

---

# II.4. Erweiterungen von Klassen und fortgeschrittene Konzepte

- 1. Unterklassen und Vererbung
- 2. Abstrakte Klassen und Interfaces
- 3. Modularität und Pakete

# Ähnliche Programmteile

```
public class Bruchelement {  
    Bruch wert;  
    Bruchelement next; ... }  
}
```

```
public class Bruchliste {
```

```
    Bruchelement kopf;
```

```
    void fuegeVorneEin (Bruch wert) {  
        ... }  
}
```

```
    Bruchelement suche (Bruch wert) {  
        ... }  
}
```

```
public class Worтеlement {  
    Wort wert;  
    Worтеlement next; ... }  
}
```

```
public class Wortliste {
```

```
    Worтеlement kopf;
```

```
    void fuegeVorneEin (Wort wert) {  
        ... }  
}
```

```
    Worтеlement suche (Wort wert) {  
        ... }  
}
```

# Allgemeine Liste

```
public class Bruchelement {  
    Bruch wert;  
    Bruchelement next; ... }  
}
```

```
public class Bruchliste {  
    Bruchelement kopf;  
    void fuegeVorneEin (Bruch wert) {  
        ... }  
    Bruchelement suche (Bruch wert) {  
        ... }  
}
```

```
public class Element {  
    Object wert;  
    Element next; ... }  
}
```

```
public class Liste {  
    Element kopf;  
    void fuegeVorneEin (Object wert) {  
        ... }  
    Element suche (Object wert) {  
        ... }  
}
```

# Verwendung der allgemeinen Liste

```
Bruch b1 = new Bruch (1,2) ,  
      b2 = new Bruch (5,4) ;
```

```
Element e;
```

```
Liste l = new Liste ();
```

```
l.fuegeVorneEin (b1) ;  
l.fuegeVorneEin (b2) ;
```

```
e = l.suche (b1) ;
```

```
l.fuegeVorneEin ("hallo") ;  
e = l.suche ("hallo") ;
```

Listen mit beliebigen  
Objekten durcheinander

```
public class Element {  
    Object wert;  
    Element next; ... }  
}
```

```
public class Liste {
```

```
    Element kopf;
```

```
    void fuegeVorneEin (Object wert) {  
        ... }  
}
```

```
    Element suche (Object wert) {  
        ... }  
}
```

# Erweiterung durch Delegation

```
public class Bruchliste {  
  
    Liste dieListe;  
  
    void fuegeVorneEin (Bruch wert) {  
  
        dieListe.fuegeVorneEin (wert);  
  
    }  
  
    Element suche (Bruch wert) {  
  
        dieListe.suche (wert);  
  
    }  
}
```

```
public class Element {  
  
    Object wert;  
    Element next; ... }  
  
public class Liste {  
  
    Element kopf;  
  
    void fuegeVorneEin (Object wert) {  
        ... }  
  
    Element suche (Object wert) {  
        ... }  
}
```

# Gleichheit in der Klasse Liste

```
public class Element {  
    Object wert;  
    Element next; ... }  
}
```

```
public class Liste {
```

```
    Element kopf;
```

```
    Element suche (Object wert) {  
        return suche (wert, kopf);  
    }  
}
```

Vergleicht Objekte  
nicht inhaltlich

```
    static Element suche (Object wert, Element kopf) {  
        if (kopf == null) return null;  
        else if (wert == kopf.wert) return kopf;  
        else return suche (wert, kopf.next);  
    }  
}
```

# Abstrakte Klasse

```
public abstract class Vergleichbar {  
  
    public abstract boolean gleich (Vergleichbar zuvergleichen); ...}  
  
    public class Bruch extends Vergleichbar {  
  
        private int zaehler, nenner;  
  
        public boolean gleich (Vergleichbar zuvergleichen) {  
  
            Bruch b;  
  
            if (zuvergleichen instanceof Bruch) {  
                b = (Bruch) zuvergleichen;  
                return (zaehler * b.nenner == b.zaehler * nenner);  
            }  
  
            else {System.out.println("Kein Bruchvergleich");  
                return false;  
            }  
  
            ...}  
}
```

# Abstrakte Klasse

```
public abstract class Vergleichbar {  
    public abstract boolean gleich (Vergleichbar zuvergleichen); ...}
```

```
public class Bruch extends Vergleichbar {
```

```
    private int zaehler, nenner;
```

```
    public boolean gleich (Vergleich
```

```
        Bruch b;
```

```
        if (zuvergleichen instanceof
```

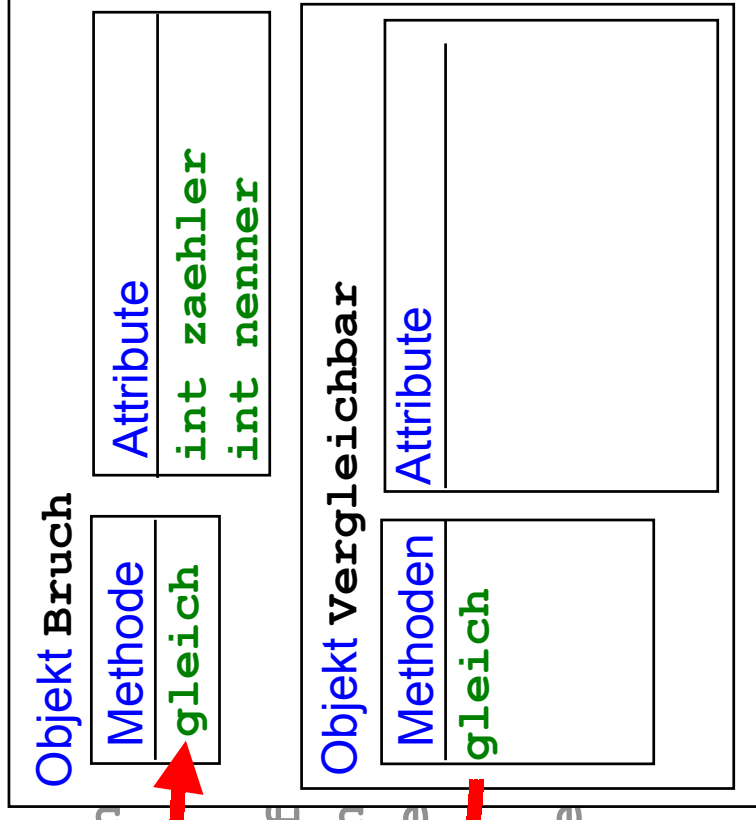
```
            b = (Bruch) zuvergleichen
```

```
                return (zaehler * b.nenne
```

```
        }  
    }
```

```
        else {System.out.println("Ke
```

```
            return false;  
        }  
    }  
    ...}
```





# Liste mit abstrakter Klasse

```
public abstract class Vergleichbar {

    public abstract boolean gleich (Vergleichbar zuvergleichen); ...}

public class Bruch extends Vergleichbar { ... }

public class Wort extends Vergleichbar { ... }

public class Element
    Vergleichbar wert; Element next; ... }

public class Liste {

    Element kopf;

    Element suche (Vergleichbar wert) {return suche (wert, kopf);}

    static Element suche (Vergleichbar wert, Element kopf) {
        if (kopf == null)
            return null;
        else if (wert.gleich(kopf.wert))
            return kopf;
        else
            return suche (wert, kopf.next);
    }
}
```

# Mehrere Anforderungen an Klassen

```
public abstract class Vergleichbar {  
    public abstract boolean gleich (Vergleichbar zuvergleichen); ... }
```

```
public abstract class Aenderbar {  
    public abstract void aenderung (); ... }
```

Geht nicht, Java hat  
nur Einfachvererbung

```
public class Bruch extends Aenderbar, Vergleichbar, Aenderbar {  
    public boolean gleich (Vergleichbar zuvergleichen) { ... }  
    public void aenderung () { ... }  
    ... }
```

```
public class Wort extends Vergleichbar {  
    public boolean gleich (Vergleichbar zuvergleichen) { ... }
```

# Mehrere Anforderungen an Klassen

```
public interface Vergleichbar  
{  
    boolean gleich (Vergleichbar zuvergleichen);  
}
```

```
public interface Aenderbar  
{  
    void aenderung ();  
}
```

```
public class Bruch implements Vergleichbar, Aenderbar {  
    public boolean gleich (Vergleichbar zuvergleichen) { ... }  
    public void aenderung () { ... }  
}
```

```
public class Wort implements Vergleichbar {  
    public boolean gleich (Vergleichbar zuvergleichen) { ... }  
}
```

# Interfaces und abstrakte Klassen

```
public interface Vergleichbar  
    boolean gleich (Vergleichbar zuvergleichen);
```

```
public interface Aenderbar  
    void aenderung ();
```

```
public abstract class Zahl implements Vergleichbar  
    protected abstract int runde ();  
    public String rundungsinfo ()  
        return "in etwa " + runde ();
```

```
public class Bruch extends Zahl implements Aenderbar  
    public boolean gleich (Vergleichbar zuvergleichen) { ... }  
    protected int runde () { ... }  
    public void aenderung () { ... }
```

```
public class Int extends Zahl  
    public boolean gleich (Vergleichbar zuvergleichen) { ... }  
    public int runde () { ... }
```

```
public class Wort implements Vergleichbar  
    public boolean gleich (Vergleichbar zuvergleichen) { ... } ... }
```

# Liste mit Interfaces

```
public interface Vergleichbar
    boolean gleich (Vergleichbar zuvergleichen);

public abstract class Zahl implements Vergleichbar { ... }
public class Bruch extends Zahl implements Aenderbar { ... }
public class Int extends Zahl { ... }
public class Wort implements Vergleichbar { ... }
```

```
public class Element
    {
    Vergleichbar wert; Element next; ... }
}
```

```
public class Liste {
    Element kopf;

    Element suche (Vergleichbar wert) {return suche (wert, kopf);}

    static Element suche (Vergleichbar wert, Element kopf) {
        if (kopf == null) return null;
        else if (wert.gleich(kopf.wert)) return kopf;
        else return suche (wert, kopf.next);
    }
}
```

# Datenzugriff mit Interfaces

```
public interface I
    int x = 4, y = 6;
    void b (int i);
    void q (int n);
public interface J
    int x = 3;
    void b (double d);
    void q (int n);
```

```
public interface I_and_J extends I, J
    ... }
```

```
public class C implements I_and_J
    public void b (int i) { ... }
    public void b (double d) { ... }
    public void q (int n) { ... }
```

```
C z = new C ();
```

```
I i = z;
```

```
J j = i;
```

```
i.b (5); j.b (5);
```

```
i.q (5); j.q (5);
```

```
System.out.println(I.x + ", " + J.x + ", " + C.x + ", " + C.y);
```

nicht erlaubt, stattdessen J j = (C) i;

nicht erlaubt, da nicht eindeutig