# Objektorientierte Programmierung

# Kapitel 14: Überladene Methoden

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/oop18/

# Inhalt

# Einführung, Motivation (1)

- Es kann mehrere Methoden (Funktionen, Prozeduren) mit gleichem Namen geben, die sich in der Parameterliste (Anzahl oder Typen der Argumente) unterscheiden.

  Der Spezialfall, dass in unterschiedlichen Klassen Funktionen (Methoden) gleichen Namens definiert sind, wurde oben schon betrachtet: Durch den Datentyp (Klasse) des Objektes, auf das die Methode angewendet wird, kann der Compiler entscheiden, welche Funktion er aufrufen soll. Dieses Konzept funktioniert auch allgemeiner.

- Diese Methoden nichts mit einander zu tun:
  Der Programmierer muss für jede Methode eine eigene Implementierung (Methodenrumpf) schreiben.

  Man kann es sich so vorstellen, dass der interne Name einer Methode, den der Compiler verwendet, auch die Typen der Argumente enthält.

# Einführung, Motivation (2)

- Zweck dieses Sprachelementes ist es, semantisch ähnlichen Aktionen auf unterschiedlichen Datentypen den gleichen Namen geben zu können.

  Das ist aber nur Konvention/guter Stil. Theoretisch könnten die Methoden ganz unterschiedliche Dinge tun.

- Überladen ist bereits bekannt von Bibliotheksfunktionen wie `println`: Es gibt viele Methoden mit diesem Namen, z.B. für die Typen `int`, `double`, `char`, `String`, `boolean`.

  Es gibt insgesamt 9 Methoden, die `print` heißen.
  [http://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html]

- Sonst musste man Namen wie `print_int`, `print_str` etc. verwenden, oder die Typsicherheit aufgeben.

  Z.B. `printf` in C: Compiler kann Typkorrektheit nicht (immer) prüfen.

# Einführung, Motivation (3)

- Überladene Operatoren gibt es in Programmiersprachen schon länger: Die Addition + ist für ganze Zahlen und für Gleitkomma-Zahlen durch völlig unterschiedliche Maschinenbefehle implementiert.

  Früher hatten viele CPUs (z.B. 8086) keine Gleitkomma-Operationen eingebaut. Diese Operationen wurden dann durch Aufruf einer Bibliotheksfunktion ausgeführt. Zur Effizienzsteigerung konnte man einen mathematischen Coprozessor (z.B. 8087) nachrüsten. Bei der Division ist noch offensichtlicher, dass sie für ganze Zahlen und Gleitkommazahlen unterschiedlich funktioniert.

- Auch die in Java eingebauten Operatoren sind überladen, z.B. + für Strings, für `int`-Werte, etc.

- In Sprachen wie Java geht das auch für eigene Methoden.

  In C++ kann man sogar die Operatoren wie + für eigene Datentypen überladen, das geht in Java nicht.

# Einführung, Motivation (4)

- "Most often, it is a good idea to give different functions different names, but when some functions conceptually perform the same task on objects of different types, it can be more convenient to give them the same name."
- "Using the same name for operations on different types is called overloading."

  Beide Zitate aus: [Stroustrup, 2000, Seite 149].

- Eine wichtige Anwendung von überladenen Methoden in Java sind auch Argumente, die fast immer einen bestimmten Wert haben: Man kann eine Version der Methode definieren ohne dieses Argument, die den normalen Wert automatisch einsetzt.

  In C++ kann man Default-Werte für Argumente festlegen. Java hat das nicht, aber mit Überladen erzielt man den gleichen Effekt.

# Syntax: Mehrere Methodendeklarationen

- Man kann in Java also mehrere Methoden/Funktionen innerhalb einer Klasse definieren, die sich nur in Anzahl oder Typen der Parameter unterscheiden.

  Der Compiler kennt beim Aufruf den exakten (statischen) Typ des Argumentes, kann also eine der Versionen auswählen.

- Es dürfen nur nicht exakt die gleichen Parameter-Typen sein, sonst bekommt man eine Fehlermeldung wie "`f(int)` is already defined in TestClass".

  Man kann auch nicht eine statische Methode und eine normale Methode mit gleichem Namen und gleichen Argument-Typen in einer Klasse definieren.

- Es reicht nicht, dass sich die beiden Methoden im Rückgabe-Typ (Ergebnis-Typ) unterscheiden.

  Der Rückgabe-Typ spielt bei der Auswahl keine Rolle. Der Aufrufer muss den Rückgabe-Wert nicht unbedingt verwenden, diese Information könnte also fehlen.

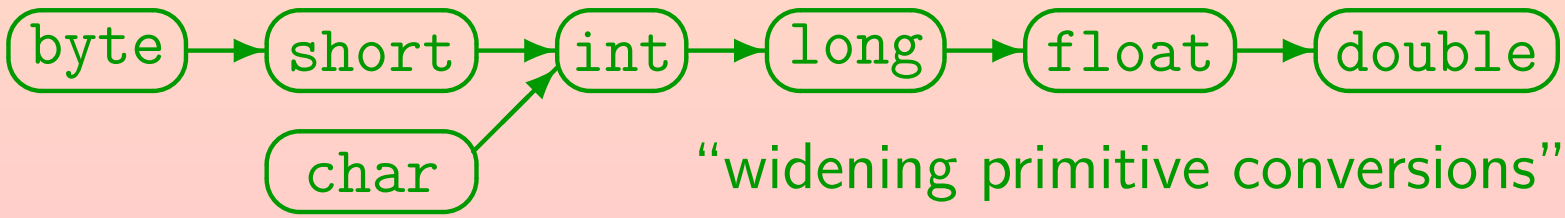
# Inhalt

# Beispiel: Speziellster Typ (1)

```
 (1)  class TestOverload1 {
 (2)
 (3)      static void output(int i) {
 (4)          System.out.println("int: " + i);
 (5)      }
 (6)
 (7)      static void output(double x) {
 (8)          System.out.println("double: " + x);
 (9)      }
(10)
(11)      public static void main(String[] args) {
(12)          output(100); // int: 100
(13)          output(1.0); // double: 1.0
(14)      }
(15)  }
```

# Beispiel: Speziellster Typ (2)

- Im obigen Beispiel gibt es zwei (statische) Methoden mit Namen "`output`",
  - eine für `int`-Werte,
  - eine für `double`-Werte.
- Wenn der Typ des Arguments ein `int` bzw. ein `double`-Wert ist, wird die jeweils passende Methode aufgerufen.
- Interessanter wird die Situation, wenn man die Methode mit einem Argument aufruft, das keinen von beiden Typen hat, aber darin umgewandelt werden kann:



# Beispiel: Speziellster Typ (3)

- Bei der Bindung eines aktuellen Parameters (Argument-Wert) an einen formalen Parameter (Variable) werden die gleichen Typ-Anpassungen durchgeführt wie bei Zuweisungen.

  Ausnahme: Kleine Konstanten (z.B. `1`) werden nicht automatisch in kleine ganzzahlige Typen (z.B. `byte`) umgewandelt. Für primitive Typen werden also nur die im Diagramm gezeigten Umwandlungen in größere Typen durchgeführt.

- Wenn man die Methode `output` mit einem `char` aufruft,

  ```
  output('a');
  ```

  erhält man die Ausgabe "`int: 97`".

  Das Zeichen "a" hat in ASCII und Unicode den Code 97. Es wurde vom Typ `char` automatisch in der Typ `int` umgewandelt.

- Das ist interessant, weil auch eine Umwandlung nach `double` (`97.0`) möglich gewesen wäre.

# Beispiel: Speziellster Typ (4)

- Etwas vereinfacht wird unter den aufrufbaren Kandidaten der Spezifischste gewählt, wobei eine Methode $m_1$ spezifischer als $m_2$ ist, wenn jeder Aufruf von $m_1$ auch für $m_2$ gültig wäre.
- So ist z.B. `output(int)` spezifischer als `output(double)`: Jeder Wert, der sich in ein `int` umwandeln läßt, kann auch weiter in `double` umgewandelt werden.
- Bei einem Aufruf mit `long`, `float` oder `double`-Werten ist `output(int)` aber kein Kandidat, daher wird in diesen Fällen `output(double)` gewählt.
- Die Regel gilt auch für Subklassen: Ist `Unter` eine Subklasse von `Ober`, so ist `m(Unter)` spezifischer als `m(Ober)`.

# Beispiel: Mehrdeutigkeit (1)

```
(1)  class TestOverload2 {
(2)
(3)      static void output(int i, double x) {
(4)          System.out.println("V1: "+i+" "+x);
(5)      }
(6)
(7)      static void output(double x, int i) {
(8)          System.out.println("V2: "+i+" "+x);
(9)      }
(10)
(11)     public static void main(String[] args) {
(12)         output(1, 2); // ? (Fehler)
(13)     }
(14) }
```

# Beispiel: Mehrdeutigkeit (2)

- Im Beispiel könnten beide Version von `output` angewendet werden:
  - Bei der ersten Version passt das erste Argument genau, das zweite muss von `int` nach `double` umgewandelt werden.
  - Bei der zweiten Version ist es umgekehrt.
- Es gibt daher keine eindeutige Entscheidung, und man erhält folgende Fehlermeldung:

```
TestOverload2.java:12: reference to output is ambiguous,
    both method output(int,double) in TestOverload2
    and method output(double,int) in TestOverload2 match
        output(1, 2);
        ^
```

- Lösung: Explizite Typumwandlung (Cast) im Aufruf.

# Beispiel: Optionales Argument (1)

```
(1)  class TestOverload3 {
(2)
(3)      static void output(int i, boolean hex) {
(4)          if(hex)
(5)              System.out.printf("%x\n", i);
(6)          else
(7)              System.out.printf("%d\n", i);
(8)      }
(9)
(10)     static void output(int i) {
(11)         output(i, false);
(12)     }
(13)
(14)     public static void main(String[] args) {
(15)         output(32); // dezimal
(16)     }
(17) }
```

# Beispiel: Optionales Argument (2)

- Im Beispiel hat die Methode `output` ein zweites Argument, mit dem man festlegen kann, ob die Zahl in Dezimal- oder Hexadezimal-Notation (Basis 16) ausgegeben werden soll.
- Fast immer will man aber die normale Dezimal-Notation.
- Deswegen wurde eine zweite Version von `output` definiert mit nur einem Argument.
- Dies dient nur der Bequemlichkeit, intern wird die andere Version aufgerufen, und der Default-Wert "`false`" für das zweite Argument übergeben.
- Diese Technik kann auch verwendet werden, wenn ein zusätzliches Argument zu einer Methode nachgerüstet werden soll, aber die Methode bereits verwendet wird.

# Inhalt

# Statischer vs. dynamischer Typ (1)

```
(1)  class Ober { }
(2)  class Unter extends Ober { }
(3)
(4)  class TestOverload3 {
(5)      static void output(Ober o) {
(6)          System.out.println("Ober!");
(7)      }
(8)
(9)      static void output(Unter u) {
(10)         System.out.println("Unter!");
(11)     }
(12)
(13)     public static void main(String[] args) {
(14)         Ober x = new Unter();
(15)         output(x); // Ober!
(16)     }
(17) }
```

# Statischer vs. dynamischer Typ (2)

- Das obige Programm druckt "`Ober!`" aus.
- Der Compiler wählt eine Version der überladenen Methode basierend auf dem statischen Typ des Argumentes.
- Im Beispiel wird ja tatsächlich
  - ein "`Unter`"-Objekt übergeben (dynamischer Typ),
  - aber die Referenz steht in einer Variablen vom Typ "`Ober`" (statischer Typ).
- Dies ist ein wesentlicher Unterschied zum Überschreiben:
  - Dort fällt die Entscheidung erst zur Laufzeit, und es entscheidet der dynamische Typ.

    In dieser Hinsicht wird das implizite 0-te Argument (`this`-Objekt) anders behandelt als die anderen Argumente.

# Statischer vs. dynamischer Typ (3)

```
(1)  class Ober {
(2)      void output() {
(3)          System.out.println("Ober!");
(4)      }
(5)  }
(6)  class Unter extends Ober {
(7)      void output() {
(8)          System.out.println("Unter!");
(9)      }
(10) }
(11)
(12) class TestOverride { // Ueberschreiben!
(13)     public static void main(String[] args) {
(14)         Ober x = new Unter();
(15)         x.output(); // Unter!
(16)     }
(17) }
```

# Überschreiben vs. Überladen (1)

```
(1)  class Ober {
(2)      void output(int i) {
(3)          System.out.println("Ober: " + i);
(4)      }
(5)  }
(6)  class Unter extends Ober {
(7)      void output(double x) {
(8)          System.out.println("Unter: " + x);
(9)      }
(10) }
(11)
(12) class TestOverload4 {
(13)     public static void main(String[] args) {
(14)         Unter u = new Unter();
(15)         u.output(1); // Ober: 1
(16)     }
(17) }
```

# Überschreiben vs. Überladen (2)

- Im Beispiel wird nicht überschrieben, sondern überladen.

  Die Methode in der Unterklasse hat zwar den gleichen Namen wie die in der Oberklasse, aber einen anderen Argument-Typ.

- Hätte man die Annotation `@Override` verwendet, so hätte man folgende Fehlermeldung bekommen:

```
TestOverload4.java:8: method does not override or
    implement a method from a supertype
    @Override
    ^
```

- Für Objekte der Unterklasse sind also beide Varianten aufrufbar.

- Es wird dann die mit dem spezifischeren Typ gewählt, das ist die Version aus der Oberklasse.