Type Checking and Interface with a Discriminator

In this lesson you will follow-up on literal type and how they can be used to discriminate a type.

WE'LL COVER THE FOLLOWING

Checking type with literal

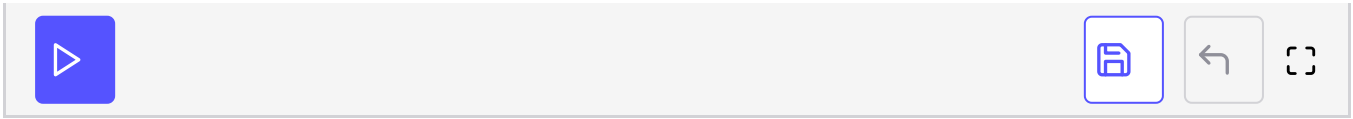
In the literal type lesson, you saw that a literal type can be used to differentiate different types. It means that you may have a function that takes a union of several types that you would like to figure out which type is being passed, to accomplish something specific to that type. You also saw the pattern of branding in a type alias that is using a discriminator in one of its patterns. As a reminder here is an example with **line 2** and **line 8** that has two string literals.

```
interface Human {
    kind: "human"; // Shared string literal
    height: number;
    weight: number;
}

interface Animal {
    kind: "animal"; // Shared string literal
    color: string;
}
type Species = Human | Animal;
```

In the code above, `Human` and `Animal` share a common member named `kind`. The name could have been anything. What is important is that they share the

same member. This is because before discriminated, TypeScript allows access to common properties. In the next example, only `kind` is available at **line 16**.

```
interface Human {
  kind: "human"; // Shared string literal
  height: number;
  weight: number;
}

interface Animal {
  kind: "animal"; // Shared string literal
  color: string;
}
type Species = Human | Animal;

let a: Human = { kind: "human", height: 72, weight: 150 };

function printData(species: Species): void {
  console.log(species.kind); // Only kind is available
}

printData(a);
```



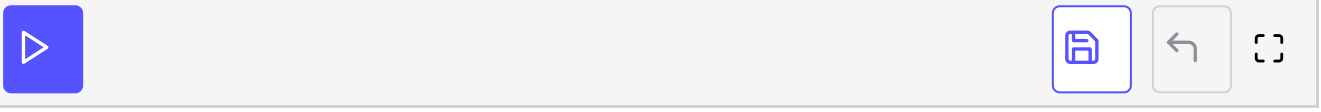
Narrowing down types

By comparing on the shared property, in this case it is the `kind` property, and creating a condition, the type narrows from the union of two to a single one giving TypeScript a hint of all the properties of the type.

In the following example, **line 17** narrows down the type to the `Human` interface. Thus, **line 18** has access to all `Human` members. Similarly, **line 19** narrows down to `Animal` and gives **line 20** all the `Animal` members.

```
interface Human {
  kind: "human";
  height: number;
  weight: number;
}

interface Animal {
  kind: "animal";
  color: string;
}
type Species = Human | Animal;

let a: Human = { kind: "human", height: 72, weight: 150 };

function printData(species: Species): void {
  console.log(species.kind); // Only kind is available
  if (species.kind == "human"){
```

```
    console.log( I am a human with the height of  ${species.height} );
  } else if (species.kind == "animal"){
    console.log(`I am an animal with the color of  ${species.color}`);

  }
}

printData(a);
```



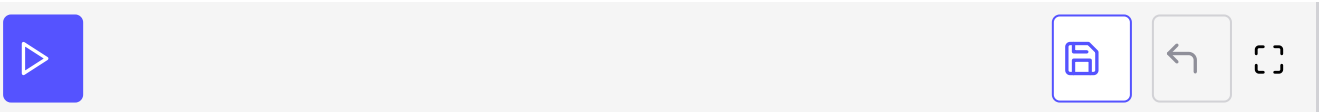
Narrowing Down Types and Inheritance

The pattern of having a string literal for each interface is not only useful for comparison as seen in a previous lesson, but also useful for narrowing down to a specific instance.

Where the discriminator will fall short is with **inheritance**. With inheritance, it is impossible to have many fields with the same name but with different types.

> Note: the below code throws an error

```
interface Human {
  kind: "human";
  height: number;
  weight: number;
}

interface Animal {
  kind: "animal";
  color: string;
}


interface SuperHumanAnimal extends Human, Animal {

}
```



The code above does not transpile. It has the error Named property `kind` of types `Human` and `Animal` are not identical. Hence, the pattern is good until the fusion of type comes into the picture. Thus, incorporating a discriminator should not be an automatic reflex but a case by case solution.