

# Monoid

Lucerne, July 2021

Ivo Kilchmann

Alcor Training

# Monoid – What's that?

Math?

Functional Programming???

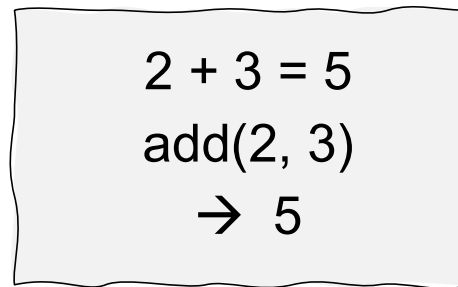
# Monoid $\Leftrightarrow$ Monad

It's about ...

Composability  
Pattern + Structure  
Rules + Benefits

Binary operation:

$f(x, y)$   
Combining  
«Closure»: same type



$2 + 3 = 5$   
 $\text{add}(2, 3)$   
 $\rightarrow 5$

## Associativity:

Rearranging the parentheses in an expression will not change the result

$$(2 + 3) + 5 = 2 + (3 + 5)$$
$$\text{add}(\text{add}(2, 3), 5) = \text{add}(2, \text{add}(3, 5))$$

Identity element:

Neutral element  
No effect when applying  
Is member of the set

**0** in addition  
 $3 + \mathbf{0} = 3$   
 $\text{add}(3, \mathbf{0}) = 3$

What's the profit for the effort?



## Some Benefits:

Pattern  
reduce / fold  
Parallel computing

```
// non-recursive
reduce(fn, id, list) {
  result = id
  for (e in list) {
    result = fn(result, e)
  }
  return result
}

reduce(add, 0, [2, 3, 5])
10
reduce(multiply, 1, [2, 3, 5])
25
```

# Recap

- You start with a bunch of things, *and* some way of combining them two at a time.
- **Rule 1 (Closure)**: The result of combining two things is always another one of the things.
- **Rule 2 (Associativity)**: When combining more than two things, which pairwise combination you do first doesn't matter.
- **Rule 3 (Identity element)**: There is a special thing called “zero” such that when you combine any thing with “zero” you get the original thing back.

Source: Scott Wlaschin, F# for fun and profit

# OK, but wait...

Functional Programming!?  
Math?

What about OO?

## Monoid in OO:

Binary operation  
instance.method(instance)

Just showing  
the idea.  
Avoid such an  
interface

```
// not recommended!
Interface Monoid<T> {
    abstract T combine(T other)
    abstract T neutralElement()

    default T reduce(T list) {
        result = neutralElement()
        for (e in list) {
            result = result.combine(e)
        }
    }
}
```

## java.util.String

```
String result = "to" + "get" + "her";
```

```
String result = "to".concat("get").concat("her");
```

- Closure

```
class String {  
    public String concat(String str);  
    ...  
}
```

- Assoziative

```
assertEquals(  
    "to".concat("get").concat("her"),  
    "to".concat(("get").concat("her"))  
);
```

- Neutral element

```
String empty = "";  
assertEquals("to" + empty, "to");
```

# Monoid in Domain-driven Design:

Value Object?!

Immutability

Change forces new instance

«Closure of Operations»

## Some Ideas

List concatenation

(btw: not in Java...)

Ranges expanding ranges

$[7..19] + [3..11] \rightarrow [3..19]$

Vector

Money

Maybe/Option

...

In Event Sourcing

Some of your domain objects...

## References

- Scott Wlaschin (F# for Fun and Profit)
- Cyrille Martraire (Monoids in Domain Modelling)
- Wikipedia (Associative property)
- Javadoc (String)
- Eric Evans (Closure of Operations)



