

# Grundlegende Konzepte von GO anhand eines RPN Taschenrechners

Lukas Gröning



# Agenda

## 1. Basics von GO

- Überblick
- Variablen, Funktionen, Kontrollstrukturen
- Datentypen (Arrays, Slices, Structs, Types)
- Reciever Funktionen

## 2. RPN Taschenrechner

- Live Demo
- Logik Aufbau
- Code Einblick

# Überblick

- Von Google (2009)
- Produktivität und Effizienz maximieren
- Kombiniert Merkmale der objektorientierten Programmierung und der funktionalen Programmierung

**Einfachheit**

**Nebenläufigkeit**

**Effizienz**

# Hello World

```
package main

import "fmt"

func main() {
    fmt.Print("Welcome to the RPN Calculator")
}
```

# Variables

```
var message string  
message = "Welcome to the RPN Calculator"
```

# Variables

```
var message string  
message = "Welcome to the RPN Calculator"
```

Tells go we're  
about to declare  
a variable

Sets the name  
of the variable

Sets the type of the  
variable

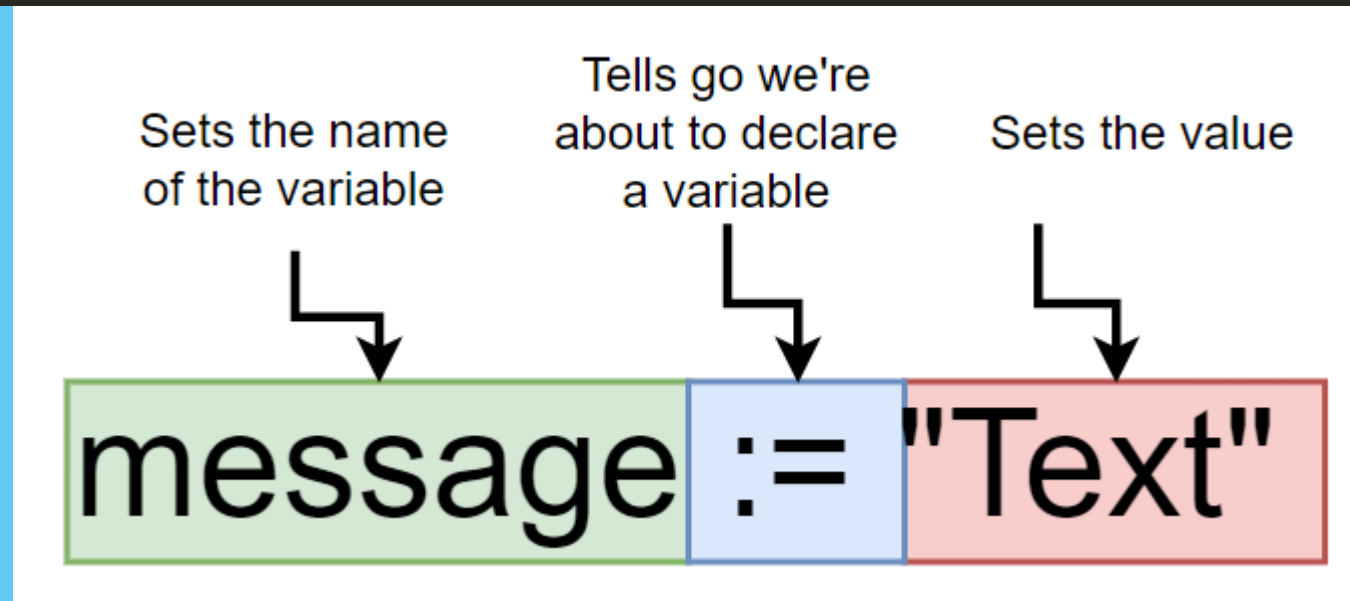
var message string

# Variables

```
message := "Welcome to the RPN Calculator"
```

# Variables

```
message := "Welcome to the RPN Calculator"
```

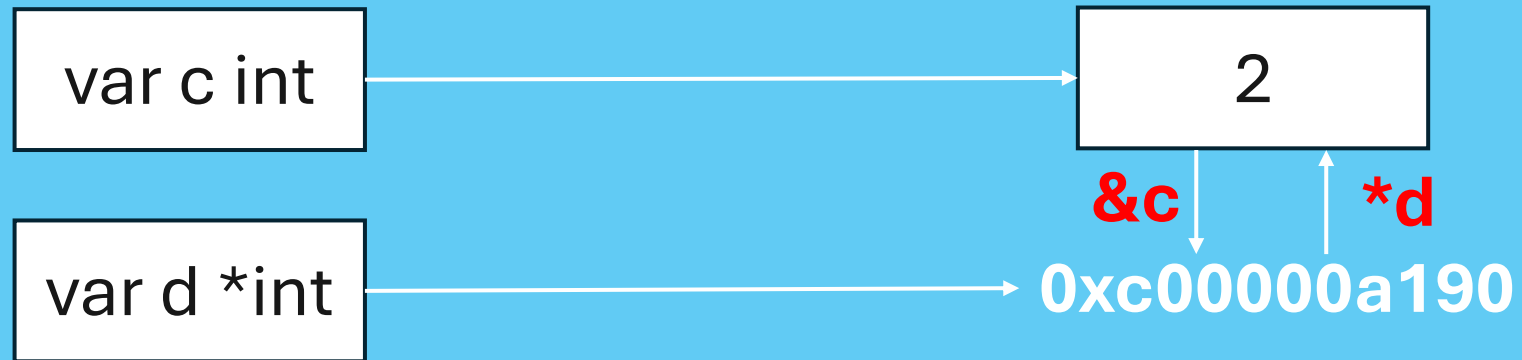


## Typinferenz & Starke Typisierung



# Variables (Pointer)

```
var c int = 2
var d *int = &c
fmt.Println(*d) // Ausgabe: 2
fmt.Println(d) // Ausgabe: 0xc00000a190
```



# Kontrollstrukturen

- If
- Schleifen
  - Iterative for Schleife
  - For each Schleife
- Switch case

# If

```
if x > 0 {  
    fmt.Println("x is positive")  
} else {  
    fmt.Println("x is not positive")  
}
```

# For Schleife

```
for i := 0; i < 10; i++ {  
    fmt.Println(i)  
}
```

# For Schleife

```
for i := 0; i < 10; i++ {  
    fmt.Println(i)  
}
```

```
count := 0  
  
// Verwendung einer for-Schleife als while-Schleife  
for count < 5 {  
    fmt.Println("Count is:", count)  
    count++ // Erhöhung der Zählvariablen  
}
```

# Endlos Schleife

```
for {  
    numberStack.Print()  
  
    input := getInputByScan()  
  
    checkInput(input, &numberStack, &latex, &history)  
}
```

# For each Schleife

```
for index, value := range s {  
    fmt.Println("value: ",value, "; at index: ",index)  
}
```

# Switch case

```
switch input {  
  case "+", "-", "*", "/", "^":  
    c.performBinaryOperation(input)  
  case "abs", "sqrt", "log", "!":  
    c.performUnaryOperation(input)  
  case "++", "**":  
    c.performMultiOperation(input)  
  default:  
    c.handleNumberInput(input)  
}
```

Ohne Fallthrough Logik!



# Funktionen

```
func add(a int, b int) int {  
    return a + b  
}
```

# Funktionen

```
func add(a int, b int) int {  
    return a + b  
}
```

Declaring a  
function

Function Name

List of arguments

Return value(s)

The diagram illustrates the components of the function declaration `func add(a int, b int) int {`. Each part is enclosed in a colored box with an arrow pointing to it from a label above:

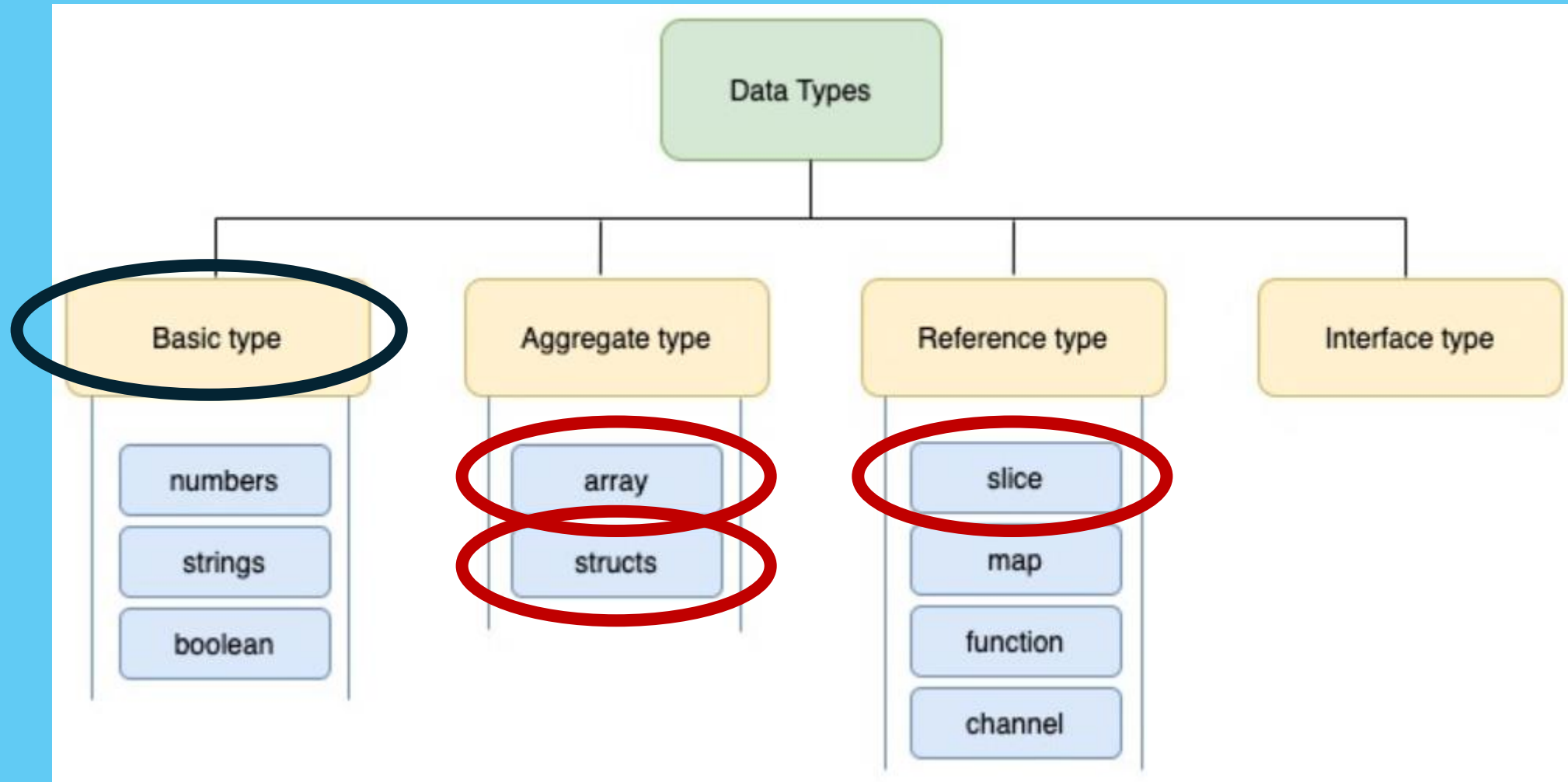
- `func` is in a light blue box, labeled "Declaring a function".
- `add` is in a light green box, labeled "Function Name".
- `(a int, b int)` is in a light yellow box, labeled "List of arguments".
- `int` is in a light purple box, labeled "Return value(s)".

The opening curly brace `{` is at the end of the first line. Below the opening brace is a large light orange box representing the function body. An arrow points from the text "Function body. Calling the function runs this code" to this box. The closing curly brace `}` is on a separate line below the body box.

# Funktionen (Error Handling)

```
func getInput() string {  
    var input string  
    _, err := fmt.Scan(&input)  
    if err != nil {  
        fmt.Println("Error while reading input: ", err.Error())  
        os.Exit(1)  
    }  
    return input  
}
```

# Datentypen



[https://miro.medium.com/v2/resize:fit:672/1\\*woZaBaFrmDR6N-RLNEjnvq.png](https://miro.medium.com/v2/resize:fit:672/1*woZaBaFrmDR6N-RLNEjnvq.png)

# Arrays

- **Statisch**
- **Feste Größe**
- **Nur ein Datentyp**
- **Wertebasiert**

# Arrays

- Statisch
- Feste Größe
- Nur ein Datentyp
- Wertebasiert

```
numbers := [3]float64{1, 2, 3}
```

# Arrays

- Statisch
- Feste Größe
- Nur ein Datentyp
- Wertebasiert

```
numbers := [3]float64{1, 2, 3}
```

```
numbers := [a]float64{}
```

Nicht dynamisch!

# Arrays

- Statisch
- Feste Größe
- Nur ein Datentyp
- Wertebasiert

```
numbers := [3]float64{1, 2, 3}
```

```
numbers := [a]float64{}
```

Nicht dynamisch!

```
numbers[0] = 1
```

Zugriff



# Arrays

vs.

# Slice

- Statisch
- Feste Größe
- Nur ein Datentyp
- Wertebasiert

- Dynamisch
- Flexible Größe
- Nur ein Datentyp
- Referenzbasiert

# Slices

## Erstellen von Slices

```
// Slice erstellen  
numbers := []int{1, 2, 3, 4, 5}    // Slice mit 1, 2, 3, 4, 5  
numbers2 := make([]int,5)         // Slice mit 0, 0, 0, 0, 0
```

# Slices

```
// Slice erstellen  
numbers := []int{1, 2, 3, 4, 5} // Slice mit 1, 2, 3, 4, 5  
numbers2 := make([]int, 5)      // Slice mit 0, 0, 0, 0, 0
```

## Append

```
// Hinzufügen eines Elements  
numbers = append(numbers, 6)  
fmt.Println("Nach Hinzufügen 6:", numbers) // Ausgabe: Nach Hinzufügen 6: [1 2 3 4 5 6]
```

# Slices

```
// Slice erstellen  
numbers := []int{1, 2, 3, 4, 5} // Slice mit 1, 2, 3, 4, 5  
numbers2 := make([]int, 5)      // Slice mit 0, 0, 0, 0, 0
```

```
// Hinzufügen eines Elements  
numbers = append(numbers, 6)  
fmt.Println("Nach Hinzufügen 6:", numbers) // Ausgabe: Nach Hinzufügen 6: [1 2 3 4 5 6]
```

## Slicing

```
// Slicing [start:end]  
numbers = numbers[1:4]  
fmt.Println("Nach slicing [1:4]: ", numbers) // Ausgabe: Nach slicing [1:4]: [2 3 4]
```

# Slices

```
func main() {  
    // Slice erstellen  
    numbers := []int{1, 2, 3, 4, 5}  
  
    // Zugriff auf Elemente  
    fmt.Println("Erstes Element:", numbers[0]) // Ausgabe: Erstes Element: 1  
  
    // Slice ausgeben  
    fmt.Println("Alle Zahlen:", numbers)  
  
    // Hinzufügen eines Elements  
    numbers = append(numbers, 6)  
    fmt.Println("Nach Hinzufügen 6:", numbers) // Ausgabe: Nach Hinzufügen 6: [1 2 3 4 5 6]  
  
    // Länge ausgeben  
    fmt.Println("Länge:", len(numbers)) // Ausgabe: Länge: 6  
}
```

# Structs & Types

```
type Stack struct {  
    items []string  
}
```

# Receiver Funktionen

```
type Stack struct {  
    items []string  
}  
  
// Pushes a new element on the Stack  
func (s *Stack) Push(item string) {  
    s.items = append(s.items, item)  
}  
  
// Pops the first element from the Stack  
func (s *Stack) Pop() (string, error) {  
    if len(s.items) == 0 {  
        return "", fmt.Errorf("Stack ist leer")  
    }  
    lastIndex := len(s.items) - 1  
    item := s.items[lastIndex]  
    s.items = s.items[:lastIndex]  
    return item, nil  
}
```

# Receiver Funktionen

*function receiver*  
↓  
`func (s *Stack)`

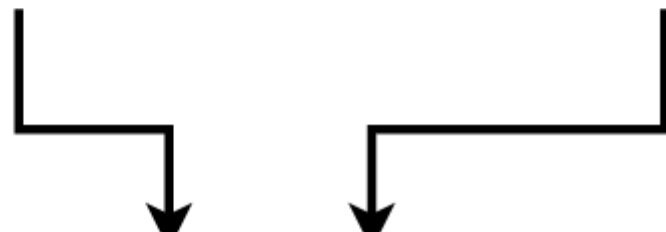
*function name*  
↓  
`push(item string) {`  
`s.items = append(s.items, item)`  
`}`



# Receiver Funktionen

*Reference to Stack  
variable, available in  
function as variable 's'*

*Every variable of  
type Stack can call  
this function*



```
func (s *Stack) push(item string) {  
    s.items = append(s.items, item)  
}
```

# RPN Calculator

# Stack

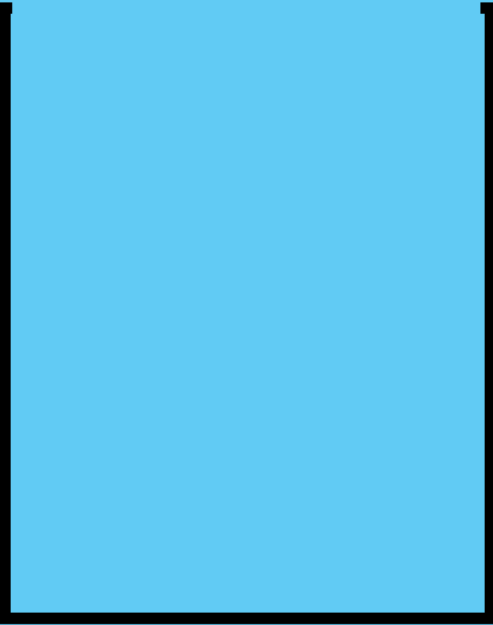
Push

Pop

Top

6

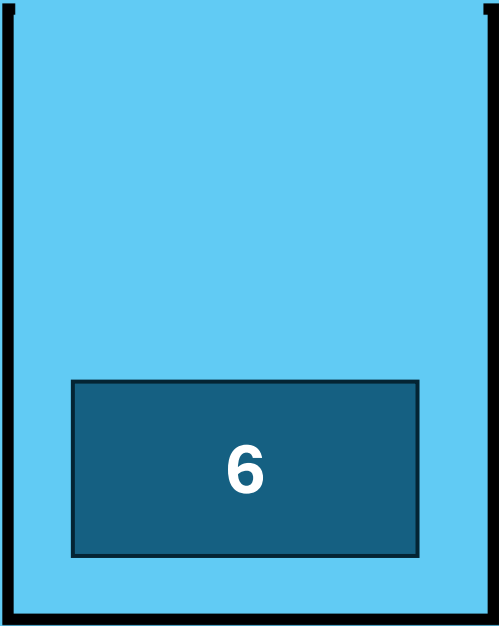
3



Stack

Stack

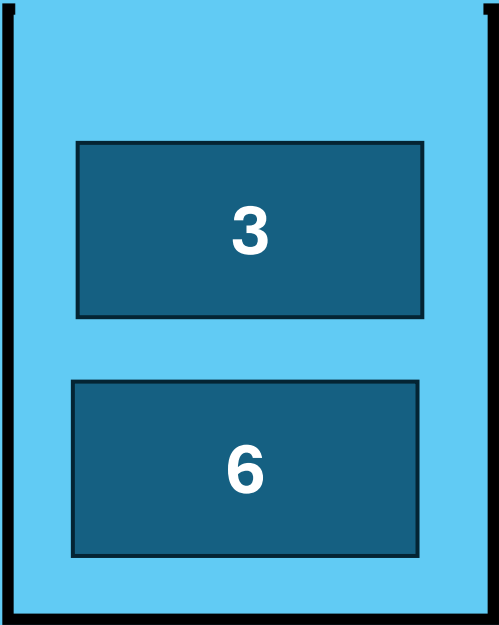
Push(6)



Stack

Stack

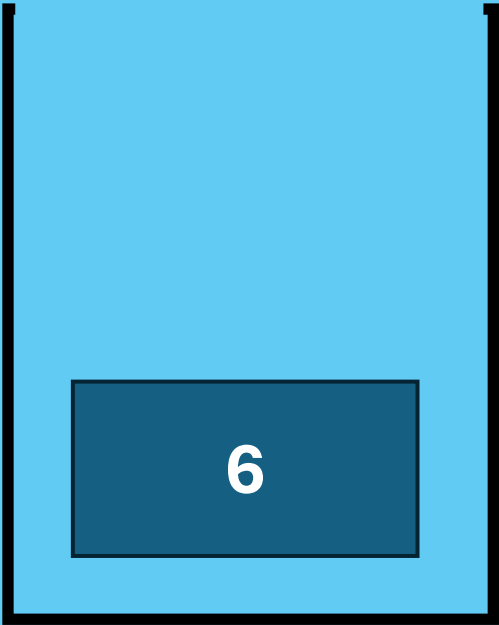
Push(3)



Stack

Stack

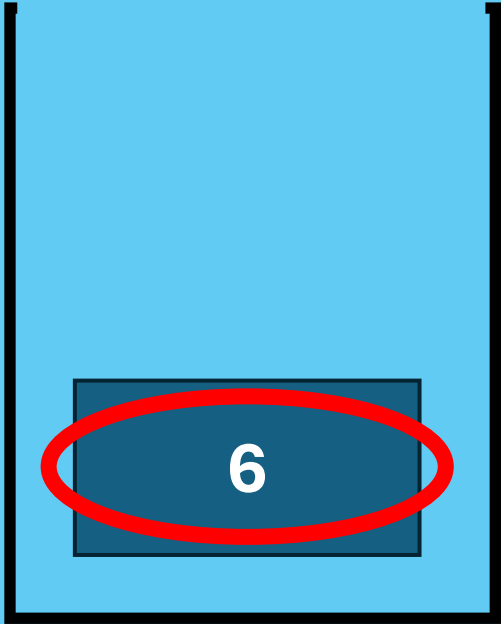
Pop



Stack

Stack

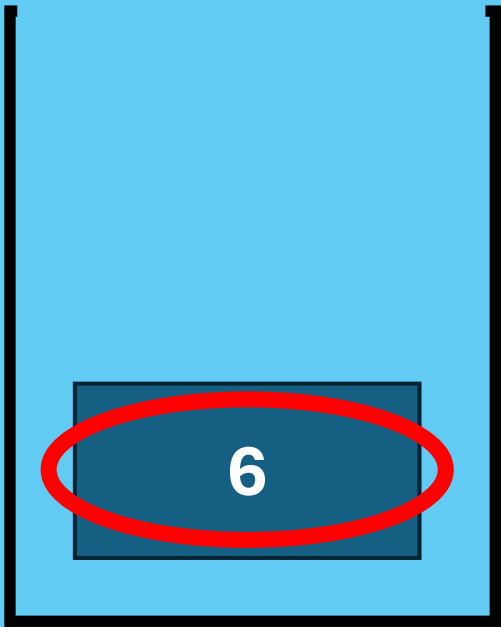
Top



Stack

Stack

Top



Stack



# Logik

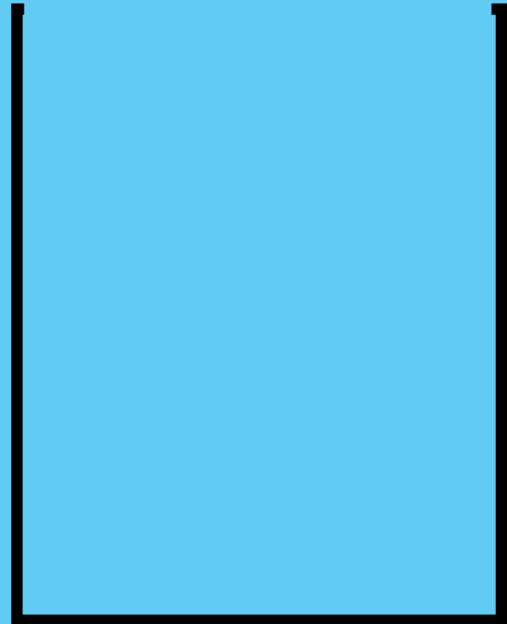
Eingabe

6

Logik?

Operation

Zahl



numberStack

# Logik

Eingabe

Logik?

Operation

Zahl

6

numberStack

# Logik

Eingabe

Logik?

Operation

Zahl

6

numberStack

# Logik

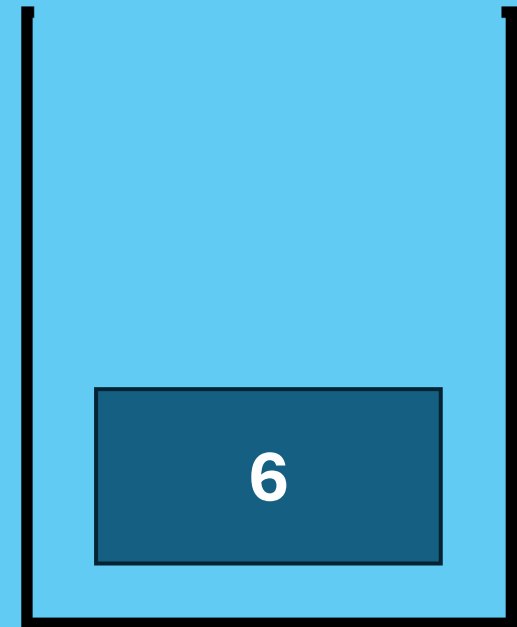
Eingabe



Logik?

Operation

Zahl



numberStack

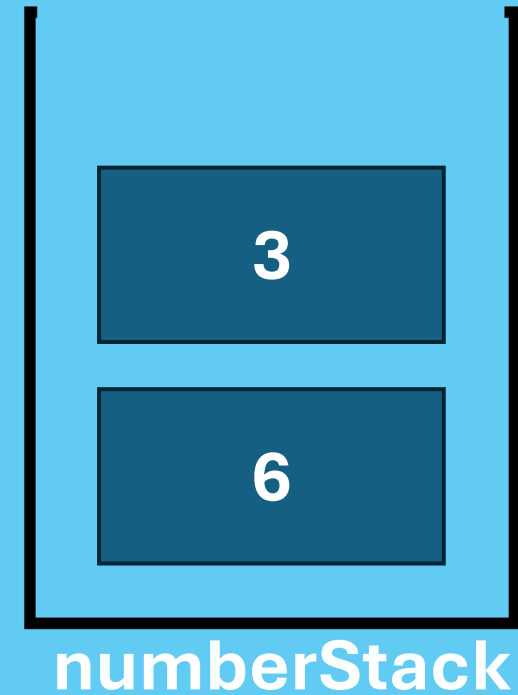
# Logik

Eingabe

Logik?

Operation

Zahl



# Logik

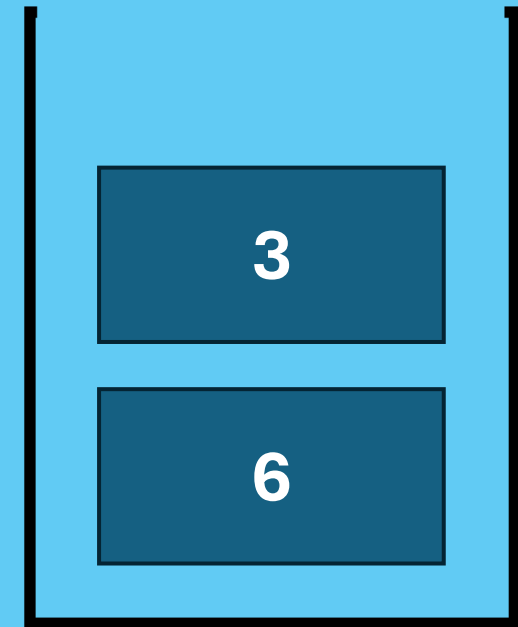
Eingabe



Logik?

Operation

Zahl



numberStack

# Logik

Eingabe

Logik?

Operation

Zahl

+

3

6

numberStack

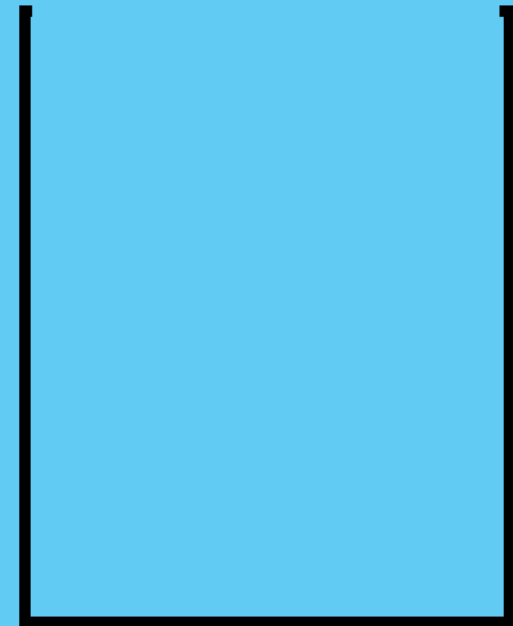
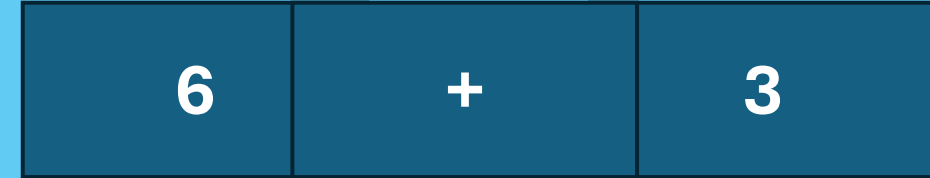
# Logik

Eingabe

Logik?

Operation

Zahl



numberStack



# Logik

Eingabe

Logik?

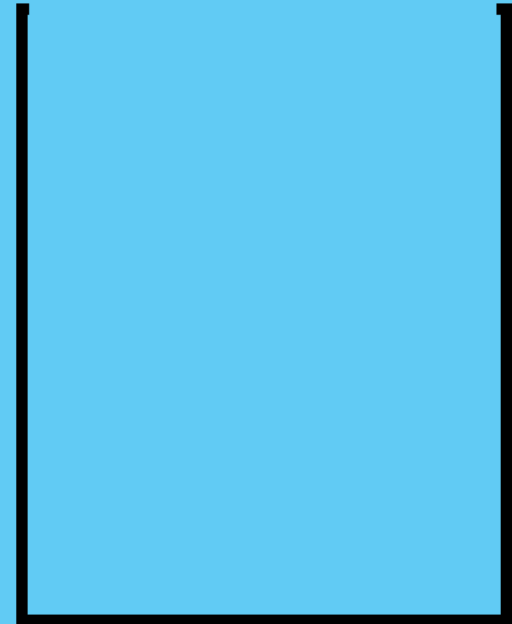
Operation

Zahl

9

numberStack

INTERNAL



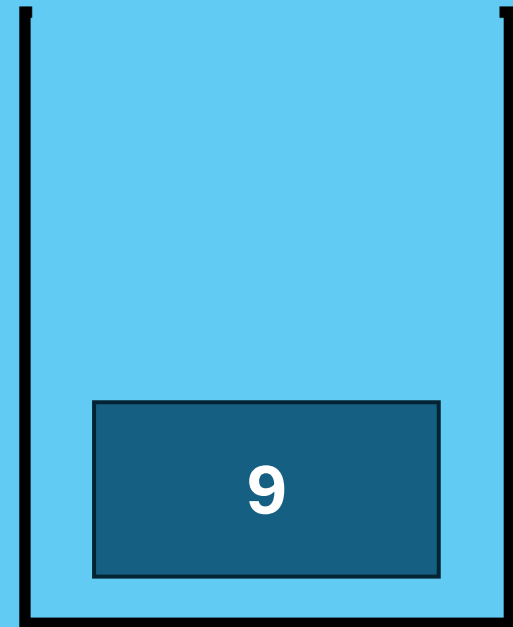
# Logik

Eingabe

Logik?

Operation

Zahl



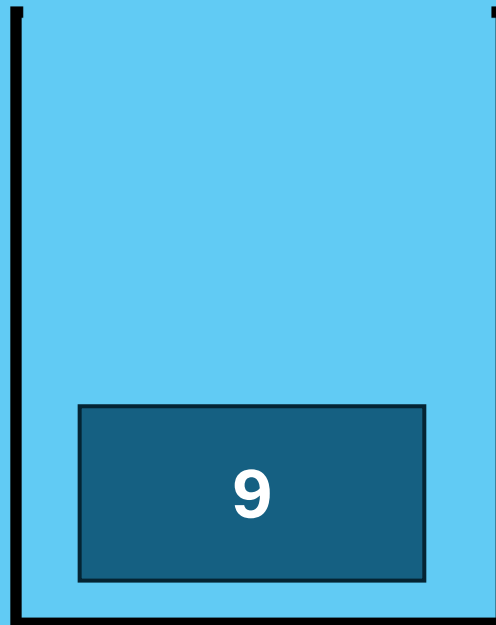
numberStack

**Logik**

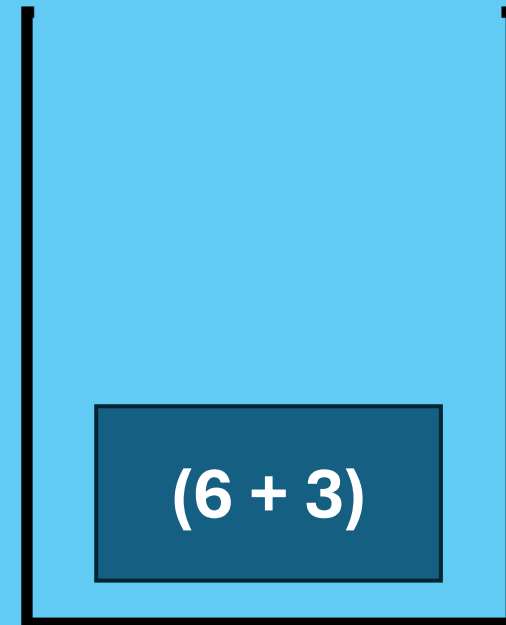
**Operation**

**Logik?**

**Zahl**



**numberStack**



**history**

INTERNAL

**Logik**

**Operation**

**Logik?**

**Zahl**

**3**

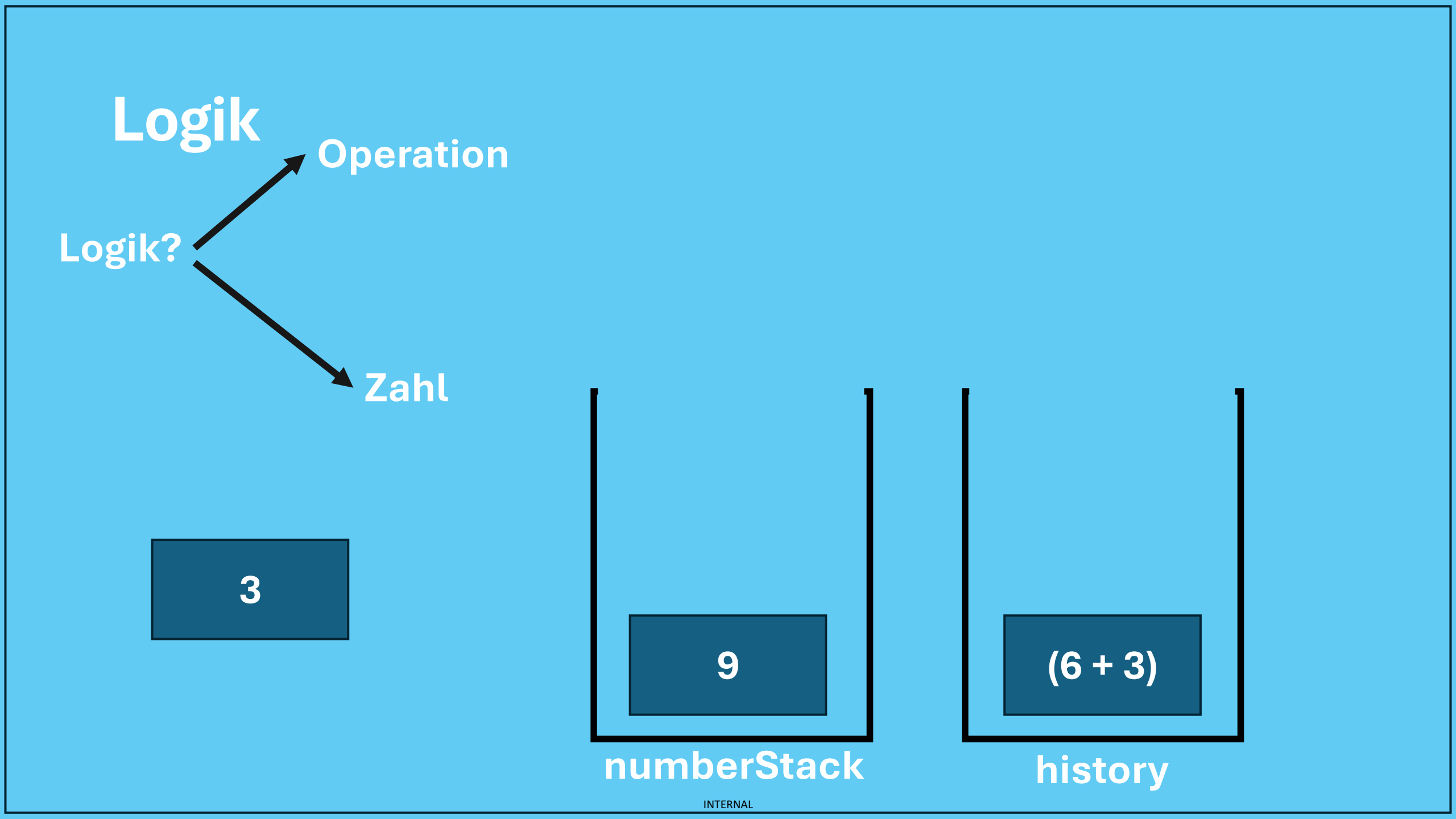
**9**

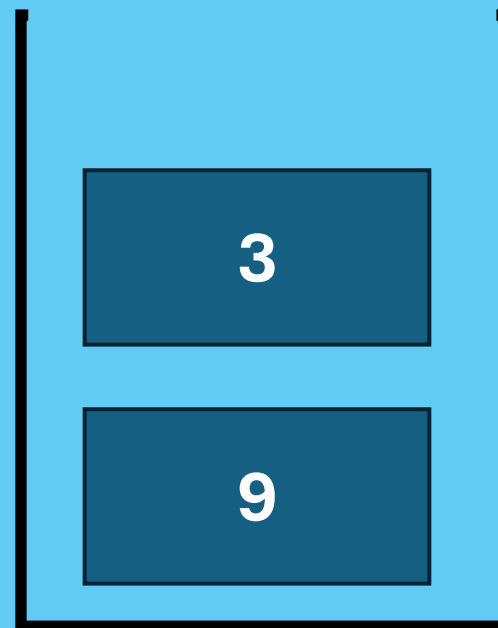
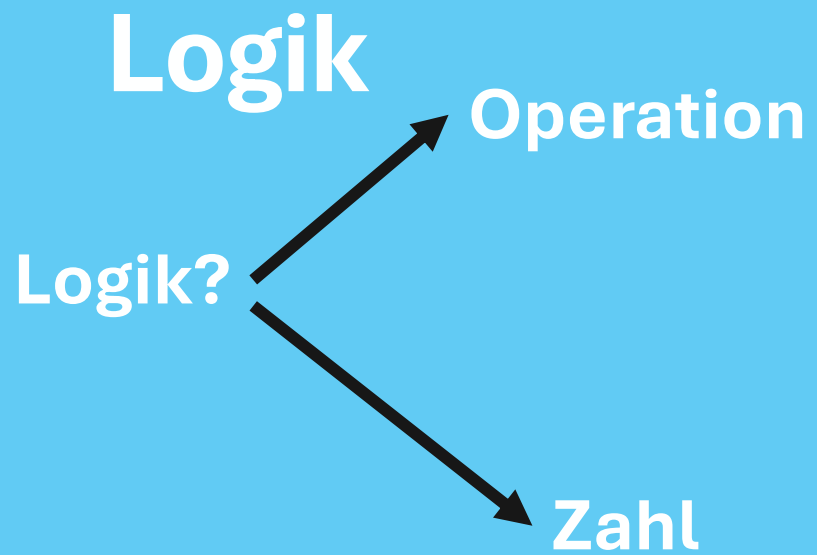
**(6 + 3)**

**numberStack**

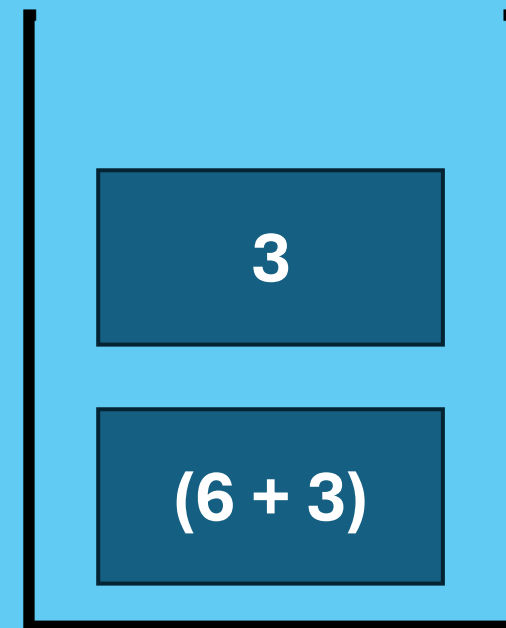
**history**

INTERNAL





numberStack



history

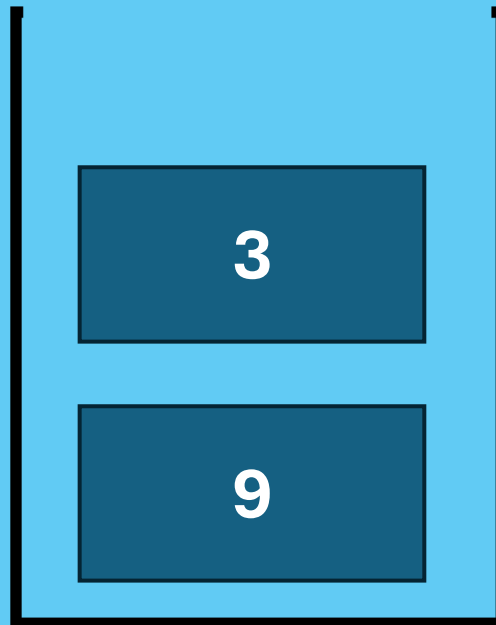
Logik

Operation

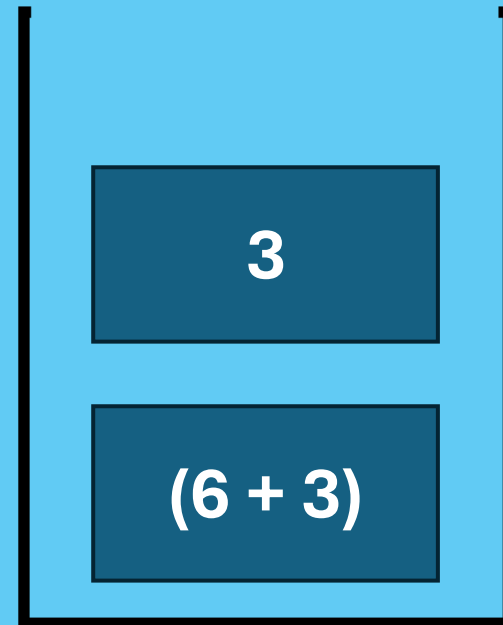
Logik?

Zahl

/



numberStack



history

Logik

Operation

Logik?

Zahl

/

3

9

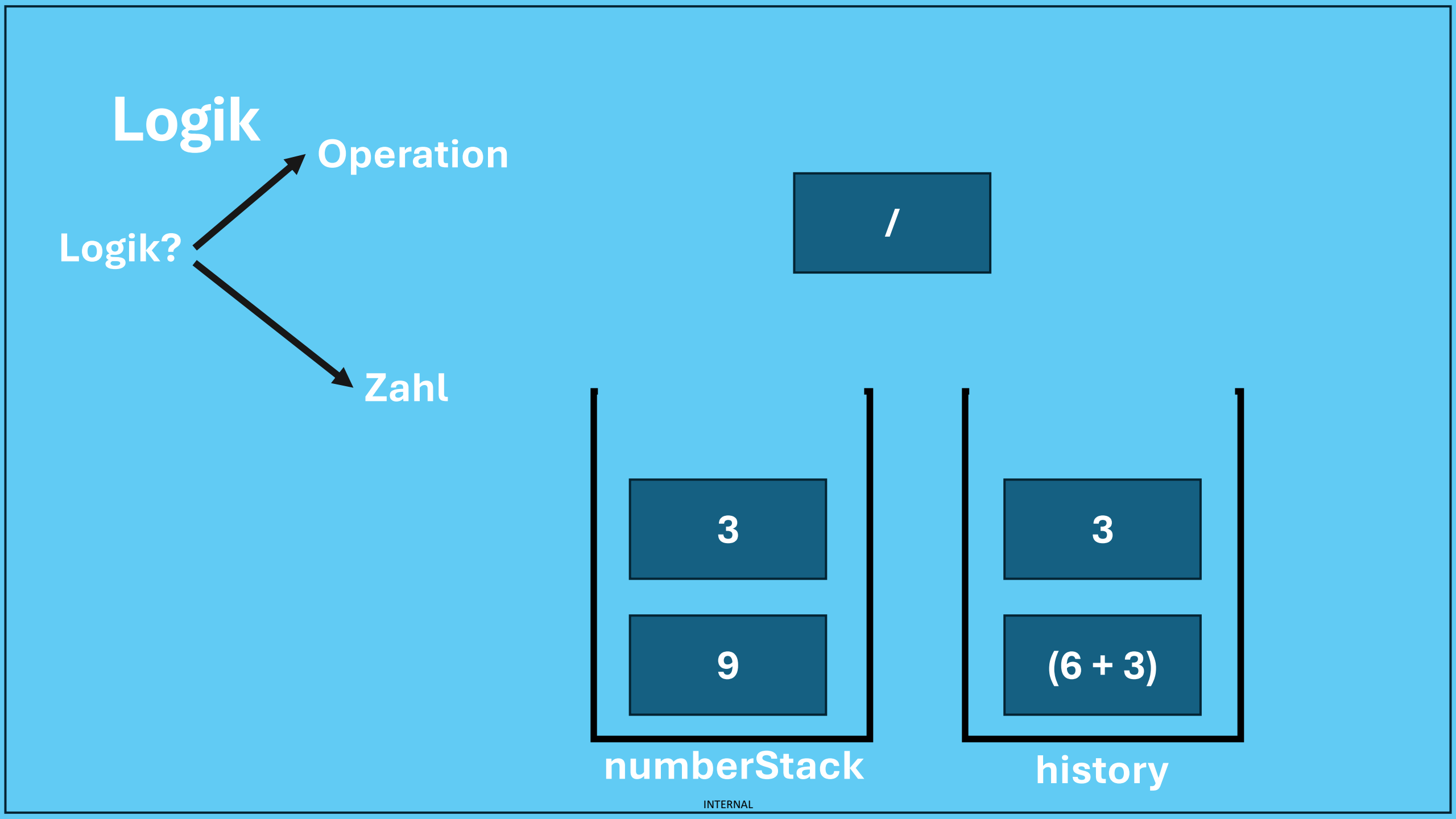
numberStack

3

(6 + 3)

history

INTERNAL

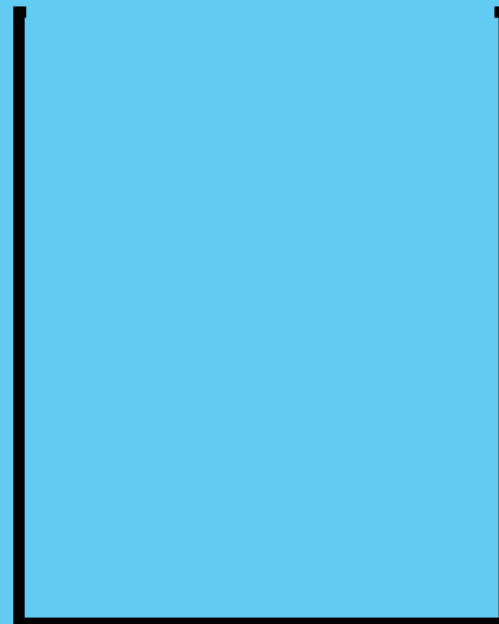


Logik

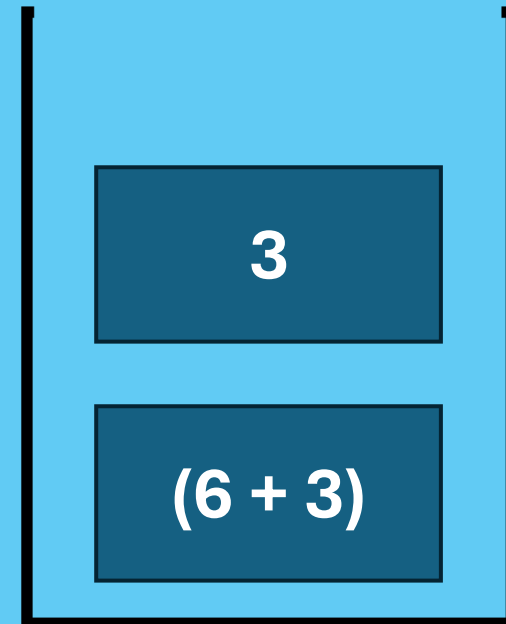
Operation

Logik?

Zahl



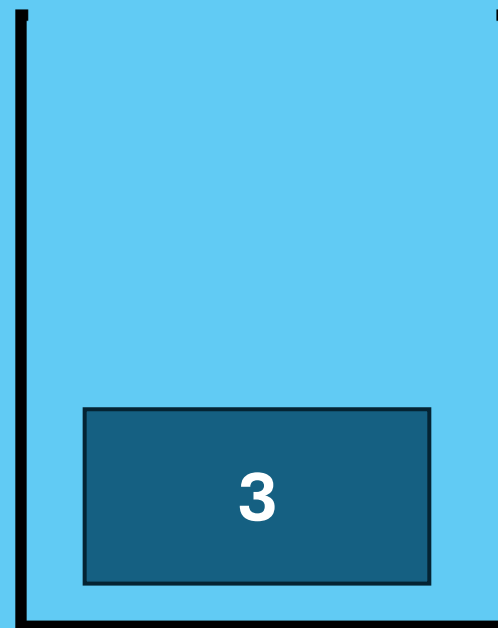
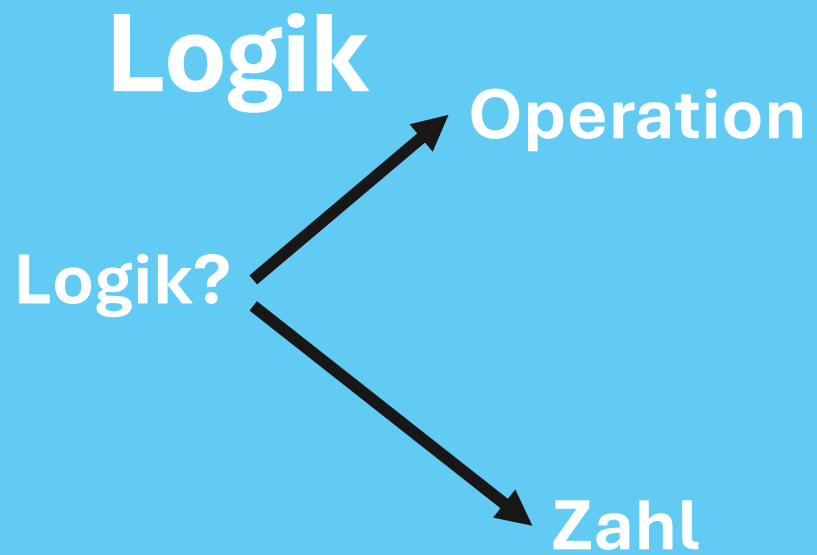
numberStack



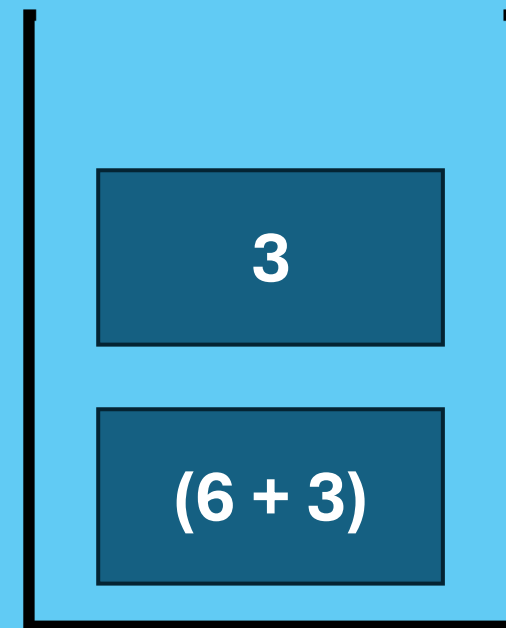
history

INTERNAL





numberStack



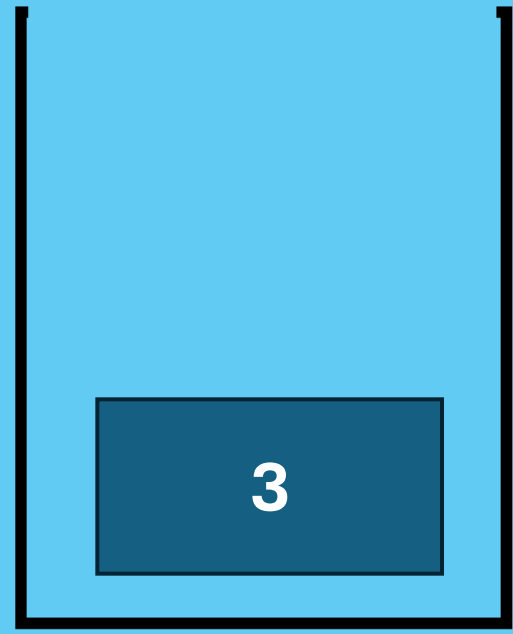
history

Logik

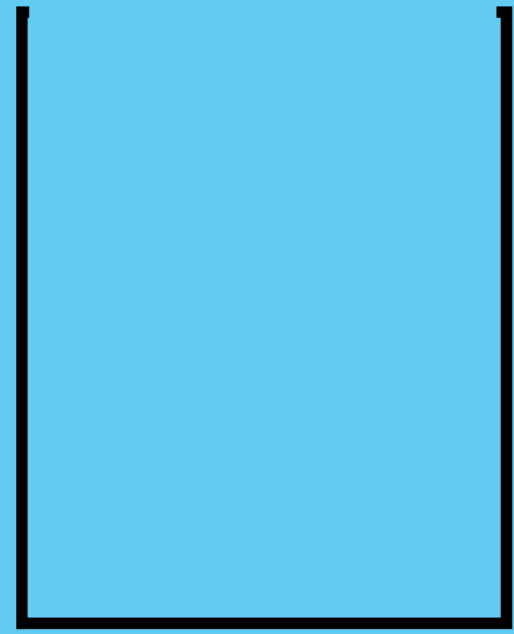
Operation

Logik?

Zahl



numberStack



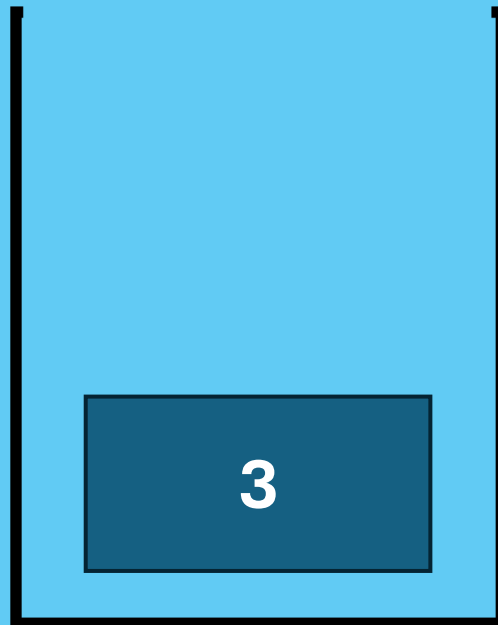
history

**Logik**

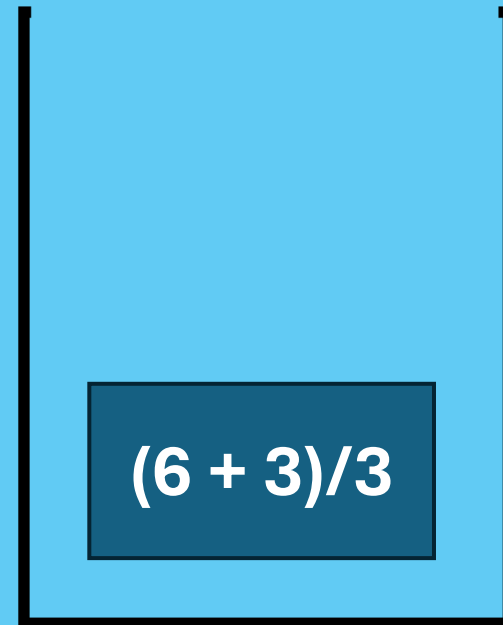
**Operation**

**Logik?**

**Zahl**



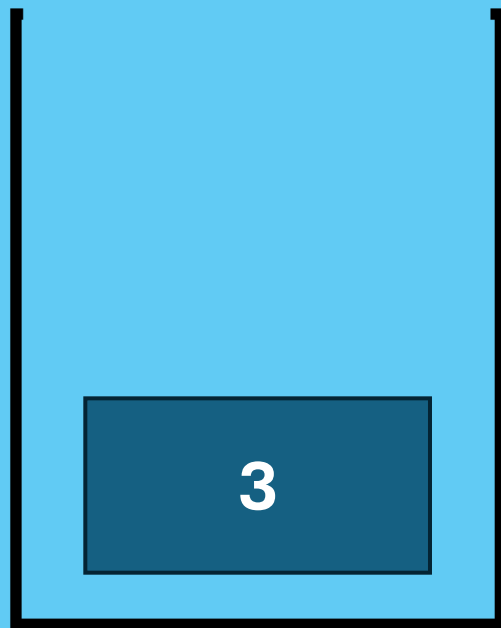
**numberStack**



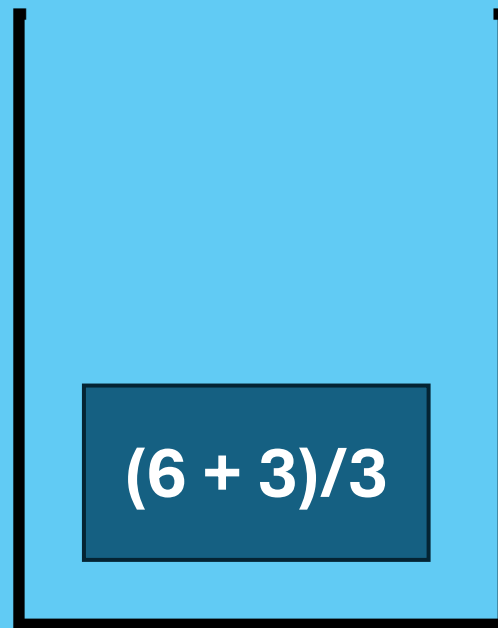
**history**

INTERNAL

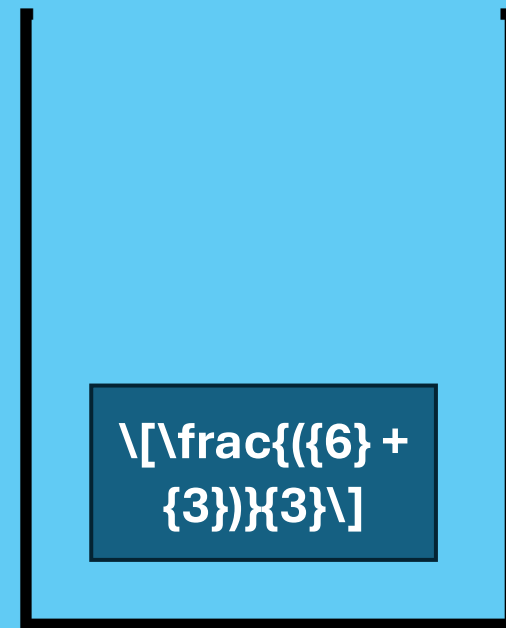
# Logik



numberStack



history



latex

# Quellen

- <https://go.dev/>
- <https://pkg.go.dev/>
- [https://en.wikipedia.org/wiki/Go\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Go_(programming_language))
- [Go: The Complete Developer's Guide \(Golang\) | Udemy](#)