Generic and Classes

This lesson delves into making classes more reusable with generic types.

WE'LL COVER THE FOLLOWING

- A simple `class`
- Adding an `interface` to our `class`
- Slight modification leading to mega mayhem
- Generic to the rescue

A simple `class`

Engineers with an object-oriented background may associate the concept of generic with classes. It is a mechanism to generalize a class, to avoid duplicating the definition for each flavor of a class.

```
// Three classes
class Human {
    public greeting: string = "Hello";
}
class Lion {
    public greeting: string = "Grrrrrr";
}
class Tulip {
    public greeting: string = "...";
}

// The class needs to use a specifies but is limited to Human
class LivingSpecies_1 {
    public species: Human; // Human only :(

    constructor(species: Human) {// Human only :(
        this.species = species;
    }
    public sayHello(): void {
        console.log(this.species.greeting);
    }
}
const species1 = new LivingSpecies_1(new Human());
species1.sayHello();
```

Code with limitation to be constraint to Human only

Adding an `interface` to our `class`

The code above is fine but would require creating a class `LivingSpecies_1` for each of the three species we have (Human, Lion, Tulip). It does not scale well if the number of classes increases and is not very shareable outside the scope of the application where the class is created.

A step towards reusability is to have an interface that is shared among the species and used in the common class.

```
// Three classes that inherit a common one
interface Greeter {
    greeting: string;
}
class Human implements Greeter { // Implement the common interface
    public greeting: string = "Hello";
}
class Lion implements Greeter {// Implement the common interface
    public greeting: string = "Grrrrrr";
}
class Tulip { // Does not implement the common interface
    public greeting: string = "...";
}

// Not limited to Human! Now any type that inherit Greeter
class LivingSpecies {
    public species: Greeter;

    constructor(species: Greeter) {
        this.species = species;
    }
    public sayHello(): void {
        console.log(this.species.greeting);
    }
}
const species1 = new LivingSpecies(new Human());
species1.sayHello();
const species2 = new LivingSpecies(new Lion());
species2.sayHello();
const species3 = new LivingSpecies(new Tulip());
species3.sayHello();
```

The code above is working, which might be surprising because `Tulip` does not implement the `Greeter` interface. TypeScript is **structural** based, and not

nominal, which means it does not rely on the name but on the signature.

`Tulip` respect the *contract* of having a `greeting` member, hence it transpiles without error. Hence, in that particular case, it works without generic. However, a small modification can cause mayhem.

Slight modification leading to mega mayhem

```
interface Greeter {
    greeting: string;
}
class Human implements Greeter {
    public humanFunction() {
        console.log("Executing a human function");
    }
    public greeting: string = "Hello";
}
class Lion implements Greeter {
    public greeting: string = "Grrrrrr";
    public lionAttack() {
        console.log("Attacking like a lion");
    }
}
class Tulip {
    public greeting: string = "...";
    public lookingGood() {
        console.log("Snazzy Flower am I");
    }
}
class LivingSpecies {
    public species: Greeter;

    constructor(species: Greeter) {
        this.species = species;
    }
    public getSpecies() {
        return this.species;
    }
}
const species1 = new LivingSpecies(new Human());
species1.getSpecies().humanFunction();
const species2 = new LivingSpecies(new Lion());
species2.getSpecies().lionAttack();
const species3 = new LivingSpecies(new Tulip());
species3.getSpecies().lookingGood();
```

The code does not compile

The code above does not work. The function does not exist in the type `Greeter` when we try to access the type passed by the constructor again. The reason is that we are passing the type `Greeter` and underneath, while it has a more narrow type of `Human` or `Lion` or `Tulip`, TypeScript follows what is specified to be an object of type `Greeter`. When returned, the `Greeter` type is returned because the notion of a more specific type is gone.

Generic to the rescue

If we modify the code again, only this time instead of relying on an interface we rely on declaring the member of a generic type, this generic type will be returned and the initial type will be recovered since it follows along its journey in the class.

```
interface Greeter {
    greeting: string;
}
class Human implements Greeter {
    public humanFunction() {
        console.log("Executing a human function");
    }
    public greeting: string = "Hello";
}
class Lion implements Greeter {
    public greeting: string = "Grrrrrr";
    public lionAttack() {
        console.log("Attacking like a lion");
    }
}
class Tulip {
    public greeting: string = "...";
    public lookingGood() {
        console.log("Snazzy Flower am I");
    }
}
class LivingSpecies<T> {
    public species: T;

    constructor(species: T) {
        this.species = species;
    }
    public getSpecies() {
        return this.species;
    }
}
const species1 = new LivingSpecies(new Human());
species1.getSpecies().humanFunction();
const species2 = new LivingSpecies(new Lion());
species2.getSpecies().lionAttack();
const species3 = new LivingSpecies(new Tulip());
species3.getSpecies().lookingGood();
```

As you can see, the change is to the class definition, where we add `<T>` specifying that it will hold a generic type `T`. The name `T` could be anything. Every reference to `T` in the class refers to the type passed in the constructor, **line 25**. Then, the variable is stored at **line 23** also as the type `T` and returned by `getSpecies`. Once out, on **lines 33, 35 and 37**, the variable's type is the actual one passed by parameter during the initialization of each `LivingSpecies`.

Generic can be used by classes in many other different ways. We are not limited to a single `<T>` and can have generic functions that introduce their own generic parameter as we will see in an upcoming lesson.