Inhalt der Vorlesung

1. Klassen und Objekte
2. Vererbung
3. Enums, Wrapper und Autoboxing
4. Interfaces
5. Generics
6. Exceptions
7. Polymorphismus
8. Grafische Oberflächen
9. Streams und Lambda Expressions
10. Leichtgewichtige Prozesse – Threads

Kapitel 7: Polymorphismus

Inhalt

Kapitel 7: Polymorphismus

Lernziele

[LZ 7.1] Das Prinzip der Substituierbarkeit für Klassen und Interfaces kennen
[LZ 7.2] Polymorphismus und Dynamisches Binden erklären können
[LZ 7.3] Die durch Polymorphismus erzielbare Flexibilität erläutern können

7. Polymorphismus

7.1 Substituierbarkeit

Substituierbarkeit
Java unterstützt das sogenannte Substitutionsprinzip, welches besagt, dass anstelle einer Oberklassenreferenz auch eine Unterklassenreferenz verwendet werden kann. Gegeben sei:

```
interface Wiegbar { ... }
class Tier implements Wiegbar { ... } class Hund extends Tier { ... } class Katze extends Tier { ... }
```

damit sind folgende Anweisungen wahr bzw. falsch (s. Kommentar):

```
Tier t = new Katze();          // OK- new Katze() liefert Unterklassen-Referenz
Hund h1 = new Hund();          // OK – gleiche Typen
Hund h2 = t;                   // Nicht ok: O4berklassen-Referenz statt Unterklassen-Ref.
Katze k = new Katze();         // OK
k = h1;                        // nicht OK, unverträgliche Typen!
t = k;                         // OK, sogenannter Upcast
```

Anmerkungen:
- „h2 = t“ ist nicht korrekt, weil man sonst zur Laufzeit über h2 auf die Schnittstelle eines Hund-Objekts zugreifen könnte. Da h2 in diesem Fall auf ein Katzen-Objekt verweist, könnte man Hund-spezifische Methoden aufrufen, die das Objekt nicht unterstützt!
- t hat eine kleinere Schnittstelle als k, d.h. man verliert also durch Upcasts Funktionalität!

Analog gilt das Substitutionsprinzip auch für Interfaces: Ist ein Interface gefordert, so kann jede Objektreferenz zu Klasse eingesetzt werden, die das Interface implementiert:

```
Figur f = new Kreis(2.5);
```

Dies gilt unter folgender Annahme: Figur ist ein Interface, welches von der Klasse Kreis implementiert wird.

Dynamisches Binden

Polymorphismus bedeutet Vielgestaltigkeit. In der Informatik ist hiermit gemeint, daß unterschiedliches Verhalten mittels derselben Schnittstelle nutzbar ist. Diese Eigenschaft nutzt man, um eine leichte Änderbarkeit bzw. Erweiterbarkeit von Software durch Austauschen von Funktionalitäten ohne Anpassung der restlichen Software zu erreichen. In Java wird dies durch die Verwendung von Oberklassen-Referenzen bzw. Interfaces unter Ausnutzung des Substitutionsprinzips erreicht:

```
class GeoFigur               { void print(){ System.out.print("GeoFigur ");  }
class Kreis extends GeoFigur { void print(){ System.out.print("Kreis ");     }
class R_Eck extends GeoFigur { void print(){ System.out.print("R_Eck ");     }
class PolymorphismusTest {
  static public void main(String[] args) {
    GeoFigur[] figuren = { new Kreis(), new R_Eck(), new Kreis() };
    for (GeoFigur geoFigur : figuren)
        (*) geoFigur.print(); // Ausgabe: Kreis R_Eck Kreis
  }
}
```

Der Compiler weiß in (*) nicht, welches Objekt referenziert wird, er geht von einem GeoFigur-Objekt bzw. dessen Ableitungen aus. Zur Laufzeit entscheidet die JVM, welches Objekt und damit welche Methode genutzt wird!

Der auf der letzten Seite beschriebene Mechanismus wird dynamisches Binden genannt.
Damit ist eine Erweiterung z.B. um eine Quadrat-Klasse ohne Änderung des restlichen Codes möglich! Die for-Schleife würde weiterhin korrekt arbeiten, wenn `figuren` zusätzlich eine Referenz vom Typ `Quadrat` hätte.
Beachte: Es funktioniert auch, wenn die Klasse `GeoFigur` abstrakt ist! Man kann dann zwar keine Objekte von `GeoFigur` erzeugen, wohl aber dürfen GeoFigur-Referenzen auf konkrete Subklassen-Objekte (`Kreis`, `R_Eck`) verweisen!

Aus der Substituierbarkeit für Interfaces ergibt sich, dass Polymorphismus auch mit Interfaces funktioniert, siehe nachfolgendes Beispiel:

```
interface IGeoFigur { void print(); }
class Kreis implements IGeoFigur{ void print(){ System.out.print("Kreis "); }
class R_Eck implements IGeoFigur{ void print(){ System.out.print("R_Eck "); }
class PolymorphismusTest {
  static public void main(String[] args) {
    IGeoFigur[] figuren = { new Kreis(), new R_Eck(), new Kreis() };
    for (GeoFigur geoFigur : figuren)
        (*) geoFigur.print(); // Ausgabe: Kreis R_Eck Kreis
}}
```

Auch hier entscheidet die JVM an der Stelle (*) zur Laufzeit, auf welches konkrete Objekt `geoFigur` verweist und ruft die passende Methode auf!