# Unterstützung für generische Programmierung

**Daniel Brecht**

# Agenda

1. Allgemeines
2. Datentypen:
    - 2.1 Primitive Datentypen
    - 2.2 Komplexe Datentypen
    - 2.3 Referenzdatentypen
    - 2.4 Interface
3. Datenstrukturen

# 1. Allgemeines

- Stark typisiert

```
var myInt int = 10
myInt = "Hello World" //kann nur Integer speichern
```

- Alias

```
//neuer Typ mit Struktur von float64
type speed float64
```

- Nominal Typing

```
//kann unterschiedliche Typen nicht vergleichen
if speed(30.5) == float64(30.5){}
```

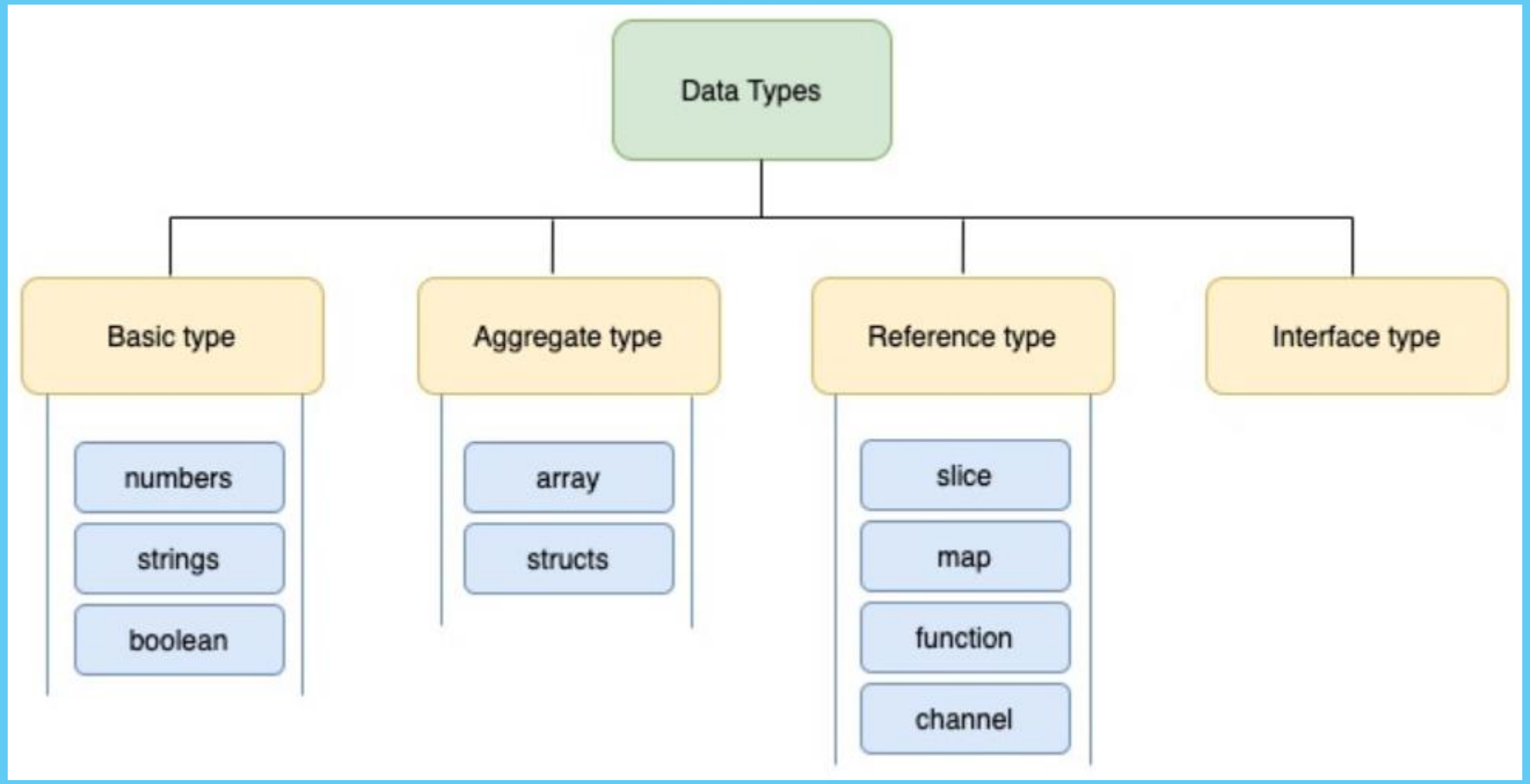
# 1. Allgemeines

- Public und private

```
var Value string = "Public" //groß geschrieben
func PublicFunc(){
    fmt.Println("This is a public function")
}
```

```
var value string = "private" //klein geschrieben
func privateFunc(){
    fmt.Println("This is a private function")
}
```

# 2. Datentypen




https://miro.medium.com/v2/resize:fit:672/1*woZaBaFrmDR6N-RLNEjnvg.png

# 2.1 Primitive Datentypen

## Ganzzahlen

| Typ | Größe | Wertebereich |
|---|---|---|
| int8 | 8 Bit | -128 bis 127 |
| int16 | 16 Bit | -32.768 bis 32.767 |
| int32 (rune) | 32 Bit | -2.147.483.648 bis 2.147.483.647 |
| int64 | 64 Bit | -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807 |
| uint8 (byte) | 8 Bit | 0 bis 255 |
| uint16 | 16 Bit | 0 bis 65.535 |
| uint32 | 32 Bit | 0 bis 4.294.967.295 |
| uint64 | 64 Bit | 0 bis 18.446.744.073.709.551.615 |
| int | **Plattformabhängig**: 32 Bit **oder** 64 Bit | Entspricht Wertebereich von int32 **oder** int64 |
| uint | **Plattformabhängig**: 32 Bit **oder** 64 Bit | Entspricht Wertebereich von uint32 **oder** uint64 |
| uintptr | **Plattformabhängig**: 32 Bit **oder** 64 Bit | **32 Bit:** 0x00000000 bis 0xFFFFFFFF<br>**64 Bit:** 0x0000000000000000 bis 0xFFFFFFFFFFFFFFFF |

# 2.1 Primitive Datentypen

## Gleitkommazahlen

| Typ | Größe | Wertebereich |
|---|---|---|
| float32 | 32 Bit | $\pm 1.18 \times 10^{-38}$ bis $\pm 3.4 \times 10^{38}$ |
| float64 | 64 Bit | $\pm 2.23 \times 10^{-308}$ bis $\pm 1.8 \times 10^{308}$ |

## Komplexe Zahlen

| Typ | Größe | Wertebereich |
|---|---|---|
| complex64 | 64 Bit | Realteil float32; Imaginärteil float32 |
| complex128 | 128 Bit | Realteil float64; Imaginärteil float64 |

## Sonstige

| Typ | Größe | Wertebereich |
|---|---|---|
| bool | 8 Bit | true oder false |
| string | Abhängig von Länge und Ihnhalt | Abhängig von Länge und Ihnhalt |

# 2.1 Primitive Datentypen

- Strings sind unveränderlich

```
myString := "Hello World"
myString = "Hallo Welt" //legt neuen String an
```

- Benötigte Bits zur Darstellung von bool: 1
  - ➢ 1 Byte (8 Bit) besser adressierbar

- Explizite Typkonvertierung
  - ➢ möglicher Datenverlust

```
var myInt int = 3
var myFloat float64 = float64(myInt)
```

```
value := 5.99
fmt.Println(int(value)) //Ausgabe: 5
```

# 2.1 Primitive Datentypen

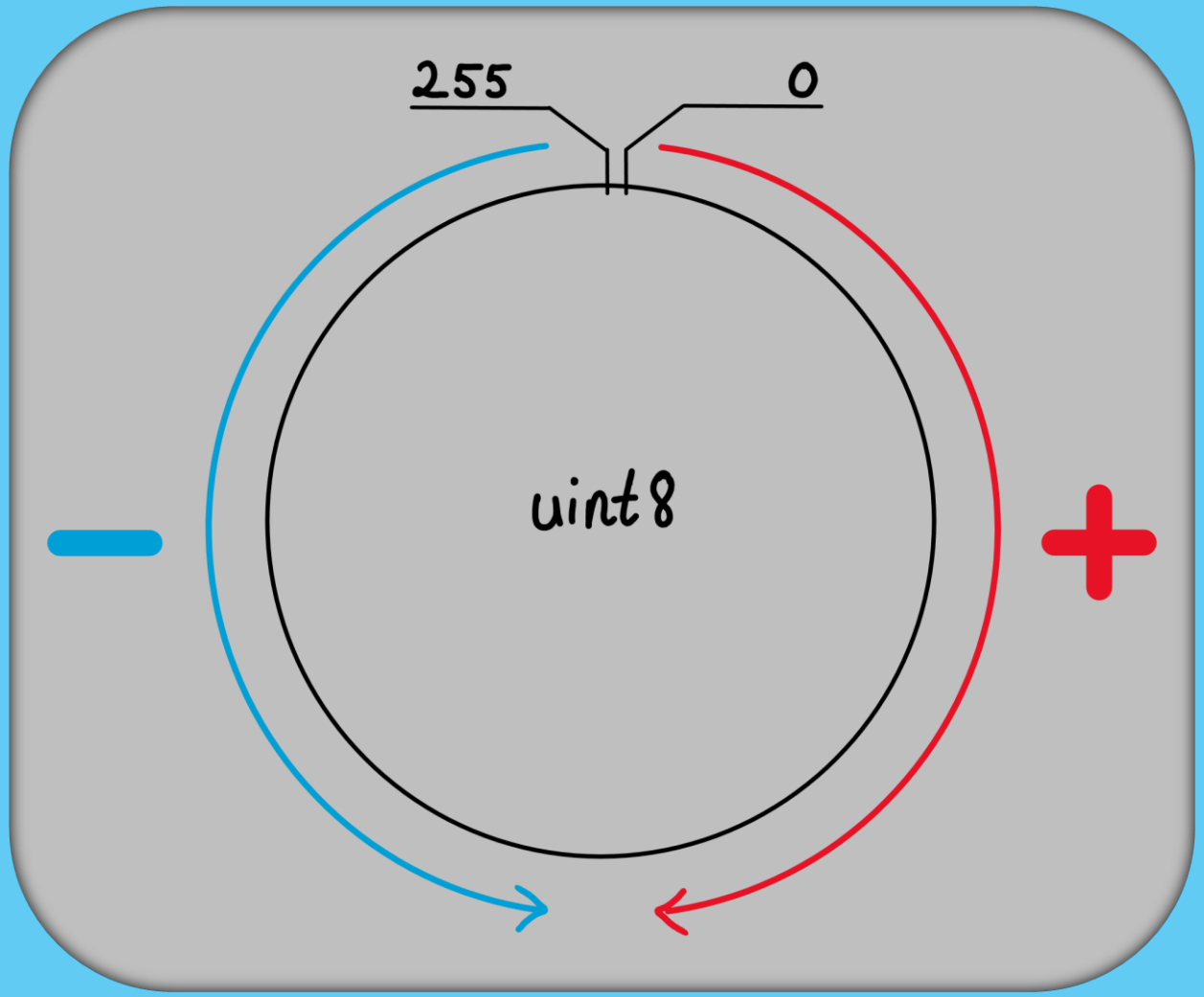
- Over- Underflow

```
var max uint8 = 255 + 1          //Fehler: Compiler erkennt Overflow
```

```
max := uint8(255)                //größter Wert
fmt.Println(max + 1)             //Ausgabe: 0
```

```
var myInt int = 256
var myUint uint8 = uint8(myInt) //Overflow durch Konvertierung
fmt.Println(myUint)              //Ausgabe: 0
```

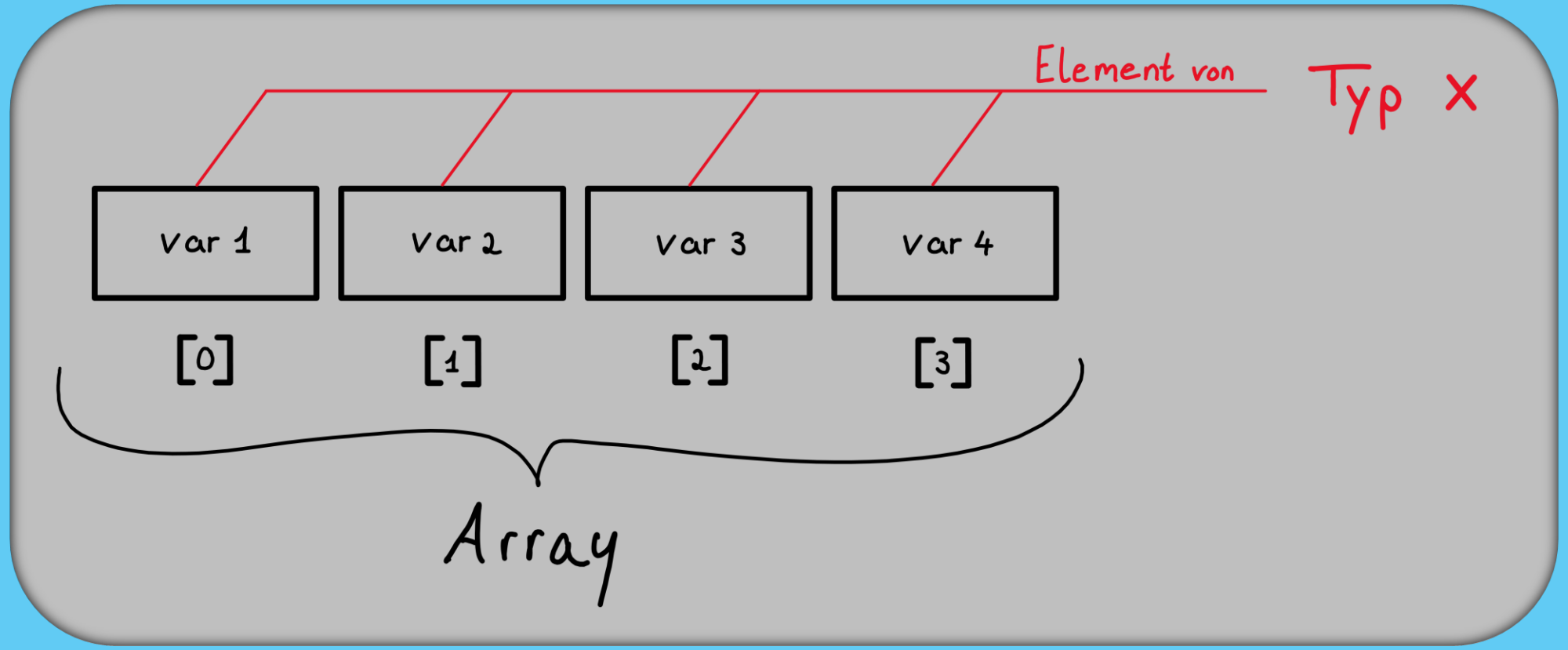
# 2.1 Primitive Datentypen




Quelle: Zeichnung von Daniel Brecht

# 2.2 Komplexe Datentypen

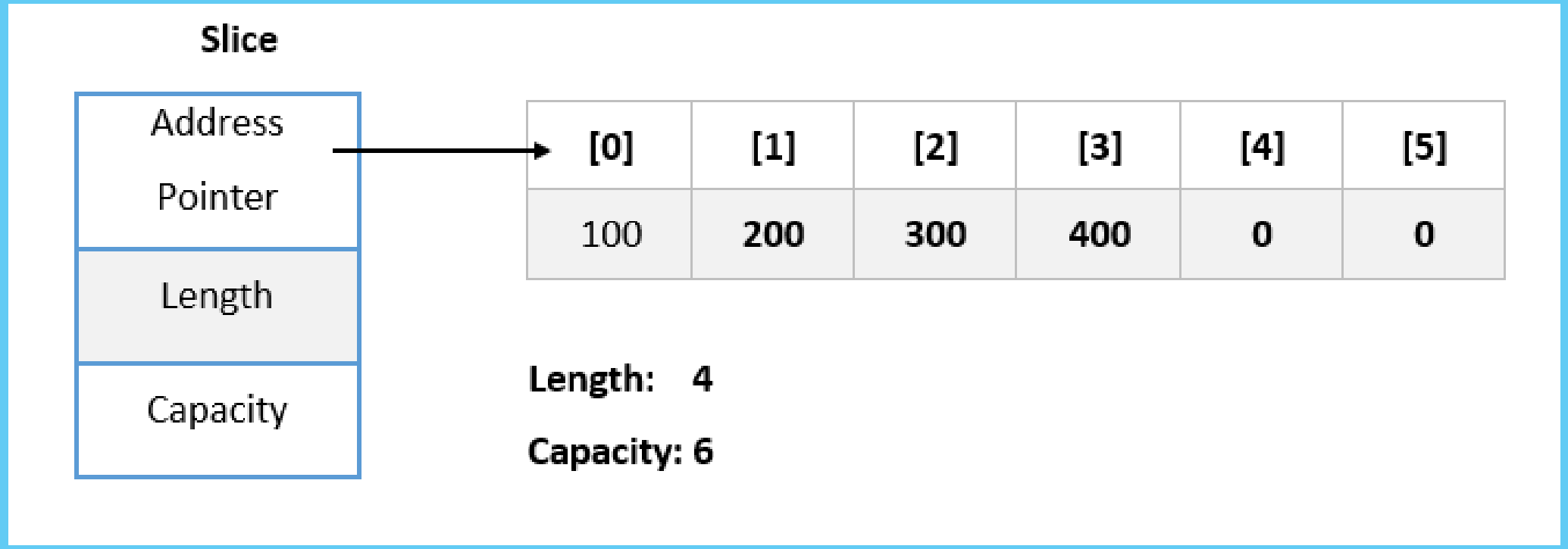
- array, struct, func
- Zusammengesetzt aus verschiedenen Datentypen




Quelle: Zeichnung von Daniel Brecht

# 2.3 Referenzdatentypen

- pointer, slice, map, channel
- Speichern Referenz auf Werte




Quelle: https://miro.medium.com/v2/resize:fit:1400/1*PW4Y8P0_gTspgYwcxfDrtQ.png

# 2.4 Interface

- Bestimmt Funktionen zur Implementierung
  - ➢Erforderlich zur Implementierung
- Implizite Implementierung

```
type shape interface {
    area() float64
}

type circle struct {
    radius float64
}

func (c *circle) area() float64 {
    result := c.radius * math.Pi
    return result
}
```

```
myCircle := circle{4}
myShape := shape(&myCircle)
fmt.Println(myShape.area()) //Polymorphie
```

# 2.4 Interface

- Generische Programmierung durch Constraints
- Constraints any & comparable

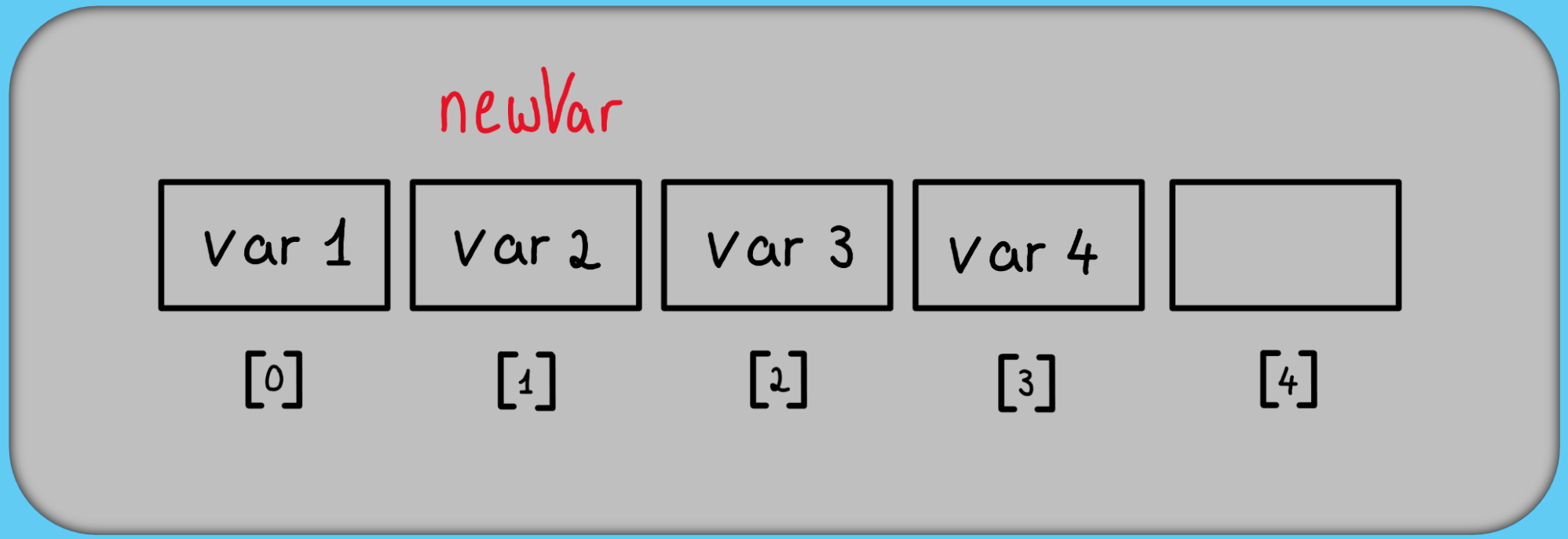
```
type number interface {
    ~int | ~int32 | ~int64
}

func add[T number](numA, numB T) T{
    return numA + numB
}
```

```
type any = interface{}
```

# Datenstrukuren:
# Liste, Stack, Queue

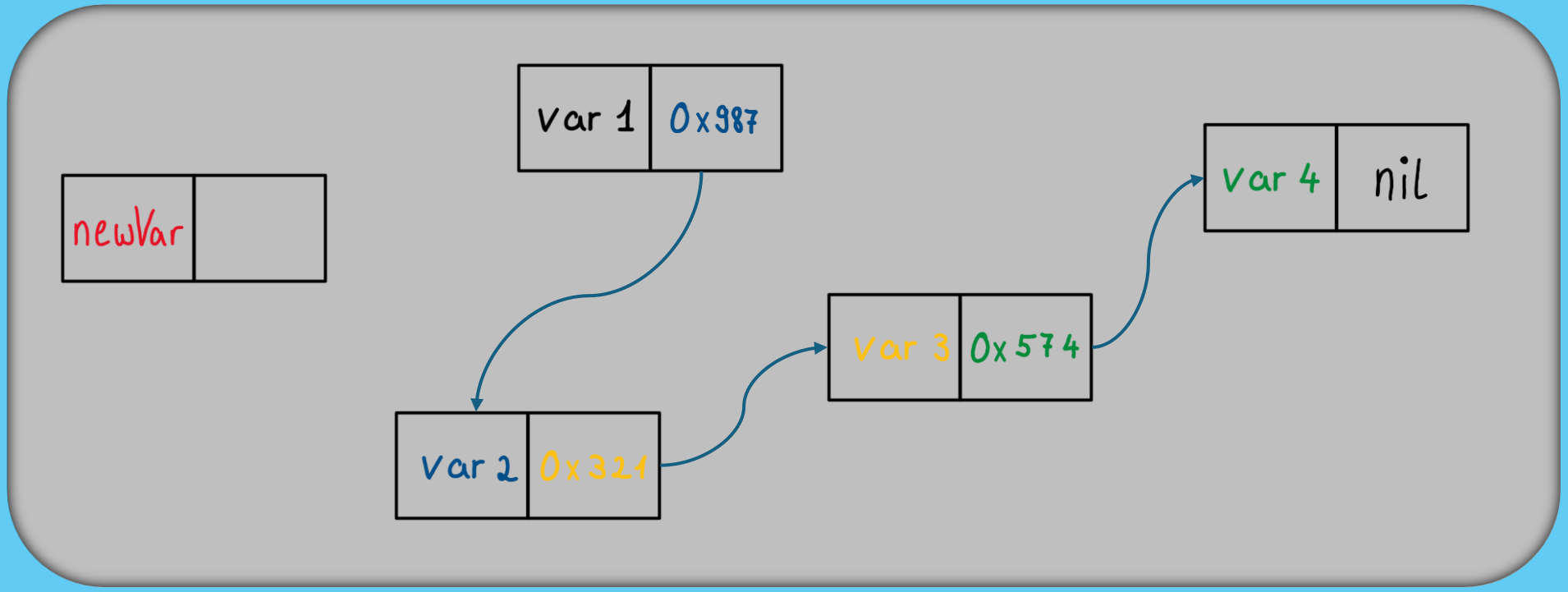
# Datenstrukuren:
# Liste, Stack, Queue






Quelle: Zeichnung von Daniel Brecht

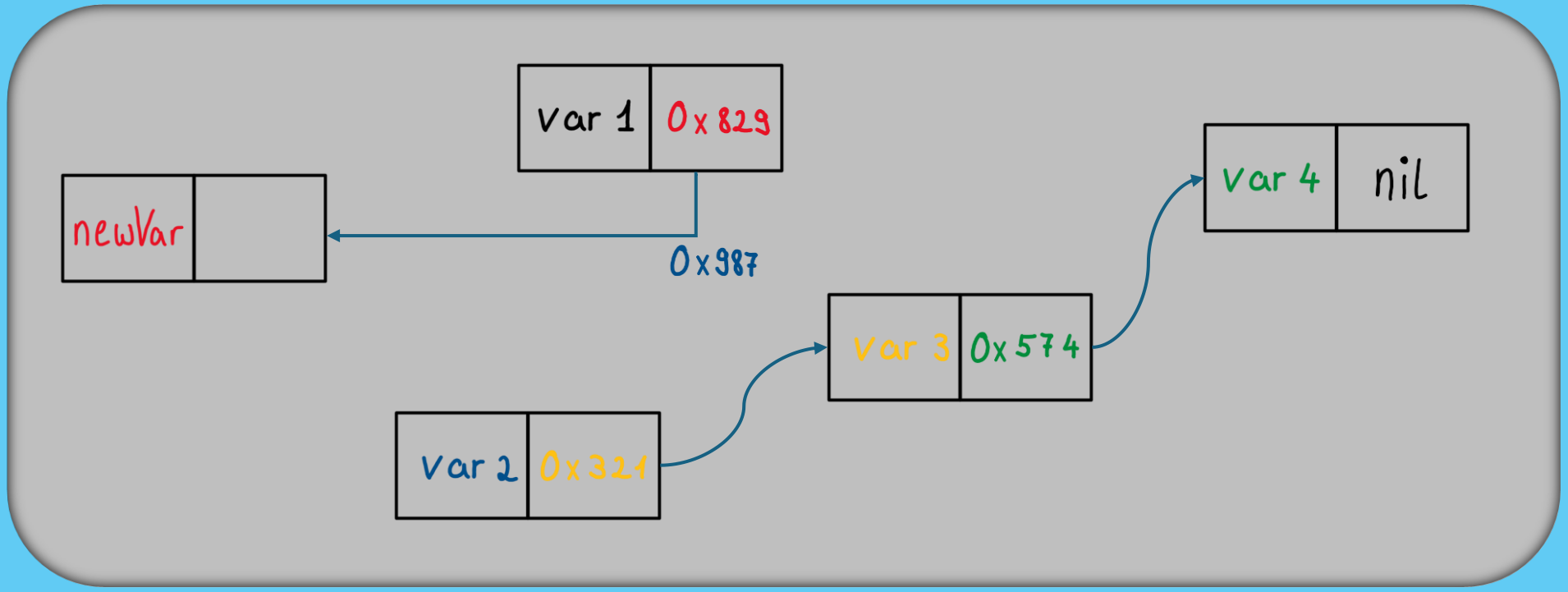
# Datenstrukuren:
# Liste, Stack, Queue






Quelle: Zeichnung von Daniel Brecht

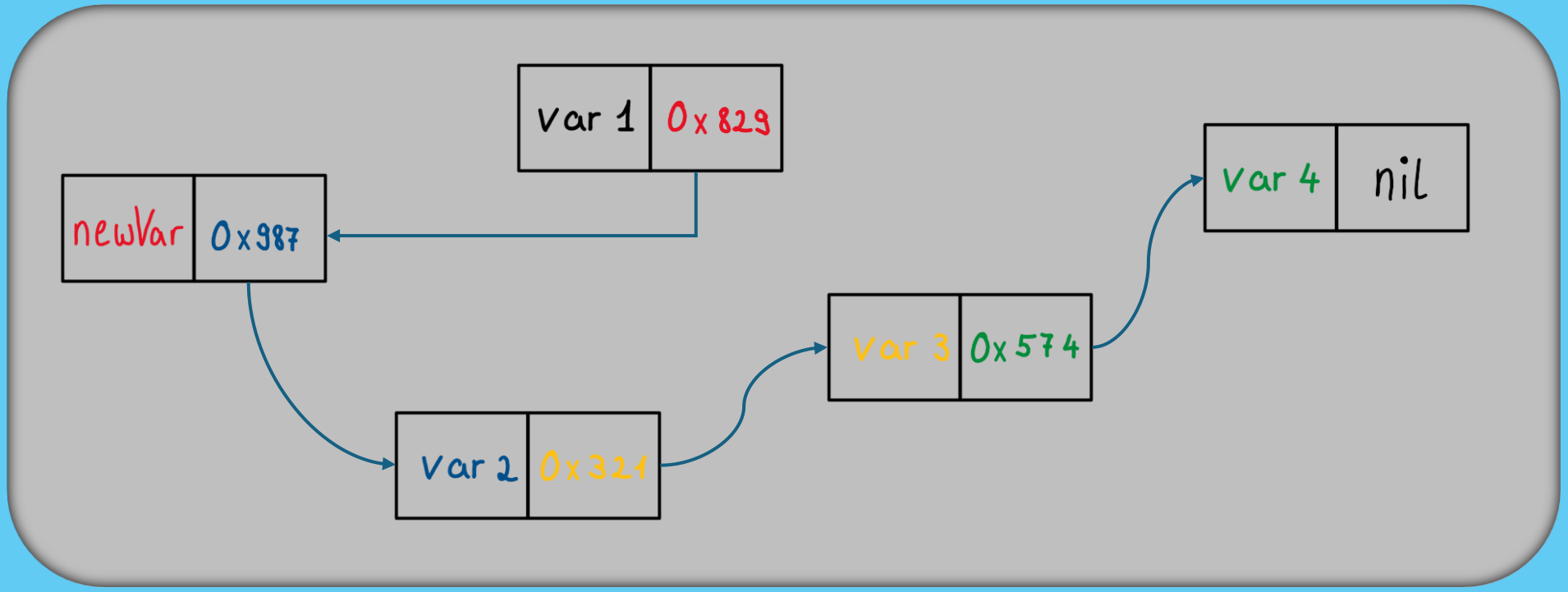
# Datenstrukuren:
# Liste, Stack, Queue






Quelle: Zeichnung von Daniel Brecht

# Datenstrukuren:
# Liste, Stack, Queue






Quelle: Zeichnung von Daniel Brecht

# Danke für eure Aufmerksamkeit