FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Konzepte moderner Programmiersprachen (KmPS)

(Wahlfach, Modulnummer 55685)

Wintersemester 2025/2026

Prof. Dr. Andreas Claßen
(zusammen mit Prof. Dr. Heinrich Faßbender)
Fachbereich 5 Elektrotechnik und Informationstechnik
FH Aachen

"Objekt-Orientierung" in Go

`struct` Datentyp

… ähnlich wie in C.

Beispiel:

```
package main
import "fmt"

type Point2D struct {
    X int
    Y int
}

func main() {
    p := Point2D{1, 2}
    p.X = 22
    fmt.Println(p) // => {22 2}
}
```

Methoden

… sind Funktionen mit einem speziellen *receiver* Parameter.
Werden "nachträglich" zum `type` hinzugefügt! Aber nur im selben Package.

```
type Point2D struct {
    X, Y int
}

func (p Point2D) Print() {
    fmt.Println("Point2D:", p.X, p.Y)  // … d.h. kein festes "this" / "self"
}

func main() {
    p := Point2D{5,6}
    p.Print()
}
```

Interfaces (Interface Types)

… als Satz von Methodensignaturen: ***method set***.

Beispiel:

```
type IMirror interface {
    Mirror() // receiver-Parameter in der der Methodensignatur nicht sichtbar
}

func main() {

    var m IMirror // Variable vom Interface-Typ.
                  // Aber kein Wert vom Typ IMirror selbst, da nicht definiert ist,
                  // wie diese Werte auszusehen hätten. Nur ihre Methoden sind "vorgezeichnet" …
    // ...
}
```

Interfaces (Interface Types)

Variable des Interface-Typs kann jeden Wert speichern,
der die geforderten Methoden implementiert.
Keine explizite Deklaration via `implements` o.ä. nötig!

```
type IMirror interface {
    Mirror()
}

type Point2D struct { X, Y int }

func (p_ptr *Point2D) Mirror() { p_ptr.X = (-1)*p_ptr.X ; p_ptr.Y = (-1)*p_ptr.Y }

func main() {
    p := Point2D{5,6}
    var m IMirror
    m = &p     // … m = p geht nicht, da Mirror() nur für Pointer-Receiver definiert!
    m.Mirror()
}
```

Prüfung der Interface-Kompatibilität

… beim *Übersetzen* des Programms.

Interfaces komplett entkoppelt von den Typen, die das Interface implementieren.
Kein `implements`, `extends` o.ä.
Weniger Aufwand und große Flexibilität bei der Programmierung.
Keine Vererbung, denn damit "programmiert" man `implements` Beziehung …

Kombiniert starke Korrektheitsprüfung durch statische Typisierung
mit Minimum an Typinformationen und Typbeziehungen!

Wiederverwendung durch **Type Embedding**

… "importiert" Attribute und Methoden aus dem "inneren, eingebetteten Typ"
in den "äußeren, einbettenden" `struct` Typ.

… damit ähnlich wie Vererbung, bei der ja auch Attribute & Methoden weitergegeben werden.

Die Komponente heisst dann wie der eingebettete Typ …

```
type User struct { Name string }
func (u User) Print() { fmt.Println("User:", u.Name) }

type Admin struct { User }

func main() {
    admin := Admin { User : User { Name: "Peter"} }
    admin.Print()
}
```

Wiederverwendung durch **Interface Embedding**

… "importiert" Methoden aus dem "inneren, eingebetteten Interface"
in das "äußere, einbettende" `interface`.

… damit ähnlich wie Vererbung, bei der ja auch Methoden weitergegeben werden.

D.h. alle Methoden des eingebetteten Interfaces können auch über den äußeren Interface-Typ aufgerufen werden.

```
type ReadWriter interface {
    Read(b Buffer) bool
    Write(b Buffer) bool
}

type File interface {
    ReadWriter
    Locker
    Close()
}
```