Generic

This lesson introduces generic types.

WE'LL COVER THE FOLLOWING

Introduction

Generic allows for improved reusability by parameterizing a type with another one. A popular front-end framework named React can have an unlimited number of components. They all have properties that are unique to their components. It would be tedious to have to inherit a base class or to have to alter the framework to accommodate all potentials possibilities. Hence, the properties of these React components use a generic.

```
// Generic Component that has properties that can change depending of the implementation
interface MyComponent<TProps> {
  name: string;
  id: number;
  props: TProps;
}

// First property that has a string
interface Props1 {
  color: string;
}

// Second property that has a number
interface Props2 {
  size: number;
}

// First component that has color in property because it is generic with Props1
const component1: MyComponent<Props1> = {
  name: "My Component One",
  id: 1,
  props: { color: "red" }
};

// Second component that has size in property because it is generic with Props2

const component2: MyComponent<Props2> = {
  name: "My Component Two",
  id: 2,
  props: { size: 100 }
};

console.log(component1);
console.log(component2);
```

Line 19 and **line 26** are where the generic component takes what type of contract will be passed along to the property `props`. The type is flexible and can scale to a future component by filling any type between `<>` to specify the expected type for `props`.

Generic and List

One of the famous examples is a list. With generic, you can define a variable that will be a list of a specific type, let's say numbers. You can develop a list class that will handle any type, and with generic, the consumer of your class can define what will be in the list without changing the core logic of the list.

Generic is not required in JavaScript or even before TypeScript implemented the notion of generic since you can use the loose type `any`.

```
let list1: number[] = [1, 2, 3];
list1.push(4); // Can only push number
console.log(list1);

let list2: any[] = [1, 2, 3];
list2.push("Here_is_a_string");
console.log(list2); // You can push any type of value
```

The drawbacks of `any` are that you cannot constrain the type passed to your class to have a minimal definition, and that you will have to cast back once the data once it is extracted from your class.

In the previous example, **line 2** and **line 6** are pushing data in two different arrays. The first one uses a generic version with `number[]` which is identical to

`Array<number>` while the second use `any[]`. The difference is that the first list leverages the generic to specify that only the type `number` can be manipulated.

Generic vs Any

Here is a quick overview of how generic can be used in an array instead of `any`. The code does not require conversion from one type to another, like the usage of `any` would require, and all members of the array can be accessed as strings. This is because it is not `any` type, but marked as `string` due to the declaration at **line 1**. At **line 3** it is possible to directly access `substr` that belongs to `string`.

```
const a: Array<string> = new Array("abc", "def");
const s: string = a[0]; // No cast required
console.log(s.substr(0,1)); // Access to string members
```

The same example with `any` will need a cast to have TypeScript helping us to find the error that on **line 3** the `substringg` has two *g* which is a mistake.

```
const a2: Array<any> = new Array("abc", "def");
const s2 = a2[0]; // No cast required
console.log(s2.substringg(0, 1)); // TypeScript does not safe guard
```

The code has an error that TypeScript cannot help because the array is not generic