Objektorientierte Programmierung

Methoden und Konstruktoren

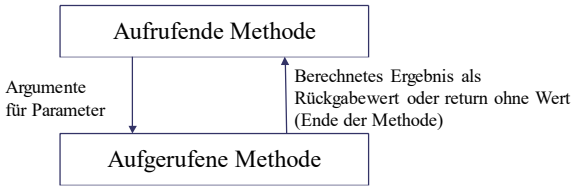
Prof. Dr. Ulrike Hammerschall
Fakultät für Informatik und Mathematik

Methoden Zusammenfassung

- Methodendefinitionen sind nur innerhalb von Klassen zulässig, nicht außerhalb, nicht innerhalb anderer Methoden!
- Es sind beliebig viele Methodendefinitionen in einer Klasse möglich.
- Die Reihenfolge der Methoden innerhalb der Klasse ist ohne Bedeutung. Konvention: erst die Konstruktoren, dann die Methoden.
- Aufruf einer Methode in der gleichen Klasse: die Methode kann unmittelbar aufgerufen werden.
- Aufruf einer Methode aus einer anderen Klasse: die Methode wird auf dem Objekt der Klasse mit Punktoperator aufgerufen.

Aufruf von Methoden - die Idee

- Übergabe von Werten (Argumenten) an Parameter transportiert Information vom Aufrufer zur Methode.
- Ergebnisrückgabe liefert Information von der Methode zurück zum Aufrufer (z.B. Ergebnisse von Berechnungen).



- Eine Methode kann beliebig viele Parameter definieren und entsprechende Werte annehmen, aber maximal einen Ergebniswert liefern.

Überladen von Methoden (Overloading)

- Methoden können überladen werden, d.h.:
  - es stehen mehrere Varianten parallel zur Verfügung.
  - Die Wahl der zu verwendenden Variante erfolgt abhängig vom Kontext.
- Umsetzung:
  - Es gibt mehrere Methoden mit gleichem Namen.
  - Die Methoden haben unterschiedliche Parameterlisten (bzgl. Anzahl und/oder Datentypen).
  - Der Namen der Parameter ist dabei ohne Bedeutung.
  - Der Ergebnistyp ist ebenfalls ohne Bedeutung.

Beispiel Klasse Rational

```
class Rational {
        // Zähler der rationalen Zahl
        private int num;
        // Nenner der rationalen Zahl
        private int denom;
        ...
        // Setzen der rationalen Zahl mit default-Werten
        void setRational() {
                num = 0;
                denom = 1;
        }
```


Überladen der Methode setRational()

```
        …
        // Setzen der Zahl mit default-Wert für den Nenner
        void setRational(int n) {
                num = n;
                denom = 1;
        }
```

```
        // Setzen neuer Werte für Zähler und Nenner
        void setRational(int n, int d) {
                num = n;
                denom = d;
        }
}
```

Verwendung überladener Methoden

- Der Aufrufer wählt die von ihm gewünschte Methode durch Angabe der entsprechenden Argumentliste aus.

```
// Initialisierung der rationalen Zahl 4/5.
Rational rational = new Rational(4, 5);
// Aufruf von setRational(int, int)
rational.setRational(2, 3);
// Aufruf von setRational()
rational.setRational();
// Aufruf von setRational(int)
rational.setRational(4);
// Fehler! Keine passende Methode!
rational.setRational(1, 2, 3);
```


Die Overload-Resolution

- Als Overload-Resolution wird die Auswahl einer passenden überladenen Methode zu einer gegebenen Argumentliste bezeichnet.
- Der Compiler wählt die Methodendefinition aus, die am besten auf den Methodenaufruf passt.
- Er prüft alle in Frage kommenden Methodenkandidaten, einschließlich der Anwendung impliziter Typkonversionen.
- Unter den Kandidaten wählt er die am genauesten passende Methode aus (die Datentypen passen am exaktesten und es sind möglichst wenig implizite Typkonversionen notwendig).
- Allgemein formuliert: Eine Methode a passt genauer als eine Methode b, wenn jeder Aufruf von a auch von b akzeptiert werden würde, aber nicht umgekehrt.

Beispiel: Auswahl der überladenen Methode durch Compiler

- Beispiel: Klasse Point für Punkte in der kartesischen Koordinatenebene mit zwei überladenen setPoint-Methoden:

```
class Point {
      public void setPoint(int x, int y){..}          (A)
      public void setPoint(double x, double y){..}    (B)
      ...
}
```

- Welche Methode wird vom Compiler gewählt?

```
setPoint(1.0, 1.0)          =>
setPoint(1, 1.0)            =>
setPoint(1.0, 1)            =>
setPoint(1, 1)              =>
```


Ergebnistypen überladener Methoden

- Überladen mit unterschiedlichen Ergebnistypen bei gleicher Parameterliste ist nicht zulässig.
- Ergebnistypen überladener Methoden dürfen abweichen, aber nur, wenn auch die Parameterliste abweicht.
- Hintergrund:
  - das Ergebnis muss nicht in jedem Fall eines Methodenaufrufs verwertet werden.
  - der Compiler kann daher einem Aufruf nicht immer ansehen, welche Methode gemeint ist.

Ergebnistypen überladener Methoden - Beispiel

- Die Methoden:

```
void setRational(int n, int d) {...}
boolean setRational(int n, int d) {...}
```

- unterscheiden sich nur hinsichtlich ihrer Ergebnistypen.
- Dem (korrekten) Aufruf:

```
setRational(2,3);          => ?
```

- lässt sich nicht ansehen, welche der beiden Methoden gemeint ist.
- Ein solcher Methodenaufruf wird daher vom Compiler als Fehler zurückgewiesen.

Übung zum Thema Überladen

1. Wird hier korrekt überladen? Falls nein, warum nicht?

```
int doSomething(int a, double b, String c) { … }
void doSomething(int a, double b, String c) { … }
```

2. Wird hier korrekt überladen? Falls nein, warum nicht?

```
int doSomething(int a, double b, String c) { … }
int doSomething (int a, int b, String c) { … }
```

3. Wird hier korrekt überladen? Falls nein, warum nicht?

```
int doSomething (int a, double b, String c) { … }
int doSomething (String c, int a, double b) { … }
```

4. Wird hier korrekt überladen? Falls nein, warum nicht?

```
void doSomething (int a, int b) { … }
void doSomething (int x, int y) { … }
```

Konstruktoren Zusammenfassung

- Spezielle Methode zur Initialisierung von Objekten.
- Signatur von Konstruktoren:
  - Name des Konstruktors entspricht Namen der Klasse
  - Kein Ergebnistyp
  - Beliebige Parameterliste
- Aufruf mit **new** Operator. Liefert als Ergebnis ein Objekt zur Klasse.
- Für Konstruktoren gelten die gleichen Regeln zum Überladen wie für Methoden:
  - Pro Klasse kann es nur einen Default-Konstruktor(kein Parameter) geben
  - Pro Klasse kann es beliebig viele Custom-Konstruktoren geben. Die Overload-Resolution muss erfüllt sein.
- In Klassen ohne Konstruktor wird automatisch ein Default-Konstruktor generiert.

Copy-Konstruktor

- Prinzipiell ein normaler Konstruktor, der den entsprechenden Regeln folgt.
- Idee: Der Copy-Konstruktor erzeugt eine Kopie eines bereits existierenden Objekts.
- Die Vorlage (das Originalobjekt) wird beim Aufruf als Wert übergeben.

```
public class Rational {
	private final int num;
	private final int denom;

	// Copy-Konstruktor
	public Rational(Rational original) {
		this.num = original.getNum();
		this.denom = original.getDenom();
	}
	…
}
```

Der Copy-Konstruktor

- Ziel: Aufruf mit anderem Objekt als „Kopiervorlage“:

```
Rational original = new Rational(2,3);
Rational copy = new Rational(original);
System.out.println(copy.toString());
```

- Merkmal eines Copy-Konstruktors: Der Typ des Parameters ist von der gleichen Klasse, wie die Klasse in der der Konstruktor definiert wurde.
- Copy-Konstruktoren sind ausreichend für das Kopieren einfacher Objekte, jedoch nicht für das Kopieren komplexer Objektgeflechte.

Konstruktoren und Objektvariablen

- Die Initialisierung von Objektvariablen findet zeitlich VOR dem Konstruktor-Aufruf statt:
  - wenn der Konstruktor aufgerufen wird, sind die Objektvariablen bereits initialisiert.
  - im Konstruktor können die Werte der initialisierten Objektvariablen verwendet und ggf. wieder überschrieben werden.
  - Methoden können die Werte auch später überschreiben.
- Auswirkung der Reihenfolge auf finale Objektvariablen:
  - Schlüsselwort final:
    - Objektvariable erhält Wert direkt oder spätestens im Konstruktor
    - Danach darf der Wert nicht mehr verändert werden.
  - Vorinitialisierte Objektvariablen können nicht im Konstruktor erneut einen Wert erhalten.

Verkettung von Konstruktoren (Constructor-Chaining)

- Konstruktoren sind manchmal aufwendig (Tests und Vorverarbeitung von Parametern, Protokollausgaben, etc).
- Oft braucht es Kopien des gleichen Codes in jedem Konstruktor.
- Besser: Code nur in einem Konstruktor, von allen anderen mitbenutzen.
- Verkettung von Konstruktoren (engl. constructor chaining):
  - Aufruf eines anderen Konstruktors der gleichen Klassen mit *this*
  - Übergabe von Werten an den Konstruktor

Beispiel Constructor-Chaining

```
public Rational {

    private final int num;
    private final int denom;

    /**
     * Default-Konstruktor für den Bruch 1/2
     */
    public Rational() {
        this(1, 2);
    }
    /*
     * Custom-Konstruktor mit zwei Parametern
     */
    public Rational(int num, int denom) {
        this.num = num;
        this.denom = denom
    }
    …
}
```

- *this(…)* dient hier wieder als Repräsentant des Objekts selbst.
- Über *this(…)* wird der Konstruktor aufgerufen, der die entsprechende Parameterliste hat.
- Einschränkungen bei verketteten Konstruktoraufrufe:
  - *this(...)* muss erste Anweisung im Konstruktorrumpf sein.
  - es ist nur ein Aufruf von *this(...)* pro Konstruktor erlaubt.

Zusammenfassung Konstruktoren

- Konstruktoren sind Methoden, die den Zweck haben Objekte zu erzeugen. Ergebnis eines Konstruktor-Aufrufs ist damit immer ein Objekt der entsprechenden Klasse.
- Es gibt: Default-Konstruktor, Custom-Konstruktoren, automatisch generierte Konstruktoren -> Jede Klasse hat damit immer mindestens einen Konstruktor zur Verfügung.
- Der Copy-Konstruktor ist ein Konstruktor, der als Ergebnis eine Kopie des Objekts liefert, das beim Aufruf als Argument übergeben wird.
- Constructor-Chaining erlaubt den Aufruf der Konstruktoren untereinander zur Reduzierung von redundantem Code.