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

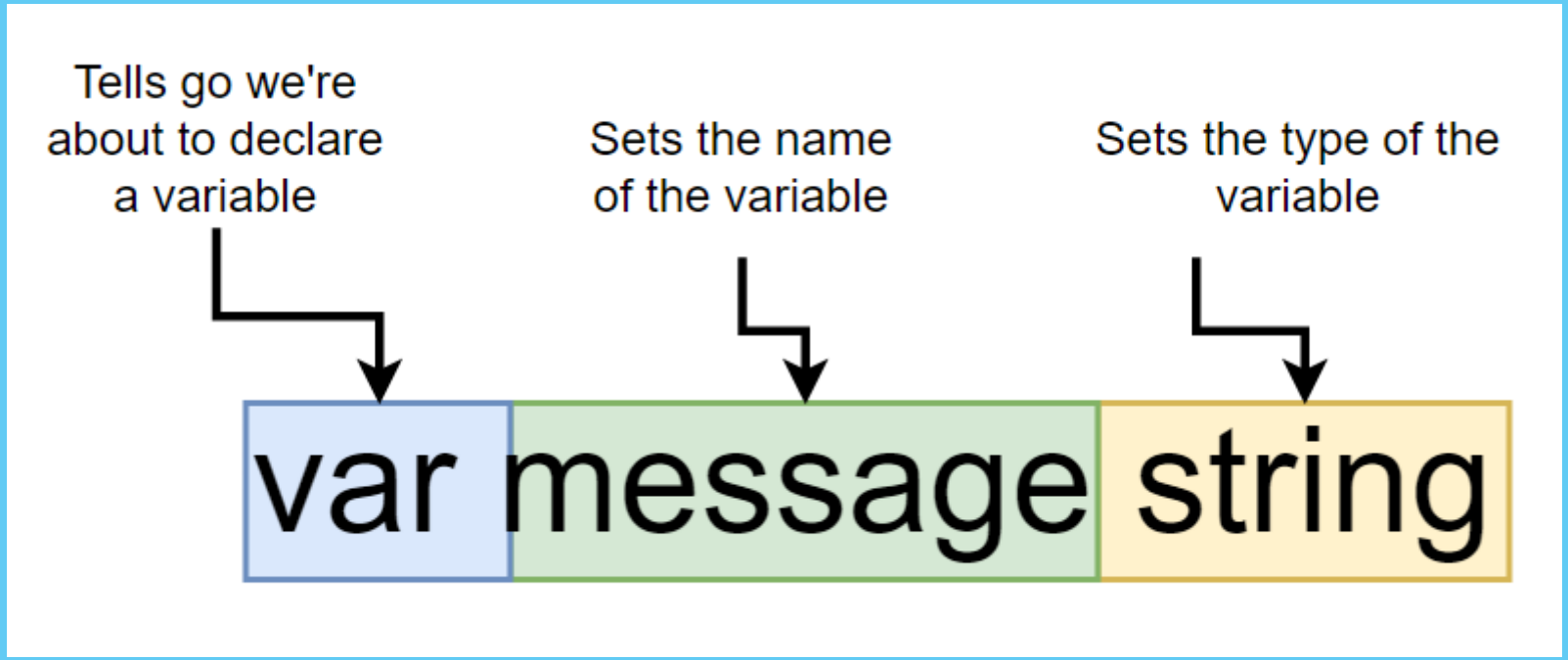
```go
package main

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"
```

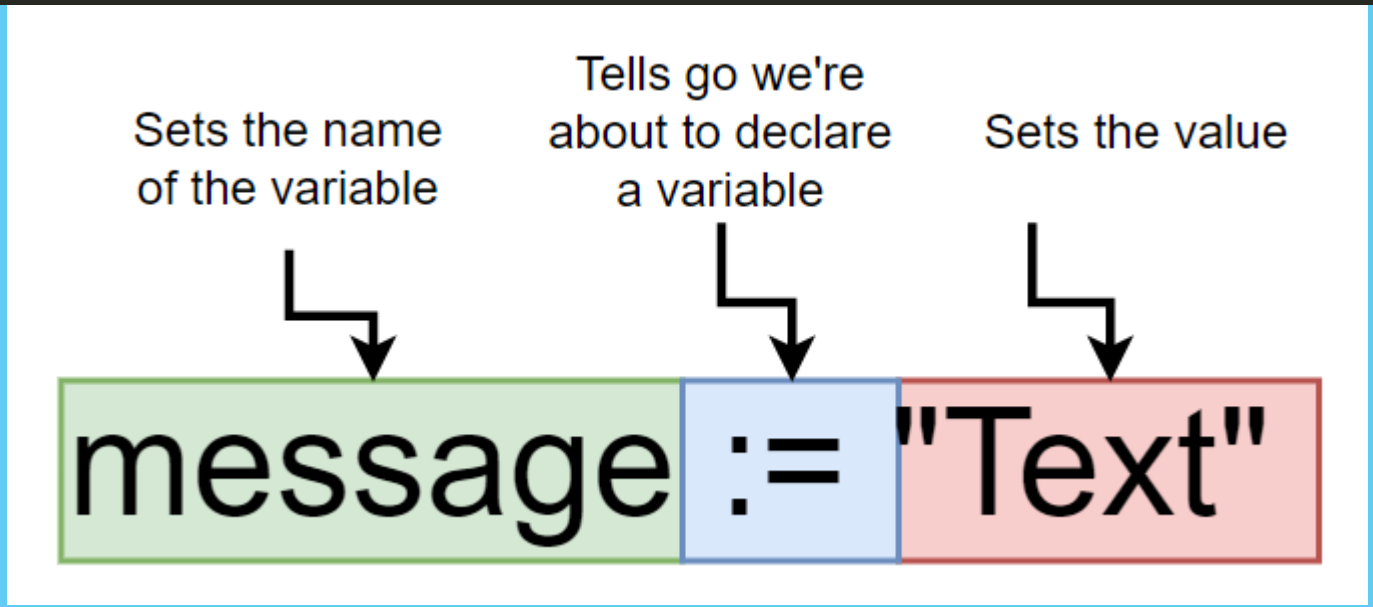


Variables

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

Variables

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



Typinferenz & Starke Typisierung

Variables (Pointer)

```go
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

```go
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

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

```go
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)
}
```

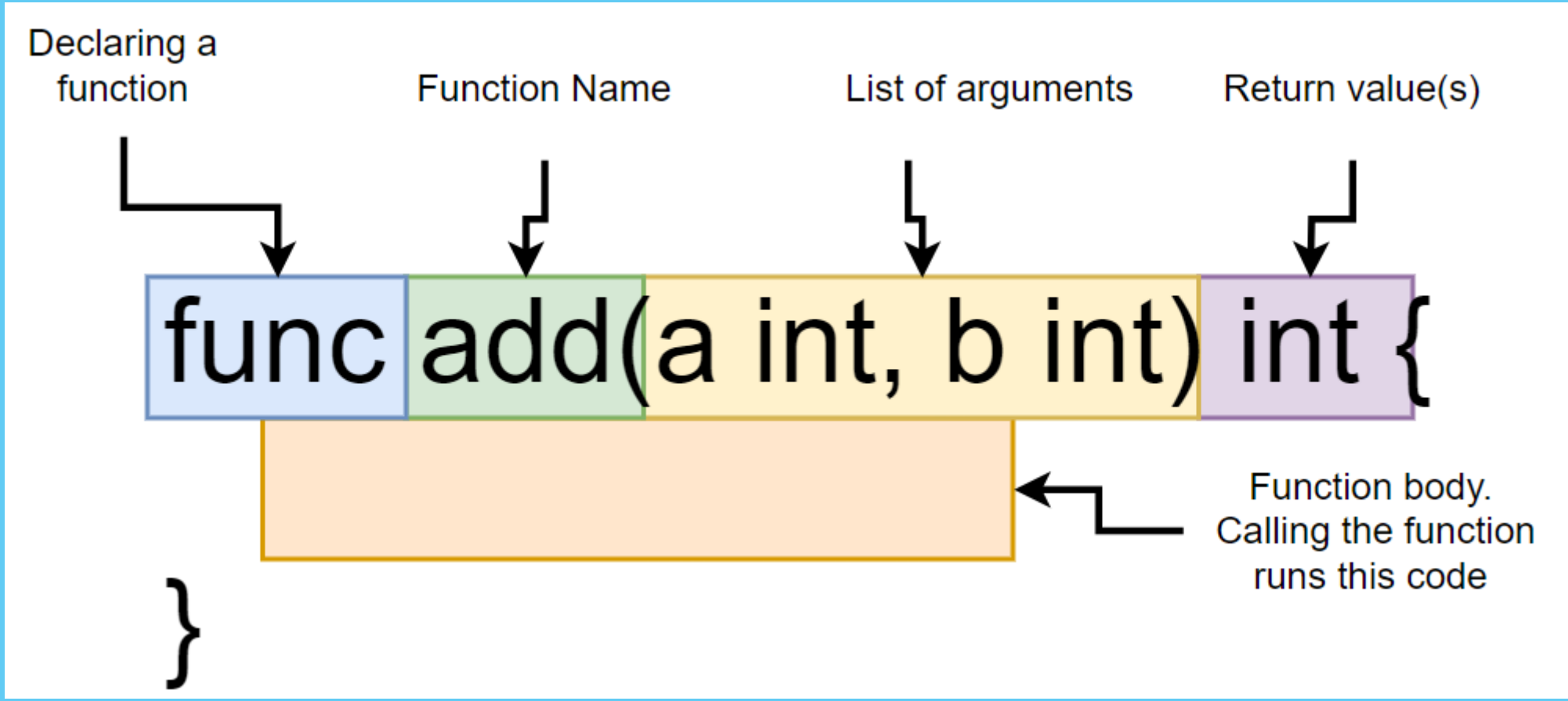
Ohne Fallthrough Logik!

Funktionen

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

Funktionen

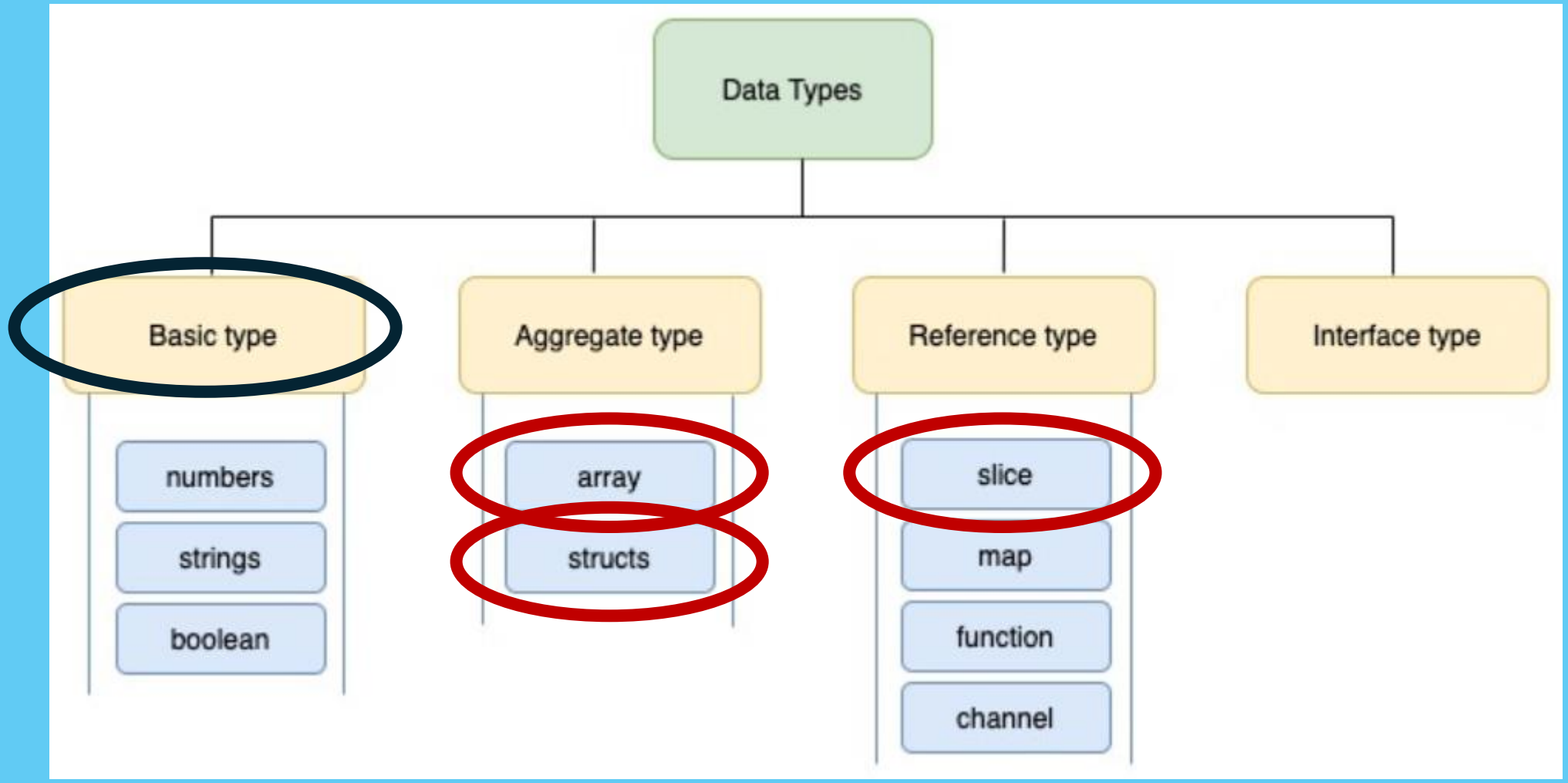
```go
func add(a int, b int) int {
    return a + b
}
```



Funktionen (Error Handling)

```go
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-RLNEjnvg.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

Arrays

- Statisch
- Feste Größe
- Nur ein Datentyp
- Wertebasiert

Slice

- 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

```go
// 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

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

Slices

```go
// 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
```

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

Slicing

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

Slices

```go
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
}
```

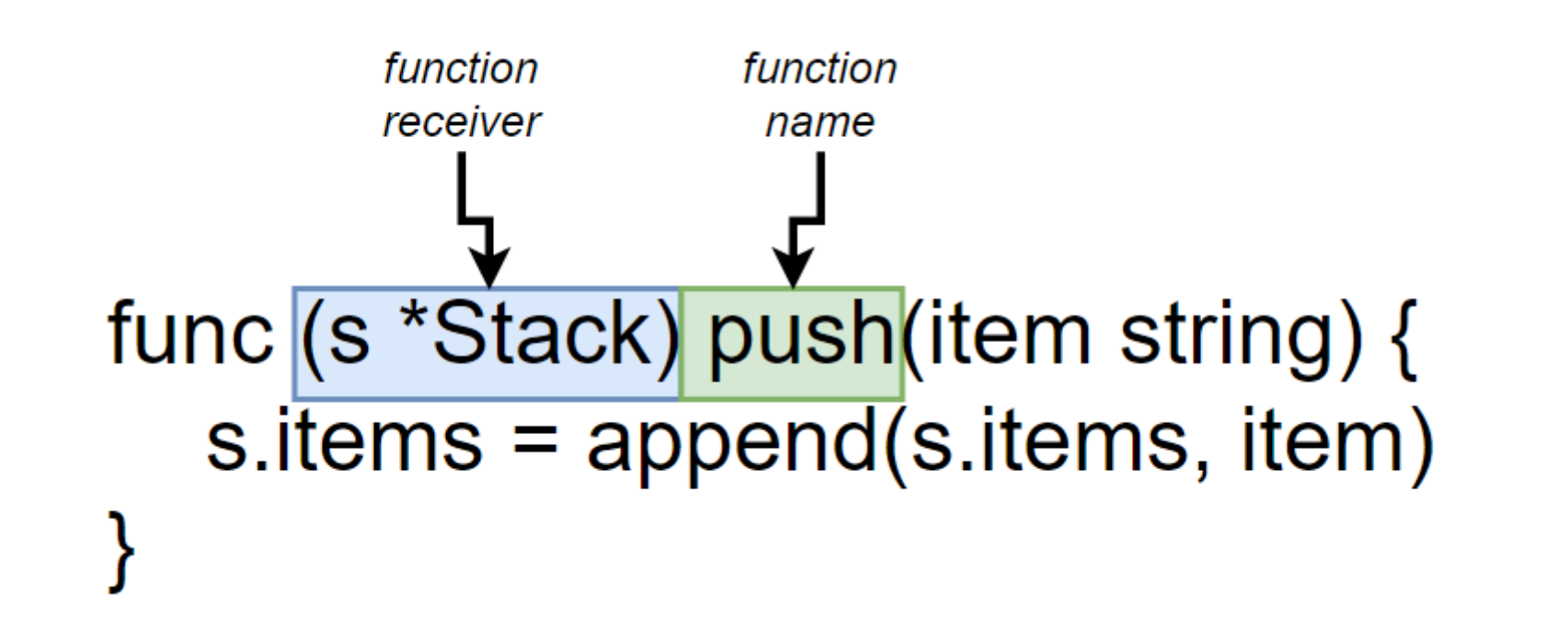
Structs & Types

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

Receiver Funktionen

```go
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



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)
}
```

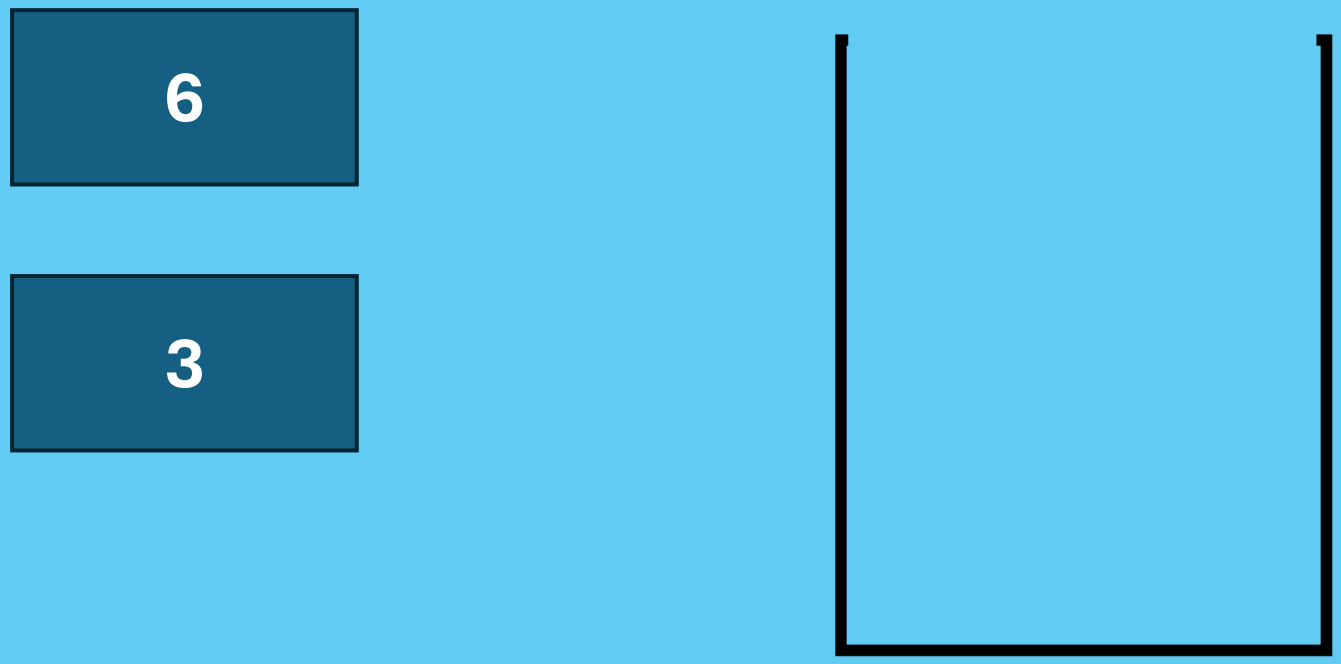
RPN Calculator

Stack

Push

Pop

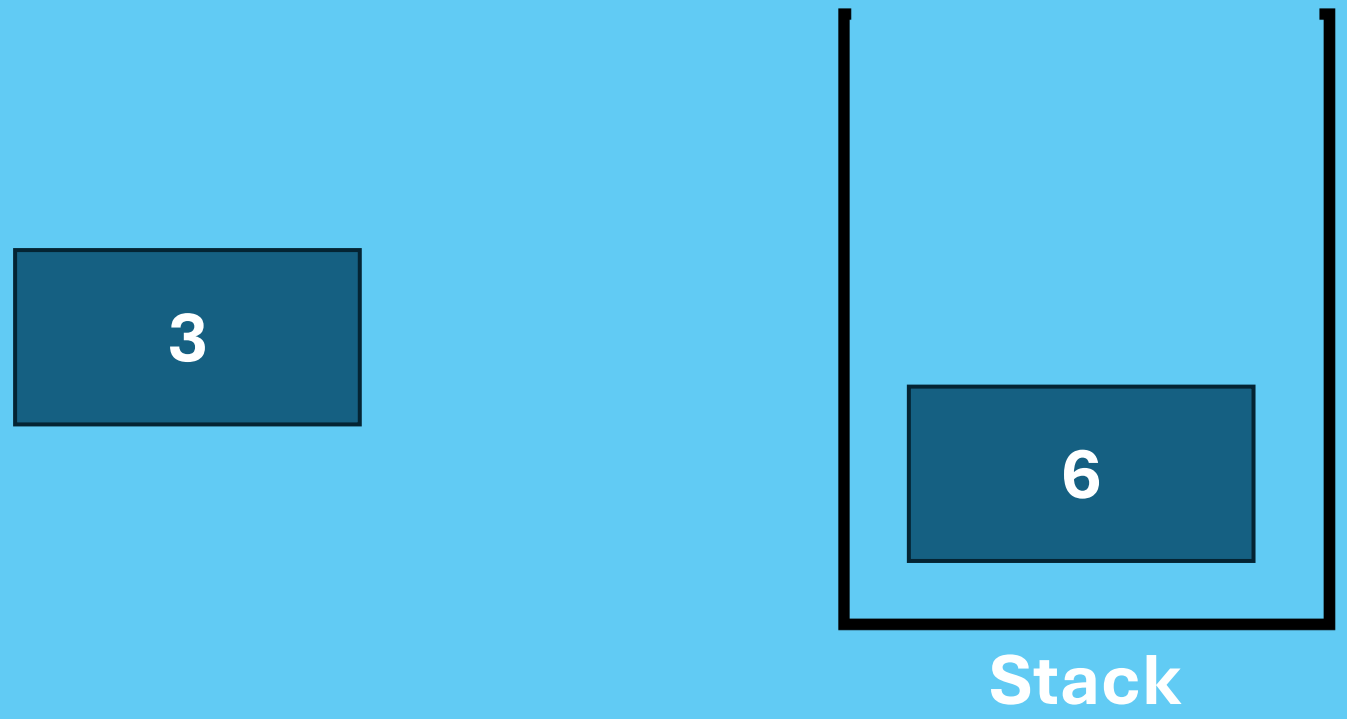
Top



Stack

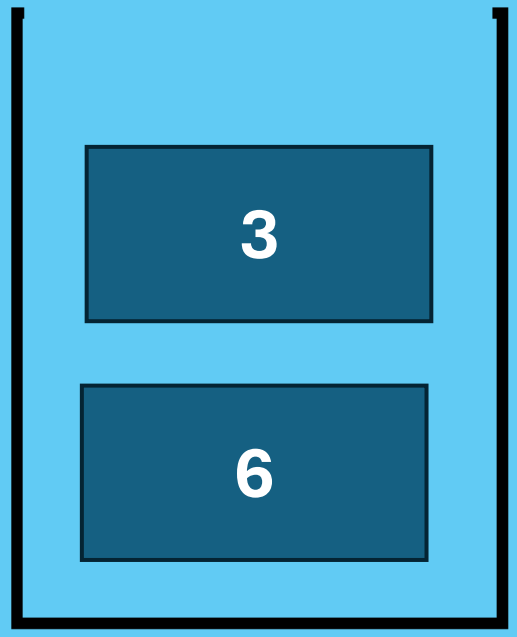
Stack

Push(6)



Stack

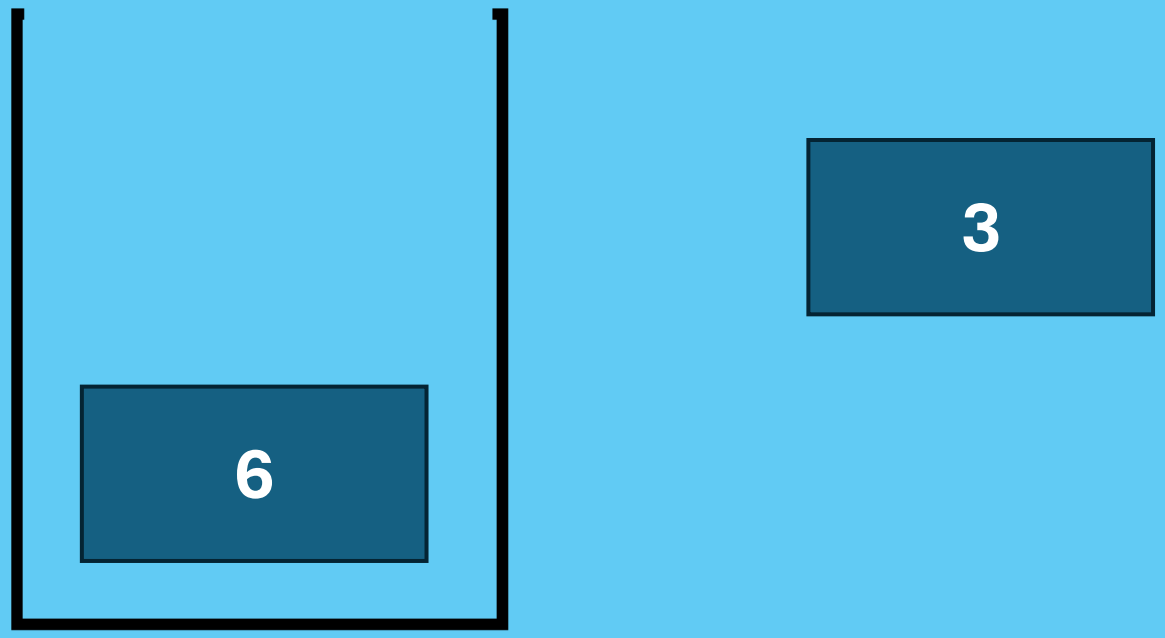
Push(3)



Stack

Stack

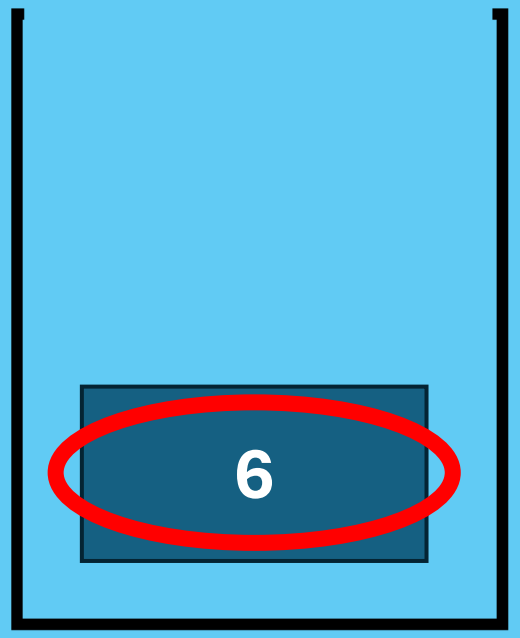
Pop



Stack

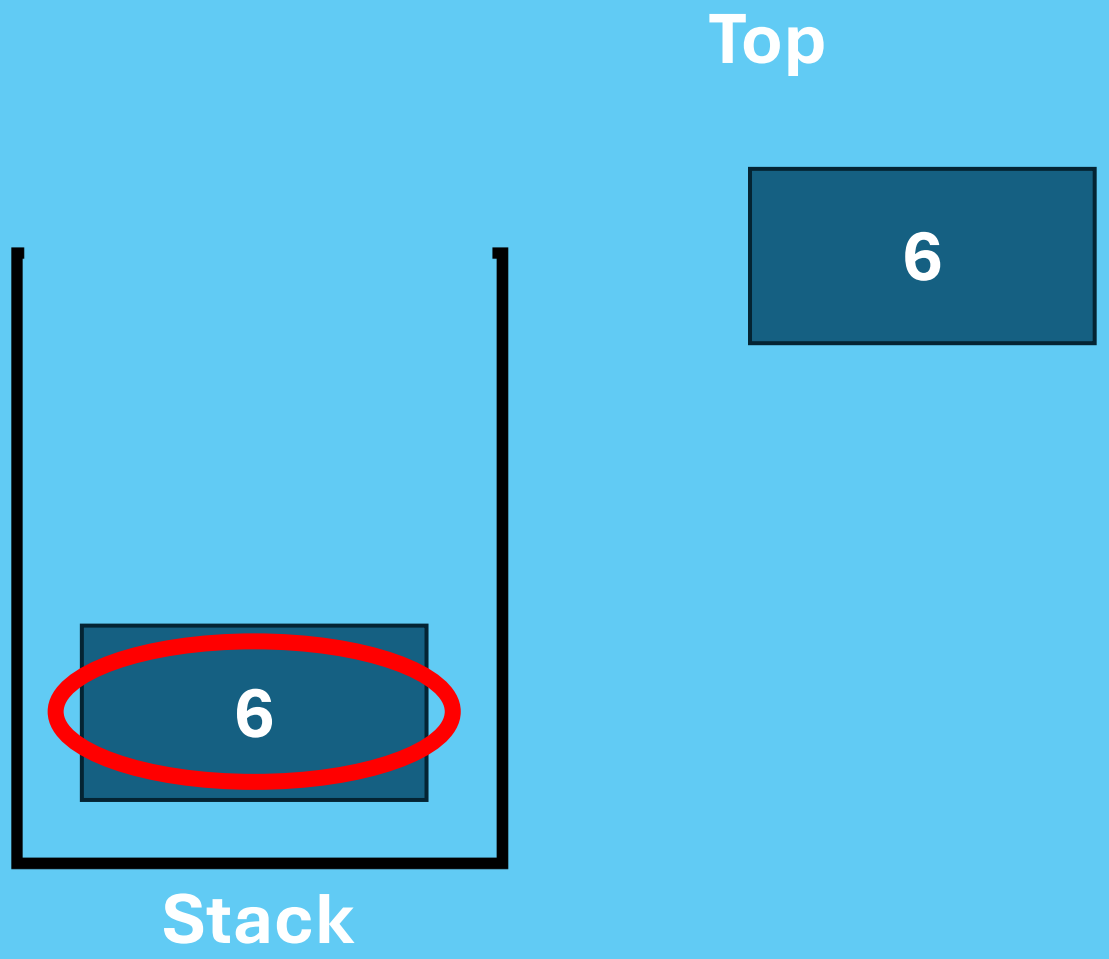
Stack

Top



Stack

Stack



Logik

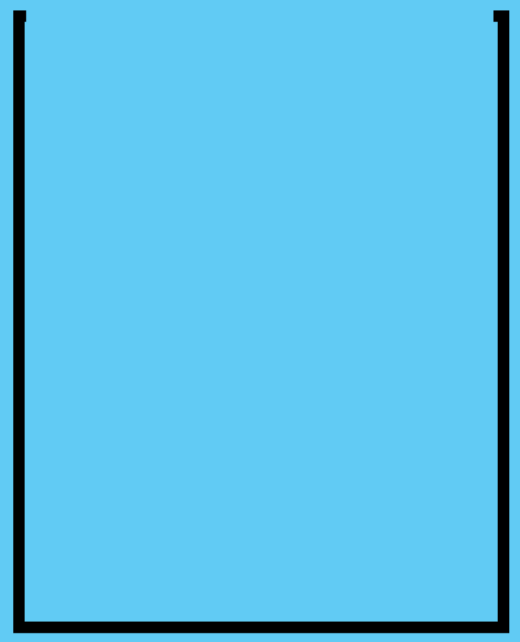
Eingabe

6

Logik?

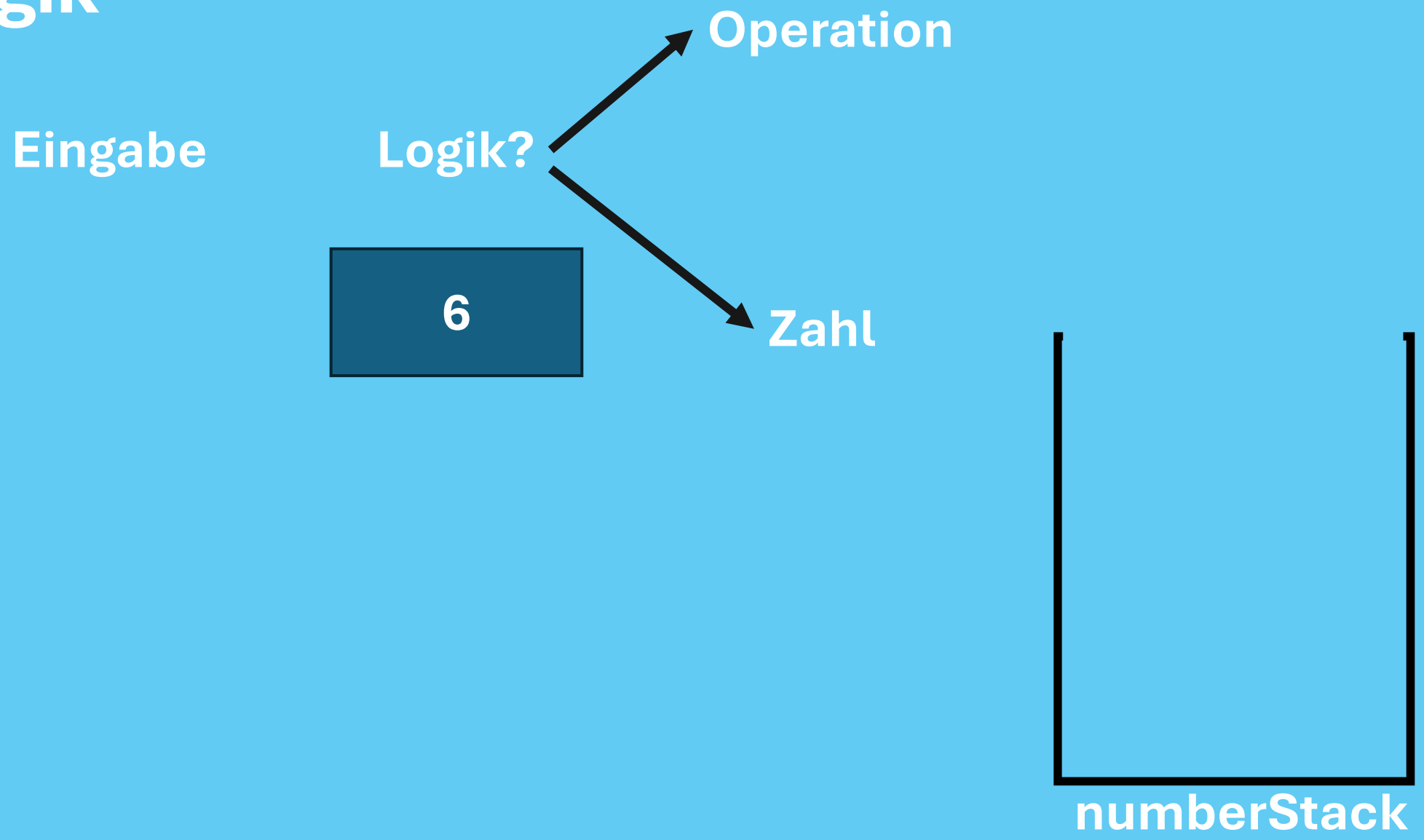
Operation

Zahl



numberStack

Logik



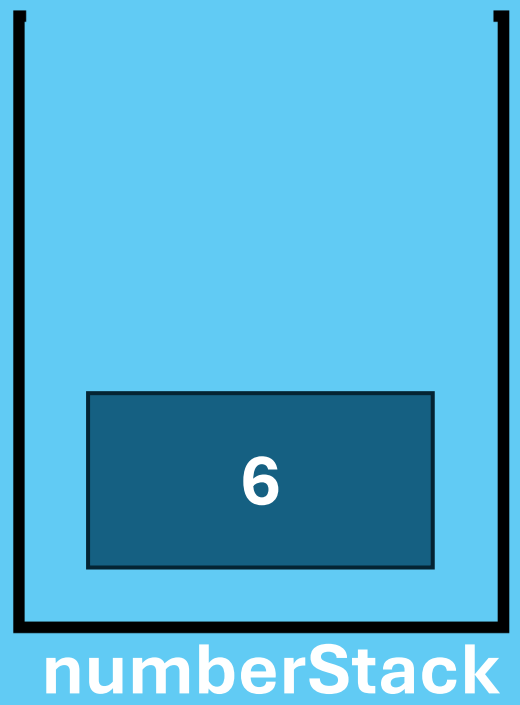
Logik

Eingabe

Logik?

Operation

Zahl



Logik

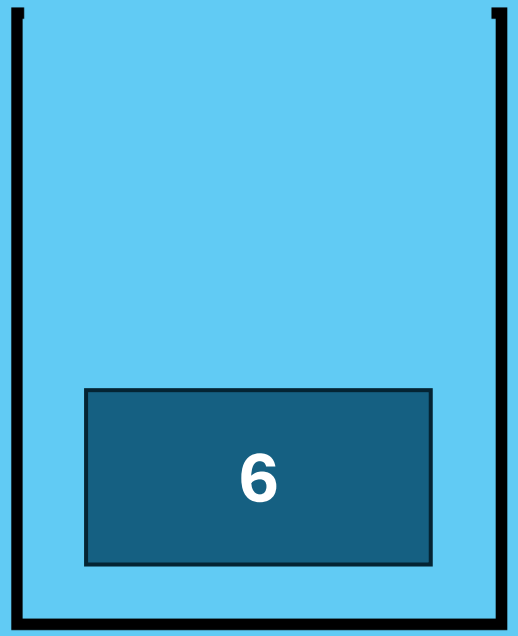
Eingabe

3

Logik?

Operation

Zahl



numberStack

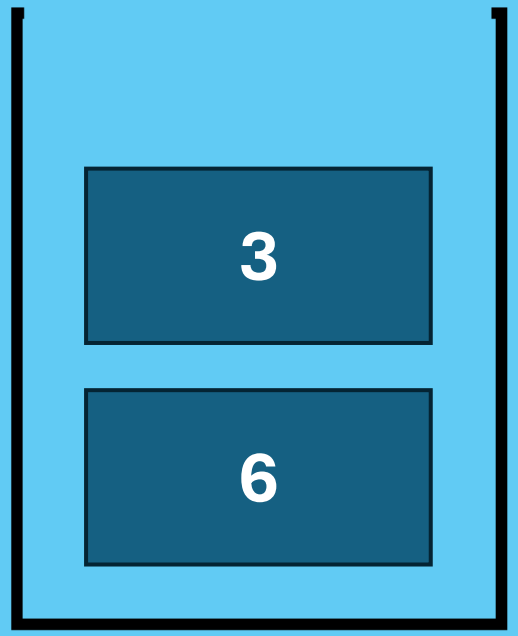
Logik

Eingabe

Logik?

Operation

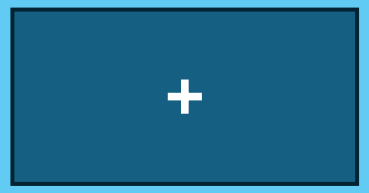
Zahl

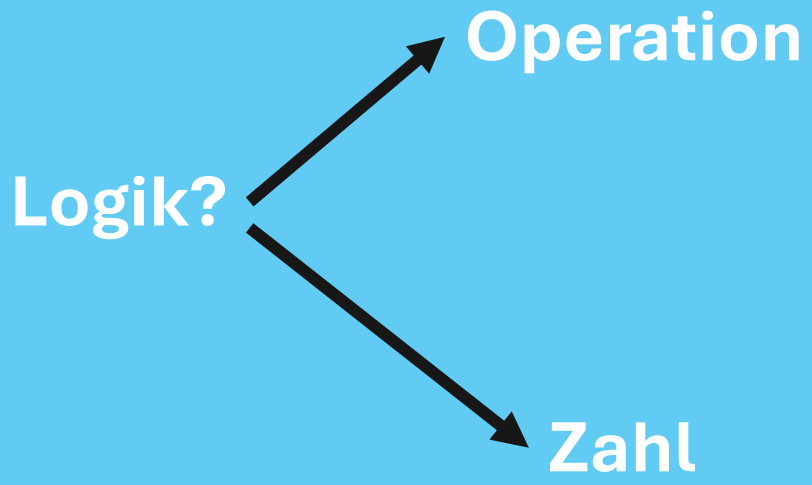


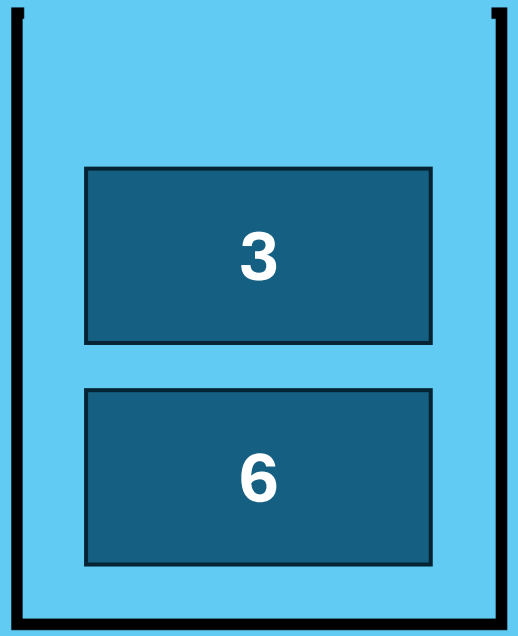
numberStack

Logik

Eingabe







numberStack

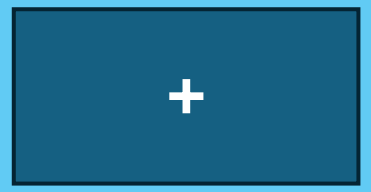
Logik

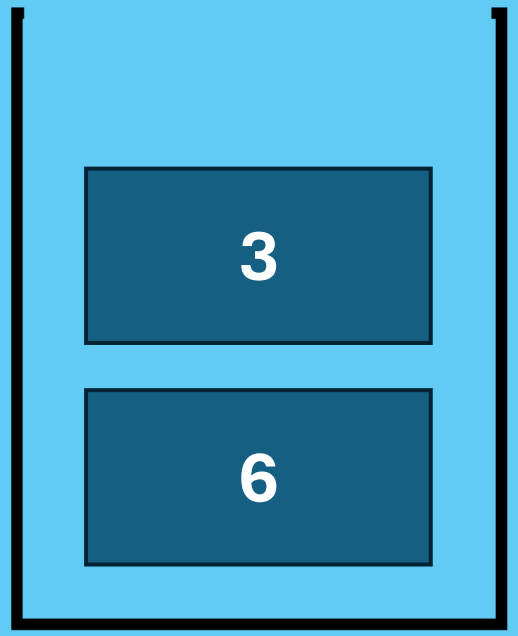
Eingabe

Logik?

Operation

Zahl





numberStack

Logik

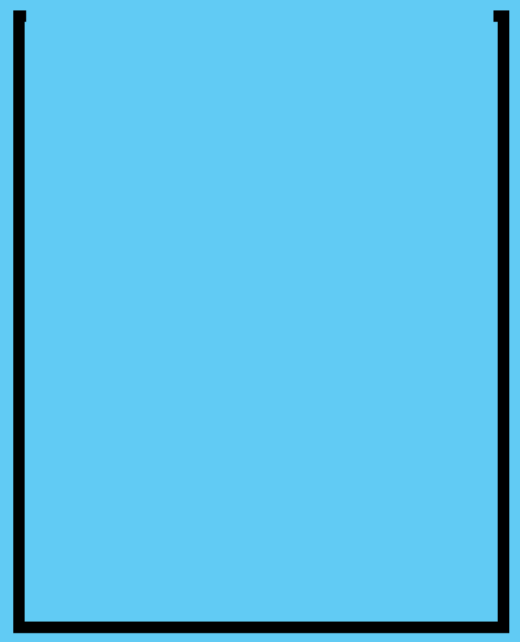
Eingabe

Logik?

Operation

Zahl

6	+	3



numberStack

Logik

Eingabe

Logik?

Operation

Zahl

9

numberStack

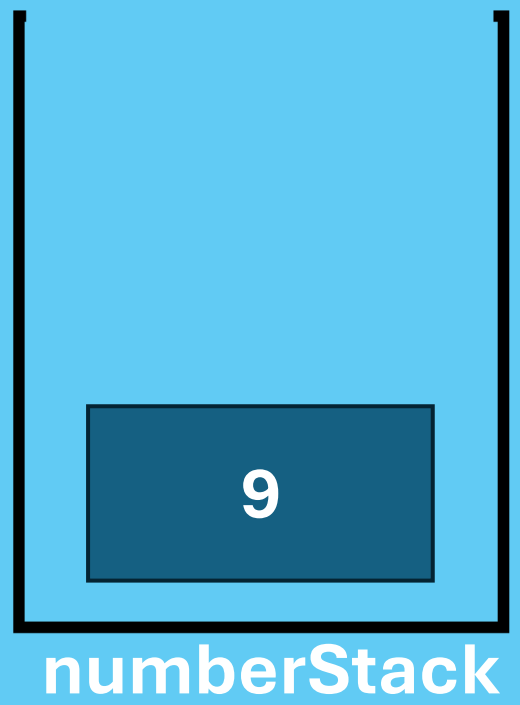
Logik

Eingabe

Logik?

Operation

Zahl

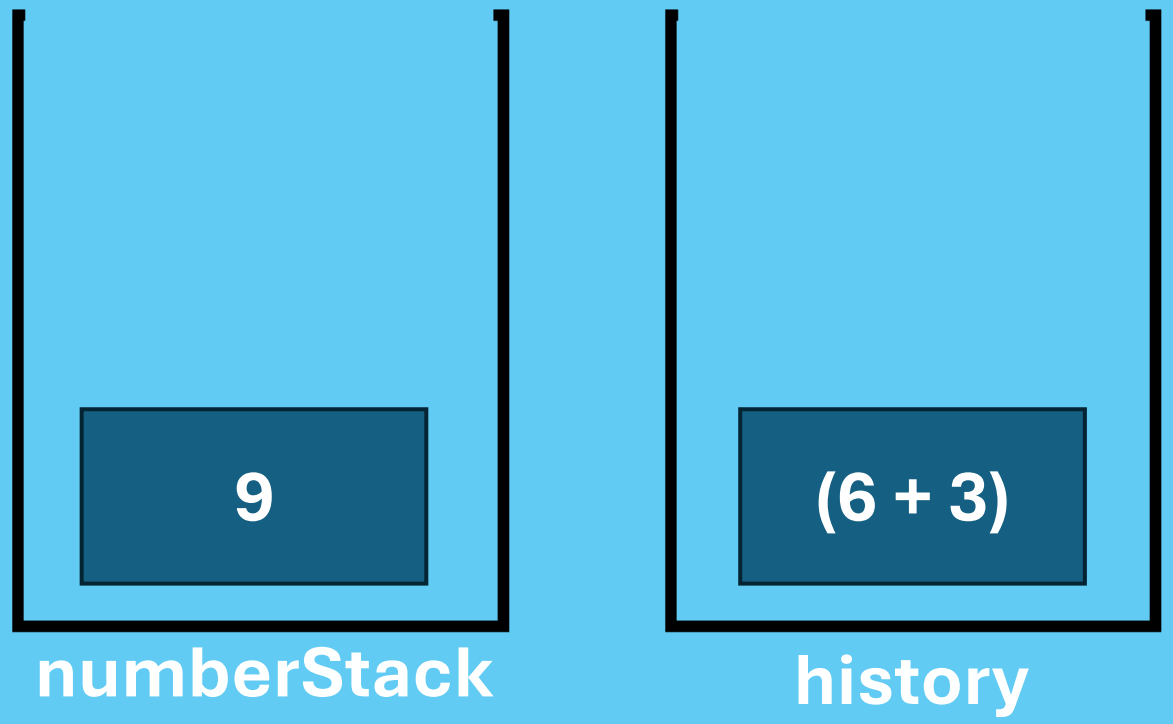


Logik

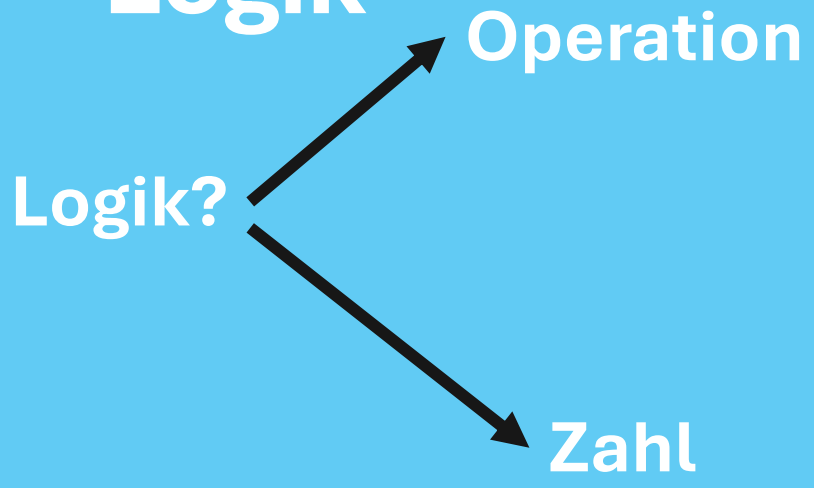
Logik?

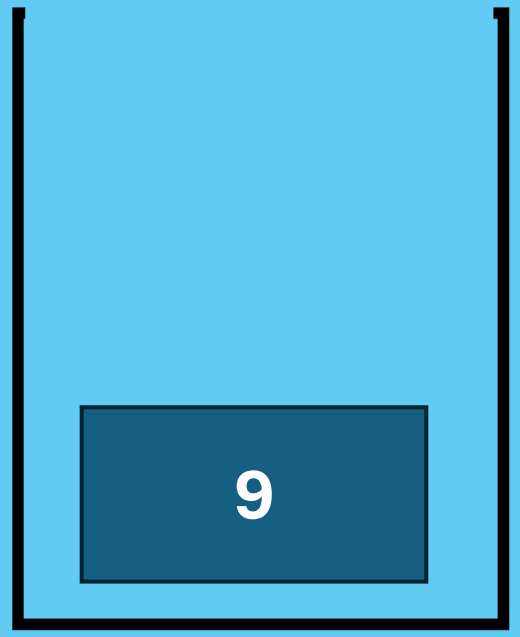
Operation

Zahl

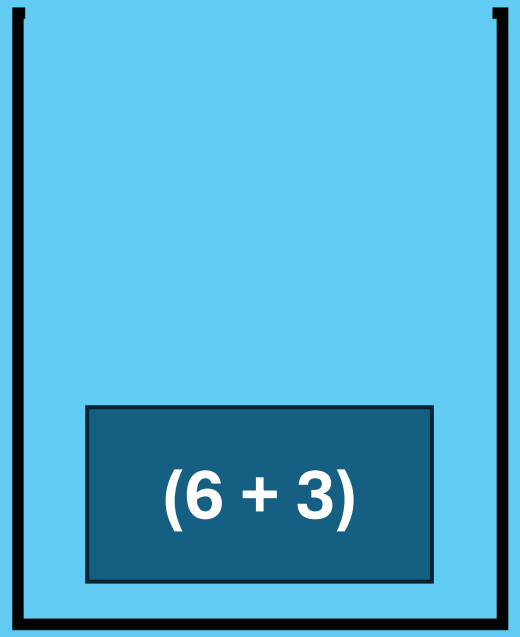


Logik



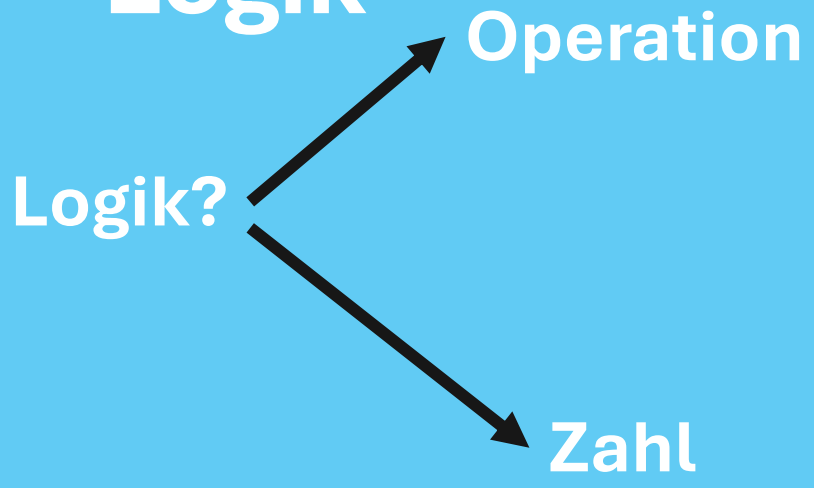


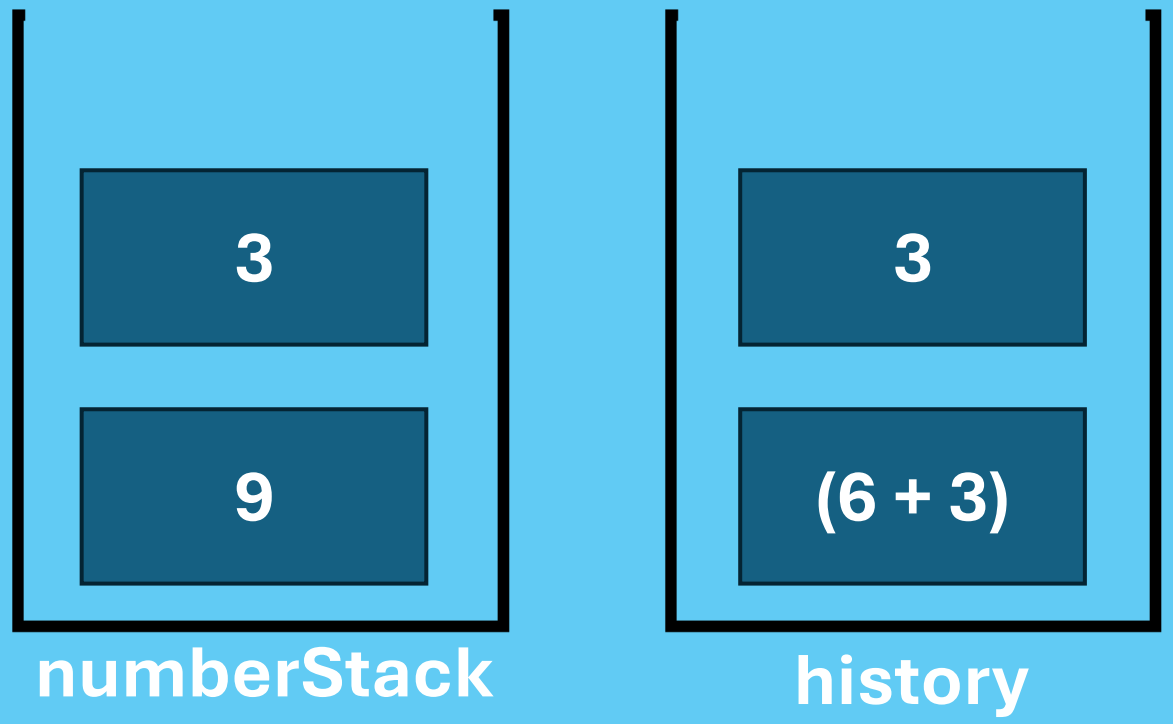
numberStack



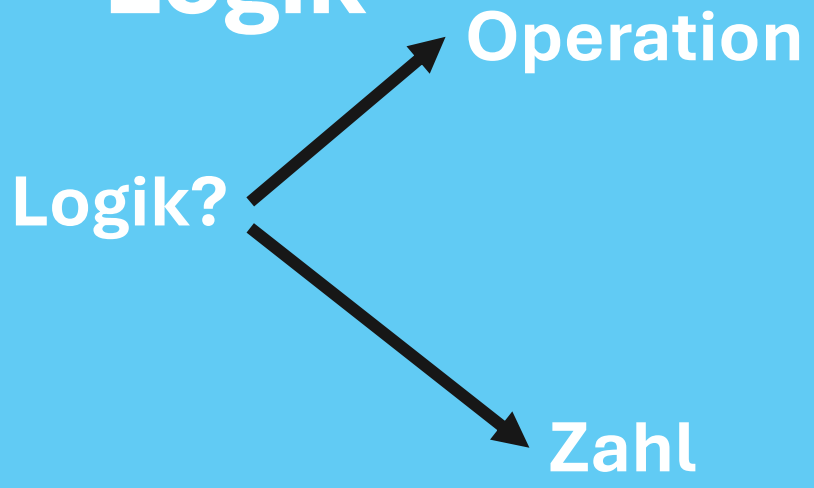
history

Logik

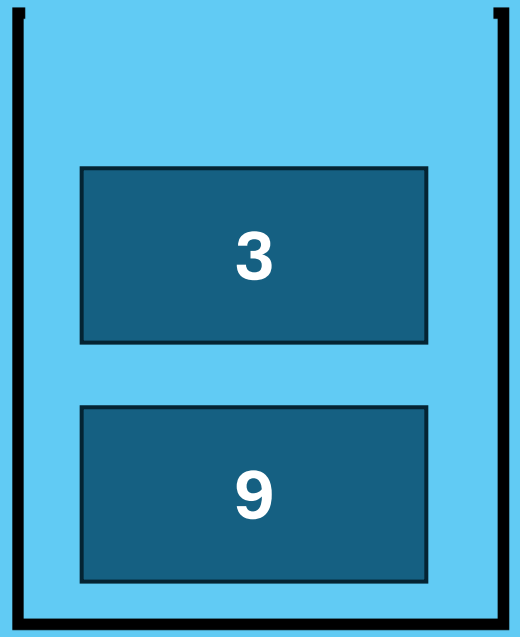




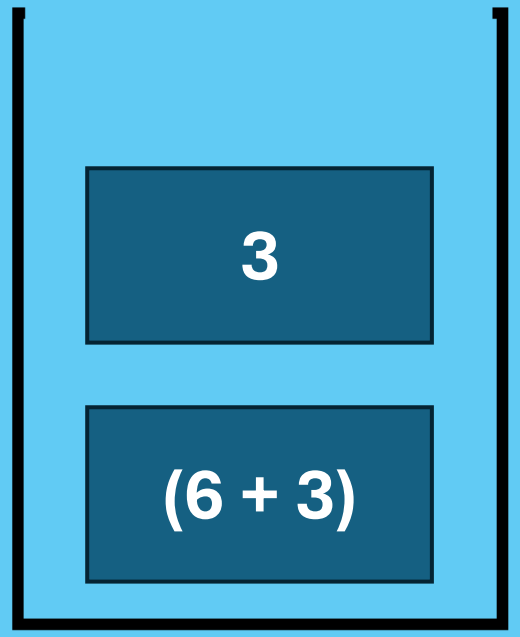
Logik



/



numberStack



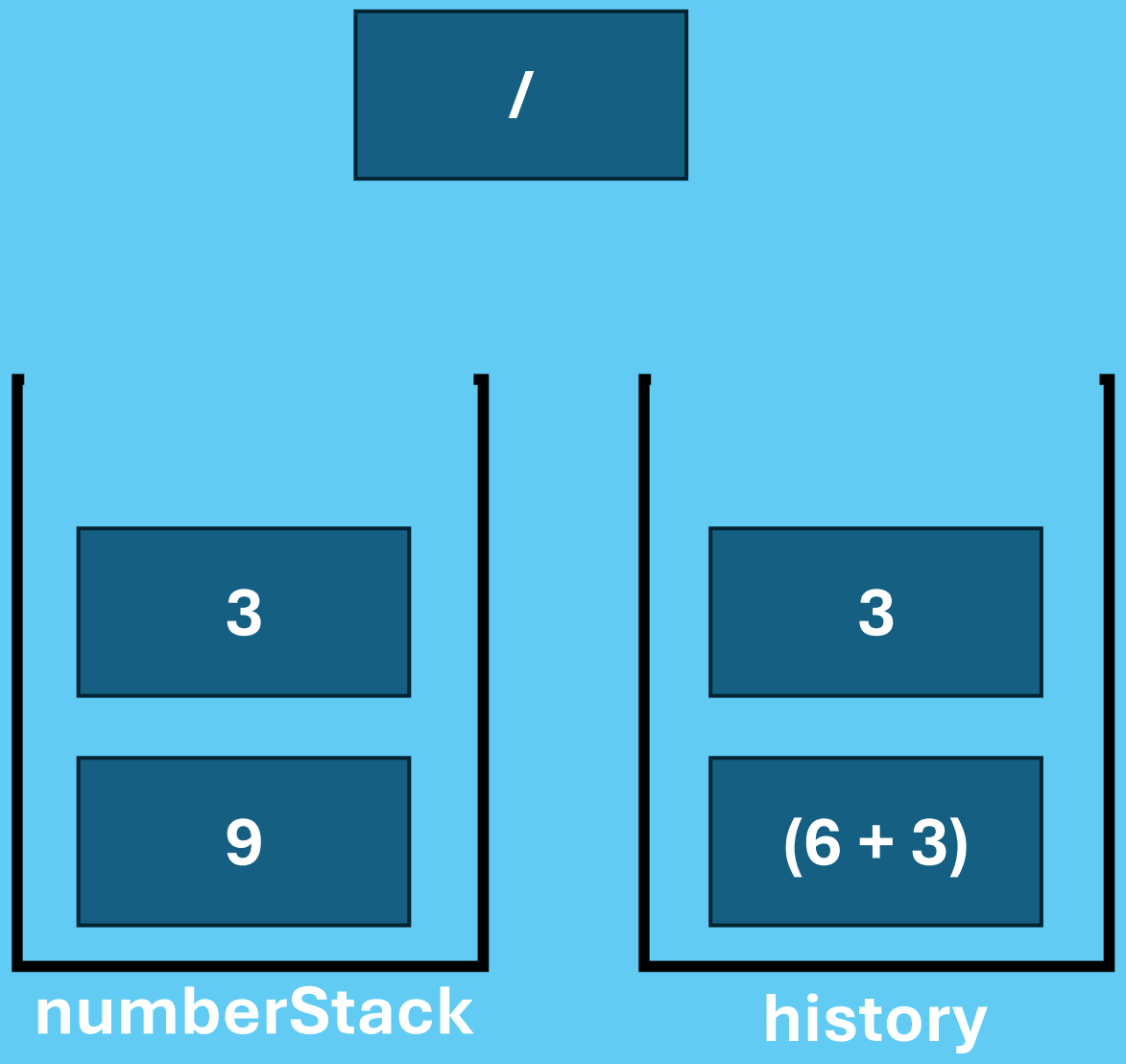
history

Logik

Logik?

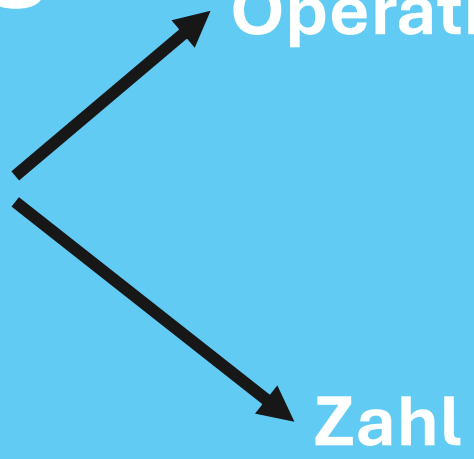
Operation

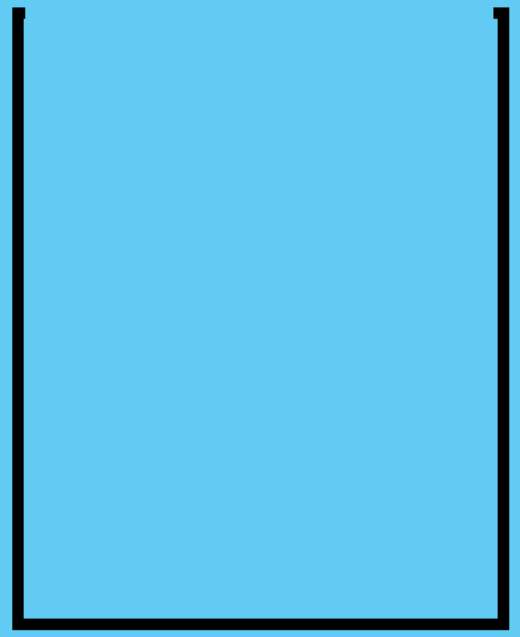
Zahl

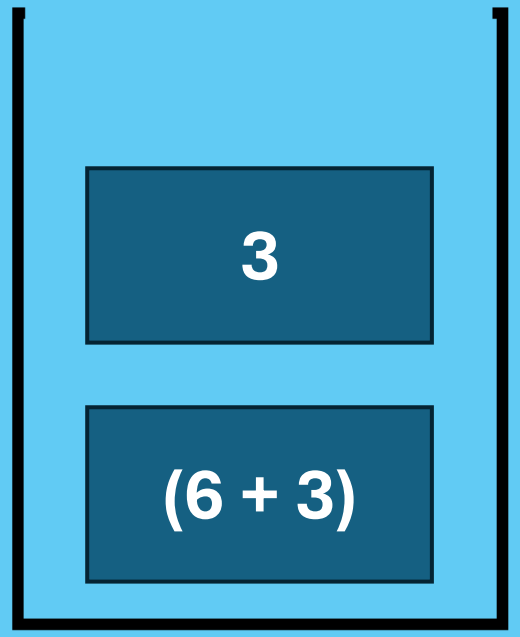


Logik

Logik?





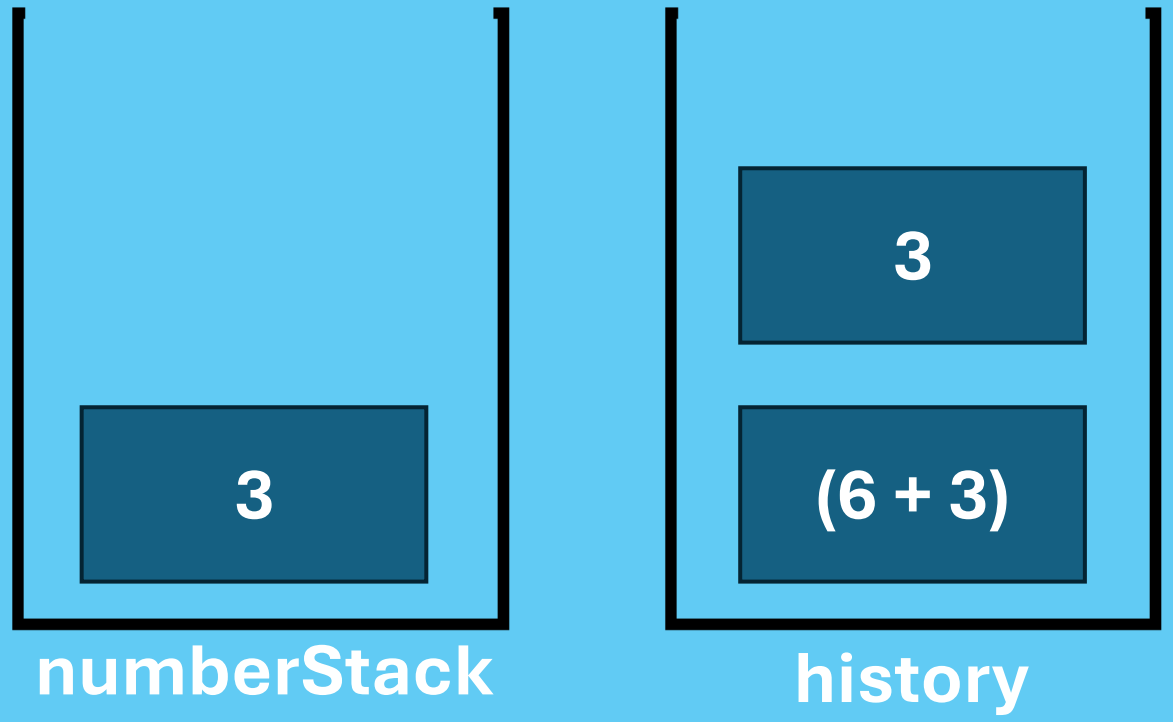


Logik

Logik?

Operation

Zahl

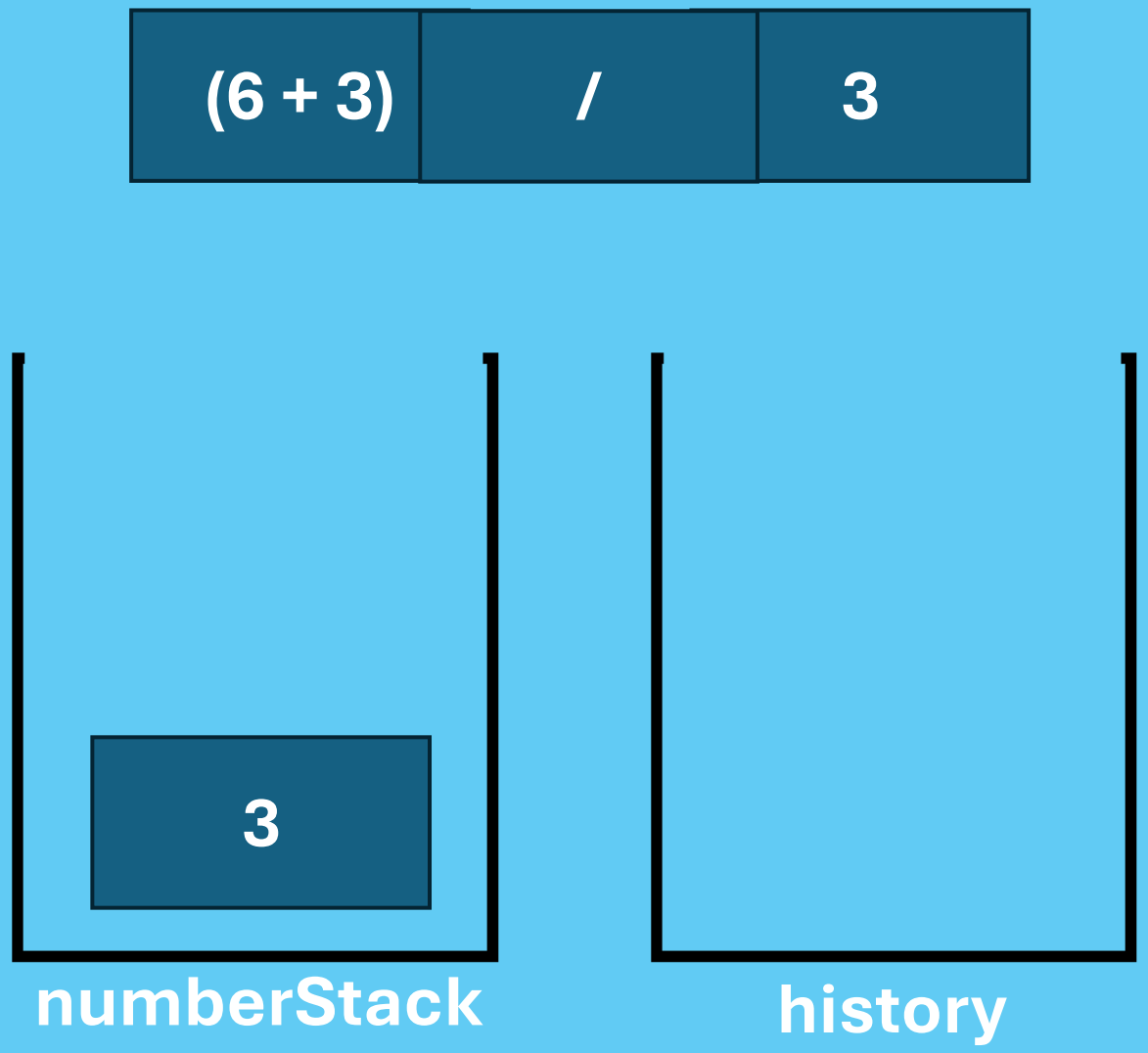


Logik

Logik?

Operation

Zahl

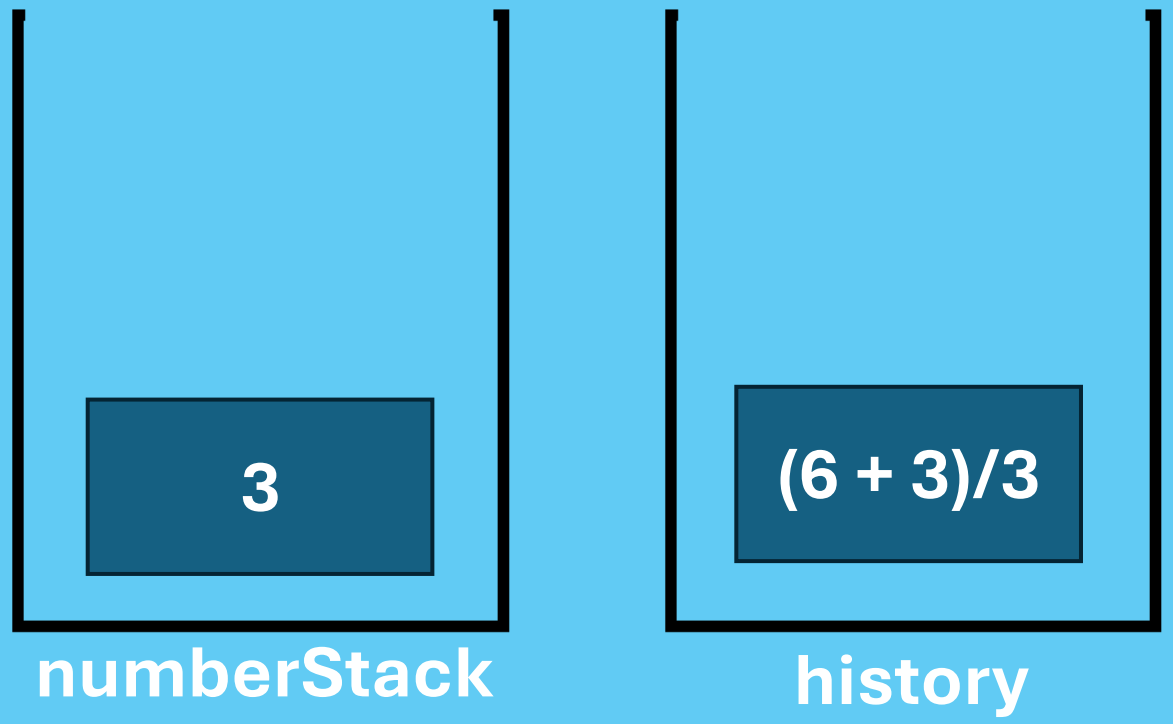


Logik

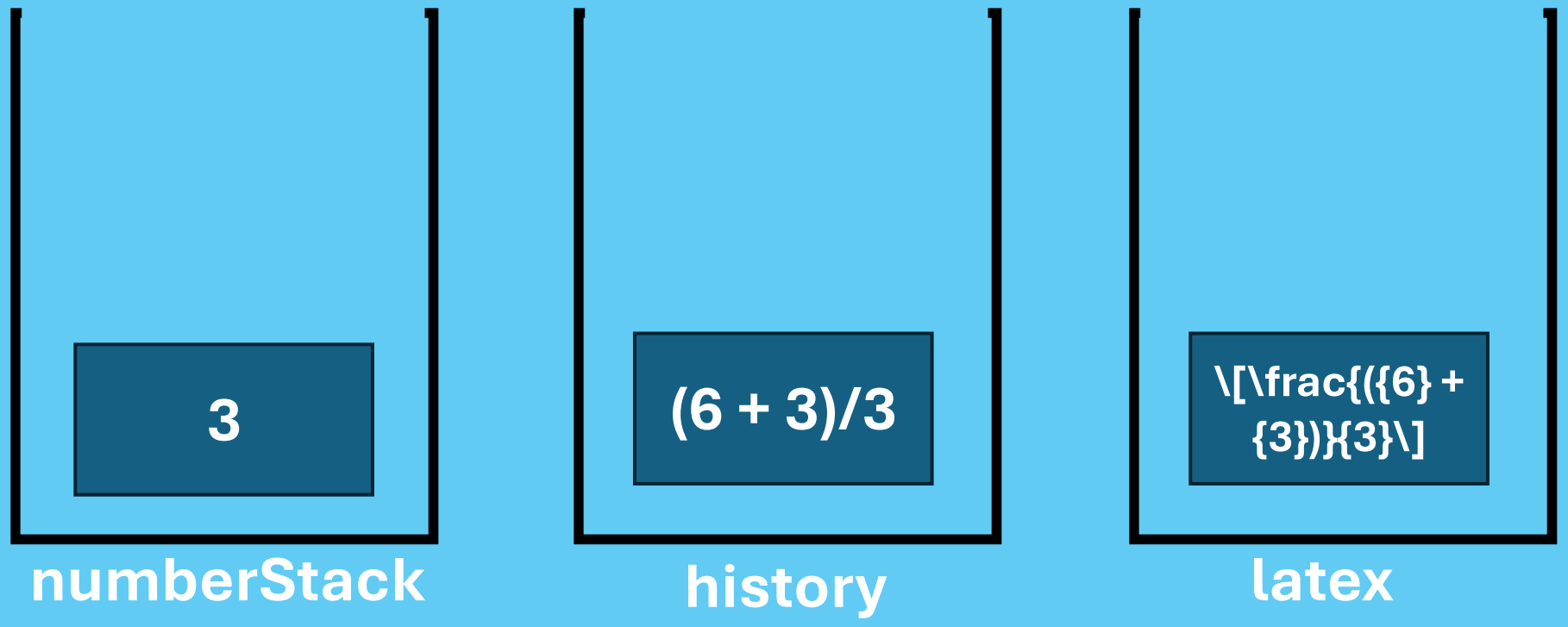
Logik?

Operation

Zahl



Logik



Quellen

- https://go.dev/
- https://pkg.go.dev/
- https://en.wikipedia.org/wiki/Go_(programming_language)
- Go: The Complete Developer's Guide (Golang) | Udemy