

OO in Java versus Go

Oliver Heim (Matrikelnr: 67890)

10. Januar 2024

Inhaltsverzeichnis

1	Beschreibung von Java Beispielen	3
1.1	Beispiel 1: Nomiale Subtypisierung	3
1.2	Beispiel 2: Virtuelle Methoden	4
2	Neuimplementierung der Beispiele in Go	6
2.1	Beispiel 1 in Go	6
2.2	Beispiel 2 in Go	7

1 Beschreibung von Java Beispielen

In diesem Abschnitt werden Java Code-Beispiele beschrieben, welche nominale Subtypisierung und virtuelle Methoden verwenden.

1.1 Beispiel 1: Nominale Subtypisierung

```
public class Person {
    String name;
    public Person(String name) {
        this.name = name;
    }

    public String sprechen() {
        return "Ich bin Person " + this.name;
    }
}

public class Mitarbeiter extends Person {
    int mitarbeiterNr;
    public Mitarbeiter(String name) {
        super(name); // Konstruktor der Oberklasse aufrufen.
    }
}

public class Kunde extends Person {
    int kundenNr;
    public Kunde(String name) {
        super(name); // Konstruktor der Oberklasse aufrufen.
    }
}

public class Main {
    public static void main(String[] args) {
        Person p = new Person("Max");
        Mitarbeiter m = new Mitarbeiter("Bernd");
        Kunde k = new Kunde("Herbert");

        if (p instanceof Mitarbeiter) { // Hier: False
            System.out.println("Die Person " + p.name + " ist ein
                                Mitarbeiter.");
        } else {
            System.out.println("Die Person " + p.name + " ist kein
                                Mitarbeiter.");
        }
    }
}
```

```

        if (m instanceof Person) { //Hier: True
            System.out.println("Die Mitarbeiter " + m.name + " ist
                               eine Person.");
        } else {
            System.out.println("Der Mitarbeiter " + m.name + " ist
                               keine Person.");
        }

        if (k instanceof Person) { //Hier: True
            System.out.println("Der Kunde " + k.name + " ist eine
                               Person.");
        } else {
            System.out.println("Der Kunde " + k.name + " ist keine
                               Person.");
        }
    }
}

```

In Java wird das nominale Subtyping realisiert. Das bedeutet, dass zwei Typen als gleich gelten, wenn sie den selben Namen haben. Außerdem gilt ein Typ nur als Untertyp eines Types, wenn dies explizit deklariert wurde (Schlüsselwort „*extends*“). Wenn zwei Untertypen vom selben Typ erben, dann haben sie den selben Basistyp. Im Oberen Beispiel ist die Oberklasse „Person“ auch vom Typ „Person“. Die Klassen „Mitarbeiter“ und „Kunde“ erben von der Personen-Klasse (`public class Mitarbeiter extends Person;` `public class Kunde extends Person;`) und sind somit Subtypen von „Person“. Es gilt `Mitarbeiter <= Person` und `Kunde <= Person`.

1.2 Beispiel 2: Virtuelle Methoden

In Java greift das Konzept der Polymorphie und dynamischen Bindung (späte Bindung). Standardmäßig wird für alle Methoden die dynamische Bindung verwendet. Alle nicht-statischen Methoden sind per default virtuelle Methoden. Polymorphie bedeutet, dass die Implementierung der Methode zur Laufzeit erfolgt, basierend auf dem Typ des Objekts, auf dem die Methode aufgerufen wird.

```

public class Mitarbeiter extends Person {
    int mitarbeiterNr;
    public Mitarbeiter(String name) {
        super(name);
    }
    @Override

```

```

        public String sprechen() {
            return "Ich bin Mitarbeiter " + this.name;
        }
    }
    public class Kunde extends Person{
        int kundenNr;
        public Kunde(String name) {
            super(name);
        }

        @Override
        public String sprechen() {
            return "Ich bin Kunde " + this.name;
        }
    }
    public class Buerger extends Person{
        public Buerger(String name) {
            super(name);
        }
    }
    public class Main {
        public static void main(String[] args) {
            Person p = new Person("Max");
            Mitarbeiter m = new Mitarbeiter("Bernd");
            Kunde k = new Kunde("Herbert");
            Buerger b = new Buerger("Manfred");

            System.out.println(p.sprechen());
            System.out.println(m.sprechen());
            System.out.println(k.sprechen());
            System.out.println(b.sprechen());
        }
    }
}

```

Betrachten wir die Klassen „Mitarbeiter“ und „Kunde“, so sehen wir dass die Methode „sprechen()“ mit dem Schlüsselwort „@Override“ versehen ist. Dies stellt sicher, dass zur Laufzeit die Methode „sprechen()“ aus der Oberklasse „Person“ (Siehe: Unterabschnitt 1.1) überschrieben wird. Falls von einer Unterklasse von „Person“ ein Objekt erstellt wird und die Methode „sprechen()“ aufgerufen wird, diese aber nicht in der Unterklasse überschrieben wurde, so wird die sprechen-Methode von „Person“ ausgeführt (siehe Klasse „Buerger“).

2 Neuimplementierung der Beispiele in Go

Im folgenden werden die Beispiele aus Abschnitt 1 in Go neu implementiert. In Go gibt es keine Vererbung und keine Klassen. Stattdessen wird structural subtyping durch Interfaces erreicht.

2.1 Beispiel 1 in Go

```
package main
import "fmt"
// Person ist eine Struktur, die das Verhalten einer Person
// darstellt.
type Person struct {
    name string
}
// Mitarbeiter ist eine Struktur, die das Verhalten eines
// Mitarbeiters darstellt.
type Mitarbeiter struct {
    Person
    mitarbeiterNr int
}
// Kunde ist eine Struktur, die das Verhalten eines Kunden
// darstellt.
type Kunde struct {
    Person
    kundenNr int
}
//Sammlung von Methoden fuer Personen
type person interface {
    //sprechen() string
}
func checkTyp(any interface{}) string {
    switch v := any.(type) {
    case Person:
        return "Der Typ ist Person und hat den Namen: " + v.name
    case Mitarbeiter:
        return "Der Typ ist Mitarbeiter und hat den Namen: " + v.name
    case Kunde:
        return "Der Typ ist Kunde und hat den Namen: " + v.name
    default:
        return "Der Typ ist weder Person noch Mitarbeiter oder Kunde"
    }
```

```

}
func main() {
    p := Person{"Max"}
    m := Mitarbeiter{Person: Person{"Bernd"}, mitarbeiterNr: 123}
    k := Kunde{Person: Person{"Herbert"}, kundenNr: 456}
    fmt.Printf(checkTyp(p) + "\n")
    fmt.Printf(checkTyp(m) + "\n")
    fmt.Printf(checkTyp(k))
}

```

Go erlaubt leere Schnittstellen (interface), die Werte unterschiedlicher Typen halten können. Sie werden oft verwendet, um Funktionen oder Datenstrukturen zu erstellen, die mit verschiedenen Typen arbeiten können, ohne explizit eine Schnittstelle mit spezifischen Methoden zu deklarieren. Mit einem Type-Switch (switch v := any.(type)) kann der konkrete Typ überprüft werden (ähnlich wie instanceof in Java). Der Einsatz von Typschaltern verleiht dem Code eine gewisse Flexibilität. Dadurch kann eine Funktion mit verschiedenen Typen arbeiten, ohne im Voraus genau zu wissen, welchen Typ der übergebene Parameter hat. Im Vergleich zu festen Schnittstellen mit vordefinierten Methoden erlauben leere Schnittstellen und Typschalter in Go eine flexiblere Handhabung von unterschiedlichen Typen, ohne die explizite Festlegung von Schnittstellenanforderungen.

2.2 Beispiel 2 in Go

```

package main

import "fmt"

// Person ist eine Struktur, die das Verhalten einer Person
// darstellt.
type Person struct {
    name string
}

// Mitarbeiter ist eine Struktur, die das Verhalten eines
// Mitarbeiters darstellt.
type Mitarbeiter struct {
    Person
    mitarbeiterNr int
}

```

```

// Kunde ist eine Struktur, die das Verhalten eines Kunden
// darstellt.
type Kunde struct {
    Person
    kundenNr int
}

// Buerger ist eine Struktur, die das Verhalten eines Buerger
// darstellt.
type Buerger struct {
    Person
}

// Sammlung von Methoden fuer Personen
type person interface {
    sprechen() string
}

func (p Person) sprechen() string {
    return "Ich bin Person " + p.name
}
func (m Mitarbeiter) sprechen() string {
    return "Ich bin Mitarbeiter " + m.name
}
func (k Kunde) sprechen() string {
    return "Ich bin Kunde " + k.name
}
func main() {
    p := Person{"Max"}
    m := Mitarbeiter{Person: Person{"Bernd"}, mitarbeiterNr: 123}
    k := Kunde{Person: Person{"Herbert"}, kundenNr: 456}
    b := Buerger{Person: Person{"Manfred"}}

    fmt.Printf(p.sprechen() + "\n")
    fmt.Printf(m.sprechen() + "\n")
    fmt.Printf(k.sprechen() + "\n")
    fmt.Printf(b.sprechen())
}

// Beliebiger Subtyp von Person moeglich
func sprechen(x person) string {
    return x.sprechen()
}

```

In Go gibt es keine klassische Vererbung wie in objektorientierten Sprachen, sondern die Beziehung wird durch das Einbetten von Typen erreicht. Hier wird für jeden Typ eine Struktur angelegt. Die Strukturen „Mitarbeiter“, „Kunde“ und „Buerger“ beinhalten die Struktur „Person“ und somit auch alle Methoden und Felder von „Person“. Anschließend wird für die Funktionen, die eine Person verfügen soll ein Interface („person“) angelegt. Interfaces sind in Go auch Typen. Die Funktionen werden dann von den Strukturen implementiert (unterschiedliche Receiver im Funktionsdefinition). Das Interface „person“ wird verwendet um polymorphe Methodenaufrufe zu ermöglichen. Beim Aufruf der Funktion „sprechen“ wird dann je nach Subtyp(Person, Mitarbeiter, Kunde, Buerger) die richtige Funktion aufgerufen.