100 XP

Implement generics with custom types and classes

5 minutes

Using generics with primitive types, like `number`, `string`, or `boolean`, illustrate the concepts of generics well, but the most powerful uses come from using them with custom types and classes.

This example has a base class called `Car` and two subclasses, `ElectricCar` and `Truck`. The `accelerate` function accepts a generic instance of `Car` and then returns it. By telling the `accelerate` function that `T` must extend `Car`, TypeScript knows which functions and properties you can call within the function. The generic also returns the specific type of car (`ElectricCar` or `Truck`) passed into the function, rather than a non-specific `Car` object.

TypeScript

```
class Car {
    make: string = 'Generic Car';
    doors: number = 4;
}
class ElectricCar extends Car {
    make = 'Electric Car';
    doors = 4
}
class Truck extends Car {
    make = 'Truck';
    doors = 2
}
function accelerate<T extends Car> (car: T): T {
    console.log(`All ${car.doors} doors are closed.`);
    console.log(`The ${car.make} is now accelerating!`)
    return car
}

let myElectricCar = new ElectricCar;
accelerate<ElectricCar>(myElectricCar);
let myTruck = new Truck;
accelerate<Truck>(myTruck);
```

The output to the console is:

```bash
"All 4 doors are closed."
"The Electric Car is now accelerating!"
"All 2 doors are closed."
"The Truck is now accelerating!"
```

Using generic constraints with custom types and classes

Earlier in the module, you learned how to use generic constraints to limit types. Generic constraints can not only be applied to native types, but also to classes.

You can do this by defining an interface and then using the `extend` keyword with the type variable to extend it. The previous example constrained the `T` type by attaching a restriction to it – `T` must extend `Car`.

Next unit: Lab - Declare a class by using a generic

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆