Frage: Wie ist das genauer mit den stärkeren Bindungen?

Antwort: Dies kann individuell festgelegt werden, üblich ist, dass „Und“ stärker bindet als „Oder“, also $a \vee b \wedge c$ äquivalent zu $a \vee (b \wedge c)$ ist. Weiterhin erfolgt die Abarbeitung von links nach rechts, was nur relevant ist, wenn es Seiteneffekte gibt, wie es in Programmiersprachen möglich ist. Da diese sogenannten Bindungsregeln in Programmiersprachen unterschiedlich sein können, ist das setzen von Klammern immer eine Alternative. Bei den Bindungsregeln spielen alle Operatoren eine Rolle, z. B. der Cast-Operator () und Bit-Shift >>.

Erinnerung: Entwicklung mit rekursiven Strukturen

Meist gibt es zwei Varianten die Formel-Bibliothek zu erweitern. Die meist bessere Variante ist es, alle speziellen Erweiterungen in die passenden Unterklassen auszugliedern und in der Ober-Klasse Formel die Standardlösung zu schreiben. In der zweiten Variante wird die gesamte Lösung rekursiv in der Ober-Klasse Formel umgesetzt. Im folgenden Beispiel sollen Und und Oder gegeneinander ausgetauscht werden.

Lösung 1

in Formel:

```
public Formel undOderTauschen() {
  List<Formel> tmp = new ArrayList<>();
  for (Formel f : this.operanden) {
    tmp.add(f.undOderTauschen());
  }
  this.operanden = tmp;
  return this;
}
```

in Und:

```
@Override
public Formel undOderTauschen() {
  Formel[] opArray = new Formel[super.operanden.size()];
  for(int i = 0; i < super.operanden.size(); i++) {
    opArray[i] = super.operanden.get(i).undOderTauschen();
  }
  return new Oder(opArray);
}
```

in Oder:

```
@Override
public Formel undOderTauschen() {
  Formel[] opArray = new Formel[super.operanden.size()];
  for(int i = 0; i < super.operanden.size(); i++) {
    opArray[i] = super.operanden.get(i).undOderTauschen();
  }
  return new Und(opArray);
}
```

Lösung 2 in Formel:

```
public Formel undOderTauschen2() {
  List<Formel> opera = new ArrayList<>();
  for (int i = 0; i < this.operanden.size(); i++) {
    opera.add(this.operanden.get(i).undOderTauschen2());
  }
  Formel[] farray = new Formel[opera.size()];
  switch (this.typ) {
    case UND: {
      return new Oder(opera.toArray(farray));
    }
    case ODER: {
      return new Und(opera.toArray(farray));
    }
    default: {
      this.operanden = opera;
      return this;
    }
  }
}
```

Beispielnutzung:

```
public static void main(String[] args) {
  Formel f1 = new Und(new Oder(new Atom("a"), new Atom("b")),
      new Und(new Atom("a"), new Atom("b")));
  Formel f2 = (Formel)f1.clone();
  System.out.println(f1.zeigen());
  System.out.println(f1.undOderTauschen().zeigen());
  System.out.println(f2.undOderTauschen2().zeigen());
}
```

Ausgabe:

```
((a OR b) AND (a AND b))
((a AND b) OR (a OR b))
((a AND b) OR (a OR b))
```

Hinweis: Abhängig von Ihre bisherigen Java-Ausbildung, hier zwei Sprachkonstrukte, die Sie eventuell noch nicht gesehen haben, aber in dieser Veranstaltung vorkommen.

Bei Methoden kann beim letzten Parameter hinter dem Typen … geschrieben werden, wodurch beliebig viele Parameter dieses Typen beim Aufruf kommasepariert am Ende stehen können (ähnliche Ideen gibt es in C++ und anderen Sprachen). Formal entspricht dies einem Array dieses Typens, der wie üblich bearbeitet werden kann.

```
public class DynamischeParameteranzahl {
  public static int summ(int... pars) {
    int erg = 0;
    for(int i=0; i < pars.length; i++) { // "forEach" geht auch
      erg += pars[i];
    }
    return erg;
  }
```

```
  public static void main(String... s) {
    System.out.println("1: " + summ());
    System.out.println("2: " + summ(42));
    System.out.println("3: " + summ(43 ,44, 45, 46, 47));
  }
}
```

Ausgabe:

```
1: 0
2: 42
3: 225
```

Man kann statt `public static void main(String[] args1)` zum Start auch `public static void main(String… args2)` schreiben. Der formale Unterschied ist, dass args1 null sein kann, args2 bei einem Aufruf ohne Parameter ein leerer Array ist.

In Java gibt es die Möglichkeit in Interfaces default-Implementierungen der Methoden anzugeben, die genutzt werden können, wenn eine das Interface implementierende Klasse diese Methode nicht selber realisiert. Formal verwischt damit der Unterschied zu abstrakten Klassen. Der Vorteil ist aber die einfache Erweiterbarkeit. Sollte ein klassisches Interface um eine Methode ergänzt werden, bedeutete dies, dass alle Klassen, die dieses Interface realisieren bearbeitet werden mussten, um die neue Methode zu ergänzen. Diese Notwendigkeit fällt mit default-Methoden weg und die Erweiterung von Interfaces ist so problemlos möglich. Natürlich kann in diesen Implementierungen auf keine Objektvariablen zugegriffen werden, die Nutzung anderer Klassen und Methoden ist wie üblich erlaubt. Generell bleibt die dynamische Polymorphie das mit Abstand wichtigste Konzept der Objektorientierung.

```
public interface InterfaceMitDefaultmethode {
  int wasBinIch = 1;  // hat nichts mit default zu tun

  public default void zeigeZahl(int wert) {
    System.out.println("Zahl: " + wert);
  }
}

public class Bsp implements InterfaceMitDefaultmethode {
  public static void main(String[] args) {
    Bsp bsp = new Bsp();
    bsp.zeigeZahl(42);
  }
}
```

Die Ausgabe ist:

```
Zahl: 42
```

Im obigen Beispiel steht noch eine Variable `wasBinIch`, die zunächst für eine Objektvariable gehalten werden könnte. Das ist nicht der Fall. Da bei Interfaces redundante Informationen weggelassen werden können, sind hier die Schlüsselworte „public“ und „static“ und „final“ (Interfaces können niemals Objektvariablen haben) weggelassen. Meine Empfehlung ist, sie hinzuschreiben, da das Programm so für nicht-javaerfahrene Personen lesbarer wird. Das „public“ vor dem default kann

nebenbei auch weggelassen werden ohne dass sich die Sichtbarkeit der Methode ändert, da ja das Interface schon public ist. Eine Nutzung kann wie folgt aussehen.

```
public class WildeKonstantennutzung {

  public static void main(String[] args) {
    System.out.println("Aha: " + InterfaceMitDefaultmethode.wasBinIch);
  }
}
```

Die Ausgabe ist:

```
Aha: 1
```