Enum With and Without Values

In this lesson, you will discover how to use an enum with explicit and implicit values.

We'll cover the following

The role of `enum`

An `enum` is a structure that proposes several allowed values for a variable. It is a way to constrain variable values by defining specific possible entries.

`enum` with values#

`enum` can be of `string` type. In that case, every member requires a value without exception.

```
enum MyStringEnum {
    ChoiceA = "A",
    ChoiceB = "B",
}
```

A mixed `enum` value type is acceptable if every member is defined. For

example, you can have one item be an integer and another be a string type.

It is recommended not to mix types since it might be more confusing than pragmatic.

```
enum MyStringAndNumberEnum {
    ChoiceA, // 0
    ChoiceB = "B",
    ChoiceC = 100
}
```

`enum` without values#

`enum` is a type that enforces a limited and defined group of constants. `enum` must have a name and accepted values. Afterward, you can use the `enum` as a type. The consumer must use the `enum` with its name followed by a dot and a potential value from the defined list.

```
enum MyEnum {
    ChoiceA,
    ChoiceB,
    ChoiceC,
}
let x: MyEnum = MyEnum.ChoiceA;
console.log(x);
```

The values are all constants starting from 0 for the first item and increasing by one until the end. This type of `enum` has **implicit** value. Developers can specify a specific value by equating to an integer. In that case, the `enum` is **explicit**.

```
enum MyEnum {
    ChoiceA,
    ChoiceB,
```

```
    ChoiceC,
}

enum MyEnum2 {
    ChoiceA, // 0
    ChoiceB = 100, // 100
    ChoiceC, // 101
    ChoiceD = MyEnum.ChoiceC, // 2
}
console.log(MyEnum2.ChoiceA);
console.log(MyEnum2.ChoiceB);
console.log(MyEnum2.ChoiceC);
console.log(MyEnum2.ChoiceD);
```

`enum` members' values can be set directly or by using computation. There are two types of computation:

1. a constant one
2. a purely computed one

A computed constant is a value provided by another `enum` or a value computed by addition, subtraction, bitwise, modulo, multiplication, division, "or," "and," "xor" operator, or complement operator (~). Purely computed values come from a **function**.

`enum` generates a function in JavaScript with a set that allows us to specify the number or name used to access the value. Here is the output of the two previously studied `enum`:

```
/*
enum MyEnum {
    ChoiceA,
    ChoiceB,
    ChoiceC,
} */
// Became in JavaScript:

var MyEnum;
(function (MyEnum) {
    MyEnum[MyEnum["ChoiceA"] = 0] = "ChoiceA";
```

```
    MyEnum[MyEnum["ChoiceB"] = 1] = "ChoiceB";
    MyEnum[MyEnum["ChoiceC"] = 2] = "ChoiceC";
})(MyEnum || (MyEnum = {}));

/*
enum MyEnum2 {
    ChoiceA, // 0
    ChoiceB = 100, // 100
    ChoiceC, // 101
    ChoiceD = MyEnum.ChoiceC, // 2
}
*/

//Because in JavaScript
var MyEnum2;
(function (MyEnum2) {
    MyEnum2[MyEnum2["ChoiceA"] = 0] = "ChoiceA";
    MyEnum2[MyEnum2["ChoiceB"] = 100] = "ChoiceB";
    MyEnum2[MyEnum2["ChoiceC"] = 101] = "ChoiceC";
    MyEnum2[MyEnum2["ChoiceD"] = 2] = "ChoiceD";
})(MyEnum2 || (MyEnum2 = {}));
```

`enum` with bitwise values

`enum` is a good candidate for *bitwise operations* since the value can be explicitly set (value set during the definition of the `enum`) and you can use the bit shift operator. Once defined, you can use it as any variable to determine if it contains the one you need or use the ampersand (`&`) to check if the one you want is present. The pipe symbol (`|`) lets you add many `enum` choices to a variable.

The following code not only initializes the value with the `|` but also checks the value. With bitwise, we cannot directly use an equal sign. The reason is that bitwise operation returns a number, not a boolean. Hence, we need to compare the number to the desired comparison value. Line 10 demonstrates how to check the value of an `enum`.

```
enum Power {
    None = 0, // Value 0 in decimal (00 in binary)
```

```
    Invincibility = 1 << 0, // Value 1 in decimal (01 in binary)
    Telepathy = 1 << 1, // Value 2 in decimal (10 in binary)
    Invisibility = 1 << 2, // Value 3 in decimal (11 in binary)
    Everything = Invincibility | Telepathy | Invisibility,
}
let power: Power = Power.Invincibility | Power.Telepathy;
console.log("Power values:" + power);
if (Power.Telepathy === (power & Power.Telepathy)) {
    console.log("Power of telepathy available");
}
```

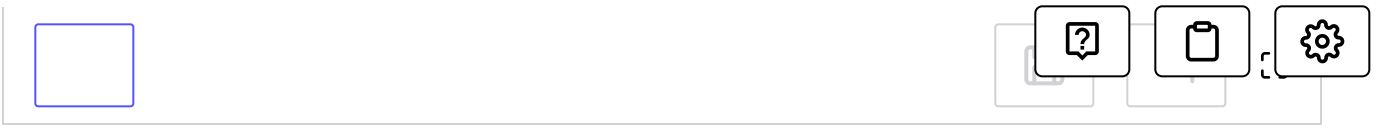
The value of the previous example is 3 because the `Invincibility` value is `1<<2`, which is binary `10`.

The `Telepathy` value is `1<<1` which gives the binary `01` and the `or` operation provided by the pipe symbol gives binary `11` which is `3`.

It is possible to remove a value from a bitwise `enum` on the fly by using `&= ~` which performs an `and` operation on the inverse of the value.

For example, the following code supplements the previous example by removing the `Telepathy` power. Line 13 has the remove operation.

```
enum Power {
    None = 0, // Value 0 in decimal (00 in binary)
    Invincibility = 1 << 0, // Value 1 in decimal (01 in binary)
    Telepathy = 1 << 1, // Value 2 in decimal (10 in binary)
    Invisibility = 1 << 2, // Value 3 in decimal (11 in binary)
    Everything = Invincibility | Telepathy | Invisibility,
}
let power: Power = Power.Invincibility | Power.Telepathy;
console.log("Power values:" + power);
if (Power.Telepathy === (power & Power.Telepathy)) {
    console.log("Power of telepathy available");
}
power &= ~Power.Telepathy;
console.log("Power values:" + power);
if (Power.Telepathy === (power & Power.Telepathy)) {
    console.log("Power of telepathy available");
}
```



The value is `1` because from the `3`, (which is in binary `11`) you use `and` of the inverse of `10` which is `01`. `11 and 01 = 01` which is 1.

Adding a value on the fly uses the pipe as when we initialized the value. Line 18 shows that not only can you use `Power.Everything` to set all the values of the `enum`, but we can also directly use a number that represents the binary of the values. In that case, `111` sets the first three powers to true.

```
enum Power {
    None = 0,
    Invincibility = 1 << 0,
    Telepathy = 1 << 1,
    Invisibility = 1 << 2,
    Everything = Invincibility | Telepathy | Invisibility,
}
let power: Power = Power.Invincibility | Power.Telepathy;
console.log("Power values:" + power);
if (power & Power.Telepathy) {
    console.log("Power of telepathy available");
}
power &= ~Power.Telepathy;
console.log("Power values:" + power);
if (power & Power.Telepathy) {
    console.log("Power of telepathy available");
}
power |= 111;
console.log("Power values:" + power);
if (power & Power.Everything) {
    console.log("Everything");
}
```

Great, now that we've covered the two types of `enum`, let's see how to access `enum` values in the next lesson.

Back

Quiz

Accessing Enum Values

Mark as Completed

Report an Issue