```



An alternative with a big constraint

Another way to get a subset of members from an existing type is to use `Extract` and `Record`. The `Extract` mapped type takes all properties from the first generic type that are also in the second generic type.

The `Record` creates a type from the list extracted. The caveat of this approach is that every field must be from a type defined in the second argument of `Record`.

In the following example, the `LivingThing` will have all his fields to be `string`. Hence, line 19 shows the age to be a `string`.

```
interface Animal {
    age: number;
    name: string;

    maximumDeepness: number;

    numberOfLegs: number;
    canSwim: boolean;
    runningSpeed: number;
}
interface Human {
    age: number;
    name: string;
}

// Create a Type from the intersection of Animal and HUman that will be of type string
type LivingThing = Record<Extract<keyof Animal, keyof Human>, string>;
const creature: LivingThing = {
    age: "1",
    name: "John",
};
console.log(creature);
```

An alternative with a constraint