

Softwareentwicklung II (IB)

Interfaces

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 18.05.2018 18:21

Inhaltsverzeichnis

| | |
|--|----------|
| Idee | 1 |
| Defaultmethoden und statische Methoden in Interfaces | 1 |
| Beispiel: Komplexe Zahlen | 2 |
| Interface <code>Complex</code> | 2 |
| Arithmetik mit komplexen Zahlen | 2 |
| Interface vs. Klasse | 3 |
| Implementieren von Interfaces | 3 |
| Klasse zum Interface | 3 |
| Beispiel <code>Cartesian</code> | 3 |
| Alternative Implementierung <code>Polar</code> | 4 |
| Gleichrangige Implementierungen | 5 |
| Interfaces als Typen | 6 |
| Methodenauswahl | 6 |
| Dynamisches Binden | 7 |
| Statischer Typ | 7 |
| Dynamischer Typ | 8 |
| Parameterübergabe | 8 |
| Beispiel: Addition komplexer Zahlen | 9 |
| Ergebnisrückgabe | 11 |
| Einsatz von Interfaces | 11 |
| Variablen in Interfaces | 12 |

Idee

- **Interface** = isolierte Schnittstelle, ohne Implementierung
- Definitionsschema:

```
interface Name {  
    public void method(...);  
    ...  
}
```

- Ähnlich wie Klassendefinition, aber mit reserviertem Wort `interface`
- Im Interface fehlen ...
 - Rümpfe der `public`-Methoden, statt dessen nur „;“
 - andere als `public`-Methoden
 - Konstruktoren
 - Objektvariablen

Defaultmethoden und statische Methoden in Interfaces

- seit Java 8 dürfen Interfaces sog. Defaultmethoden und statische Methoden enthalten
- diese werden wir im Rahmen dieser Lehrveranstaltung **nicht** betrachten!

Beispiel: Komplexe Zahlen

- Komplexe Zahlen = Punkte in der komplexen Zahlenebene (auch: kartesischen Zahlenebene)
- Reelle Achse (Abszisse, quer) und imaginäre Achse (Ordinate, senkrecht) .Darstellungen einer komplexen Zahl:
 - **Kartesische Darstellung** mit Realteil r und Imaginärteil i
 - **Polardarstellung** mit Abstand d vom Ursprung (Betrag) und Drehwinkel φ (ab positiver Abszisse gegen den Uhrzeigersinn)

Interface Complex

- Interface `Complex` für komplexe Zahlen:

```
interface Complex {  
    ...  
}
```

- Eigenschaften einer komplexen Zahl: Realteil, Imaginärteil, Betrag, Winkel
- Getter:

```
interface Complex {
    public double getReal();
    public double getImag();
    public double getDistance();
    public double getAngle();
}
```

Arithmetik mit komplexen Zahlen

- Grundrechenarten für komplexe Zahlen
- Stellvertretend

Addition: In kartesischer Darstellung Realteil und Imaginärteil getrennt addieren:

$$(r_1, i_1) + (r_2, i_2) = (r_1 + r_2, i_1 + i_2)$$

Multiplikation: In Polardarstellung Abstände multiplizieren, Winkel addieren:

$$(d_1, \varphi_1) \cdot (d_2, \varphi_2) = (d_1 \cdot d_2, \varphi_1 + \varphi_2)$$

- Im Interface als Methoden `add`, `mult` mit Wertesemantik:

```
interface Complex {
    ...
    public Complex add(Complex that);
    public Complex mult(Complex that);
}
```

Interface vs. Klasse

- Interface = Vertrag, aber Interface \neq Klasse
- Legt nur Anforderungen fest, liefert keine Implementierung
- Keine Objekte von Interfaces
- Wie bei Klassen: Eine Interfacedefinition pro Quelltextdatei, Dateiname = Interfacename + `.class`
- Beispiel: Interface `Complex` in der Quelltextdatei `Complex.java`
- Bytecode: ohne ausführbare Anweisungen
- Elemente eines Interfaces implizit `public`, alles andere unzulässig

Implementieren von Interfaces

Klasse zum Interface

- Isoliertes Interface nutzlos
- Konkrete Klassen implementieren das Interface = definieren die Methoden des Interface
- implements koppelt Klasse mit Interface. Schematisch:

```
class Name implements Interface {  
    ...  
}
```

- Klassendefinition ansonsten normal

Beispiel Cartesian

- Beispiel: Klasse Cartesian für komplexe Zahlen in kartesischer Darstellung:

```
class Cartesian implements Complex {...}
```

- double-Objektvariable für Real- und Imaginärteil:

```
class Cartesian implements Complex {  
    private final double real;  
    private final double imag;  
    ...  
}
```

- Cartesian muss alle Methoden des Interfaces Complex mit den gleichen Köpfen definieren:

```
class Cartesian implements Complex {  
    private final double real;  
    private final double imag;  
    @Override  
    public double getReal() {  
        return real;  
    }  
    @Override  
    public double getImag() {  
        return imag;  
    }  
}
```

```

@Override
public double getDistance() {
    return Math.hypot(real, imag);
}
@Override
public double getAngle() {
    return Math.atan2(imag, real);
}
@Override public Complex add(Complex that) {...}
@Override public Complex mult(Complex that) {...}
...
}

```

- Zusätzliche Methoden erlaubt
- Beispiel Konstruktor:

```

class Cartesian implements Complex {
    Cartesian(double r, double i) {
        real = r;
        imag = i;
    }
    ...
}

```

- `@Override` ist eine sog. Annotation
 - zeigt an, dass es eine Methode mit dieser Signatur im Interface gibt und diese hier implementiert werden soll

Alternative Implementierung Polar

- Klasse Polar für komplexe Zahlen in Polardarstellung, definiert ebenfalls das Interface Complex:

```

class Polar implements Complex {...}

```

- Objektvariablen für Betrag und Winkel:

```

class Polar implements Complex {
    private final double distance;
    private final double angle;
    ...
}

```

- Auch Polar definiert alle Methoden von Complex, aber mit anderen Rümpfen als Cartesian:

```
class Polar implements Complex {
    private final double distance;
    private final double angle;
    Polar(double d, double a) {
        distance = d;
        angle = a;
    }
    @Override
    public double getReal() {
        return distance * Math.cos(angle);
    }

    @Override
    public double getImag() {
        return distance * Math.sin(angle);
    }
    @Override
    public double getDistance() {
        return distance;
    }
    @Override
    public double getAngle() {
        return angle;
    }
    @Override public Complex add(Complex that) {...}
    @Override public Complex mult(Complex that) {...}
    ...
}
```

Gleichrangige Implementierungen

- Cartesian und Polar implementieren gleichrangig das Interface Complex
- Gemeinsames Interface gibt Methodenköpfe vor, identisch in Cartesian und Polar
- Cartesian und Polar unabhängig, isoliert, ohne gegenseitigen Bezug
- Objektvariablen, Methodenrumpfe unterschiedlich
- Ausgenommen vom Interface: Konstruktoren
- Interfaces kennen keine konkreten Objekte, keine Objektvariablen \Rightarrow können keine Konstruktoren

Interfaces als Typen

- Interfaces sind **Typen**, ebenso wie Klassen

- Zulässig für Variablendefinitionen, Parameterlisten, Methodenergebnis, Arrayelemente, ...
- Beispiel: Variable `c` vom Typ `Complex`:

```
Complex c;
```

- Objekte des Interface gibt es nicht. Was könnte `c` überhaupt zugewiesen werden?
- Alle implementierenden Klassen kompatibel zum Interface
- Beispiel: An `c` kann ein `Cartesian`-Objekt zugewiesen werden:

```
Complex c = new Cartesian(1, 2);
```

Methodenauswahl

- Methodenaufruf mit Variable von Interfacetyp wirft Fragen auf
- Einfacher Fall: Nur ein Kandidat: `getReal`-Methode der Klasse `Cartesian`

```
Complex c = new Cartesian(1, 2);  
System.out.println(c.getReal());
```

⇒ `getReal`-Methode der Klasse `Cartesian`

- `Cartesian` und `Polar` implementieren beide `Complex`.
- Vorhergehendes Beispiel mit `Polar`-Objekt, jetzt `getReal`-Methode der Klasse `Polar`:

```
Complex c = new Polar(1.7, 0.8); // vorher Cartesian  
System.out.println(c.getReal());
```

- Isolierte Anweisung: Welche `getReal`-Methode?

```
System.out.println(c.getReal());
```

- Im allgemeinen Fall: Der Compiler kann keine `getReal`-Methode auswählen!
- Beispiel mit Bedingung `b`, die erst zur Laufzeit entschieden wird (bspweise Benutzereingabe):

```
Complex c;  
if (b) {  
    c = new Cartesian(1, 2);  
} else {  
    c = new Polar(1.7, 0.8);  
}  
// c = Cartesian oder Polar?  
System.out.println(c.getReal());
```

Dynamisches Binden

- Auswahl einer konkreten Methode fällt erst zur Laufzeit, der Compiler trifft keine Entscheidung
- JVM sucht pro Aufruf eine Methode des momentan zugewiesenen Objekts
- Bezeichnung: **Dynamisches Binden** = Zuordnung Aufruf ↔ Rumpf zur Laufzeit
- Dynamisches Binden = weitere Form des Polymorphismus
- Beispiel: Wechsel in jedem Schleifendurchgang:

```
Complex c;  
for (int i = 0; i < 10; i++) {  
    if(i%2 == 0) {  
        c = new Cartesian(i, 2);  
    } else {  
        c = new Polar(i, 0.8);  
    }  
    System.out.println(c.getReal());  
}
```

Statischer Typ

- **Statischer Typ** = Typ einer Variablen gemäß Definition
- Beispiel: Statischer Typ von `c = Complex`

```
Complex c;
```

- Für eine Variable eines Interfacetyp: Der Compiler prüft, ob aufgerufene Methoden im Interface definiert sind
- Zur Laufzeit wird der Variablen ein Objekt einer konkreten, kompatiblen Klasse zugewiesen sein

- jede kompatible Klasse implementiert das Interface
- jede kompatible Klasse definiert jede Methode des Interface

⇒ irgend eine passende Methode muss existieren Was immer an `c` zugewiesen wird, die Methode `getReal` existiert:

```
Complex c;  
c = ...;  
System.out.println(c.getReal());
```

- Der Compiler kann für einen Methodenaufruf ...
 - sicherstellen, dass irgend eine passende Methode existiert

- nicht entscheiden, welche konkrete Methode das sein wird
- Compiler kann nicht vor dem Wert `null` schützen: Programmabbruch mangels Objekt

Dynamischer Typ

- **Dynamischer Typ** = Typ des tatsächlich an eine Variable zugewiesenen Objekt

```
Complex c = new Cartesian(1, 2);
```

c hat den ...

| | |
|-----------------|---|
| statischen Typ | <code>Complex</code> (gemäß Variablendefinition) |
| dynamischen Typ | <code>Cartesian</code> (das tatsächlich zugewiesene Objekt) |

- Der ...
 - statische Typ bleibt immer gleich
 - dynamische Typ kann sich ändern
- Beim Übersetzen ist der statische Typ entscheidend (Compiler)
- Zur Laufzeit ist der dynamische Typ entscheidend (JVM)

Parameterübergabe

- Parameterübergabe = versteckte Wertzuweisung
- Übergabe kompatibler Typen zulässig \Rightarrow Parameter können Interfacetyp haben
Kopier-Konstruktor von `Cartesian`. Naive Implementierung mit `Cartesian`-Parameter:

```
class Cartesian implements Complex {
    Cartesian(Cartesian that) { // Parametertyp Cartesian
        real = that.real; // Zugriff auf Objektvariable
        imag = that.imag;
    }
    ...
}
```

Nachteil: Kann nur komplexe Zahlen in der kartesischen Darstellung duplizieren

- Flexibler: Kopier-Konstruktor mit `Complex`-Parameter:

```
class Cartesian implements Complex {
    Cartesian(Complex that) { // Parametertyp Complex
```

```

    real = that.getReal(); // Getter
    imag = that.getImag(); // Getter
}
...
}

```

Vorteil: Kann alle komplexen Zahlen kopieren, insbes. auch Polardarstellung

- Zugriff auf Original-Objekt nur über Getter
- Entsprechender Kopier-Konstruktor in Polar:

```

class Polar implements Complex {
    Polar(Complex that) { // Paramertyp Complex
        distance = that.getDistance(); // Getter
        angle = that.getAngle();
    }
    ...
}

```

- Dynamischer Typ der Parameterobjekte für Konstruktoren irrelevant
- Alle Mischungen von Cartesian- und Polar-Objekten möglich:

```

Complex c;
c = new Cartesian(new Cartesian(1, 2));
c = new Cartesian(new Polar(1.5, 0.8));
c = new Polar(new Cartesian(1, 2));
c = new Polar(new Polar(1.5, 0.8));

```

Beispiel: Addition komplexer Zahlen

- Methode add addiert zwei komplexe Zahlen
- Kopf von add im Interface festgelegt, für alle komplexen Zahlen verfügbar:

```

interface Complex {
    ...
    public Complex add(Complex c);
}

```

- add-Parameter vom Typ Complex, d.h. beliebige komplexe Zahl
- Implementierende Klassen müssen add mit dem gleichen Kopf definieren:

```

class Cartesian implements Complex {
    ...
    @Override public Complex add(Complex c) { ... }
}

```

```

class Polar implements Complex {
    ...
    @Override public Complex add(Complex c) { ... }
}

```

- Implementierungen von add kennen den dynamischen Typ des Parameters c nicht, müssen mit Gettern auskommen •

```

class Cartesian implements Complex {
    ...
    @Override
    public Complex add(Complex c) {
        return new Cartesian(real + c.getReal(),
                               imag + c.getImag());
    }
}

```

```

class Polar ... // entsprechend

```

- Im Aufruf von add mehrfach dynamisches Binden:

```

Complex c = ...;
Complex d = ...;
if(c.add(d))
    ...

```

- Ablauf im konkreten Einzelfall recht komplex. Kann aber weitgehend ignoriert werden ...
 1. der Compiler stellt sicher, dass alle erforderlichen Methoden existieren
 2. dynamisches Binden der JVM ruft passende Implementierungen der Methoden auf
- Klassen können isoliert und unabhängig entwickelt, getestet, freigegeben, ... werden

Ergebnisrückgabe

- Schnittstellen von Arithmetik-Methoden im Interface:

```

interface Complex {
    ...
    public Complex add(Complex that);
    public Complex mult(Complex that);
}

```

- **Covarianter Ergebnistyp:**
Implementierende Klasse mit abweichendem Ergebnistyp
- Beispiel: Implementierung der Addition in `Cartesian`:

```
class Cartesian implements Complex {
    ...
    // statt public Complex add...
    @Override
    public Cartesian add(Complex c) {
        return new Cartesian(real + c.getReal(),
                               imag + c.getImag());
    }
}
```

Einsatz von Interfaces

- Entwicklung, Modifikation von Interface-Implementierungen ohne Rücksicht auf andere Klassen zum gleichen Interface
- Beispiel: neue, dritte Implementierung namens `Perplex`:
 - Existenz von `Cartesian` und `Polar` kann ignoriert werden
 - Anwenderprogramme können sofort mit `Perplex` arbeiten
 - Reibungsloses Zusammenspiel automatisch sichergestellt
- Anwendung: Berechnen der Potenz c^n (für ganzzahlige $n \geq 1$)

```
Complex power(Complex c, int n) {
    while(n > 1) {
        c = c.mult(c);
        n--;
    }
    return c;
}
```

Variablen in Interfaces

- Öffentliche Konstanten als Variablen in Interfaces zulässig
- Wenn nicht genannt: Compiler ergänzt automatisch Modifier `public static final`
- Initialisierung Pflicht
- Beispiel: Interface mit Definition der Kreiszahl π :

```
interface MathConstants {  
    double PI = 3.14;  
}
```

Äquivalent zu:

```
interface MathConstants {  
    public static final double PI = 3.14;  
}
```

- Andere, widersprechende Modifier unzulässig