1 Statische ausprogrammierte Methoden in Schnittstellen

In der Regel deklariert eine Schnittstelle Operationen, also abstrakte Objektmethoden, die eine Klasse später implementieren muss. Die in Klassen implementierte Schnittstellenmethode kann später wieder überschrieben werden, nimmt also ganz normal an der dynamischen Bindung teil. Einen Objektzustand kann die Schnittstelle nicht deklarieren, denn Objektvariablen sind in Schnittstellen tabu – jede deklarierte Variable ist automatisch statisch, also eine Klassenvariable.

In Schnittstellen sind statische Methoden erlaubt und lassen sich als Utility-Methoden neben Konstanten stellen. Es gibt also statische Klassenmethoden und statische Schnittstellenmethoden; beide werden nicht dynamisch gebunden.

Beispiel

Im vorangehenden Kapitel 5 hatten wir eine Schnittstelle `Buyable` deklariert. Die Idee ist, dass alles, was käuflich ist, diese Schnittstelle implementiert und einen Preis hat. Zusätzlich gibt es eine Konstante für einen Maximalpreis:

```
interface Buyable {
  int MAX_PRICE = 10 000 000;
  double price();
}
```

Hinzufügen lässt sich nun eine statische Methode `isValidPrice(double)`, die prüft, ob sich ein Kaufpreis im gültigen Rahmen bewegt:

```
interface Buyable {
  int MAX_PRICE = 10_000_000;
  static boolean isValidPrice( double price ) {
    return price >= 0 && price < MAX_PRICE;
  }
  double price();
}
```

Von außen ist dann der Aufruf `Buyable.isValidPrice(123)` möglich.

Alle deklarierten Eigenschaften sind implizit immer `public`, sodass dieser Sichtbarkeitsmodifizierer redundant ist. Konstanten sind implizit immer statisch. Statische Methoden müssen den Modifizierer `static` tragen, andernfalls gelten sie als abstrakte Operationen.

Hinweis

Statische Schnittstellenmethoden erlauben eine neue Möglichkeit zur Deklaration der `main(…)`-Methode:

```
interface HelloWorldInInterfaces {
  static void main( String[] args ) {
    System.out.println( "Hallo Welt einmal anders!" );
  }
}
```

Das Schlüsselwort `interface` ist vier Zeichen länger als `class`, doch mit der Einsparung von `public` und einem Trenner ergibt sich eine Kürzung von drei Zeichen – wieder eine neue Möglichkeit zum Längefeilschen.

Der Zugriff auf eine statische Schnittstellenmethode ist ausschließlich über den Namen der Schnittstelle möglich, bzw. die Eigenschaften können statisch importiert werden. Bei statischen Methoden von Klassen ist im Prinzip auch der Zugriff über eine Referenz erlaubt (wenn auch unerwünscht), etwa wie bei `new Integer(12).MAX_VALUE`. Allerdings ist das bei statischen Methoden von Schnittstellen nicht zulässig. Implementiert etwa `Car` die Schnittstelle `Buyable`, würde `new Car().isValidPrice(123)` zu einem

Compilerfehler führen. Selbst `Car.isValidPrice(123)` ist falsch, was doch ein wenig verwundert, da statische Methoden normalerweise vererbt werden.

Fassen wir die erlaubten Eigenschaften einer Schnittstelle zusammen:

	Attribut	**Methode**
Objekt-	nein, nicht erlaubt	ja, üblicherweise abstrakt
Statische(s)	ja, als Konstante	ja, immer mit Implementierung

Tabelle 1.1: Erlaubte Eigenschaften einer Schnittstelle

Gleich werden wir sehen, dass Schnittstellenmethoden durchaus eine Implementierung besitzen können, also nicht zwingend abstrakt sein müssen.

Design

Eine Schnittstelle mit nur statischen Methoden ist ein Zeichen für ein Designproblem und sollte durch eine finale Klasse mit privatem Konstruktor ersetzt werden. Schnittstellen sind immer als Vorgaben zum Implementieren gedacht. Wenn nur statische Methoden in einer Schnittstelle vorkommen, erfüllt die Schnittstelle nicht ihren Zweck, Vorgaben zu machen, die unterschiedlich umgesetzt werden können.

2 Erweitern von Schnittstellen

Sind Schnittstellen einmal deklariert und in einer großen Anwendung verbreitet, so sind Änderungen nur schwer möglich, da sie schnell die Kompatibilität brechen. Wird der Name einer Parametervariablen umbenannt, ist das kein Problem. Bekommt aber eine Schnittstelle eine neue Operation, führt das zu einem Übersetzungsfehler, wenn nicht bereits alle implementierenden Klassen diese neue Methode implementieren. Framework-Entwickler müssen also sehr darauf achten, wie sie Schnittstellen modifizieren, doch sie haben es in der Hand, wie weit die Kompatibilität gebrochen wird.

Geschichtsstunde

Schnittstellen später zu ändern, wenn schon viele Klassen die Schnittstelle implementieren, ist eine schlechte Idee. Denn erneuert sich die Schnittstelle, etwa wenn nur eine Operation hinzukommt oder sich ein Parametertyp ändert, dann sind plötzlich alle implementierenden Klassen kaputt. Sun selbst hat dies bei der Schnittstelle `java.sql.Connection` riskiert. Beim Übergang von Java 5 auf Java 6 wurde die Schnittstelle erweitert, und keine Treiberimplementierung konnte mehr compiliert werden.

*Code-Kompatibilität und Binär-Kompatibilität **

Es gibt Änderungen, wie zum Beispiel neu eingeführte Operationen in Schnittstellen, die zwar zu Compilerfehlern führen, aber zur Laufzeit in Ordnung sind. Bekommt eine Schnittstelle eine neue Methode, so ist das für die JVM überhaupt kein Problem. Die Laufzeitumgebung arbeitet auf den Klassendateien selbst, und sie interessiert es nicht, ob eine Klasse brav alle Methoden der Schnittstelle implementiert; sie löst nur Methodenverweise auf. Wenn eine Schnittstelle plötzlich „mehr“ vorschreibt, hat sie damit kein Problem.

Während also fast alle Änderungen an Schnittstellen zu Compilerfehlern führen, sind einige Änderungen für die JVM in Ordnung. Wir nennen das *Binär-Kompatibilität*. Wenn zum Beispiel die Schnittstelle verändert, neu übersetzt und in den Klassenpfad gesetzt wird, ist Folgendes in Ordnung:

- neue Methoden in Schnittstelle hinzufügen
- Schnittstelle erbt von einer zusätzlichen Schnittstelle.
- Hinzufügen oder Löschen einer `throws`-Ausnahme
- letzten Parametertyp von `T[]` in `T...` ändern
- neue Konstanten, also statische Variablen hinzufügen

Es gibt Änderungen, die jedoch nicht binärkompatibel sind und zu einem JVM-Fehler führen:

- Ändern des Methodennamens
- Ändern der Parametertypen und Umsortieren der Parameter
- formalen Parameter hinzunehmen oder entfernen

Strategien zum Ändern von Schnittstellen

Falls die Schnittstelle nicht weit verbreitet wurde, so lassen sich einfacher Änderungen vornehmen. Ist der Name einer Operation zum Beispiel schlecht gewählt, wird ein Refactoring in der IDE den Namen in der Schnittstelle genauso ändern wie auch alle Bezeichner in den implementierenden Klassen. Problematischer ist es, wenn externe Nutzer sich auf die Schnittstelle verlassen. Dann müssen Klienten ebenfalls Anpassungen durchführen, oder Entwickler müssen auf „Schönheitsänderungen“ wie das Ändern des Methodenamens einfach verzichten.

Kommen Operationen hinzu, hat sich eine Konvention etabliert, die im Java-Universum oft anzutreffen ist: Soll eine Schnittstelle um Operationen erweitert werden, so gibt es eine neue Schnittstelle, die die alte

erweitert und auf „2“ endet; `java.awt.LayoutManager2` ist ein Beispiel aus dem Bereich der grafischen Oberflächen, `Attributes2`, `EntityResolver2`, `Locator2` für XML-Verarbeitung sind weitere.[1]

Default-Methoden sind eine weitere Möglichkeit zur späteren Erweiterung von Schnittstellen. Sie erweitern die Schnittstelle, bringen aber gleich schon eine vorgefertigte Implementierung mit, sodass Unterklassen nicht zwingend eine Implementierung anbieten müssen. Das schauen wir uns jetzt an.

[1] Ein Blick auf die API vom Eclipse-Framework zeigt, dass bei mehr als 3.700 Typen dieses Muster um die sechzigmal angewendet wurde (*http://help.eclipse.org/mars/topic/org.eclipse.platform.doc.isv/reference/api/index.html?overview-summary.html*).

3 Default-Methoden

Ist eine Schnittstelle einmal verbreitet, so sollte es dennoch möglich sein, Operationen hinzuzufügen. Entwicklern sollte es erlaubt sein, neue Operationen einzuführen, ohne dass Unterklassen verpflichtet werden, diese Methoden zu implementieren. Damit das möglich ist, muss die Schnittstelle eine Standardimplementierung mitbringen. Auf diese Weise ist das Problem der „Pflicht-Implementierung“ gelöst, denn wenn eine Implementierung vorhanden ist, haben die implementierenden Klassen nichts zu meckern und können bei Bedarf das Standardverhalten überschreiben. Oracle nennt diese Methoden in Schnittstellen mit vordefinierter Implementierung *Default-Methoden*[2]. Schnittstellen mit Default-Methoden heißen *erweiterte Schnittstellen*.

Eine Default-Methode unterscheidet sich syntaktisch in zwei Aspekten von herkömmlichen implizit abstrakten Methodendeklarationen:

- Die Deklaration einer Default-Methode beginnt mit dem Schlüsselwort `default`.[3]
- Statt eines Semikolons markiert bei einer Default-Methode ein Block mit der Implementierung in geschweiften Klammen das Ende der Deklaration. Die Implementierung wollen wir Default-Code nennen.

Sonst verhalten sich erweiterte Schnittstellen wie normale Schnittstellen. Eine Klasse, die eine Schnittstelle implementiert, erbt alle Operationen, sei es die abstrakten Methoden oder die Default-Methoden. Falls die Klasse nicht abstrakt sein soll, muss sie alle von der Schnittstelle geerbten abstrakten Methoden realisieren; sie kann die Default-Methoden überschreiben, muss das aber nicht, denn eine Vorimplementierung ist ja schon in der Default-Methode der Schnittstelle gegeben.

Hinweis

Erweiterte Schnittstellen bringen „Code“ in eine Schnittstelle, doch das ging vorher auch schon, indem zum Beispiel eine implizite öffentliche und statische Variable auf eine Realisierung verweist:

```
interface Comparators {
  Comparator<String> TRIM_COMPARATOR = new Comparator<String>() {
    @Override public int compare( String s1, String s2 ) {
      return s1.trim().compareTo( s2.trim() );
    } };
}
```

Die Realisierung nutzt hier eine innere anonyme Klasse, ein Konzept, das genauer in Kapitel 8, „Äußere.innere Klassen“, beleuchtet wird.

3.1.1 Erweiterte Schnittstellen deklarieren und nutzen

Realisieren wir dies in einem Beispiel. Für Spielobjekte soll ein Lebenszyklus möglich sein; der besteht aus `start()` und `finish()`. Der Lebenszyklus ist als Schnittstelle vorgegeben, die Spielobjektklassen implementieren können. Version 1 der Schnittstelle sieht also so aus:

```
interface GameLifecycle {
  void start();
  void finish();
}
```

---

[2] Der Name hat sich während der Planung für dieses Feature mehrfach gewandelt. Ganz am Anfang war der Name „defender methods“ im Umlauf, dann lange Zeit „virtuelle Erweiterungsmethoden“ (engl. virtual extension methods).

[3] Am Anfang sollte `default` hinter dem Methodenkopf stehen, doch die Entwickler wollten `default` so wie einen Modifizierer wirken lassen; da Modifizierer aber am Anfang stehen, rutschte auch `default` nach vorne. Eigentlich ist ein Modifizierer auch gar nicht nötig, denn wenn es eine Implementierung, also einen Codeblock, in `{}` gibt, ist klar, dass es eine Default-Methode wird. Doch die Entwickler wollten eine explizite Dokumentation, so wie auch `abstract` eingesetzt wird – auch dieser Modifizierer bei Methoden wäre eigentlich gar nicht nötig, denn es gibt keinen Codeblock, wenn eine Methode abstrakt ist.

Klassen wie `Player`, `Room`, `Door` können die Schnittstelle erweitern, und wenn sie dies tun, müssen sie die beiden Methoden implementieren. Bei Spielobjekten, die diese Schnittstelle implementieren, kann unser Hauptprogramm, das Spiel, diese Methoden aufrufen und den Spielobjekten Rückmeldung geben, ob sie gerade in das Spiel gebracht wurden oder ob sie aus dem Spiel entfernt wurden.

Je länger Software lebt, desto mehr offenbaren sich Fehlentscheidungen beim Design. Die Umstellung einer ganzen Architektur ist eine Mammutaufgabe, einfache Änderungen wie das Umbenennen sind über ein Refactoring schnell erledigt. Nehmen wir an, dass es auch bei unserer Schnittstelle einen Änderungswunsch gibt – nur die Initialisierung und das Ende zu melden, reicht nicht. Geht das Spiel in einen Pausemodus, soll ein Spielobjekt die Möglichkeit bekommen, im Hintergrund laufende Programme anzuhalten. Das soll durch eine zusätzliche `pause()`-Methode in der Schnittstelle realisiert werden. Hier spielen uns Default-Methoden perfekt in die Hände, denn wir können die Schnittstelle erweitern, aber eine leere Standardimplementierung mitgeben. So müssen Unterklassen die `pause()`-Methode nicht implementieren, können dies aber; Version 2 der nun erweiterten Schnittstelle `GameLifecycle`:

```
interface GameLifecycle {
  void start();
  void finish();
  default void pause() {}
}
```

Klassen, die `GameLifecycle` schon genutzt haben, bekommen von der Änderung nichts mit. Der Vorteil: Die Schnittstelle kann sich weiterentwickeln, aber alles bleibt binärkompatibel, und nichts muss neu compiliert werden. Vorhandener Code kann auf die neue Methode zurückgreifen, die automatisch mit der „leeren“ Implementierung vorhanden ist. Außerdem verhalten sich Default-Methoden wie andere Methoden von Schnittstellen auch: Es bleibt bei der dynamischen Bindung, wenn implementierende Klassen die Methoden überschreiben. Wenn eine Unterklasse wie `Flower` zum Beispiel bei der Spielpause nicht mehr blühen möchte, so überschreibt sie die Methode und lässt etwa den Timer pausieren. Eine Tür dagegen hat nichts zu stoppen und kann mit dem Default-Code in `pause()` gut leben. Das Vorgehen ist ein wenig vergleichbar mit normalen nichtfinalen Methoden: Sie können, müssen aber nicht überschrieben werden.

Hinweis

Statt des leeren Blocks könnte der Rumpf auch `throw new UnsupportedOperationException ("Not yet implemented");` beinhalten, um anzukündigen, dass es keine Implementierung gibt. So führt eine hinzugenommene Default-Methode zwar zu keinem Compilerfehler, aber zur Laufzeit führen nicht überschriebene Methoden zu einer Ausnahme. Erreicht ist das Gegenteil vom Default-Code, weil eben keine Logik standardmäßig ausgeführt wird; das Auslösen einer Ausnahme zum Melden eines Fehlers wollen wir nicht als Logik ansehen.

Kontext der Default-Methoden

Default-Methoden verhalten sich wie Methoden in abstrakten Klassen und können alle Methoden der Schnittstelle (inklusive der geerbten Methoden) aufrufen.[4] Die Methoden werden später dynamisch zur Laufzeit gebunden.

Nehmen wir eine Schnittstelle `Buyable` für käufliche Objekte:

```
interface Buyable {
  double price();
}
```

Leider schreibt die Schnittstelle nicht vor, ob Dinge überhaupt käuflich sind. Eine Methode wie `hasPrice()` wäre in `Buyable` ganz gut aufgehoben. Was kann aber die Default-Implementierung sein? Wir können auf `price()` zurückgreifen und testen, ob die Rückgabe ein gültiger Preis ist. Das soll gegeben sein, wenn der Preis echt größer 0 ist.

[4] Und damit lässt sich das bekannte Template-Design-Pattern realisieren.

```
interface Buyable {
  double price();
  default boolean hasPrice() { return price() > 0; }
}
```

Implementieren Klassen die Schnittstelle `Buyable`, müssen sie `price()` implementieren, da die Methode keine Default-Methode ist. Doch es ist ihnen freigestellt `hasPrice()`, zu überschreiben, mit eigener zu Logik füllen und nicht die Default-Implementierung zu verwenden. Wenn implementierende Klassen keine neue Implementierung wählen, bekommen sie den Default-Code und erben eine konkrete Methode `hasPrice()`. In dem Fall geht ein Aufruf von `hasPrice()` intern weiter an `price()` und dann genau an die Klasse, die `Buyable` und die Methode `price()` implementiert. Die Aufrufe sind dynamisch gebunden und landen bei der tatsächlichen Implementierung.

Hinweis

Eine Schnittstelle kann die Methoden der absoluten Oberklasse `java.lang.Object` ebenfalls deklarieren, etwa um mit Javadoc eine Beschreibung hinzuzufügen. Allerdings ist es *nicht* möglich, mittels Default-Code Methoden wie `toString()` oder `hashCode()` vorzubelegen.

Neben der Möglichkeit, auf Methoden der eigenen Schnittstelle zurückzugreifen, steht auch die `this`-Referenz zur Verfügung. Das ist sehr wichtig, denn so kann der Default-Code an Utility-Methoden delegieren und einen Verweis auf sich selbst übergeben. Hätten wir zum Beispiel schon eine `hasPrice(Buyable)`-Methode in einer Utility-Klasse `PriceUtils` implementiert, so könnte der Default-Code aus einer einfachen Delegation bestehen:

```
class PriceUtils {
  public static boolean hasPrice( Buyable b ) { return b.price() > 0; }
}
interface Buyable {
  double price();
  default boolean hasPrice() { return PriceUtils.hasPrice( this ); }
}
```

Dass die Methode `PriceUtils.hasPrice(Buyable)` für den Parameter den Typ `Buyable` vorsieht und sich der Default-Code mit `this` auf genauso ein `Buyable`-Objekt bezieht, ist natürlich kein Zufall, sondern bewusst gewählt. Der Typ der `this`-Referenz zur Laufzeit entspricht dem der Klasse, die die Schnittstelle implementiert hat und deren Objektexemplar gebildet wurde.

Haben die Default-Methoden weitere Parameter, so lassen sich auch diese an die statische Methode weiterreichen:

```
class PriceUtils {
  public static boolean hasPrice( Buyable b ) { return b.price() > 0; }
  public static double defaultPrice( Buyable b, double defaultPrice ) {
    if ( b != null && b.price() > 0 )
      return b.price();
    return defaultPrice;
  }
}
interface Buyable {
  double price();
  default boolean hasPrice() { return PriceUtils.hasPrice( this ); }
  default double defaultPrice( double defaultPrice ) {
   return PriceUtils.defaultPrice( this, defaultPrice );
  }
}
```

Da Schnittstellen auch statische Utility-Methoden mit Implementierung enthalten können, kann der Default-Code auch hier weiterleiten. Allerdings sind die statischen Schnittstellen-Methoden immer öffentlich, und

vielleicht möchte der Default-Code an eine geschützte paketsichtbare Methode weiterleiten. Außerdem ist es vorzuziehen, die Implementierung auszulagern, damit die Schnittstellen nicht so codelastig werden. Nutzt das JDK Default-Code, so gibt es in der Regel immer eine statische Methode in einer Utility-Klasse.

3.1.2 Erweiterte Schnittstellen, Mehrfachvererbung und Mehrdeutigkeiten *

Hintergrund zur Einführung von Default-Methoden war die Notwendigkeit, Schnittstellen im Nachhinein ohne nennenswerte Compilerfehler mit neuen Operationen ausstatten zu können. Ideal ist, wenn neue Default-Methoden hinzukommen und Standardverhalten definieren und es dadurch zu keinem Compilerfehler für implementierende Klassen kommt oder zu Fehlern bei Schnittstellen, die erweiterte Schnittstellen erweitern.

Erweiterte Schnittstellen mit Default-Code nehmen ganz normal an der objektorientierten Modellierung teil, können vererbt und überschrieben werden und werden dynamisch gebunden. Nun gibt es einige Sonderfälle, die wir uns anschauen müssen. Es kann vorkommen, dass zum Beispiel

- eine Klasse von einer Oberklasse eine Methode erbt, aber gleichzeitig auch von einer Schnittstelle Default-Code für die gleiche Methode oder
- eine Klasse von zwei erweiterten Schnittstellen unterschiedliche Implementierungen angeboten bekommt.

Gehen wir verschiedene Fälle durch.

Überschreiben von Default-Code

Eine Schnittstelle kann andere Schnittstellen erweitern und neuen Default-Code bereitstellen. Mit anderen Worten: Default-Methoden können andere Default-Methoden aus Oberschnittstellen überschreiben und mit neuem Verhalten implementieren.

Führen wir eine Schnittstelle `Priced` mit einer Default-Methode ein:

```
interface Priced {
  default boolean hasPrice() { return true; }
}
```

Eine andere Schnittstelle kann die Default-Methode überschreiben:

```
interface NotPriced extends Priced {
  @Override default boolean hasPrice() { return false; }
}
public class TrueLove implements NotPriced {
  public static void main( String[] args ){
    System.out.println( new TrueLove().hasPrice() );          // false
  }
}
```

Implementiert die Klasse `TrueLove` die Schnittstelle `NotPriced`, so ist alles in Ordnung, und es entsteht kein Konflikt. Die Vererbungsbeziehung ist linear `TrueLove` → `NotPriced` → `Priced`.

Klassenimplementierung geht vor Default-Methoden

Implementiert eine Klasse eine Schnittstelle und erbt außerdem von einer Oberklasse, kann Folgendes passieren: Die Schnittstelle hat Default-Code für eine Methode, und die Oberklasse vererbt ebenfalls die gleiche Methode mit Code. Dann bekommt die Unterklasse von zwei Seiten eine Implementierung. Zunächst muss der Compiler entscheiden, ob so etwas überhaupt syntaktisch korrekt ist. Ja, das ist es!

```
interface Priced {
  default boolean hasPrice() { return true; }
}
class Unsaleable {
  public boolean hasPrice() { return false; }
}
public class TrueLove extends Unsaleable implements Priced {
```

```
  public static void main( String[] args ) {
    System.out.println( new TrueLove().hasPrice() );   // false
  }
}
```

`TrueLove` erbt die Implementierung `hasPrice()` von der Oberklasse `Unsaleable` und auch von der erweiterten Schnittstelle `Priced`. Der Code compiliert und führt zu der Ausgabe `false` – die Klasse mit dem Code „gewinnt“ also gegen den Default-Code. Merken lässt sich das ganz einfach an der Reihenfolge `class … extends … implements …` – es steht `extends` am Anfang, also haben Methoden aus Implementierungen hier eine höhere Priorität als die aus erweiterten Schnittstellen.

*Default-Methoden aus speziellen Oberschnittstellen ansprechen **

Eine Unterklasse kann eine konkrete Methode der Oberklasse überschreiben, aber dennoch auf die Implementierung der überschriebenen Methode zugreifen. Allerdings muss der Aufruf über `super` erfolgen, da sich sonst ein Methodenaufruf rekursiv verfängt.

Default-Methoden können andere Default-Methoden aus Oberschnittstellen ebenfalls überschreiben und mit neuem Verhalten implementieren. Doch genauso wie normale Methoden können sie mit `super` auf Default-Verhalten aus dem übergeordneten Typ zurückgreifen.

Nehmen wir für ein Beispiel unsere bekannte Schnittstelle `Buyable` und eine neue erweiterte Schnittstelle `PeanutsBuyable` an:

```
interface Buyable {
  double price();
  default boolean hasPrice() { return price() > 0; }
}
interface PeanutsBuyable extends Buyable {
  @Override default boolean hasPrice() {
    return Buyable.super.hasPrice() && price() < 50_000_000;
  }
}
```

In der Schnittstelle `Buyable` sagt der Default-Code von `hasPrice()` aus, dass alles einen Preis hat, was größer als 0 ist. `PeanutsBuyable` dagegen nutzt eine erweiterte Definition und implementiert daher das Default-Verhalten neu. Nach den berühmten kopperschen Peanuts[5] ist alles unter 50 Millionen problemlos käuflich und verursacht – zumindest für die Deutsche Bank – keine Schmerzen. In der Implementierung von `hasPrice()` greift `PeanutsBuyable` auf den Default-Code von `Buyable` zurück, um vom Obertyp eine Entscheidung über die Preiseigenschaft zu bekommen, die aber mit der Und-Verknüpfung noch spezialisiert wird.

*Default-Code für eine Methode von mehreren Schnittstellen erben **

Wenn eine Klasse aus zwei erweiterten Schnittstellen den gleichen Default-Code angeboten bekommt, führt das zu einem Compilerfehler. Die Klasse `RockAndRoll` zeigt dieses Dilemma:

```
interface Sex {
  default boolean hasPrice() { return false; }
}
interface Drugs {
  default boolean hasPrice() { return true; }
}
public class RockAndRoll implements Sex, Drugs { } // Compilerfehler
```

Selbst wenn beide Implementierungen identisch wären, müsste der Compiler das ablehnen, denn der Code könnte sich ja jederzeit ändern.

---

5 *https://de.wikipedia.org/wiki/Hilmar_Kopper#.E2.80.9EPeanuts.E2.80.9C*

Mehrfachvererbungsproblem mit super lösen

Die Klasse `RockAndRoll` lässt sich so nicht übersetzen, weil die Klasse aus zwei Quellen Code bekommt. Das Problem kann aber einfach gelöst werden, indem in `RockAndRoll` die `hasPrice()`-Methode überschrieben und dann an eine Methode delegiert wird. Um rekursive Aufrufe zu vermeiden, kommt wieder `super` mit der neuen Schreibweise ins Spiel:

```
interface Sex {
  default boolean hasPrice() { return false; }
}
interface Drugs {
  default boolean hasPrice() { return true; }
}
class RockAndRoll implements Sex, Drugs {
  @Override public boolean hasPrice() { return Sex.super.hasPrice(); }
}
```

Abstrakte überschriebene Schnittstellenoperationen nehmen Default-Methoden weg

Default-Methoden haben die interessante Eigenschaft, dass Untertypen den Status von „hat Implementierung“ in „hat keine Default-Implementierung“ ändern können:

```
interface Priced {
  default boolean hasPrice() { return false; }
}
interface Buyable extends Priced {
  @Override boolean hasPrice();
}
```

Die Schnittstelle `Priced` bietet eine Default-Methode. `Buyable` erweitert die Schnittstelle `Priced`, aber überschreibt die Methode – jedoch nicht mit Code! Dadurch wird sie in `Buyable` abstrakt. Eine abstrakte Methode kann also durchaus eine Default-Methode überschreiben. Klassen, die `Buyable` implementieren, müssen also nach wie vor eine `hasPrice()`-Methode implementieren, wenn sie nicht selbst abstrakt sein wollen. Es ist schon ein interessantes Java-Feature, dass die Implementierung einer Default-Methode in einem Untertyp wieder „weggenommen“ werden kann. Bei der Sichtbarkeit ist das zum Beispiel nicht möglich: Ist eine Methode einmal öffentlich, kann eine Unterklasse die Sichtbarkeit nicht einschränken.

Das Verhalten des Compilers hat einen großen Vorteil: Bestimmte Veränderungen der Oberschnittstelle sind erlaubt und haben keine Auswirkungen auf die Untertypen. Nehmen wir an, `hasPrice()` hätte es in `Priced` vorher nicht gegeben, sondern nur abstrakt in `Buyable`. Default-Code ist ja nur eine nette Geste, und diese sollte schmerzlos in `Priced` integriert werden können. Anders gesagt: Entwickler können in den Basistyp so eine Default-Methode ohne Probleme aufnehmen, ohne dass es in den Untertypen zu Fehlern kommt. Obertypen lassen sich also ändern, ohne die Untertypen anzufassen. Im Nachhinein kann aber zur Dokumentation die Annotation `@Override` an die Unterschnittstelle gesetzt werden.

Nicht nur eine Unterschnittstelle kann die Default-Methoden „wegnehmen“, sondern auch eine abstrakte Klasse:

```
abstract class Food implements Priced {
  @Override public abstract double price();
}
```

Die Schnittstelle `Priced` bringt eine Default-Methode mit, doch die abstrakte Klasse `Food` nimmt diese wieder weg, sodass erweiternde `Food`-Klassen auf jeden Fall `price()` implementieren müssen, wenn sie nicht selbst `abstract` sein wollen.

3.1.3 Bausteine bilden mit Default-Methoden *

Default-Methoden geben Bibliotheksdesignern ganz neue Möglichkeiten. Heute ist noch gar nicht richtig abzusehen, was Entwickler damit machen werden und welche Richtung die Java-API einschlagen wird. Auf jeden Fall wird sich die Frage stellen, ob eine Standardimplementierung als Default-Code in eine

Schnittstelle wandert oder wie bisher eine Standardimplementierung als abstrakte Klasse bereitgestellt wird, von der wiederum andere Klassen ableiten. Als Beispiel sei auf die Datenstrukturen verwiesen: Eine Schnittstelle `Collection` schreibt Standardverhalten vor, `AbstractCollection` gibt eine Implementierung so weit wie möglich vor, und Unterklassen wie Listen setzen dann noch einmal auf diese Basisimplementierung auf. Erweiterte Schnittstellen können Hierarchien abbauen, denn auf eine abstrakte Basisimplementierung kann verzichtet werden. Auf der anderen Seite kann aber eine abstrakte Klasse einen Zustand über Objektvariablen einführen, was eine Schnittstelle nicht kann.

Default-Methoden können aber noch etwas ganz anderes: Sie können als Bauelemente für Klassen dienen. Eine Klasse kann mehrere Schnittstellen mit Default-Methoden implementieren und erbt im Grunde damit Basisfunktionalität von verschiedenen Stellen. In anderen Programmiersprachen ist das als *Mixin* bzw. *Trait* bekannt. Das ist ein Unterschied zur Mehrfachvererbung, die in Java nicht zulässig ist. Schauen wir uns diesen Unterschied jetzt einmal genauer an.

Default-Methoden zur Entwicklung von Traits nutzen

Was ist das Kernkonzept der objektorientierten Programmierung? Wohl ohne zu zögern können wir Klassen, Kapselung und Abstraktion nennen. Klassen und Klassenbeziehungen sind das Gerüst eines jeden Java-Programms. Bei der Vererbung wissen wir, dass Unterklassen Spezialisierungen sind und das liskovsche Substitutionsprinzip (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**, „**Fehler! Verweisquelle konnte nicht gefunden werden.**“) gilt: Falls ein Typ gefordert ist, können wir auch einen Untertyp übergeben. So sollte perfekte Vererbung aussehen: Eine Unterklasse spezialisiert das Verhalten, aber erbt nicht einfach von einer Klasse, weil sie nützliche Funktionalität hat. Aber warum eigentlich nicht? Als Erstes ist zu nennen, dass das Erben aufgrund der Nützlichkeit oft gegen die Ist-eine-Art-von-Beziehung verstößt und dass uns Java zweitens nur Einfachvererbung mit nur einer einzigen Oberklasse erlaubt. Wenn eine Klasse etwas Nützliches wie Logging anbietet und unsere Klasse davon erbt, kann sie nicht gleichzeitig von einer anderen Klasse erben, um zum Beispiel Zustände in Konfigurationsdaten festzuhalten. Eine unglückliche Vererbung verbaut also eine spätere Erweiterung. Das Problem bei der „Funktionalitätsvererbung“ ist also, dass wir uns nur einmal festlegen können.

Wenn eine Klasse eine gewisse Funktionalität einfach braucht, woher soll diese denn dann kommen, wenn nicht aus der Oberklasse? Eigentlich gibt es hier nur eine naheliegende Variante: Die Klasse greift auf andere Objekte per Delegation zurück. Wenn ein Punkt mit Farbe nicht von `java.awt.Point` erben soll, kann ein Farbpunkt einfach in einer internen Variablen einen `Point` referenzieren. Das ist eine Lösung, aber dann nicht optimal, wenn eine Ist-eine-Art-von-Beziehung besteht. Und Schnittstellen wurden ja gerade eingeführt, damit eine Klasse mehrere Typen besitzt. Abstraktionen über Schnittstellen und Oberklassen sind wichtig, und Delegation hilft hier nicht. Gewünscht ist eine Technik, die einen Programmbaustein in eine Klasse setzen kann – im Grunde so etwas wie Mehrfachvererbung, aber doch anders, weil die Bausteine nicht als komplette Typen auftreten; der Baustein selbst ist nur ein Implantat und allein uninteressant. Auch ein Objekt kann von diesem Bausteintyp nicht erzeugt werden.

Am ehesten sind die Bausteine mit abstrakten Klassen vergleichbar, doch das wären Klassen, und Nutzer könnten nur einmal von diesem Baustein erben. Mit den erweiterten Schnittstellen gibt es ganz neue Möglichkeiten: Sie bilden die Bausteine, von denen Klassen Funktionalität bekommen können, wir nennen das Mixin bzw. Trait.[6] Diese Bausteine sind nützlich, denn so lässt sich ein Algorithmus in eine Extra-Kompilationseinheit setzen und leichter wiederverwenden. Ein Beispiel: Nehmen wir zwei erweiterte Schnittstellen `PersistentPreference` und `Logged` an. Die erste erweiterte Schnittstelle soll mit `store()` Schlüssel-Wert-Paare in die zentrale Konfiguration schreiben, und `get()` soll sie auslesen:

```
import java.util.prefs.Preferences;
interface PersistentPreference {
  default void store( String key, String value ) {
    Preferences.userRoot().put( key, value );
  }
  default String get( String key ) {
```

[6] Siehe etwa *http://scg.unibe.ch/archive/papers/Scha02aTraitsPlusGlue2002.pdf*.

```
    return Preferences.userRoot().get( key, "" );
  }
}
```

Die zweite erweiterte Schnittstelle ist `Logged` und bietet uns drei kompakte Logger-Methoden:

```
import java.util.logging.*;
interface Logged {
  default void error( String message ) {
    Logger.getLogger( getClass().getName() ).log( Level.SEVERE, message );
  }
  default void warn( String message ) {
    Logger.getLogger( getClass().getName() ).log( Level.WARNING, message );
  }
  default void info( String message ) {
    Logger.getLogger( getClass().getName() ).log( Level.INFO, message );
  }
}
```

Eine Klasse kann diese Bausteine nun einbauen:

```
class Player implements PersistentPreference, Logged {
  // …
}
```

Die Methoden sind nun Teil vom `Player` und können auch von Unterklassen überschrieben werden. Als Aufgabe für den Leser bleibt, die Implementierung von `store()` im `Player` zu verändern, sodass der Schlüssel immer mit „player.“ beginnt. Die Frage, die der Leser beantworten sollte, ist, ob `store()` von `Player` auf das `store()` von der erweiterten Schnittstelle zugreifen kann.

Default-Methoden weitergedacht

Für diese Bausteine, also die erweiterten Schnittstellen, gibt es viele Anwendungsfälle. Da die Java-Bibliothek schon an die 20 Jahre alt ist, würden heute einige Typen anders aussehen. Dass sich Objekte mit `equals(…)` vergleichen lassen können, könnte heute zum Beispiel in einer erweiterten Schnittstelle stehen, etwa so:[7]

```
interface Equals {
  default boolean equals( Object that ) {
    return this == that;
  }
}
```

So müsste `java.lang.Object` die Methode nicht für alle vorschreiben, wobei das jetzt sicherlich kein Nachteil ist. Natürlich gilt das Gleiche auch für die `hashCode()`-Methode, die heutzutage aus einer erweiterten Schnittstelle `Hashable` stammen könnte.

Und `java.lang.Number` ist ein weiteres Beispiel. Die abstrakte Basisklasse für Werte repräsentierende Objekte deklariert die abstrakten Methoden `doubleValue()`, `floatValue()`, `intValue()`, `longValue()` und die konkreten Methoden `byteValue()` und `shortValue()`. Bisher erben `AtomicInteger`, `AtomicLong`, `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short` von dieser Oberklasse. Auch diese Funktionalität ließe sich mit einer erweiterten Schnittstelle umsetzen.

Zustand in den Bausteinen?

Nicht jeder wünschenswerte Baustein ist mit erweiterten Schnittstellen möglich. Ein Grund ist, dass die Schnittstellen keinen Zustand einbringen können. Nehmen wir zum Beispiel einen Container als Datenstruktur, der Elemente aufnimmt und verwaltet. Einen Baustein für einen Container können wir nicht

---

[7] Die Schnittstelle compiliert mit dem jetzigen Java SE nicht, da eine Default-Methode keine Methode aus `Object` überschreiben kann.

so einfach implementieren, da ein Container Kinder verwaltet, und hierfür ist eine Objektvariable für den Zustand nötig. Schnittstellen haben nur statische Variablen, und die sind für alle sichtbar; und selbst wenn die Schnittstelle eine modifizierbare Datenstruktur referenzieren würde, würde jeder Nutzer des Container-Bausteins von den Veränderungen betroffen sein. Da es keinen Zustand gibt, existieren auch für Schnittstellen keine Konstruktoren und folglich auch nicht für solche Bausteine. Denn wo es keinen Zustand gibt, gibt es auch nichts zu initialisieren. Wenn eine Default-Methode einen Zustand benötigt, muss sie selbst diesen Zustand erfragen. Hier lässt sich eine Technik einsetzen, die Oracles Java Language Architect Brian Goetz „virtual field pattern“[8] nennt. Wie das geht, zeigt das folgende Beispiel.

Referenziert ein Behälter eine Menge von Objekten, die sortierbar sind, können wir einen Baustein `Sortable` mit einer Methode `sort()` realisieren. Die Schnittstelle `Comparable` soll die Klasse nicht direkt implementieren, da ja nur die referenzierten Elemente sortierbar sind, nicht aber Objekte der Klasse selbst; zudem soll eine neue Methode `sort()` in `Sortable` hinzukommen. Damit das Sortieren gelingt, muss die Implementierung irgendwie an die Daten gelangen, und hier kommt ein Trick ins Spiel: Zwar ist `sort()` eine Default-Methode, doch die erweiterte Schnittstelle `Sortable` besitzt eine abstrakte Methode `getValues()`, die die Klasse implementieren muss, und dem Sortierer die Daten gibt. Im Quellcode sieht das so aus:

Listing 3.1: SortableDemo.java, Teil 1

```
import java.util.*;
interface Sortable<T extends Comparable<?>> {
  T[] getValues();
  void setValues( T[] values );
  default void sort() {
    T[] values = getValues();
    Arrays.sort( values );
    setValues( values );
  };
}
```

Fassen wir zusammen: Damit `sort()` an die Daten kommt, erwartet `Sortable` von den implementierenden Klassen eine Methode `getValues()`, und damit die Daten nach dem Sortieren wieder zurückgeschrieben werden können, eine zweite Methode `setValues(…)`. Der Clou ist, dass die spätere Implementierung von `Sortable` mit den beiden Methoden dem Sortierer Zugriff auf die Daten gewährt – allerdings auch jedem anderem Stück Code, da die Methoden öffentlich sind. Da bleibt ein unschönes „Geschmäckle“ zurück.

Ein Nutzer von `Sortable` soll `RandomValues` sein; die Klasse erzeugt intern Zufallszahlen.

Listing 3.2: SortableDemo.java, Teil 2

```
class RandomValues implements Sortable<Integer> {
  private List<Integer> values = new ArrayList<>();
  public RandomValues() {
    Random r = new Random();
    for ( int i = r.nextInt( 20 ) + 1; i > 0; i-- )
      values.add( r.nextInt(10000) );
  }
  @Override public Integer[] getValues() {
    return values.toArray( new Integer[values.size()] );
  }

  @Override public void setValues( Integer[] values ) {
    this.values.clear();
    Collections.addAll( this.values, values );
```

[8] http://mail.openjdk.java.net/pipermail/lambda-dev/2012-July/005171.html

```
  }
}
```

Damit sind die Typen vorbereitet, und eine Demo schließt das Beispiel ab:

Listing 3.3: SortableDemo.java, Teil 3

```
public class SortableDemo {
  public static void main( String[] args ) {
    RandomValues r = new RandomValues();
    System.out.println( Arrays.toString( r.getValues() ) );
    r.sort();
    System.out.println( Arrays.toString( r.getValues() ) );
  }
}
```

Aufgerufen kommt auf die Konsole zum Beispiel:

```
[2732, 4568, 4708, 4302, 4315, 5946, 2004]
[2004, 2732, 4302, 4315, 4568, 4708, 5946]
```

So interessant diese Möglichkeit auch ist, ein Problem wurde schon angesprochen: Jede Methode in einer Schnittstelle ist `public`, ob sie nun eine abstrakte oder eine Default-Methode ist. Es wäre schön, wenn die Datenzugriffsmethoden nicht öffentlich sind, aber das geht nicht.

Wo wir gerade bei der Sichtbarkeit sind: Gibt es im Default-Code Codeduplizierung, so kann der gemeinsame Code bisher nicht in private Methoden ausgelagert werden, da es private Operationen in Schnittstellen nicht gibt. Allerdings läuft gerade ein Test, ob so etwas eingeführt werden soll.

Warnung!

Natürlich lässt sich mit Rumgetrickse ein Speicherort finden, der Exemplarzustände speichert. Es lässt sich zum Beispiel in der Schnittstelle ein Assoziativspeicher referenzieren, der eine `this`-Instanz mit einem Objekt assoziiert. Ein Container-Baustein, der mit `add()` Objekte in eine Liste setzt und sie mit `iterable()` herausgibt, könnte so aussehen:

```
interface ListContainer<T> {
  Map<Object,List<Object>> $ = new HashMap<>();
  default void add( T e ) {
    if ( ! $.containsKey( this ) )
      $.put( this, new ArrayList<Object>() );
    $.get( this ).add( e );
  }
  default public Iterable<T> iterable() {
    if ( ! $.containsKey( this ) )
      return Collections.emptyList();
    return (Iterable<T>) $.get( this );
  }
}
```

Nicht nur die öffentliche Konstante `$` ist ein Problem, sondern auch, dass es ein großartiges doppeltes Speicherloch ist. Ein Exemplar der Klasse, die diese erweiterte Schnittstelle nutzt, kann nicht so einfach entfernt werden, denn in der Sammlung ist noch eine Referenz auf das Objekt, und diese Referenz verhindert eine automatische Speicherbereinigung. Selbst wenn dieses Objekt weg wäre, hätten wir noch all die referenzierten Kinder der Sammlung in der `Map`. Das Problem ist nicht wirklich zu lösen, und hier müsste mit schwachen Referenzen tief in die Java-Voodoo-Kiste gegriffen werden. Alles in allem, keine gute Idee, und Java-Chefentwickler Brian Goetz macht auch klar:

„Please don't encourage techniques like this. There are a zillion ‚clever' things you can do in Java, but shouldn't. We knew it wouldn't be long before someone suggested this, and we can't stop you. But please, use your power for good, and not for evil. Teach people to do it right, not to abuse it."[9]

Daher: Es ist eine schöne Spielerei, aber der Zustand sollte eine Aufgabe der abstrakten Basisklassen oder vom Delegate sein.

Zusammenfassung

Was wir in den letzten Beispielen zu den Bausteinen gemacht haben, war, ein Standardverhalten in Klassen einzubauen, ohne dass dabei der Zugriff auf die nur einmal existierende Basisklasse nötig war und ohne dass die Klasse an Hilfsklassen delegiert. In dieser Arbeitsweise können Unterklassen in jedem Fall die Methoden überschreiben und spezialisieren. Wir haben es also mit üblichen Klassen zu tun und mit erweiterten Schnittstellen, die nicht selbst eigenständige Entitäten bilden. In der Praxis wird es immer Fälle geben, in denen für eine Umsetzung eines Problems entweder eine abstrakte Klasse oder eine erweiterte Schnittstelle in Frage kommt. Wir sollten uns dann noch einmal an die Unterschiede erinnern: Eine abstrakte Klasse kann Methoden aller Sichtbarkeiten haben und sie auch final setzen, sodass sie nicht mehr überschrieben werden können. Eine Schnittstelle dagegen ist mit puren virtuellen und öffentlichen Methoden darauf ausgelegt, dass die Implementierung überschrieben werden kann.

[9] http://mail.openjdk.java.net/pipermail/lambda-dev/2012-July/005166.html