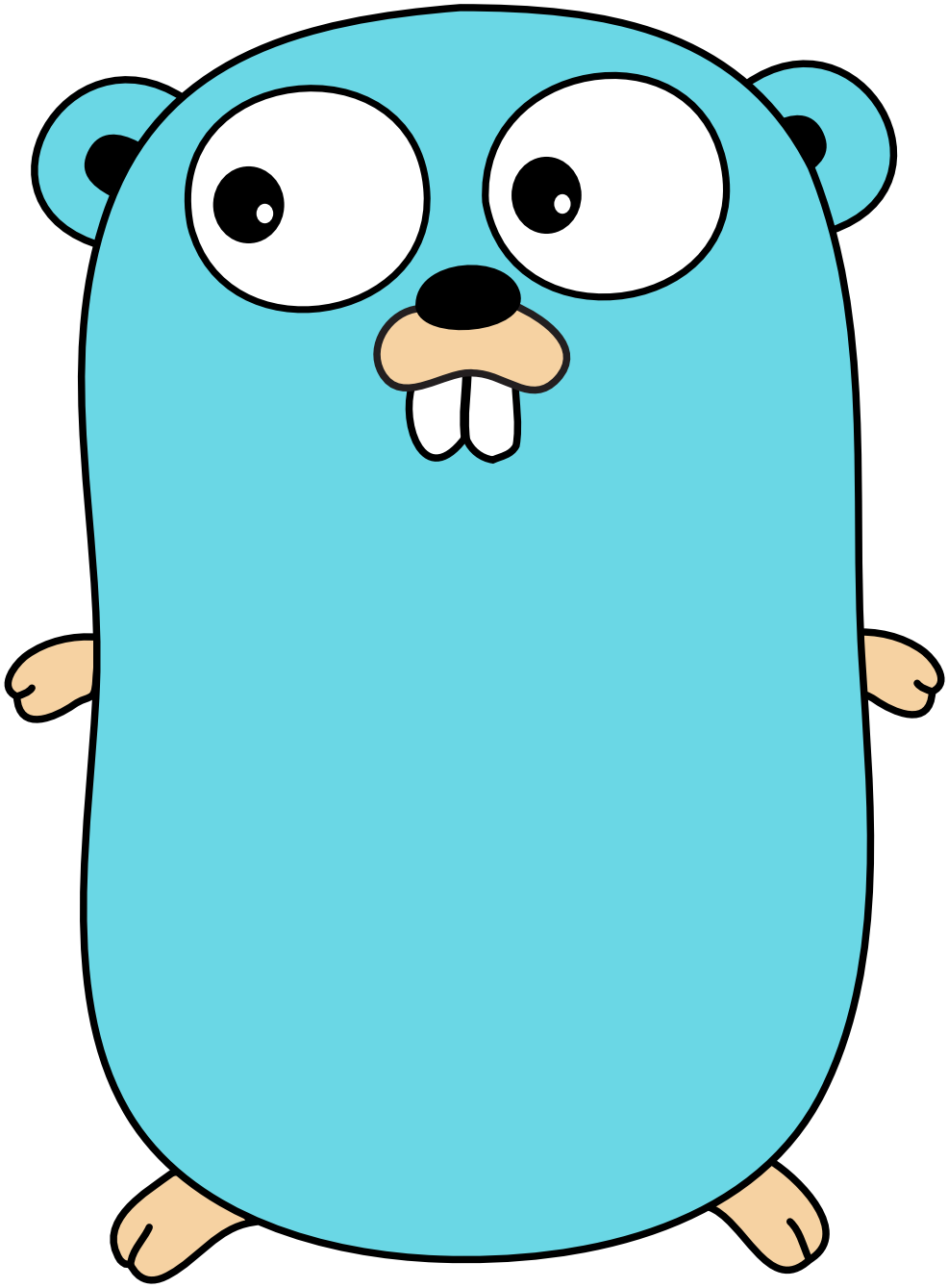
Objektorientierte Programmierung in Go



Inhaltsverzeichnis

Methoden

Die Programmiersprache Go hat keine Klassen. Es ist aber möglich Methoden an Typen zu definieren. Eine **Methode** ist eine Funktion mit einem speziellen *Receiver*-Argument. Der **Receiver** wird in Klammern zwischen dem `func`-Schlüsselwort und dem Namen geschrieben und macht eine Funktion zu einer Methode. Der Name des Receivers kann frei gewählt werden und entspricht dem `this`-Pointer in anderen Sprachen der C-Familie wie beispielsweise *Java* oder *C#*.

```
type Circle struct {
    R float64
}

func (k Circle) Area() float64 {
    return math.Pow(k.R, 2) * math.Pi
}
```

Methoden können nur für Typen definiert werden welche aus dem selben Package stammen. Es ist nicht möglich nachträglich Methoden zu bereits vorhandenen Typen hinzuzufügen. Methoden können allerdings auch an Typen definiert werden die keine `struct`-Typen sind.

```
type Double float64

func (d Double) Round() Double {
    return Double(math.Round(float64(d)))
}
```

Um die definierten Methoden aufzurufen muss dann zuerst ein **Objekt** des Typs **instanziiert** werden. Eine Methode wird dann mit dem Member-Opertor (`.`) direkt an dem Objekt aufgerufen.

```
func main() {
    var k = Circle{R: 2.5}
    var area = k.Area()
    fmt.Println(area)
    var d = Double(4.89)
    fmt.Println(d.Round())
}
```

Pointer Receiver

Wie bei Parametern gilt auch beim Reveiver, dass der Wert der Methode als Kopie übergeben wird (**Call-By-Value**). Wenn ich aus einer Methode heraus Werte des Objekts ändern möchte, muss der Receiver ein Pointer sein.

```
func (k *Circle) SetRadius(r float64) {
    k.R = r
}
```

Struct Embedding

Ein `struct`-Typ kann in einen anderen `struct`-Typ eingebettet werden, dann *erbt* der neue `struct`-Typ alle Methoden und Eigenschaften des eingebetteten `struct`-Typs.

```
type Person struct {
    FirstName, LastName string
}

type Customer struct {
    Person
    CustomerID int
}

func main() {
    var customer Customer
    customer.FirstName = "John"
    customer.LastName = "Doe"
    customer.CustomerID = 123
}
```

Interfaces

Ein **Interface**-Typ definiert eine Sammlung von Methoden-Signaturen. Ein Wert mit einem **Interface**-Typ kann jeden Wert enthalten der alle definierten Methoden implementiert.

```
type Shape interface{
    Area() float64
    Circumference() float64
}
```

In der Programmiersprache Go werden Interfaces implizit dadurch implementiert, dass alle im Interface Definierten Methoden vorhanden sind. Eine explizite Angabe mit beispielsweise einem "implements"-Schlüsselwort ist nicht möglich und auch nicht notwendig. Der Nullwert eines mit einem Interface-Typ ist `nil`. Wenn Methoden an einem Typ mit Pointer-Receivern definiert sind, muss der Wert welcher einer Variable mit einem Passendem Interface-Typ zegewiesen werden soll ein Pointer sein.

```
var shape Shape // Nullwert ist nil
shape = &Circle{7.3} // Pointer auf Circle-Objekt zuweisen
```

Interface embedding

Ein Interface-Typ kann beliebig viele andere Interface-Typen einbetten. Der neue Interface-Typ *erbt* dann alle Methoden-Signaturen der eingebetteten Interface-Typen.

```
type Named interface {
    Name() string
}
type NamedShape interface {
    Shape, Named
    fmt.Stringer
    SomeMethod(someParam string) string
}
```

Type assertions

Das Zuweisen eines konkreten Wertes zu einer Varaible mit einem akstrakten Interface-Typ funktioniert ohne Typenumwandlung. Bei einer Zuweisung in die andere Richtung (abstrakt zu konkret) muss der Typ aber explizit umgewandelt werden.

Eine Type-Assertion wandelt den Typ eines Interface-Typ-Wertes um. Optional kann mit einem weiten, Boolschen Wert geprüft werden ob die Umwandlung erfolgreich war.

```
    var shape Shape = Circle{5.6}
    var circle, ok = shape.(Circle)
```

Type switches

Mit einem Type-Switch kann der Datentyp eines Wertes zur Laufzeit ermittelt werden.

```
switch t := shape.(type) {
    case Circle:
    // ...
    case Square:
    // ...
    default:
    // ...
}

var x any = 5
switch t := x.(type) {
    case int:
    // ...
    case string:
    // ...
    default:
    // ...
}
```

Generics

Typen-Parameter

Funktionen können mit Hilfe von Typen-Parametern so geschrieben werden, das sie mit unterschiedlichen Datentypen verwendet werden können. Ein oder mehrere Typen-Parameter können in eckigen Klammern (`[ ]`) zwischen dem Namen under Parameter-Liste geschrieben werden. Der Type-Constraint eines Typen-Parameters schränkt die möglichen Typen ein. Neben den Built-In-Constraints `any` und `comparable` können dort beliebige andere Typen angegeben werden.

```
func Index[T comparable](s []T, x T) int {
    for i, v := range s {
        if v == x { return i }
    }
    return -1
}
```

Mit | getrennt können auch mehrer Type-Constraints angegeben werden.

```
func Sum[T int|int32|float64](args ...T) {
    var sum T
    for _, num := range args {
        sum += num
    }
    return sum
}
```

Interface-Typen können verwendet werden um Custom-Type-Constraints erstellt zu werden.

```
type Number interface {
    int | int8 | int16 | int32 | int64 | float32 | float64
}

func Sum[T Number](args ...T) {
    var sum T
    for _, num := range args {
        sum += num
    }
    return sum
}
```

Generic types

Zusätzlich zu Generics durch Typen-Parameter bei Funktionen können auch `struct`- und `interface`-Typen mit Typen-Parametern definiert werden.

```
type ImmutablePair[T any] interface {
 Left() T
 Right() T
}

type Pair[T any] struct {
 left, right T
}
```

Bei der Definition der Mothoden eines Typs können die am Typ definierten Typen-Parameter ebenfalls verwendet werden.

```

func (p Pair[T]) Left() T {
 return p.left
}

func (p *Pair[T]) SetLeft(left T) {
 p.left = left
}

func (p Pair[T]) Right() T {
 return p.right
}

func (p *Pair[T]) SetRight(right T) {
 p.right = right
}
```