Softwareentwicklung II (IB)

Methoden

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 28.01.2020 17:34

Inhaltsverzeichnis

Was Sie bereits wissen müssen: Siehe Softwareentwicklung I (IB)

Überladen

Definition überladener Methoden

- **Überladen** (engl. overloading) = mehrere Methoden mit gleichen Namen aber unterschiedlichen Parameterlisten
- Sinnvoll für verwandte Methoden mit ähnlichem Zweck
- drei Methoden `set` zum Festsetzen eines Bruchs:

```
class Rational {
  void set() {
    numer = 0;
    denom = 1;
  }
  void set(int n) {
    numer = n;
    denom = 1;
  }
  void set(int n, int d) {
    numer = n;
    denom = d;
  }
  ...
}
```

- Überladen mit unterschiedlicher Parameteranzahl oder unterschiedlichen Parametertypen oder beidem
- Namen der Parameter ohne Bedeutung

Aufruf überladener Methoden

- Argumentliste des Aufrufers wählt überladene Methode aus

```
r.set();          // Aufruf von set()
r.set(1);         // Aufruf von set(int)
r.set(1, 2);      // Aufruf von set(int, int)
r.set(1, 2, 3);   // Fehler! Keine passende Methode
                  // definiert
```

- Überladene Methoden führen zu Polymorphismus, entsprechend zu arithmetischen Operatoren

Overload Resolution

- **overload resolution** = Auswahl einer passenden überladenen Methode zu einer gegebenen Argumentliste — manchmal nicht ganz einfach!
- Zuerst: Alle in Frage kommenden Kandidaten sammeln, einschließlich der Anwendung impliziter Typkonversionen
- Unter den Kandidaten diejenige Methode aufrufen, die am genauesten passt
- Was bedeutet: „passt am genauesten?“
  Eine Methode a „passt genauer“ als eine Methode b, wenn jeder Aufruf von a auch von b akzeptiert werden würde, aber nicht umgekehrt.

Beispiel für Overload-Resolution

- Klasse `Point` für Punkte in der kartesischen Koordinatenebene mit zwei überladene `set`-Methoden:

```
class Point {
  void set(int x, int y)          {...}
  void set(double x, double y)    {...}
  ...
}
```

Konstruktoren

Ziel

- Ziel: Automatisch sinnvoller Startzustand für neu geschaffene Objekte
- Beispiel `Rational`: $\frac{0}{1}$ sinnvoll, aber nicht $\frac{1}{0}$ oder $\frac{0}{0}$.
- Idee: Aufruf einer normalen Methode, zum Beispiel `set` für `Rational`-Objekte:

  ```
  Rational r = new Rational();
  r.set(0, 1);
  ```

  Technisch in Ordnung, aber unzuverlässig
- Besser: **Konstruktoren** (engl. constructors = ctors): Spezielle Methoden, werden automatisch bei jedem `new` aufgerufen

Definition

- Definition eines Konstruktors wie normale Methode, außer …
  - derselbe Name wie die Klasse
  - kein Vorsatz von `void` (oder anderem Ergebnistyp)
- Abgesehen davon: Kopf, Parameterliste, Rumpf wie andere Methoden
- Konstruktor mit leerer Parameterliste = **Default-Konstruktor** (engl. def-ctor)

Beispiele

- Default-Konstruktor von `Rational`:

  ```
  class Rational {
    Rational() {
      numer = 0;
      denom = 1;
    }
    ...
  }
  ```

- `new` ruft automatisch Konstruktor auf:

  ```
  // Aufruf von Rational()
  Rational r = new Rational();
  r.print(); // gibt 0/1 aus
  ```

Konstruktoren mit Parametern

- **Custom-Konstruktor** = Konstruktor mit nicht-leerer Parameterliste
- Beispiel `Rational`:

```
class Rational {
  Rational(int n, int d) {
    numer = n;
    denom = d;
  }
  ...
}
```

- Im `new`-Aufruf passende Argumente angeben:

```
// Aufruf von Rational(int, int)
Rational r = new Rational(2, 3);
r.print();       // gibt 2/3 aus
```

- Syntaktisch erkennbar: Nach `new` steht ein Konstruktoraufruf

Defaultwerte von Objektvariablen

- Lokale Variablen starten nicht initialisiert (ohne Wert)
- Gegensatz: Objektvariablen automatisch mit **Defaultwerten** initialisiert
- Defaultwerte abhängig vom Typ:

Typ	**Defaultwert**
`int`	`0`
`double`	`0.0`
`boolean`	`false`
`char`	`\u0000`
Referenztypen	`null`

- Beispiel: Ohne Konstruktor starten `Rational`-Objekte mit $\frac{0}{0}$ (unbrauchbar wegen 0 im Nenner)

Explizite Initialisierung von Objektvariablen

- Objektvariablen können bei der Definition explizit initialisiert werden

```
class Rational {
  int numer = 0;
```

```
  int denom = 1;
}
```

- In einfachen Fällen (wie `Rational`) Ersatz für Konstruktor
- Explizite Initialisierung überschreibt Defaultwerte
- Explizite Initialisierung zeitlich vor Konstruktoraufrufen
  ⇒ in Konstruktoren sind Objektvariablen bereits initialisiert

Automatisch definierter Konstruktor

- Klasse ohne explizit definierten Konstruktor: Compiler erzeugt automatisch Default-Konstruktor
- Rumpf eines automatisch erzeugten Default-Konstruktors leer
- Beliebige explizite Definition unterbindet automatische Definition
- Früheres Beispiel: Klasse hat nur Custom-Konstruktor, keinen Default-Konstruktor ⇒ Aufruf des (nicht definierten) Default-Konstruktors scheitert:

```
new Rational(2, 3);
new Rational();     // Fehler
```

- Fazit: Jede Klasse hat immer wenigstens einen Konstruktor

Kopier-Konstruktor

- **Kopier-Konstruktor** (engl. copy-ctor) erzeugt eine Kopie eines bereits existierenden Objekts
- Vorlage (= Original-Objekt) wird als Parameter `that` übergeben

```
class Rational {
  Rational(Rational that) {
    numer = that.numer;
    denom = that.denom;
  }
  ...
}
```

- Aufruf mit anderem Objekt als „Kopiervorlage“:

```
Rational original = new Rational(2, 3);
Rational copy = new Rational(original);
copy.print(); // gibt 2/3 aus
```

- Merkmal eines Kopier-Konstruktors: Parameter der gleichen Klasse

- Kopier-Konstruktor ausreichend für einfache Klassen, zu wenig für komplexere Klassen

Verkettete Konstruktoren

- Konstruktoren manchmal aufwändig (Tests und Vorverarbeitung von Parametern, Protokollausgaben, …)
- Falls mehrere überladene Konstruktoren: Kopien des gleichen Codes in jedem Konstruktor
- Besser: Code nur in einem Konstruktor, von allen anderen mitbenutzen
- Konstruktor-Verkettung (engl. constructor chaining) = Aufruf eines anderen Konstruktors der gleichen Klassen
- Syntax: `this` als Repräsentant des „eigenen Objektes“

Beispiel: `Rational` mit verketteten Konstruktoren

```
class Rational {
  Rational(int n, int d) {
    /* 1.) Überprüfen ob d != 0
     * 2.) n und d kürzen
     * 3.) Falls d < 0: Vorzeichen von
     * n und d umdrehen
     */
    numer = n;
    denom = d;
  }
  Rational() {
    this(0); // Aufruf von Rational(int)
  }

  Rational(int n) {
    this(n, 1); // Aufruf von Rational(int, int)
  }
  Rational(Rational that) {
    // Aufruf von Rational(int, int)
    this(that.numer, that.denom);
  }
  ...
}
```

- Einschränkungen verketteter Konstruktoraufrufe

  - `this(…)` muss erste Anweisung im Konstruktorrumpf sein
  - nur ein Aufruf von `this(…)` erlaubt

Initializer

- **Initializer** = namenloser Block neben Objektvariablen, Konstruktoren und anderen Methoden
- Syntax

```
class Classname {
    {
        ...
    }
}
```

- Inhalt des Initializers wird an den Anfang jedes Konstruktors kopiert
- Ausnahme: verkettete Konstruktoren
- Laufzeit: Initializer wird einmal vor dem (ersten) Konstruktor ausgeführt

Beispiel: `Rational` mit Initializer

```
class Rational {
  // Initializer
  {
    System.out.println("initializer");
  }

  Rational() {
    // hier Kopie des Initializers
    numer = 0;
    denom = 1;
    System.out.println("Rational()");
  }

  Rational(int n) {
    // hier keine Kopie des Initializers
    this(n, 1);
    System.out.printf("Rational(%d)%n", n);
  }

  Rational(int n, int d) {
    // hier Kopie des Initializers
```

```
    numer = n;
    denom = d;
    System.out
        .printf("Rational(%d, %d)%n", n, d);
  }
}
```

Ausgaben

- `new Rational()`:

  ```
  initializer
  Rational()
  ```

- `new Rational(1)`:

  ```
  initializer
  Rational(1, 1)
  Rational(1)
  ```

- `new Rational(2, 1)`:

  ```
  initializer
  Rational(2, 1)
  ```

- Nützlich im Zusammenhang mit Anonymen Klassen

Ergebnisrückgabe

Idee

- Parameterübergabe transportiert Information vom Aufrufer zur Methode
- **Ergebnisrückgabe** liefert Information von der Methode zurück zum Aufrufer

	Aufrufer	
Parameterübergabe	↓ ↑	Ergebnisrückgabe
	Methode	

- Eine Methode kann beliebig viele Parameterwerte annehmen, aber nur einen Ergebniswert liefern

Definition

- Zwei gekoppelte Maßnahmen zur Ergebnisrückgabe:
  1. Typ des Ergebniswertes im Methodenkopf
  2. `return`-Anweisung im Methodenrumpf
- Schema:

```
type methodname(...) {
  ...
      return expression;
}
```

- Typ von expression in der `return`-Anweisung kompatibel zu type im Methodenkopf
- Ausdruck „Methodentyp“ = Typ des Ergebniswertes der Methode

Prozedur und Funktion

- Auch kurz: „Typ-Methode“ = Methode die ein Ergebnis des Typs liefert
- liefert eine Methode kein Ergebnis (`void`), so heißt sie eine **Prozedur**
- liefert eine Methode ein Ergebnis, so heißt sie eine **Funktion**
- umgekehrt:
  - eine Prozedur oder eine Funktion die Member einer Klasse ist, heißt **Methode**

Beispiel

```
class Rational {
  ...
  int getNumer() {
    return numer;
  }
  int getDenom() {
    return denom;
  }
  double getReal() {
    // Typecast nötig, sonst int-Division
    return ((double) numer) / denom;
  }
}
```

Mehrere `return`-Anweisungen

- Mehrere `return`-Anweisungen im Rumpf erlaubt
- Methode kehrt zurück, sobald zur Laufzeit das erste `return` erreicht wird
- Statische Reihenfolge der `return`-Anweisungen unerheblich, konkreter Ablauf zur Laufzeit entscheidet
- Beispiel: `signum` gibt `+1`, `0` oder `-1` zurück, wenn der Bruch positiv, null oder negativ ist:

```
int signum() {
  if (numer * denom > 0) {
    return 1;
  } else {
    if (numer == 0) {
      return 0;
    } else {
      return -1;
    }
  }
}
```

- Rückkehr im `then`-Zweig ⇒ `else` in solchen Fällen unnötig:

```
...
int signum() {
  if (numer*denom > 0) {
    return 1;
  }
  if (numer == 0) {
    return 0;
  }
  return -1;
}
```

Falsches oder fehlendes `return`

- Compiler prüft: Typen aller `return`-Anweisungen kompatibel zum Ergebnistyp?
- Beispiel: Korrekt wegen Kompatibilität `int`→`double` (wenn auch nicht sehr sinnvoll):

```
class Rational {
  ...
  double getReal() {
```

```
    return numer / denom; // int-Division
  }
}
```

- Compiler prüft: Methode muss in jedem Fall ein `return` erreichen, Rückkehr ohne Wert unzulässig

Beispiel

- Fehlerhaft weil ohne Rückgabewert für negative Brüche:

```
class Rational {
  ...
  int signum() {
    if (numer * denom > 0) {
      return 1;
    }
    if (numer == 0) {
      return 0;
    }
    // kein return!
  }
}
```

Grenzen des Compilers

- Compiler arbeitet defensiv: Kann Code-Logik nicht immer durchschauen
- Beispiel: Wird nicht übersetzt, obwohl immer ein Ergebnis:

```
boolean isEven(int n) {
  if(n%2 == 0) {
    return true;
  }
  if(n%2 == 1) {
    return false;
  }
}
```

- wenn der Compiler so etwas überprüfen würde, wäre der Compile-Vorgang u.U. sehr langsam

Ergebnislose Methoden

- Rückkehr ohne Ergebnis: Angabe des Pseudo-Typ `void` (= überhaupt kein Wert, siehe frühere Beispiele)
- Automatische Rückkehr am Ende des Methodenrumpfes
- Rückkehr mitten im Rumpf mit `return`-Anweisung ohne Ausdruck

Beispiel

- Methode `reduce` zum Kürzen, kehrt sofort zurück falls ggT = 1:

```
class Rational {
  ...
  void reduce() {
    int gcd = ...;
    if (gcd == 1)
      // bereits gekürzt, nichts zu tun
      return;
    numer /= gcd;
    denom /= gcd;
    // automatische Rückkehr
  }
}
```

Ergebnistyp überladener Methoden

- Ergebnistypen überladener Methoden dürfen abweichen
- Beispiel:

```
int max(int a, int b) {...}

Rational max(Rational a, Rational b) {...}
```

- Aber: Überladen mit Ergebnistyp alleine (gleiche Parameterlisten) unzulässig
- Beispiel, wird nicht übersetzt:

```
int upperLimit() {...}

double upperLimit() {...}
```

- Grund: Compiler kann einem Aufruf nicht ansehen, welche Methode gemeint ist:

```
System.out.println(upperLimit());
```

Konstruktoren

- Definition von Konstruktoren ohne Ergebnistyp, auch nicht `void`
- Resultat liegt automatisch fest: neues Objekt
- `return` in Konstruktoren unzulässig
- Keine Möglichkeit zur Rückkehr mitten aus dem Rumpf