enum

This lesson explains enum, its use and properties.

WE'LL COVER THE FOLLOWING

`enum`

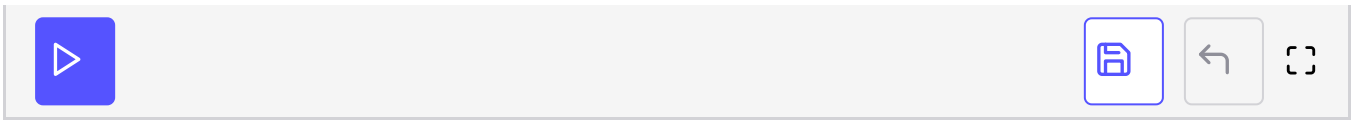
`enum` is the feature that enables defining named constant values.

Effects of magic constants on code quality

Let's have a look at this code:

```
import std.stdio;

void main() {

    int first = 16;
    int second = 8;
    int result;

    int operation = 2;

    if (operation == 1) {
    result = first + second;

    } else if (operation == 2) {
    result = first - second;

    } else if (operation == 3) {
    result = first * second;
```

```
    } else if (operation == 4) {
    result = first / second;
    }

    writeln(result);

}
```



The integer literals `1`, `2`, `3` and `4` in the code above are called **magic constants**. It is not easy to determine what each of those literals mean in the program. One must examine the code in each scope to determine that 1 means addition, 2 means subtraction and so on. This task is relatively easy for the above code because all of the scopes contain just a single line. It would be considerably more difficult to decipher the meanings of magic constants in longer and complex programs.
Magic constants must be avoided because they impair two important qualities of source code:

- readability
- maintainability

`enum` enables names to be given to these constants and, consequently, makes the code more readable and maintainable. Each condition would be readily understandable if the following `enum` constants were used:

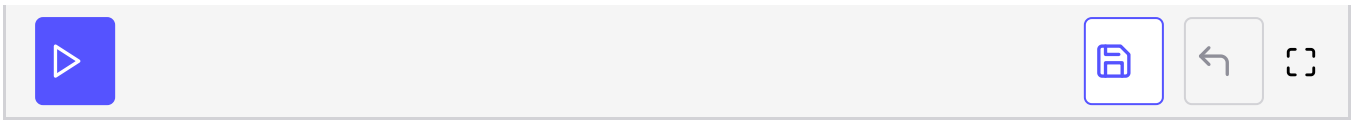
```
if (operation == Operation.add) {
    result = first + second;

} else if (operation == Operation.subtract) {
    result = first - second;

} else if (operation == Operation.multiply) {
    result = first * second;

} else if (operation == Operation.divide) {
    result = first / second;
}
```

The `enum` type `Operation` used above obviates the need for magic constants 1, 2, 3 and 4, and can be defined like this:

2, 3 and 4, and can be defined like this:

```
enum Operation { add = 1, subtract, multiply, divide }
```

```
import std.stdio;

void main() {

    enum Operation { add = 1, subtract = 2, multiply = 3, divide  = 4}

    int first = 16;
    int second = 8;
    int result;

    int operation = 2;

    if (operation == Operation.add) {
    result = first + second;

    } else if (operation == Operation.subtract) {
        result = first - second;

    } else if (operation == Operation.multiply) {
        result = first * second;

    } else if (operation == Operation.divide) {
        result = first / second;
    }

    writeln(result);
}
```



The `enum` syntax

`enum` is commonly defined as follows:

```
enum TypeName { ValueName_1, ValueName_2, /* etc. */ }
```

Sometimes it is necessary to specify the actual type (the base type) of the values as well:

```
enum TypeName : base_type { ValueName_1, ValueName_2, /* etc. */ }
```

We will see how this is used in the next section.

TypeName defines what the constants collectively mean. All the member

constants of an `enum` type are listed within curly brackets. Here are some examples:

```
enum HeadsOrTails { heads, tails }
enum Suit { spades, hearts, diamonds, clubs }
enum Fare { regular, child, student, senior }
```

Each set of constants becomes part of a type specified by TypeName. For example, `heads` and `tails` become members of the type `HeadsOrTails`. The new type can now be used like other fundamental types when defining variables:

```
HeadsOrTails result; // default initialized
auto ht = HeadsOrTails.heads; // inferred type
```

When `enum` is used to define variables for e.g., Operation op, then we could also use op.add or op.subtract in the if condition. As has been seen in the pieces of code above, the members of `enum` types are always specified by the name of their `enum` type:

```
if (result == HeadsOrTails.heads) {
    // ...
}
```

Actual values and base types

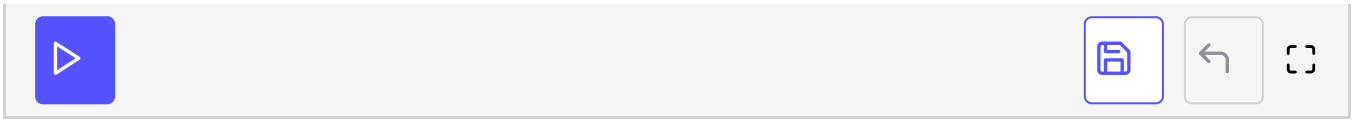
The member constants of `enum` types are implemented as `int` values by default. In other words, although on the surface they appear as just names (such as heads and tails), they also have numerical values.

> **Note:** It is possible to choose a type other than `int` when needed.

Unless explicitly specified by the programmer, the numerical values of `enum` members start at 0 and are incremented by one for each member. For example, the two members of the `HeadsOrTails` have numerical values 0 and 1:

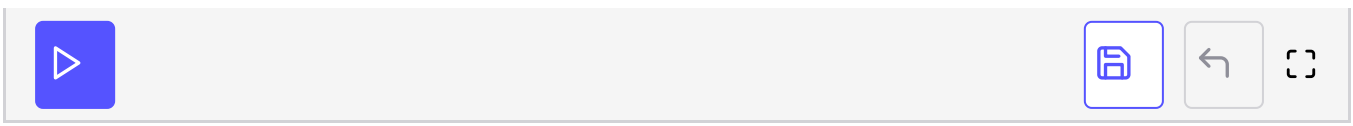
```
import std.stdio;

void main() {
```

```
    enum HeadsOrTails { heads, tails }

    writeln("heads is 0: ", (HeadsOrTails.heads == 0));
    writeln("tails is 1: ", (HeadsOrTails.tails == 1));

}
```



It is possible to manually (re)set the values at any point in the `enum`. That was the case when we specified the value of `Operation.add` as `1` above. The following example manually sets a new value twice:

```
import std.stdio;

void main() {

    enum Test { a, b, c, ç = 100, d, e, f = 222, g, ğ }
    writefln("%d %d %d", Test.b, Test.ç, Test.ğ);

}
```



If `int` is not suitable as the base type of the `enum` values, the base type can be specified explicitly after the name of the `enum`:

```
enum NaturalConstant : double { pi = 3.14, e = 2.72 }
enum TemperatureUnit : string { C = "Celsius", F = "Fahrenheit" }
```

`enum` values that are not of an `enum` type

We have discussed that it is important to avoid magic constants and instead to take advantage of the `enum` feature.
However, sometimes it may not be natural to come up with `enum` type names just to use named constants. Let's assume that a named constant is needed to represent the number of seconds per day. It should not be necessary to also define an `enum` type to contain this constant value. All that is needed is a constant value that can be referred to by its name. In such cases, the type name of the `enum` and the curly brackets are omitted:

```
enum secondsPerDay = 60 * 60 * 24;
```

The type of the constant can be specified explicitly, which is required if the type cannot be inferred from the right-hand side:

```
enum int secondsPerDay = 60 * 60 * 24;
```

Since there is no `enum` type to refer to, such named constants can be used in code simply by their names:

```
totalSeconds = totalDays * secondsPerDay;
```

`enum` can be used for defining named constants of other types as well. For example, the type of the following constant would be `string`:

```
enum fileName = "list.txt";
```

Such constants are `rvalues`, and they are called **manifest constants**.
It is possible to create manifest constants of arrays and associative arrays as well. However, as we will see later in the immutability chapter, `enum` arrays and associative arrays may have hidden costs.

Properties

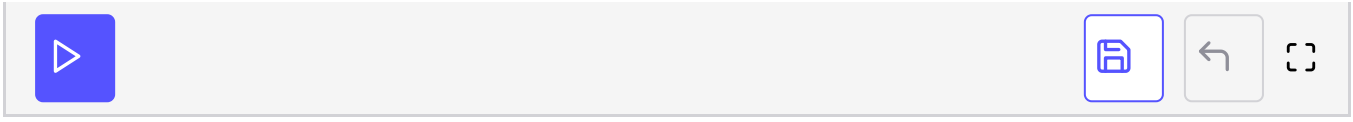
The `.min` and `.max` properties are the minimum and maximum values of an `enum` type. When the values of the `enum` type are consecutive, they can be iterated over in a `for` loop within these limits:

```
enum Suit { spades, hearts, diamonds, clubs }

for (auto suit = Suit.min; suit <= Suit.max; ++suit) {
    writefln("%s: %d", suit, suit);
}
```

Format specifiers `"%s"` and `"%d"` produce different outputs:

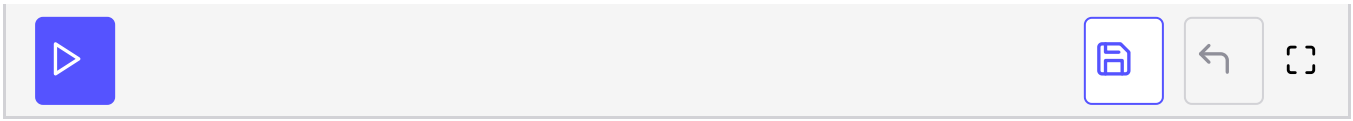
```
import std.stdio;

void main(){

    enum Suit { spades, hearts, diamonds, clubs }
```

```
    for (auto suit = Suit.min; suit <= Suit.max; ++suit) {
        writefln("%s: %d", suit, suit);


    }
}
```



Note that a `foreach` loop over that range would leave the `.max` value out of the iteration:

```
import std.stdio;

void main(){

    enum Suit { spades, hearts, diamonds, clubs }

    foreach (suit; Suit.min .. Suit.max) {
        writefln("%s: %d", suit, suit);
    }
}
```

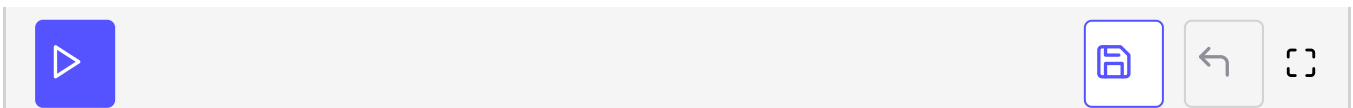


The output:

```
spades: 0
hearts: 1
diamonds: 2
           ← clubs is missing
```

For that reason, a correct way of iterating the overall values of an `enum` is using the `EnumMembers` template from the `std.traits` module:

```
import std.traits;
import std.stdio;

void main() {
    enum Suit { spades, hearts, diamonds, clubs }

    foreach (suit; EnumMembers!Suit) {
      writefln("%s: %d", suit, suit);
    }
}
```



> Note: The `!` character above is for *template instantiations*.

```
spades: 0
hearts: 1
diamonds: 2
clubs: 3       ← clubs is present
```

Converting from the base type

As we saw in the formatted outputs above, an `enum` value can automatically be converted to its base type (e.g. to `int`). The reverse conversion(`int` to `enum`) is not automatic:

```
Suit suit = 1; // ← compilation ERROR
```

One reason for this is to avoid ending up with invalid `enum` values:

```
suit = 100; // ← would be an invalid enum value
```

The values that are known to correspond to valid `enum` values of a particular `enum` type can still be converted to that type by an explicit type cast:

```
suit = cast(Suit)1; // now hearts
```

It would be the programmer's responsibility to ensure the validity of the values when an explicit cast is used.

In the next lesson, you will find a quiz based on the concepts of literals and enum.