Polymorphismus

- Definition
- Überdecken
- Überladen
- Beispiele

Definition: Polymorphismus

- ***Polymorphismus*** bedeutet, dass die *gleiche Anfrage an ein Objekt* (= Methodenaufruf) in *unterschiedliches Verhalten* (gleichnamige Methoden in verschiedenen Klassen oder mit unterschiedlichen Parametern) umgesetzt werden kann
- In Java: Es werden unterschiedliche Methoden(rümpfe) ausgeführt
- Beispiel 1 – Methode *berechneDeineFläche*:
  Ein Kreis berechnet seine Fläche anders als ein Rechteck oder ein Dreieck
  Wie die Berechnung aussieht, wird in der entsprechenden Methode in der jeweiligen *Klasse* einheitlich festgelegt

```
public static void main( String[] args ) {
    GeometrischeFigur[] figuren = new GeometrischeFigur[2];
    figuren[0] = new Rechteck(2, 3);
    figuren[1] = new Kreis(4);

    for (GeometrischeFigur f : figuren)
        println("Fläche: " + f.berechneDeineFläche());
}
```

Definition: Polymorphismus

- ***Polymorphismus*** bedeutet, dass die *gleiche Anfrage an ein Objekt* (= Methodenaufruf) in *unterschiedliches Verhalten* (gleichnamige Methoden in verschiedenen Klassen oder mit unterschiedlichen Parametern) umgesetzt werden kann
- In Java: Es werden unterschiedliche Methoden(rümpfe) ausgeführt
- Beispiel 1 – Methode *berechneDeineFläche*:
  Ein Kreis berechnet seine Fläche anders als ein Rechteck oder ein Dreieck
  Wie die Berechnung aussieht, wird in der entsprechenden Methode in der jeweil

```
public static void main( String[] args
    GeometrischeFigur[] figuren = new G
    figuren[0] = new Rechteck(2, 3);
    figuren[1] = new Kreis(4);

    for (GeometrischeFigur f : figuren)
        println("Fläche: " + f.berechnel
}
```

```
public class Rechteck implements GeometrischeFigur {

    public double berechneDeineFläche() {
        return länge * breite;
    }

}
```

```
public class Kreis implements GeometrischeFigur {
    public double berechneDeineFläche() {
        return radius * radius * Math.PI;
    }

}
```

Definition: Polymorphismus

- ***Polymorphismus*** bedeutet, dass die *gleiche Anfrage an ein Objekt* (= Methodenaufruf) in *unterschiedliches Verhalten* (gleichnamige Methoden in verschiedenen Klassen oder mit unterschiedlichen Parametern) umgesetzt werden kann
- In Java: Es werden unterschiedliche Methoden(rümpfe) ausgeführt
- Beispiel 2 – Methode *println* (oder auch *print*):
  Je nachdem, mit welchen *aktuellen Parametern* die Methode *println* aufgerufen wird, werden unterschiedliche *println*-Methoden ausgeführt, die Werte unterschiedlicher Typen auf die Konsole oder in eine Datei ausgegeben

```
println("Hello world!");
println(true);
println(4712);
```

Polymorphismus

- Achtung:
  Im Sinne der Verständlichkeit/Nachvollziehbarkeit von Programmen sollte das "unterschiedliche Verhalten" nach Außen immer *sinngemäß gleich* wirken!

- Beispiel:
  Die Methode *berechneDeineFläche* sollte bei Dreiecken nicht zum Drehen des Objekts führen...

Polymorphismus 1/2: überdeckte Methoden

- In Java gibt es ***überdeckte Methoden***
  - *Verschiedene Klassen* können Methoden mit *gleichem Namen* und *gleichen formalen Parametern* enthalten
    Beispiel:
    → *berechneDeineFläche* für Kreise, Rechtecke etc.
    → *toString* in den Klassen *Versicherter*, *Student*, *Rentner* etc.
    Anmerkung:
    Der Ergebnistyp der Methoden spielt keine Rolle für die Unterscheidung: Er *muss* immer gleich sein (Compiler!)
  - Im Falle überdeckter Methoden sorgt die VM mittels ***dynamischer Bindung*** zur Laufzeit dafür, dass anhand des Empfängerobjektes die "passendste" Methode ausgewählt und ausgeführt wird

Ablauf bei der Auswahl *dynamisch gebundener Methoden*

- Beispiel:
  objekt.*methode*(*parameter*$_1$, …, *parameter*$_n$)
- Bei der *Übersetzung* prüft der Compiler für jeden Methodenaufruf, ob für den statischen Typ des *objekt mindestens eine* Methode auf die angegebenen aktuellen Parameter anwendbar ist, dazu
  1. bestimmt er die (statischen) Typen des *objekt* und aller Parameter
  2. stellt er fest, ob in der Klasse des *objekt* oder in einer von deren Oberklassen (mindestens) eine Methode existiert, zu deren Parametertypen die Typen der aktuellen Parameter zuweisbar (z.B. *int* an *int* oder *int* an *long*) sind

```
for (GeometrischeFigur f : figuren)
    println("Fläche: " + f.berechneDeineFläche());
```

```
public interface GeometrischeFigur {

    double berechneDeineFläche();

}
```

  Falls nein: Compilerfehler
  Falls ja: Bytecode erzeugen *inkl. eventueller Casts* der Parameter
- Bei der *Programmausführung* bestimmt die virtuelle Maschine anhand des ***dynamischen Typs*** des *objekt* die *passendste* Methode, dazu
  1. bestimmt sie den dynamischen Typ des *objekt*
  2. wählt sie die speziellste passende Methode für die aktuellen Parameter mit dem vom Compiler bestimmten/gecasteten Typ, und führt diese aus

Ablauf bei der Auswahl *dynamisch gebundener Methoden*

- Beispiel:
  objekt.
- Bei der *Üb*
  für den sta
  angegebe
  1. bestim
  2. stellt er
     Oberkl
     Parame
     int oder

Hier steht
“dynamisch gebunden”:
das funktioniert *nicht* mit
statischen (*static*) Methoden!

```
for (GeometrischeFig
    println("Fläche:
```

```
rischeFigur {

neFläche();

}
```

  Falls nein: Compilerfehler
  Falls ja: Bytecode erzeugen *inkl. eventueller Casts* der Parameter

- Bei der *Programmausführung* bestimmt die virtuelle Maschine anhand des ***dynamischen Typs*** des *objekt* die *passendste* Methode, dazu
  1. bestimmt sie den dynamischen Typ des *objekt*
  2. wählt sie die speziellste passende Methode für die aktuellen Parameter mit dem vom Compiler bestimmten/gecasteten Typ, und führt diese aus

Polymorphismus 2/2: überladene Methoden

- In Java gibt es ***überladene Methoden***
  - *Eine Klasse* kann mehrere *gleichnamige Methoden* enthalten, sofern diese sich durch die *Anzahl und/oder die Typen ihrer formalen Parameter unterscheiden*
    Beispiel:
    → *println* in der Klasse *MakeItSimple*

```
public void println(String text) {

public void println(boolean b) {

public void println(int number) {
```

  - Anmerkung:
    Der Ergebnistyp der Methoden spielt auch hier keine Rolle für die Unterscheidung

- Nur der *Compiler* kümmert sich um die Überladung, die *VM* hat *nicht*s damit zu tun

Ablauf bei der Auswahl *überladener Methoden*

- Beispiel:
  a. *objekt*.*methode*(*parameter*$_1$, …, *parameter*$_n$) oder
  b. *Klassenname*.*methode*(*parameter*$_1$, …, *parameter*$_n$) // für statische Methoden
- Fall a) Bei der *Übersetzung* prüft der Compiler für jeden Methodenaufruf, ob für den ***statischen Typ*** des *objekt* mindestens eine Methode auf die angegebenen aktuellen Parameter anwendbar ist, dazu
  1. bestimmt er den (statischen) Typ des *objekt* (Fall a) und aller Parameter
  2. stellt er fest, ob in der Klasse des *objekt* oder in einer von deren Oberklassen (mindestens) eine Methode existiert, zu deren Parametertypen die Typen der aktuellen Parameter zuweisbar sind

  Falls nein: Compilerfehler
  Falls ja: Bytecode erzeugen *inkl. eventueller Casts* der Parameter
- Bei der *Programmausführung* bestimmt die virtuelle Maschine anhand des ***dynamischen Typs*** des *objekt* die passendste Methode, dazu
  1. bestimmt sie den dynamischen Typ des *objekt*
  2. wählt sie die speziellste passende Methode für die aktuellen Parameter mit dem vom Compiler bestimmten/gecasteten Typ, und führt diese aus

Ablauf bei der Auswahl *überladener Methoden*

- Beispiel:
  a. *objekt.methode*(*parameter*$_1$, …, *parameter*$_n$) oder
  b. *Klassenname.methode*(*parameter*$_1$, …, *parameter*$_n$) // für statische Methoden
- Fall b) Bei der *Übersetzung* prüft der Compiler für jeden Methodenaufruf, ob aus der angegebenen Klasse mindestens eine Methode auf die angegebenen aktuellen Parameter anwendbar ist, dazu
  1. verwendet er die angegebene Klasse und bestimmt den (statischen) Typ aller Parameter
  2. stellt er fest, ob in der angegebenen Klasse oder in einer von deren Oberklassen (mindestens) eine Methode existiert, zu deren Parametertypen die Typen der aktuellen Parameter zuweisbar sind

  Falls nein: Compilerfehler
  Falls ja: Bytecode erzeugen *inkl. Angabe der identifizierten Klasse* und *inkl. eventueller Casts* der Parameter
- Bei der *Programmausführung* führt die virtuelle Maschine die vom Compiler identifizierte Methode mit den vom Compiler bestimmten/gecasteten aktuellen Parametern aus

Beispiel für überladene Methoden in einer Klasse

- Die folgende Methode sortiert ein Integer-Array auf- oder absteigend, je nach Wert des zweiten Parameters

```
void sort ( int [ ] array, boolean ascending )        { ... }
```

- Die folgende Methode sortiert ein Array von Gleitkommazahlen entsprechend

```
void sort ( float [ ] array, boolean ascending )      { ... }
```

- Die folgenden Methoden rufen die vorherigen auf, indem sie dafür sorgen, dass immer aufsteigend sortiert wird (das erspart etwas Schreibarbeit)

```
void sort ( int [ ] array )       { sort( array, true ); }
void sort ( float [ ] array )     { sort( array, true ); }
```

- Diese Methoden unterscheiden sich in ihren formalen Parametern (Anzahl, Typ); sie können daher problemlos zusammen in einer Klasse definiert werden
- Beim Aufruf wird je nach Anzahl und Typ der aktuellen Parameter die passende Methode ausgewählt

Aufgabe 1/2: Auswahl der "passendsten" Methode

- Stellen Sie sich eine Klasse *Ober* mit den folgenden Methoden vor
  a. void m () { ... }
  b. void m (double f) { ... }
- Zusätzlich gebe es eine Klasse *Unter*, welche von *Ober* erbt mit den folgenden Methoden
  c. void m () { ... }
  d. void m (int i) { ... }
  e. void m (float f) { ... }
  f. void m (String s) { … }

- Welche Methoden werden in den folgenden Situationen laufen und warum?

```
Ober o = new Unter();
```

1. o.m();
2. o.m(3.0);
3. o.m(3);
4. o.m(3.0f);
5. o.m("hello");

Notieren Sie Ihre Vermutung *mit Begründung*!
Folgen Sie dem Schema
  – Der Compiler ...
  – Die VM ...

Aufgabe 2/2: Auswahl der "passendsten" Methode

- Stellen Sie sich eine Klasse *Ober* mit den folgenden Methoden vor
  a. void m () { ... }
  b. void m (double f) { ... }
- Zusätzlich gebe es eine Klasse *Unter*, welche von *Ober* erbt mit den folgenden Methoden
  c. void m () { ... }
  d. void m (int i) { ... }
  e. void m (float f) { ... }
  f. void m (String s) { … }

- Welche Methoden werden in den folgenden Situationen laufen und warum?

```
Ober o = new Ober();
Unter u = new Unter();
```

1. o.m();
2. o.m(3);
3. o.m(3.0f);
4. u.m();
5. u.m(3);
6. u.m(3.0d);
7. u.m(3.0f);

Notieren Sie Ihre Vermutung *mit Begründung*!
Folgen Sie dem Schema
  – Der Compiler ...
  – Die VM ...

Problem: Welche Methode wird ausgeführt?

- Gegeben sei eine Klasse mit einer überladenen Methode

  a. whichOne(float f, int i) { ... }
  b. whichOne(int i, float f) { ... }

- Welche Ausgaben erwarten Sie, wenn Sie die Methode *whichOne* mit folgenden Parametern aufrufen:

  1. whichOne(3, 4f);
  2. whichOne(3f, 4);
  3. whichOne(3f, 4f);
  4. whichOne(3, 4);