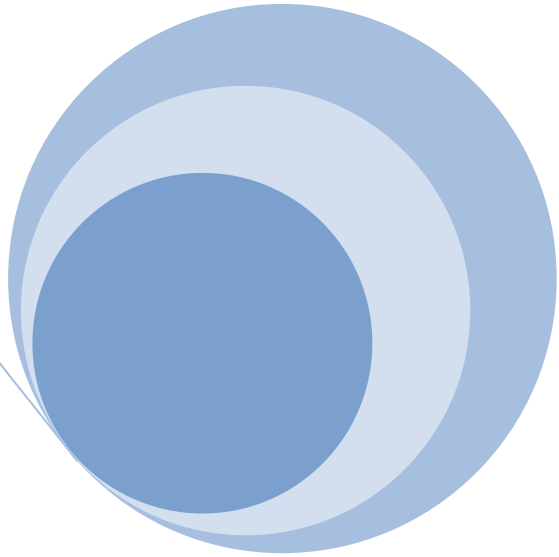
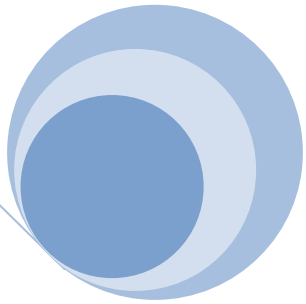
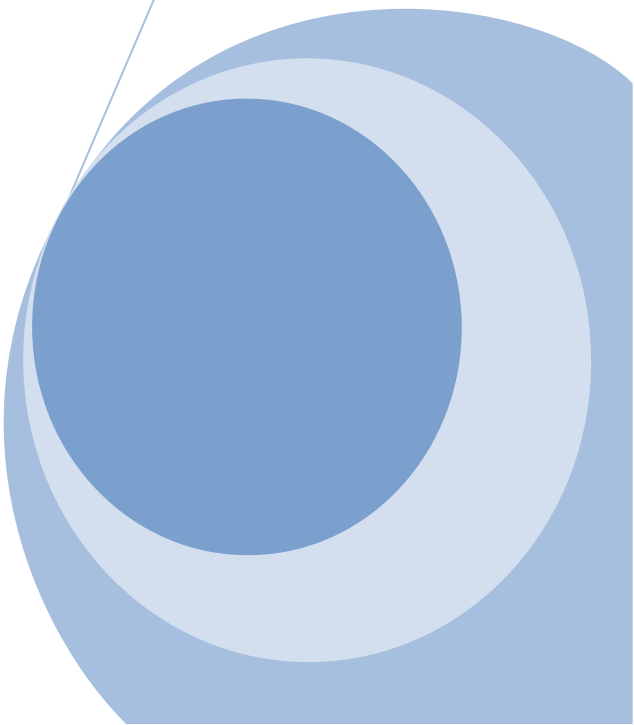




Generische Einheiten in Java, ein Tutorial

Von Steven Busack, Version 08.04.2011

Dieses Tutorial geht sowohl auf die grundlegenden als auch die weiterführenden Konzepte von generischen Einheiten in Java (Java Generics) ein.



Inhalt

Motivation

Mit der Bewilligung des JSR[1] 14 im Mai 1999 beschloss Sun Microsystems die Programmiersprache Java um generische Einheiten zu erweitern. Der Entwicklungsprozess erstreckte sich über 5 Jahre, bis im September 2004, mit der Veröffentlichung von Java 1.5, generische Einheiten in die Programmiersprache Java Einzug hielten. Im Zuge der Einführung wurden bereits bestehende APIs, wie das Java Collection API, überarbeitet und generisch re-implementiert. Generische Einheiten in Java (Java Generics) sind wohl die umfangreichste Java 1.5 Spracherweiterung. Vier Jahre nach ihrer Einführung in Java, sind sie jedoch noch ein relativ neues Konzept. Erstaunlicherweise gibt es nur sehr wenig umfangreiche Fachliteratur zum Thema generische Einheiten in Java. Die meisten aktuellen Java-Fachbücher beschäftigen sich zwar mit generischen Einheiten, beschreiben sie jedoch nur sehr grundlegend. Wichtige Themen, wie die Vereinbarung generischer Einheiten, werden oft nur sehr kurz beschrieben. Auf die Modellimplementierung wird in den meisten Fällen gar nicht eingegangen. Die Modellimplementierung gibt viel Aufschluss darüber, was mit generischen Einheiten in Java möglich ist. Java-Programmierer, die generische Einheiten verwenden, sollten zumindest grundlegende Kenntnisse über die Modellimplementierung besitzen.

Während meines Studiums habe ich generische Einheiten in zahlreichen Projekten eingesetzt. Dabei bin ich häufig auf (für mich) erstaunliche Phänomene gestoßen. Dies hat mich dazu veranlasst, mich näher mit generischen Einheiten in Java zu beschäftigen. Mir wurde schnell bewusst, dass sie keine triviale Spracheigenschaft sind. Um sie professionell einsetzen zu können, ist ein umfassendes Wissen nötig.

Mit dieser Arbeit soll ein Tutorial über generische Einheiten entstehen. Das Tutorial soll möglichst viele interessante Fälle behandeln, kann aber nicht vollständig sein. Die Schwerpunkte liegen dabei auf saubere, eindeutige Begriffe und leicht verständlichen Erklärungen. Ein Anspruch ist es, sowohl jenen gerecht zu werden, die einen schnellen Einstieg in das Thema generische Einheiten erhalten wollen (ohne näher ins Detail zu gehen), als auch jenen, die tiefer in das Thema einsteigen möchten. Das Tutorial wird deshalb in Grundlagen und Feinheiten unterteilt sein.

[1] Java Specification Request

Zusammenfassung

Diese Arbeit beschäftigt sich mit generischen Einheiten in Java. Ziel war es, ein Tutorial über generische Einheiten in Java zu verfassen, das alle interessanten Fälle behandelt. Das Tutorial unterteilt sich in die folgenden Abschnitte:

- Einleitung
- Generische Einheiten (Typen und Methoden)
- Modellimplementierung
- Formale Typparameter
- Aktuelle Typparameter
- Praktische Tipps
- Glossar

Kapitel 1 gibt eine Einleitung in das Thema generische Einheiten und erläutert die Ziele, Zielgruppe und die Voraussetzungen für dieses Tutorial. Mit einem kurzen Exkurs in das Gebiet der Typprüfung wird verdeutlicht, warum der Wunsch bestand, Java um generische Einheiten zu erweitern. Dabei werden auch zu generischen Einheiten alternative Lösungsansätze angedeutet. Am Ende folgt eine Zusammenfassung der wichtigsten Argumente für generische Einheiten.

Der Abschnitt über generische Einheiten ist in drei Kapitel unterteilt. In Kapitel 2 werden die Grundlagen über generische Typen erläutert. Ohne auf Spezialfälle einzugehen, wird gezeigt, wie generische Typen vereinbart werden. Desweiteren werden die Rohtyp-Instanziierung und konkrete Instanziierung erläutert. Kapitel 2 ist vor allem für jene interessant, die einen schnellen Einstieg in die Vereinbarung und Benutzung generischer Typen erhalten möchten, ohne näher ins Detail zu gehen. Kapitel 4 baut auf Kapitel 2 auf und beschäftigt sich mit den Feinheiten im Umgang mit generischen Typen. Schwerpunkte sind hier die Implementierung generischer Schnittstellen und die Erweiterung generischer Klassen. Weiter wird die Jokertyp-Instanziierung beschrieben, die es ermöglicht, generische Typen mit Jokerzeichen zu instanziieren. Kapitel 5 beschreibt wie generische Methoden vereinbart, beziehungsweise benutzt werden. Einen Schwerpunkt bildet dabei der Argumentenrückschluss, der die aktuellen Typparameter beim Aufruf einer generischen Methode automatisch herleitet.

Die Kapitel 3 und 8 beschreiben die Modellimplementierung. Das Kapitel 3 beschäftigt sich dabei mit den Grundlagen der Modellimplementierung. Hier geht es im Wesentlichen um die Anforderungen, die an die Spracherweiterung gestellt wurden und darum, wie diese Anforderungen umgesetzt wurden. Eine wesentliche Rolle spielt hier die während der Übersetzung durchgeführte Typentfernung. Kapitel 8 baut auf Kapitel 3 auf und befasst sich näher mit der Typentfernung und den damit verbundenen Techniken wie der Brückenmethoden-Generierung.

Generische Einheiten (generische Typen und generische Methoden) zeichnen sich durch formale Typparameter aus. Diese werden separat im Kapitel 6 erläutert. Die Schwerpunkte sind hier die Sichtbarkeit, die Gemeinsamkeiten und Unterschiede zu Java-Typen und die Bindung an Java-Typen. Die aktuellen Typparameter werden dagegen im Kapitel 7 beschrieben. Neben der Jokerbindung liegt der Fokus auf die als aktueller Typparameter zulässigen Typen.

Das Kapitel 9 enthält verschiedene Tipps im Umgang mit generischen Einheiten. Hier werden verschiedene Anwendungsfälle und Lösungsansätze skizziert. Die eingeführten englischen und deutschen Fachbegriffe rund um generische Einheiten in Java werden im Kapitel 10 erläutert. Neben der Definition enthalten die Erläuterungen auch zahlreiche Beispiele.

Während der Arbeit an diesem Tutorial ist die Anwendung JavaTEV entstanden. TEV steht für „type erasure viewer“. Der JavaTEV ist eine leichtgewichtige Anwendung, mit deren Hilfe die Auswirkung der Typentfernung auf den Code nachvollzogen werden kann. Die Anwendung enthält ausgewählte Beispiel-Quellcodes für verschiedene Anwendungsfälle. Ausführliche Informationen entnehmen Sie bitte dem Handbuch.

1 Einleitung

Die Veröffentlichung von Java 1.5 brachte zahlreiche Spracherweiterungen für die Programmiersprache Java. Eine dieser Spracherweiterungen waren generische Einheiten, die unter dem Begriff Java Generics Einzug in Java hielten. Wenn im Folgenden von generischen Einheiten gesprochen wird, sind damit immer generische Einheiten in Java gemeint. Natürlich gibt es diese Spracheigenschaft auch in anderen Programmiersprachen. Vergleiche oder Verweise auf diese Spracheigenschaft in anderen Programmiersprachen werden deshalb besonders hervorgehoben.

1.1 Ziele, Zielgruppe und Voraussetzungen

Generische Einheiten sind in Java keine triviale Spracheigenschaft. Um sie professionell benutzen zu können, ist ein umfassendes Wissen nötig. Dieses Tutorial versucht dieses Wissen mit klaren Begriffen zu vermitteln. Umfassend bedeutet jedoch nicht vollständig. Frau Angelika Langer hat auf ihrer Webseite[1] eine sehr umfangreiche Fragen-und-Antworten Sammlung (FAQ) über generische Einheiten veröffentlicht. Dieses Werk umfasst in der aktuellen Version rund 430 Seiten. Anders als das FAQ von Frau Langer, hat dieses Tutorial nicht den Anspruch, alle Aspekte von generischen Einheiten in Java zu behandeln. Das wäre aus meiner Sicht auch nicht sinnvoll. Vielmehr soll dieses Tutorial eine Einführung in, aus meiner Sicht, alle wichtigen Aspekte generischer Einheiten in Java geben.

Die Zielgruppe sind all diejenigen, die den Umgang mit generischen Einheiten in Java lernen möchten. Die Beispiele und Erklärungen sind so gewählt, dass sie auch von Personen mit grundlegenden Programmierkenntnissen nachvollzogen werden können. Ganz ohne Vorkenntnisse kommt dieses Tutorial jedoch nicht aus. Es sollten zumindest grundlegende Kenntnisse in den Bereichen objektorientierter Softwareentwicklung, der Programmiersprache Java sowie dem Java Collection API vorhanden sein.

Die Fachsprache der Informatik ist Englisch. Ich habe mich dennoch dafür entschieden, dieses Tutorial komplett in Deutsch zu schreiben. Das umfasst auch die Fachbegriffe. Es wurde versucht, für englische Begriffe geeignete deutsche Begriffe zu finden. Die entsprechenden englischen Begriffe werden bei der Einführung eines neuen Fachbegriffes genannt. Das Kapitel 10 enthält ein Glossar mit allen eingeführten Fachbegriffen des Bereiches der generischen Einheiten in Java. Dort werden sowohl die deutschen als auch die englischen Fachbegriffe aufgelistet.

1.2 Warum generische Einheiten?

Bevor man sich mit einer so umfangreichen Spracheigenschaft auseinandersetzt, sollte man sich zunächst über ihren Nutzen im Klaren sein. Dieses Kapitel beschäftigt sich mit der Frage warum der Wunsch bestand, Java um generische Einheiten zu erweitern.

Definition Eine generische Einheit ist (in Java) ein generischer Typ oder eine generische Methode. Generische Einheiten sind mit formalen Typparametern parametrisiert.

Das Java Collection API hat wohl am stärksten von der Einführung generischer Einheiten in Java profitiert. In der Java-Version 1.5 wurde das gesamte API überarbeitet und generisch re-implementiert. Anhand von Sammlungen lässt sich wohl auch am besten der Nutzen von generischen

[1] http://www.angelikalanger.com/

Einheiten in Java demonstrieren. Unter Sammlung versteht man in Java eine Instanz einer Klasse oder eine Schnittstelle, die `java.util.Collection` implementiert, beziehungsweise erweitert. Eine solche Sammlung hat eine Aufgabe: Das Sammeln von Objekten (Elementen). Unter sammeln versteht man das Hinzufügen von Elementen zu einer Sammlung, das Entfernen von Elementen aus einer Sammlung und das Suchen von Elementen in einer Sammlung. Neben diesen primären Aufgaben gibt es noch weitere Aufgaben oder besser Anforderungen, die an Sammlungen gestellt werden. Zu den wichtigsten gehören universelle Wiederverwendbarkeit und Typsicherheit. Universelle Wiederverwendbarkeit setzt eine vom Elementtyp unabhängige Implementierung voraus. Bis zur Einführung generischer Einheiten wurde dies durch die Verwendung der Wurzelklasse `Object` erreicht. Der Elementtyp jeder Sammlung war `Object`. Dadurch konnte eine Sammlung zum sammeln aller Referenztypen benutzt werden. Nur primitive Typen konnten mit einer solchen Sammlung nicht gesammelt werden. Die Verwendung von `Object` als Elementtyp hatte jedoch auch einige Nachteile. Um mit den Elementen einer Sammlung sinnvoll arbeiten zu können, waren verengende Typumwandlungen (engl. down casts) nötig. Das folgende Beispiel zeigt vereinfacht einen typischen Anwendungsfall:

```
1 java.util.ArrayList strings = new java.util.ArrayList();
2 strings.add("eine Zeichenkette");
3 String firstElement = (String) strings.get(0);
```

Die Methode `get(int)` der Klasse `ArrayList` liefert ein Element vom Typ `Object`. Um mit dieses Element sinnvoll verwenden zu können, war in der Regel eine verengende Typumwandlung, wie in Zeile 3, nötig. Verengende Typumwandlungen sind potentiell gefährlich, denn sie können zur Laufzeit fehlschlagen. Aus diesem Grund müssen sie vom Programmierer explizit angeordnet werden. Der Programmierer übernimmt damit die Verantwortung für den Erfolg der Typumwandlung. Das Beispiel verdeutlicht die Schwachstelle von nicht-generischen Sammlungen. Die universelle Wiederverwendbarkeit von Sammlungen konnte vor der Einführung generischer Einheiten nur auf Kosten der Typsicherheit erreicht werden.

Typsicherheit unterteilt sich in statische und dynamische Typsicherheit. Typprüfungen werden unter anderem bei Zuweisungen oder Methodenaufrufen durchgeführt. Ein Codeabschnitt kann sowohl statisch als auch dynamisch typsicher sein. Statische Typprüfungen haben einen entscheidenden Vorteil: Sie können durchgeführt werden, ohne den betreffenden Codeabschnitt ausführen zu müssen. In Java werden statische Typprüfungen in der Regel vom Übersetzer (engl. Compiler) durchgeführt. Dynamische Typprüfungen werden hingegen erst zur Laufzeit (in Java von der JVM) durchgeführt. Schlägt eine Typprüfung fehl, führt das zu einem Typfehler. In Java werden Typfehler immer gefunden (entweder statisch bei der Übersetzung oder dynamisch zur Laufzeit). Den Unterschied zwischen statischer und dynamischer Typsicherheit verdeutlicht der folgende Quellcodeausschnitt:

```
1 Integer int1 = new Integer(10);   // statisch typsicher
2 Number num   = int1;              // statisch typsicher
3 Integer int2 = (Integer) num;     // kann zur Laufzeit fehlschlagen
4 Double dou1  = (Double) num;      // schlägt zur Laufzeit fehl
5 String str   = (String) int1;     // statischer Typfehler: inkompatible
                                       Typen
```

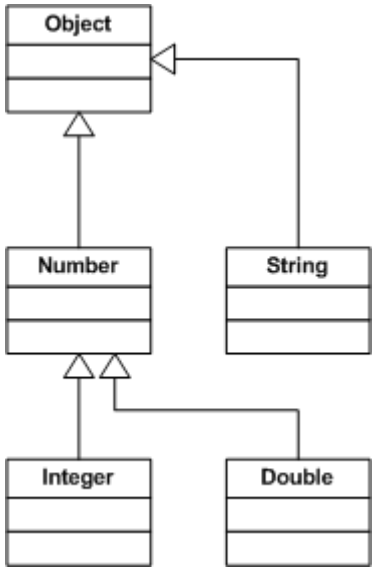


Abbildung 1 Ausschnitt aus der Java-Vererbungshierarchie

Die fünf Codezeilen zeigen verschiedene Zuweisungen. Bei allen fünf Zuweisungen kann der Übersetzer statische Typprüfungen durchführen. Statisch typsicher sind jedoch nur die Zuweisungen in den Zeilen 1 und 2. Typprüfungen werden mit Hilfe der Vererbungshierarchie durchgeführt. Abbildung 1 zeigt den für das Beispiel relevanten Teil der Java-Vererbungshierarchie. In den Zeilen 2 bis 5 werden verschiedene Typumwandlungen durchgeführt. Der Erfolg einer Typumwandlung setzt voraus, dass sich sowohl Quelltyp, als auch Zieltyp im selben Zweig der Vererbungshierarchie befinden. In Zeile 1 ist sowohl die Referenz `int1`, als auch das Objekt, das ihr zugewiesen wird, vom selben Typ (`Integer`). Hier muss keine Typumwandlung durchgeführt werden. Eine solche Zuweisung ist statisch typsicher. Zeile 2 enthält eine erweiternde Typumwandlung (engl. up cast) von `Integer` nach `Number`. Sowohl `Integer` als auch `Number` befinden sich im selben Zweig der Vererbungshirarchie. Eine erweiternde Typumwandlung ist statisch typsicher und wird vom Übersetzer implizit eingefügt. Aus diesem Grund muss in Zeile 2 keine explizite Typumwandlung angeordnet werden. Zeile 3 enthält eine verengende Typumwandlung von `Number` nach `Integer`. Wie bereits angedeutet, können verengende Typumwandlungen zur Laufzeit fehlschlagen und sind daher potentiell gefährlich. Der Übersetzer kann hier lediglich prüfen, ob sich Quell- und Zieltyp im selben Zweig der Vererbungshierarchie befinden. Über den Erfolg der Typumwandlung kann er jedoch keine Aussage treffen. Zeile 4 zeigt warum. Auch hier befinden sich Quell- und Zieltyp im selben Zweig der Vererbungshierarchie. Die Typumwandlung schlägt dennoch immer fehl. Zum Zeitpunkt der Zuweisung referenziert `num` auf ein Objekt vom Typ `Integer`. Die Typen `Double` und `Integer` sind inkompatibel. Die Folge ist eine `ClassCastException` zur Laufzeit. Der Übersetzer hat keinerlei Möglichkeit statisch festzustellen, von welchem Typ das Objekt ist, das `num` zu diesem Zeitpunkt referenziert. Zeile 5 enthält eine Typumwandlung, bei der sich Quell- und Zieltyp in verschiedenen Zweigen der Vererbungshierarchie befinden. In diesem Fall kann der Übersetzer statisch feststellen, dass diese Typumwandlung immer fehlschlagen wird.

Dieses kurze Beispiel zeigt bereits, dass mit Hilfe statischer Typprüfung viele Typfehler bereits vor der Übersetzung (und somit bevor der Code das erste Mal ausgeführt wird) gefunden werden können. Das folgende Beispiel zeigt die potentielle Gefahr verengender Typumwandlungen im Bezug auf Sammlungen:

```
1 ArrayList strings = new ArrayList();
2 strings.add("eine Zeichenkette");
3 strings.add(new Integer(10)); // möglich, jedoch unerwünscht
4
5 for(int i = 0; i < strings.size(); i++) {
6     String current = (String) strings.get(i); // schlägt für i == 1 fehl
7     // ...
8 }
```

In der in Zeile 1 vereinbarten Sammlung `strings` sollen Zeichenketten gesammelt werden. Vor der Einführung generischer Einheiten gab es jedoch keine Möglichkeit, den Elementtyp einer Sammlung statisch festzulegen. Infolge konnte (wie in Zeile 3) jedes beliebige Objekt zu einer Sammlung hinzugefügt werden. Ein solches „irrtümlich“ hinzugefügtes Element kann später zu unerwarteten

Ausnahmen führen. In Zeile 6 wird eine verengende Typumwandlung nach `String` durchgeführt unter der Annahme, dass die Sammlung ausschließlich Elemente vom Typ `String` enthält. Für das in Zeile 3 hinzugefügte `Integer`-Objekt schlägt diese Typumwandlung zur Laufzeit mit einer `ClassCastException` fehl. Um solche Fehler zu vermeiden, muss verhindert werden, dass unerwünschte Objekte zu einer Sammlung hinzugefügt werden können. Ein Lösungsansatz wäre eine dynamisch typsichere Sammlung. Der folgende Codeausschnitt zeigt den Ansatz einer solchen Sammlung:

```
1  public class DynamicTypeSafeCollection implements Collection {
2
3     private final Class desiredElementType;
4
5     public DynamicTypeSafeCollection(Class desiredElementType) {
6           this.desiredElementType = desiredElementType;
7     }
8
9     public boolean add(Object toAdd) {
10          if(this.desiredElementType.isInstance(toAdd)) {
11                // Element hinzufügen
12                return true;
13          }
14          else throw new ClassCastException("Unaccepted element type.");
15    }
16    // ...
17 }
```

Der Sammlung wird im Konstruktor eine `Class`-Referenz übergeben. Diese Referenz repräsentiert den Elementtyp. Methoden zum Hinzufügen von Elementen `add(Object)` prüfen, ob das übergebene Objekt vom gewünschten Typ ist. Das Hinzufügen unerwünschter Objekte führt zu einer `Exception`. Obwohl nun sichergestellt werden kann, dass die Sammlung ausschließlich Elemente vom gewünschten Typ enthält, bleiben einige Schwachstellen bestehen:

- Keine statische Typsicherheit
  - Der benutzende Code muss ausgeführt werden, um mögliche Programmfehler zu finden
  - Programmfehler, die nur bei bestimmten Bedingungen auftreten (zum Beispiel innerhalb einer **`if-else`**-Bedingung), bleiben eventuell lange unentdeckt
- Der Quellcode wird aufgebläht
  - Der Quellcode enthält viele prüfende Codezeilen, die ihn unübersichtlich und schwer wartbar machen
  - Geworfene Ausnahmen sollten (müssen) entsprechend behandelt werden: Ausnahmen, wie in Zeile 14, sollten im benutzenden Code behandelt werden. Das macht den Code unübersichtlich und schwer wartbar.

Alle diese Schwachstellen lassen sich mit generischen Einheiten beseitigen. Sie ermöglichen es unter anderem, den Elementtyp einer Sammlung statisch festzulegen und Typfehler, die vorher erst zur Laufzeit auftraten, bereits bei der Übersetzung zu finden.

Zusammenfassung

Der vorherige Abschnitt zeigt deutlich warum der dringende Wunsch bestand, Java um generische Einheiten zu erweitern. Generische Einheiten können einen nicht unerheblichen Qualitätsanstieg mit sich bringen. Darüber hinaus können sie dabei helfen, Entwicklungskosten zu sparen, denn viele Typfehler werden bereits vor der Übersetzung und nicht erst zur Laufzeit gefunden.

Die wichtigsten Argumente für generische Einheiten zusammengefasst:

- Ausweitung statischer Typprüfungen → Das Risiko von Typfehlern zur Laufzeit wird verringert
- Verengende, potentiell gefährliche Typumwandlungen werden nicht manuell vom Programmierer, sondern automatisch vom Übersetzer eingefügt
- Sammlungen können sowohl statisch typsicher als auch typunabhängig vereinbart werden

2 Generische Typen I

Nachdem sich das vorherige Kapitel mit dem Nutzen generischer Einheiten beschäftigte, geht es in diesem Kapitel um die Grundlagen von generischen Typen. Dieses Kapitel zeigt, wie generische Typen vereinbart und benutzt werden. Generische Typen sind (in Java) Referenztypen mit formalen Typparametern. Ein formaler Typparameter ist ein Platzhalter für einen beliebigen Referenztyp. Er ermöglicht es, universelle und zugleich statisch typsichere Typen zu vereinbaren.

Definition Ein generischer Typ ist (in Java) ein Referenztyp mit formalen Typparametern.

2.1 Generische Typen vereinbaren

Von der Ebene des Quellcodes aus betrachtet, unterscheiden sich generische Typen von nicht-generischen Typen allein durch formale Typparameter. Ein formaler Typparameter ist ein Platzhalter für einen Java-Typ. Ein Typ sollte immer dann generisch vereinbart werden, wenn die Vereinbarung sowohl universell als auch statisch typsicher sein soll. Die formalen Typparameter werden in spitzen Klammern hinter dem Typnamen vereinbart. Mehrere formale Typparameter werden durch Kommata getrennt. Beispiele:

```
1 class Twins<E> { }
2 class Pair<T, S> { }
3 interface Comparable<T> { }
```

Die in den Zeilen 1 und 3 vereinbarten Typen `Twins` und `Comparable` haben jeweils einen formalen Typparameter, der Typ `Pair` (in Zeile 2) hat zwei formale Typparameter. Der Bezeichner für einen formalen Typparameter kann frei gewählt werden. Es empfiehlt sich jedoch, der Konvention zu folgen und formale Typparameter mit einem Großbuchstaben zu benennen. Nach der Vereinbarung kann ein formaler Typparameter im gesamten nicht-statischen Kontext des umgebenen Typen benutzt werden. Bei einer generischen Schnittstelle sind das die Methodenvereinbarungen. Der folgende Quellcode zeigt die generische Schnittstelle `Comparable` aus der Java-Standardbibliothek:

```
1 public interface Comparable<T> {
2     int compareTo(T other);
3 }
```

In der Methode `compareTo(T)` wird der formale Typparameter `T` als formaler Parameter benutzt. Auf diese Art können beliebige Objekte gleichen Typs verglichen werden.

Von ein paar Ausnahmen abgesehen, können formale Typparameter im gesamten nicht-statischen Kontext anstelle von Referenztypen benutzt werden. Es ist immer dann sinnvoll, einen formalen Typparameter zu verwenden, wenn man sich nicht auf einen konkreten Referenztyp festlegen möchte. Wie bereits erwähnt, können Objekte gleichen Typs mit Hilfe der Schnittstelle `Comparable` verglichen, beziehungsweise geordnet werden. Der Typ der zu ordnenden Objekte spielt dabei für die Vereinbarung der Schnittstelle keine Rolle. Ähnlich ist dies bei Sammlungen. Der folgende Quellcode zeigt die generische Klasse `Twins`. Die Klasse `Twins` ist zwar keine Sammlung im eigentlichen Sinne (erweitert nicht `java.util.Collection`), soll jedoch zwei typgleiche Objekte zusammenfassen. Jedes Objekt der Klasse `Twins` repräsentiert ein Paar typgleicher Objekte.

```
 1 public class Twins<E> {
 2
 3    private E first;
 4
 5    private E second;
 6
 7    public Twins() {}
 8
 9    public Twins(E first, E second) {
10          this.first = first;
11          this.second = second;
12    }
13
14    public E getFirst() { return this.first; }
15
16    public void setFirst(E first) { this.first = first; }
17
18    public E getSecond() { return this.second; }
19
20    public void setSecond(E second) { this.second = second; }
21 }
```

Der formale Typparameter `E` wird hier an verschiedenen Stellen benutzt. In den Zeilen 3 und 5 wird er als Typ einer Variablen benutzt, in den Zeilen 16 und 20 als formaler Parameter und in den Zeilen 14 und 18 als Rückgabetyp. Die Klasse `Twins` ist ein Beispiel für einen Typ, den man vor der Einführung generischer Einheiten nicht hätte statisch typsicher vereinbaren können. Vor Java 1.5 hätte man anstelle des formalen Typparameters `E` die Wurzelklasse `Object` benutzen müssen. Um die gesammelten Objekte sinnvoll nutzen zu können, wären dann jedoch verengende Typumwandlungen nötig.

2.2 Generische Typen benutzen

Um einen generischen Typ benutzen zu können, muss er zunächst instanziiert werden. Bei der Instanziierung werden alle formalen Typparameter mit aktuellen Typparametern belegt. Ein generischer Typ kann auf drei Arten instanziiert werden. Zwei der drei Typinstanziierungen werden in diesem Kapitel beschrieben. Die dritte Art wird im Kapitel 4.2.1 beschrieben. Ein generischer Typ kann beliebig oft und mit beliebigen aktuellen Typparametern instanziiert werden. In anderen Worten, von einem generischen Typ können zu einem Zeitpunkt verschiedene Instanziierungen existieren.

2.2.1 Rohtyp-Instanziierung

Bei einer Rohtyp-Instanziierung werden keine aktuellen Typparameter angegeben. Alle formalen Typparameter werden implizit durch `Object` belegt. Beispiele:

```
1 Comparable
2 Twins
3 Pair
```

Eine solche Instanz nennt man *Rohtyp*. Im Zuge der Einführung generischer Einheiten wurden einige APIs, wie das Java Collection API generisch re-implementiert. Damit die re-implementierten APIs auch im früheren (nicht-generischen Code) lauffähig sind, wurde die Rohtyp-Instanziierung (engl. raw type instantiation) erlaubt. Die Java Language Specification weist jedoch ausdrücklich darauf hin, das Rohtypen in späteren Java-Versionen eventuell nicht mehr unterstützt werden. Ein Rohtyp kann wie ein nicht-generischer Typ benutzt werden. Die Verwendung von Rohtypen hat jedoch den Nachteil,

dass der große Vorteil generischer Typen (die statische Typsicherheit) eingebüßt wird. Aus diesem Grund meldet der Übersetzer bei der Verwendung von Rohtypen Warnungen. Es empfiehlt sich, wenn möglich, parametrisierte Typen anstelle von Rohtypen zu benutzen. Das folgende Beispiel zeigt warum:

```
1 Twins strTwins = new Twins();
2
3 strTwins.setFirst(new Integer(10));
4 strTwins.setSecond("SomeClass");
5
6 String first = (String) strTwins.getFirst(); // ClassCastException
```

Den Methoden `setFirst(...)` und `setSecond(...)` kann, wie in den Zeilen 3 und 4, ein beliebiges Objekt übergeben werden. Zur Erinnerung: Jedes Objekt der Klasse `Twins` soll ein Paar typgleiche Objekte repräsentieren. Durch die Rohtyp-Instanziierung kann jedoch nicht gewährleistet werden, dass beide Objekte typgleich sind. Generell sollte man die Verwendung von Rohtypen in neuem Code vermeiden. Es gibt jedoch zwei Fälle, in denen man um die Verwendung von Rohtypen nicht herumkommt:

- **`class`**-Literal
  ```
  Twins<String>.class          // unzulässing
  Twins.class                  // zulässing
  ```

- **`instanceof`**-Ausdruck
  ```
  obj instanceof Twins<String> // unzulässing
  obj instanceof Twins         // zulässing
  ```

2.2.2 Instanziierung mit konkreten Typen

Eine weitere Art generische Typen zu instanziieren stellt die *konkrete Instanziierung* dar. Anders als bei der Rohtyp-Instanziierung werden hier die formalen Typparameter eines generischen Typen explizit mit aktuellen Typparametern belegt. Die Reihenfolge der aktuellen Typparameter entspricht dabei der Reihenfolge, in der die formalen Typparameter vereinbart wurden. Es müssen alle formalen Typparameter durch aktuelle Typparameter belegt werden. Die Belegung erfolgt in spitzen Klammern, hinter dem Typnamen. Mehrere aktuelle Typparameter werden auch hier durch Kommata getrennt. Beispiele:

```
1 Comparable<String>
2 Twins<Integer>
3 Pair<String, Integer>
```

Eine solche Instanz nennt man *konkreter parametrisierter Typ* (engl. concrete parameterized type).

Konkrete parametrisierte Typen können jedoch nicht überall dort verwendet werden, wo Rohtypen oder nicht-generische Typen verwendet werden können. In den folgenden Fällen dürfen konkrete parametrisierte Typen nicht benutzt werden:

- *Als Komponententyp bei der Erzeugung einer Reihung:* Ein parametrisierter Typ hat keinen Laufzeittyp-Repräsentanten. Der Komponententyp einer Reihung muss jedoch einen Laufzeittyp-Repräsentanten haben. Reihungen mit parametrisierten Typen als Komponententyp wären unsicher und sind daher unzulässig. Beispiel:

  ```
  new Twins<String>[10] // unzulässig
  ```

- *In einem* `class`*-Literal oder einem* `instanceof`*-Ausdruck:* Nur Typen, die zur Laufzeit durch einen entsprechenden Laufzeittyp repräsentiert werden, können in einem `class`-Literal oder einem `instanceof`-Ausdruck benutzt werden. Beispiele:

  ```
  Twins<String>.class           // unzulässig
  var instanceof Twins<String>  // unzulässig
  ```

- *Vereinbarung einer Ausnahmeklasse:* Die JVM kennt keine generischen Typen und kann daher verschiedene Instanzen eines generischen Typen nicht unterscheiden. Aus diesem Grund dürfen generische Typen nicht direkt oder indirekt `java.lang.Throwable` erweitern. Beispiel:

  ```
  class MyException<T> extends Throwable {} // unzulässig
  ```

In allen anderen Fällen können konkrete parametrisierte Typen wie nicht-generische Typen benutzt werden. Das folgende Beispiel zeigt eine konkrete parametrisierte Instanz der Klasse `Twins` und deren Benutzung:

```
1 Twins<String> strTwins = new Twins<String>();
2
3 strTwins.setFirst(new Integer(10)); // unzulässig > Übersetzungsfehler
4 strTwins.setSecond("SomeClass");
5
6 String first = strTwins.getFirst();
```

In Zeile 1 wird die Klasse `Twins` mit dem aktuellen Typparameter `String` instanziiert. Infolge ist der formale Typparameter für diese Instanz durch `String` belegt. Die Methoden `setFirst(...)` und `setSecond(...)` dieser Instanz akzeptieren nur Objekte vom Typ `String` (oder Untertypen von `String`). Der Versuch, ein Objekt mit einem anderen Typ zu übergeben, führt zu einem Übersetzungsfehler. Die Methode `getFirst()` liefert für diese Instanz ein Objekt vom Typ `String`. Aus diesem Grund ist in Zeile 6 keine Typumwandlung nötig.

3 Modellimplementierung I

Das vorherige Kapitel befasste sich mit den Grundlagen generischer Typen. Bevor es jedoch um die Feinheiten geht, ist es sinnvoll, zunächst die Grundlagen der Modellimplementierung zu verstehen. Mit Modellimplementierung ist die Standard-Implementierung von Sun Microsystems gemeint. Sie legt fest, wie generische Einheiten in die bestehende Java-Welt eingebunden werden. Die Modellimplementierung gibt viel Aufschluss darüber, was mit generischen Einheiten in Java möglich ist und welchen Grenzen sie unterliegen. Bei der Entwicklung standen folgende Anforderungen im Vordergrund[1]:

- *Abwärtskompatibilität zu früheren Java-Versionen:* Älterer Bytecode muss auf dem neuen System lauffähig sein. Die Abwärtskompatibilität bezieht sich nicht nur auf den Bytecode, sondern auch auf die Einheiten der Standardbibliothek, die generisch re-implementiert werden. Desweiteren muss älterer Quellcode auf dem neuen System kompiliert werden können, ohne Änderungen durchführen zu müssen.

- *Keine wesentlichen Performance- und Ressourceneinbußen:* Die Verwendung von generischen Einheiten darf zu keinen wesentlichen Performance- oder Ressourceneinbußen gegenüber der Verwendung nicht-generischer Einheiten führen.

- *Migration bestehender APIs:* Eine generische Re-Implementierung bestehender APIs (speziell des Java Collection API) muss ohne große Umstände möglich sein.

Neben diesen Anforderungen wurden Ziele definiert, die durch die Einführung generischer Einheiten erreicht werden sollten:

- *Ausweitung statischer Typprüfungen:* Die Benutzung generischer Einheiten soll es dem Übersetzer ermöglichen, zusätzliche statische Typprüfungen durchzuführen. Diese zusätzlichen statischen Typprüfungen mindern die Gefahr von Typfehlern zur Laufzeit.

- *Eliminierung überflüssiger Typumwandlungen:* Durch die Benutzung generischer Einheiten werden einige Typumwandlungen überflüssig. Diese überflüssigen Typumwandlungen müssen vom Programmierer im Quellcode nicht explizit angegeben werden.

Diese Anforderungen und Ziele hatten einen wesentlichen Anteil an der Entwicklung der Modellimplementierung. Eine der wichtigsten Anforderungen war es, die Abwärtskompatibilität zu früheren Java-Versionen zu gewährleisten. Generischer Code sollte problemlos mit nicht-generischem Code vermischt werden können. Um diese Anforderungen realisieren zu können, müssen bei der Übersetzung alle generischen

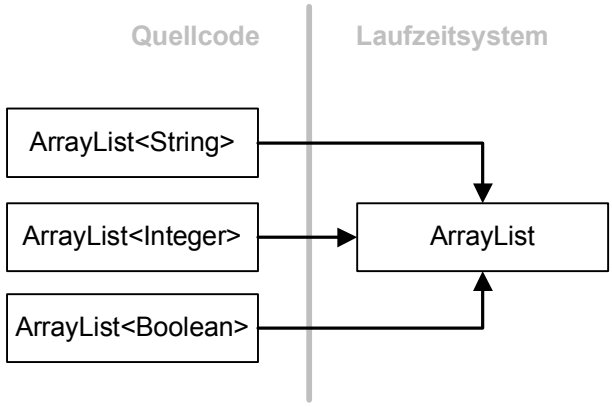


Abbildung 2 Abbildung verschiedener Instanziierungen eines generischen Typen auf einen Laufzeittyp

[1] Vgl. http://www.jcp.org/en/jsr/detail?id=14

Informationen aus dem Quellcode entfernt werden. Ähnliche Vorgehensweisen lassen sich in anderen Programmiersprachen finden. Die Template-Engine der Programmiersprache C++ erzeugt für verschiedene Instanzen jeweils eine Kopie des Template-Codes mit den entsprechenden aktuellen Typparametern. Bei intensiver Nutzung generischer Einheiten kann das jedoch zu einem hohen Ressourcenverbrauch führen. In Java hat man sich für die Typentfernung entschieden. Verschiedene Instanzen eines generischen Typen werden so auf einen nicht-generischen Laufzeittyp abgebildet (siehe Abbildung 2). Das Laufzeitsystem kennt somit keine generischen Einheiten.

3.1 Typentfernung

Bei der Typentfernung (engl. type erasure) werden nahezu alle generischen Informationen aus dem Code entfernt. Dabei werden Variablentypen und Methodensignaturen verändert. Das führt dazu, dass an einigen Stellen verengende Typumwandlungen nötig sind. Diese werden vom Übersetzer automatisch eingefügt. Das Ergebnis ist ein nicht-generischer Bytecode, der nur einige wenige generische Informationen enthält. Diese verbliebenen Informationen werden von der JVM als Kommentar ignoriert und werden lediglich vom Übersetzer für verschiedene Prüfungen[1] verwendet. Das folgende Beispiel zeigt die Auswirkungen der Typentfernung auf den Code.

Hinweis Mit der Anwendung JavaTEV kann man die Auswirkung der Typentfernung auf den Quellcode interaktiv nachvollziehen.

Ausschnitt der generischen Klasse `Twins` vor der Typentfernung:

```
1  public class Twins<E> {
2
3      private E first;
4
5      public Twins() {}
6
7      public E getFirst() { return this.first; }
8
9      public void setFirst(E first) { this.first = first; }
10
11     // ...
12 }
```

Die Klasse `Twins` hat einen formalen Typparameter (`E`). Bei der Typentfernung wird seine Vereinbarung entfernt. Danach wird `E` an allen Stellen durch `Object` ersetzt. Infolge wird der Variablentyp in Zeile 3 und die Methodensignaturen in den Zeilen 7 und 9 verändert. Die Methode `getFirst()` beispielsweise liefert vor der Typentfernung ein Objekt vom Typ `E`. Nach der Typentfernung liefert `getFirst()` ein Objekt vom Typ `Object`. Die Klasse `Twins` sieht nach der Typentfernung wie folgt aus:

[1] In größeren Projekten verwendet man oft Fremdbibliotheken. Leider stehen einige Fremdbibliotheken nur als kompilierter Bytecode (zum Beispiel als jar-Datei) zur Verfügung. Der Übersetzer prüft in diesem Fall mit Hilfe der verbliebenen generischen Informationen, ob die Verwendung einer generischen Einheit der Bibliothek zulässig ist.

```
 1 public class Twins {
 2
 3    private Object first;
 4
 5    public Twins() {}
 6
 7    public Object getFirst() { return this.first; }
 8
 9    public void setFirst(Object first) { this.first = first; }
10
11    // ...
12 }
```

Die Typentfernung hat nicht nur Auswirkungen auf den Code einer generischen Einheit selbst, sondern auch auf den benutzenden Code. Der folgende Quellcodeausschnitt zeigt zwei verschiedene generische Instanzen der Klasse `Twins` und die Verwendung dieser Instanzen:

```
1 Twins<String>  strTwins = new Twins<String>();
2 Twins<Integer> intTwins = new Twins<Integer>();
3
4 strTwins.setFirst("eine Zeichenkette");
5 String firstStr = strTwins.getFirst();
6
7 intTwins.setFirst(new Integer(10));
8 Integer firstInt = intTwins.getFirst();
```

In den Zeilen 1 und 2 wird die Klasse `Twins` mit verschiedenen aktuellen Typparametern instanziiert. Bei der Typentfernung werden alle aktuellen Typparameter entfernt und danach nötige Typumwandlungen eingefügt. Die beiden `Twins`-Instanziierungen werden auf denselben Laufzeittypen abgebildet. Nach der Typentfernung sieht der Code wie folgt aus:

```
1 Twins strTwins = new Twins();
2 Twins intTwins = new Twins();
3
4 strTwins.setFirst("eine Zeichenkette");
5 String firstStr = (String) strTwins.getFirst();
6
7 intTwins.setFirst(new Integer(10));
8 Integer firstInt = (Integer) intTwins.getFirst();
```

Durch die Veränderung der Methodensignaturen in der Klasse `Twins` sind nun in den Zeilen 5 und 8 verengende Typumwandlungen nötig. Diese werden vom Übersetzer automatisch eingefügt.

4 Generische Typen II

Nachdem es in dem Kapitel Generische Typen I um die Grundlagen von generischen Typen ging, geht es in diesem Kapitel um die Feinheiten im Umgang mit generischen Typen. Generische Typen sind parametrisierte Java-Typen. Es können jedoch nicht alle Java-Typen parametrisiert werden. Die folgenden Typen dürfen nicht generisch sein:

- *Aufzählungstypen (engl. enum types):* Ein Aufzählungstyp und seine Werte sind statisch. Formale Typparameter, die im Kopf eines Typen vereinbart wurden, können nur im nicht-statischen Kontext benutzt werden. Von einem Aufzählungstyp kann jedoch kein Objekt erzeugt werden. Innerhalb eines Aufzählungstypen können jedoch generische Einheiten (innere Klassen, Methoden) vereinbart werden.

- *Anonyme innere Klassen:* Eine anonyme Klasse hat keinen Namen. Bei einer parametrisierten Instanziierung wird jedoch der Typname benötigt. Generische anonyme innere Klassen könnten also nie parametrisiert instanziiert werden.

- *Ausnahmeklassen (engl. exception classes):* Eine generische Klasse darf die Klasse `java.lang.Throwable` nicht direkt oder indirekt erweitern. Der Ausnahmebehandlungsmechanismus (engl. exception handling mechanism) ist ein Laufzeitmechanismus. Die JVM kennt jedoch keine generischen Typen und kann verschiedene Instanzen eines generischen Typen nicht unterscheiden. Aus diesem Grund wäre sie nicht in der Lage, verschiedene Instanzen einer Ausnahmeklasse innerhalb der `catch`-Klausel zu unterscheiden.

Alle anderen Typen, wie normale Klassen, (geschachtelte) Schnittstellen, geschachtelte (statische) Klassen, innere (nicht statische) Klassen und lokale Klassen können generisch sein, das heißt parametrisiert werden. Wenn in Java von einem generischen Typ gesprochen wird, ist folgendes gemeint:

Definition Ein generischer Typ ist in Java entweder eine normale Klasse (die nicht von `Throwable` erbt), eine (geschachtelte) Schnittstelle, eine geschachtelte (statische) Klasse, eine innere (nicht statische) Klasse oder eine lokale Klasse mit formalen Typparametern.

4.1 Vereinbaren

Dieses Kapitel beschreibt, wie generische Schnittstellen implementiert und generische Klassen erweitert werden.

4.1.1 Eine generische Schnittstelle implementieren

Die Implementierung einer generischen Schnittstelle unterscheidet sich von der Implementierung einer nicht-generischen Schnittstelle. Aufgrund der formalen Typparameter kann eine generische Schnittstelle (je nach Anwendungsfall) verschieden implementiert werden. Ausschlaggebend ist dabei die Belegung der formalen Typparameter. Als Beispielschnittstelle dient die generische Schnittstelle `Comparable`:

```
1 public interface Comparable<T> {
2     public int compareTo(T other);
3 }
```

`Comparable` hat einen formalen Typparameter. Eine generische Schnittstelle kann jedoch auch mehrere formale Typparameter haben. Die folgenden Beispiele beschränken sich auf einen formalen Typparameter. Für Schnittstellen mit mehreren formalen Typparametern gelten die gleichen Regeln. Im folgenden Beispiel wird der Rohtyp von `Comparable` implementiert:

```
1 public class Twins<E> implements Comparable {
2
3     // ...
4
5     public int compareTo(Object other) {
6           // Vergleich
7     }
8 }
```

Bei einer Rohtyp-Instanz sind alle formalen Typparameter implizit mit `Object` belegt. Der formale Parameter von `compareTo(T)` muss deshalb durch `Object` ersetzt werden. Welche Folgen die Verwendung von Rohtypen hat, wurde bereits im Kapitel 2.2.1 beschrieben. Die dort beschriebenen Folgen gelten auch für die Implementierung einer generischen Schnittstelle. Grundsätzlich sollte, wie im folgenden Beispiel, immer eine parametrisierte Instanz implementiert werden:

```
1 public class Twins<E> implements Comparable<Twins<E>> {
2
3     // ...
4
5     public int compareTo(Twins<E> other) {
6           // Vergleich
7     }
8 }
```

In diesem Beispiel wurde eine parametrisierte Instanz der Schnittstelle `Comparable` implementiert. Der formale Typparameter von `Comparable` wurde dabei mit `Twins<E>` belegt. Daraus folgt, dass der formale Parameter von `compareTo(T)` durch `Twins<E>` ersetzt werden muss. Im Gegensatz zum ersten Beispiel bleibt hier die statische Typsicherheit erhalten. Man hätte den formalen Typparameter von `Comparable` auch mit dem Rohtypen von `Twins` belegen können. Dadurch hätte man jedoch auch Objekte vom Typ `Twins<String>` mit Objekten vom Typ `Twins<Integer>` vergleichen können, was man in der Regel jedoch ausschließen möchte. Im folgenden Beispiel wird eine weitere parametrisierte Instanz von `Comparable` implementiert:

```
1 public class Twins<E> implements Comparable<E> {
2
3     // ...
4
5     public int compareTo(E other) {
6           // Vergleich
7     }
8 }
```

Im Unterschied zum vorherigen Beispiel wird der formale Typparameter von `Comparable` durch den formalen Typparameter von `Twins` belegt. Als Folge muss der formale Parameter von `compareTo(T)` durch `E` ersetzt werden. Zugegeben, dieses Beispiel ist nicht besonders sinnvoll. Der

formale Typparameter von `Comparable` sollte immer mit dem Typ der implementierten Klasse belegt werden. Das Beispiel dient jedoch nur der Demonstration der Möglichkeiten.

Im Kapitel 4.2.1 wird die Möglichkeit beschrieben, generische Typen mit Jokerzeichen zu instanziieren. Ein Jokerzeichen steht nicht für einen Typ, sondern für eine Typfamilie. Eine solche Instanz nennt man parametrisierter Jokertyp. Bei der Erweiterung von Klassen (**`extends`**) oder der Implementierung von Schnittstellen (**`implements`**) dürfen Jokerzeichen jedoch nicht benutzt werden. Beispiele:

```
1 public class Twins<T> implements Comparable<?> {} // unzulässig
2 public class Twins<T> implements Comparable<? extends Number> {} // unzl.
3 public class Twins<T> implements Comparable<? super Integer> {}  // unzl.
```

4.1.2 Eine generische Klasse erweitern

Beim Erweitern einer generischen Klasse gelten die gleichen Regeln wie beim Implementieren einer generischen Schnittstelle. Die zulässigen Instanzen entsprechen denen aus Kapitel 4.1.1. Aus diesem Grund fallen die Beispiele an dieser Stelle weniger ausführlich aus. Zulässig sind auch hier Rohtypen und parametrisierte Typen (mit Ausnahme von parametrisierten Jokertypen). Der folgende Quellcodeausschnitt zeigt zulässige und unzulässige Erweiterungen:

```
1 public class SortedBox<E> extends Box {}                    // zulässig
2 public class SortedBox<E> extends Box<SortedBox<E>> {}      // zulässig
3 public class SortedBox<E> extends Box<E> {}                 // zulässig
4
5 public class SortedBox<E> extends Box<?> {}                 // unzulässig
6 public class SortedBox<E> extends Box<? extends Number> {} // unzulässig
7 public class SortedBox<E> extends Box<? super Integer> {}   // unzulässig
```

Sofern die erweiternde Klasse nicht abstrakt ist, müssen die abstrakten Methoden der Oberklassen entsprechend den aktuellen Typparametern implementiert werden. Beispiel:

```
1 public abstract class Box<E> {
2
3     public abstract E getFirst();
4
5     public E get(int i) { /* ... */ }
6
7     // ...
8 }
```

```
1 public class SortedBox<K> extends Box<Number> {
2
3     public Number getFirst() { /* ... */ }
4
5     // ...
6 }
```

Seit Java 1.5 dürfen überschriebene Methoden einen anderen Rückgabetyp haben als die Originalmethode. Das ist jedoch nur dann zulässig, wenn der Rückgabetyp in der überschriebenen Methode ein Untertyp des Rückgabetypen der Originalmethode ist. Für die in der Klasse `SortedBox` überschriebene Methode `getFirst()` wären also auch die Rückgabetypen `Integer`, `Long` oder ein anderer Untertyp von `Number` zulässig.

In einigen Fällen braucht man das Schlüsselwort **super**, um explizit auf Elemente des Obertypen zugreifen zu können. Erweitert man eine generische Klasse referenziert **super** auf die entsprechende (parametrisierte) Instanz. Beispiel:

```
1 public class SortedBox<K> extends Box<Number> {
2
3    public void someMethod() {
4          Number element = super.get(0);
5
6          // ...
7    }
8
9    // ...
10 }
```

In diesem Beispiel wurde die Instanz `Box<Number>` erweitert. Die Methode `Box.get(int)` gibt deshalb ein Objekt vom Typ `Number` zurück.

4.2 Benutzen

Im Kapitel 2.2 wurden bereits zwei verschiedene Arten generische Einheiten zu instanziieren vorgestellt. Dieses Kapitel beschäftigt sich mit einer weiteren Art.

Hinweis Die Tabelle 1 im Kapitel 11 zeigt die Unterschiede der verschiedenen Instanzen eines generischen Typen im Bezug auf ihre Verwendbarkeit.

4.2.1 Instanziierung mit Jokerzeichen

Neben der Rohtyp-Instanziierung und der Instanziierung mit konkreten Typen gibt es die Möglichkeit, generische Typen mit Jokerzeichen zu instanziieren. Beispiele:

```
1 Twins<?>
2 Twins<? extends Number>
3 Twins<? super Integer>
4 Pair<?, ?>
5 Pair<? extends Number, ?>
6 Pair<String, ?>
```

Eine solche Instanz nennt man *parametrisierter Jokertyp* (engl. wildcard parameterized type). Von einem parametrisierten Jokertyp spricht man, wenn mindestens ein aktueller Typparameter ein Jokerzeichen ist. Das Jokerzeichen steht dabei nicht für einen konkreten Referenztyp, sondern für eine Typfamilie. Anders als Rohtypen oder konkrete parametrisierte Typen können parametrisierte Jokertypen nur als Typ einer Referenz benutzt werden, jedoch nicht innerhalb eines **new**-Ausdrucks. Insgesamt stehen drei Jokerzeichen zur Verfügung:

Das Jokerzeichen `?`

Das sogenannte ungebundene Jokerzeichen (engl. unbounded wildcard) steht für die Familie aller Referenztypen. Beispiele:

```
1 Twins<?>
2 List<?>
3 Pair<?, ?>
```

Sind alle aktuellen Typparameter ungebunden, spricht man von einem *ungebundenen parametrisierten Jokertyp* (engl. unbounded wildcard parameterized type). Ein solcher Typ ist immer dann hilfreich, wenn man alle parametrisierten Instanzen des entsprechenden generischen Typen zulassen möchte. Diese hohe Flexibilität hat jedoch auch eine Kehrseite. Der Zugriff auf einen ungebundenen parametrisierten Jokertypen ist sehr eingeschränkt. Methoden, die beispielsweise einen an das Jokerzeichen `?` gebundenen aktuellen Typparameter als Parameter erwarten, akzeptieren nur noch **`null`** für diesen Parameter. Das folgende Beispiel verdeutlicht dies:

```
1 Twins<?> myTwins = new Twins<Integer>();
2 myTwins.setFirst(new Integer(10));      // unzulässig
3 myTwins.setFirst(null);                 // zulässig
```

Als Parameter für `setFirst(T)` und `setSecond(T)` ist nur noch **`null`** zugelassen. Der Übersetzer kann an dieser Stelle nicht feststellen, dass `myTwins` ein Objekt vom Typ `Twins<Integer>` referenziert. Aus diesem Grund ist nur **`null`** als Parameter erlaubt. Andersherum geben Methoden, die einen an das Jokerzeichen `?` gebundenen aktuellen Typparameter als Rückgabetyp haben, ein Objekt vom Typ `Object` zurück. Beispiel:

```
1 Twins<?> twins = new Twins<Integer>();
2 Object first = twins.getFirst(); // liefert ein Objekt vom Typ Object
```

Das Jokerzeichen `? extends TYPE`

`? extends TYPE` steht für eine obere Bindung (engl. upper bound). Die Typfamilie umfasst alle Untertypen von `TYPE` inklusive `TYPE`. Beispiele:

```
1 Twins<? extends Number> // Number inklusive aller Untertypen
2 List<? extends Number>
3 Pair<String, ? extends Number>
```

Kompatible parametrisierte Typen müssen den entsprechenden aktuellen Typparameter mit `TYPE` oder mit einem Untertyp von `TYPE` belegen. Beispiele:

```
1 Twins<? extends Number>  twins1 = new Twins<Number>();
2 Twins<? extends Number>  twins2 = new Twins<Integer>();
3 Twins<? extends Number>  twins3 = new Twins<Long>();
4 Twins<? extends Integer> twins4 = new Twins<Number>(); // unzulässig
5
6 Pair<String, ? extends Number> pair1 = new Pair<String, Number>();
7 Pair<String, ? extends Number> pair2 = new Pair<String, Integer>();
```

Das Jokerzeichen `? super TYPE`

Das Jokerzeichen `? super TYPE` ermöglicht eine sogenannte untere Bindung (engl. lower bound). Die Typfamilie umfasst dabei alle Obertypen von `TYPE` inklusive `TYPE`. Beispiele:

```
1 Twins<? super Integer>
2 Pair<? super Integer, String>
3 Pair<Cloneable, ? super Integer>
4 Pair<? super Integer, ? super Boolean>
```

Kompatible parametrisierte Typen müssen den entsprechenden aktuellen Typparameter mit `TYPE` oder einem Obertyp von `TYPE` belegen. Beispiele:

```
1 Twins<? super Integer> twins1 = new Twins<Integer>();
2 Twins<? super Integer> twins2 = new Twins<Number>();
3 Twins<? super Integer> twins3 = new Twins<Object>();
4 Twins<? super Number>  twins3 = new Twins<Integer>(); // unzulässig
5
6 Pair<String, ? super Integer> pair1 = new Pair<String, Integer>();
7 Pair<String, ? super Integer> pair2 = new Pair<String, Number>();
```

4.2.1.1 Einschränkungen

Ist mindestens ein aktueller Typparameter ein Jokerzeichen, spricht man von einem *parametrisierten Jokertyp* (engl. wildcard parameterized type). Ähnlich wie konkrete parametrisierte Typen können auch parametrisierte Jokertypen nicht überall dort benutzt werden, wo Rohtypen oder nicht-generische Typen benutzt werden können. In den folgenden Fällen sind parametrisierte Jokertypen unzulässig:

- *In einem* **new**-*Ausdruck:* Bei der Erzeugung eines Objektes muss der Übersetzer den Typ aller aktuellen Typparameter kennen. Ein parametrisierter Jokertyp steht jedoch nicht für einen Typ, sondern für eine Typfamilie. Konstrukte, wie die folgenden sind daher unzulässig:

  ```
  new Twins<?>()                // unzulässig
  new Twins<? extends Number>() // unzulässig
  new Twins<? super Integer>()  // unzulässig
  ```

- *Als Komponententyp bei der Erzeugung einer Reihung:* Gebundene parametrisierte Jokertypen (`? extends TYPE`, `? super TYPE`) sind als Komponententyp bei der Erzeugung einer Reihung nicht zugelassen. Ursache ist die zur Laufzeit stattfindende Prüfung beim Hinzufügen eines Elements zur Reihung (engl. array store check). Diese Prüfung kann nur für Typen durchgeführt werden, die einen Laufzeittyp-Repräsentanten haben. Gebundene parametrisierte Jokertypen haben jedoch keinen Laufzeittyp-Repräsentanten und wären daher unsicher. Ungebundene parametrisierte Jokertypen sind hingegen erlaubt. Sie haben einen Laufzeittyp-Repräsentanten und verhalten sich hier exakt wie Rohtypen. Beispiele:

  ```
  new Twins<?>[10]                  // zulässig
  new Twins<? extends Number>[10]   // unzulässig
  new Twins<? super Integer>[10]    // unzulässig
  ```

- *Als Obertyp:* Bei der Erweiterung einer generischen Klasse oder der Implementierung einer generischen Schnittstelle muss der konkrete Typ jedes formalen Typparameters feststehen. Da ein parametrisierter Jokertyp jedoch für eine Typfamilie steht, sind die folgenden Konstrukte unzulässig:

```
class MyClass extends Twins<?> {}                    // unzulässig
class MyClass extends Twins<? extends Number> {}     // unzulässig
class MyClass extends Twins<? super Integer> {}      // unzulässig
```

- *In einem* **instanceof**-*Ausdruck:* Nur Typen, die zur Laufzeit durch einen entsprechenden Laufzeittyp repräsentiert werden, können in einem **instanceof**-Ausdruck benutzt werden. Nur ungebundene parametrisierte Jokertypen und Rohtypen sind in einem **instanceof**-Ausdruck erlaubt, da diese beiden Typen während der Typenentfernung keine Typinformationen verlieren.

```
myTwins instanceof Twins<?>                 // zulässig
myTwins instanceof Twins<? extends Number> // unzulässig
myTwins instanceof Twins<? super Integer>  // unzulässig
```

- *In einem* **class**-*Literal:* Parametrisierte Jokertypen verlieren bei der Typenentfernung ihre aktuellen Typparameter. Daher teilen sich alle Instanzen eines generischen Typen denselben Laufzeittypen, welches der Rohtyp ist. Nur der Rohtyp existiert zur Laufzeit und kann in einem **class**-Literal benutzt werden.

```
Twins.class                                 // zulässig
Twins<?>.class                              // unzulässig
Twins<? extends Number>.class               // unzulässig
Twins<? super Integer>.class                // unzulässig
```

4.2.2 Kovarianz und generische Typen

In den bisherigen Beispielen waren sowohl die Referenz als auch das referenzierte Objekt von derselben Instanz (zum Beispiel: `Twins<String>`). Bei Reihungen weiß man, dass eine Reihungsreferenz eine Reihung referenzieren kann, deren Komponententyp ein Untertyp des eigenen Komponententypen ist. Eine Reihung mit dem Komponententyp `Number` ist Obertyp einer Reihung mit dem Komponententyp `Integer`. Diese Eigenschaft von Reihungen nennt man Kovarianz. Reihungen sind in Java kovariant. Beispiel:

```
1 Number[] numArr = new Integer[10];
```

Für generische Typen gilt dies jedoch nicht. Verschiedene Instanzen eines generischen Typen sind nicht kovariant zueinander. Auch nicht dann, wenn alle aktuellen Typparameter der einen Instanz Obertypen der aktuellen Typparameter der anderen Instanz sind. Beispiel:

```
1 Twins<Number> twins = new Twins<Integer>(); // unzulässig
```

Eine konkrete parametrisierte Referenz kann nur `null` sein, ein Objekt vom gleichen Typ oder ein Objekt vom Typ des entsprechenden Rohtypen referenzieren. Beispiele:

```
1 Twins<Number> twins1 = null;
2 Twins<Number> twins2 = new Twins<Number>();
3 Twins<Number> twins3 = new Twins();          // führt zu Warnungen
```

Die Zuweisung in Zeile 3 führt bei der Übersetzung zu einer Warnung. Der Übersetzer empfiehlt anstelle des Rohtypen, einen parametrisierten Typen zu benutzen.

Mit einem parametrisierten Typen anstelle eines konkreten parametrisierten Typen lässt sich die Liste gültiger Referenzen erweitern. Beispiele (alle Zuweisungen sind zulässig):

```
1 Twins<?> twins1 = new Twins<String>();
2 twins1 = new Twins<Integer>();
3
4 Twins<? extends Number> twins2 = new Twins<Integer>();
5 twins2 = new Twins<Double>();
6
7 Twins<? super Integer> twins3  = new Twins<Number>();
8 twins3 = new Twins<Object>();
```

Wichtig Generische Typen sind in Java immer invariant. Jokerzeichen sind oft nötig um eine Implementierung flexibler zu machen. An der Invarianz ändern sie jedoch nichts.

Unabhängig davon kann eine parametrisierte Referenz ein Objekt referenzieren, dessen Typ ein Untertyp des eigenen Typen ist. Dabei müssen jedoch die formalen Typparameter der beiden Typen identisch belegt sein. Beispiele:

```
1 List<Number> numList       = new ArrayList<Number>();
2 Collection<String> strColl = new LinkedList<String>();
3 Map<String, String> map    = new HashMap<String, String>();
```

4.2.3 Verschachtelungen

Parametrisierte Typen können beliebig verschachtelt werden, das heißt ein parametrisierter Typ kann aktueller Typparameter eines anderen parametrisierten Typen sein. Beispiel:

```
1 Twins<Collection<String>>
```

Der Typ `Collection<String>` ist in diesem Beispiel aktueller Typparameter des Typen `Twins<T>`. In Verbindung mit Jokerzeichen kann dies schnell zu Verwirrungen führen. Beispiel[1]:

```
1 Collection<Pair<String, Integer>> c1 =
2     new ArrayList<Pair<String, Integer>>();
3
4 Collection<Pair<String, ?>> c3 = c1; // unzulässig
```

Die Zuweisung in Zeile 4 ist unzulässig. Das verwirrt auf den ersten Blick, denn die folgende Zuweisung ist zulässig:

```
1 Pair<String, ?> pair = new Pair<String, Integer>();
```

Trotzdem führt die Zuweisung in Zeile 4 zu einem Übersetzungsfehler. Das liegt daran, dass der parametrisierte Typ `Collection<Pair<String, Integer>>` eine homogene Sammlung von Paaren bestehend aus `String` und `Integer` ist. Der parametrisierte Typ `Collection<Pair<String, ?>>` ist hingegen eine heterogene Sammlung von Paaren bestehend aus `String` und einem unbekannten Typ. Eine solche heterogene Sammlung kann zum Beispiel auch Objekte vom Typ `Pair<String, Date>` enthalten. Die Zuweisung in Zeile 4 würde somit die Typsicherheit außer Kraft setzen und ist daher unzulässig. Die Lösung für dieses Problem stellt eine Jokerzeichenbindung dar. Beispiel:

```
5 Collection<? extends Pair<String, ?>>  c4 = c1; // zulässig
```

Das Jokerzeichen `? extends Pair<String, ?>` steht für alle Untertypen von `Pair<String, ?>`. Zu dieser Familie gehören auch Typen, wie:

```
1 Pair<String, Long>
2 Pair<String, ? extends Number>
3 Pair<String, ?>
```

[1] Vergleiche http://www.angelikalanger.com/GenericsFAQ/FAQSections/TypeArguments.html „What do multi-level wildcards mean?"

5 Generische Methoden

Neben Typen können auch Methoden parametrisiert werden. Dieses Kapitel beschäftigt sich mit der Vereinbarung und Benutzung generischer Methoden. Unter einer generischen Methode versteht man eine Methode, die ihre eigenen formalen Typparameter vereinbart. Methoden, die formale Typparameter des umgebenen Typen benutzen und keine eigenen formalen Typparameter vereinbaren, sind damit nicht gemeint. Die Methoden `getFirst()` und `getSecond()` der generischen Klasse `Twins` sind keine generischen Methoden, obwohl sie mit formalen Typparametern arbeiten. Eine generische Methode darf überall dort vereinbart werden, wo auch nicht-generische Methoden vereinbart werden dürfen. Sie kann, muss jedoch nicht, innerhalb eines generischen Typen vereinbart werden.

Definition Eine generische Methode ist eine Methode mit eigenen formalen Typparametern.

Die Java Standard-Bibliothek enthält viele generische Methoden. Mit generischen Methoden lassen sich typunabhängige und zugleich statisch typsichere Algorithmen implementieren. Ein Beispiel dafür ist die Klasse `java.util.Collections`. Sie stellt verschiedene generische Methoden bereit, die den Umgang mit Sammlungen erleichtern. Das Paket `java.util` enthält sowohl die Schnittstelle `Collection` als auch die Klasse `Collections`. Beide Typen werden im Folgenden verwendet, sollten jedoch nicht verwechselt werden.

5.1 Vereinbaren

Die formalen Typparameter einer generischen Methode werden nicht, wie bei generischen Typen, hinter dem Namen vereinbart, sondern vor dem Rückgabetyp. Auch hier gilt: Innerhalb der spitzen Klammern können beliebig viele formale Typparameter vereinbart werden. Mehrere formale Typparameter werden durch Kommata getrennt. Der folgende Codeabschnitt zeigt die generische Methode *`emptyList`*`()` der Klasse `java.util.Collections`, die eine Liste mit dem Elementtyp `T` zurück gibt:

```
1 public static final <T> List<T> emptyList() { /* ... */ }
```

In diesem Beispiel wurde der formale Typparameter `T` vor dem Rückgabetyp `List<T>` vereinbart. Nach seiner Vereinbarung kann ein formaler Typparameter in der gesamten Methode, inklusive Methodenkopf, benutzt werden. Eine nicht-statische generische Methode hat zudem Zugriff auf die formalen Typparameter ihrer umgebenden Klasse.

5.2 Benutzen

Generische Methoden können wie nicht-generische Methoden aufgerufen werden. Die explizite Belegung der formalen Typparameter ist optional. Das heißt, die formalen Typparameter einer generischen Methode können, müssen jedoch nicht, beim Aufruf durch aktuelle Typparameter belegt werden. Das folgende Beispiel zeigt einen Aufruf der generischen Methode *`emptyList`*. Der formale Typparameter wurde explizit mit `String` belegt.

```
1 List<String> list = Collections.<String>emptyList();
```

`emptyList``()` liefert bei diesem Aufruf eine leere Liste mit dem Elementtyp `String`. Die aktuellen Typparameter werden in spitzen Klammern vor dem Methodennamen angegeben. Mehrere aktuelle

Typparameter werden auch hier durch Kommata voneinander getrennt. Bei expliziter Belegung müssen immer alle formalen Typparameter belegt werden. Zudem muss die Methode qualifiziert aufgerufen werden, das heißt inklusive umgebendem Typen. Das ist auch nötig, wenn sich der Aufruf im selben Typ der Methodenvereinbarung befindet oder wenn die Methode statisch importiert wurde. Beispiel:

```
1 Collections.<String>emptyList();  // zulässig
2 <String>emptyList();              // unzulässig
```

Gültige aktuelle Typparameter sind Referenztypen und formale Typparameter. Primitive Typen und Jokertypen, wie `?` oder `? extends Number` sind unzulässig. Beispiele:

```
1 Collections.<String>emptyList();              // zulässig
2 Collections.<T>emptyList();                   // zulässig
3 Collections.<int>emptyList();                 // unzulässig
4 Collections.<? extends Number>emptyList();    // unzulässig
```

Der Übersetzer ist jedoch auch in der Lage, die aktuellen Typparameter aus dem Aufrufkontext herzuleiten, sodass die formalen Typparameter nicht explizit belegt werden müssen. Diese Technik nennt man Argumentenrückschluss (engl. argument type inference). Das folgende Beispiel zeigt einen Aufruf von `emptyList()` ohne explizite Belegung der aktuellen Typparameter:

```
1 List<String> list = Collections.emptyList();
```

`emptyList()` liefert in diesem Beispiel eine leere Liste mit dem Elementtyp `String`. Der Übersetzer leitet den aktuellen Typparameter aus dem Aufrufkontext her. `emptyList()` gibt eine Liste vom Typ `List<T>` zurück. Der Rückgabewert wird in diesem Beispiel einer Referenz vom Typ `List<String>` zugewiesen. Der Übersetzer schließt daraus, dass der aktuelle Typparameter mit `String` belegt werden muss. Werden die formalen Typparameter nicht explizit belegt, muss die Methode auch nicht qualifiziert aufgerufen werden. Beispiele:

```
1 Collections.emptyList();    // zulässig
2 emptyList();                // zulässig
```

In dem gerade gezeigten Beispiel werden die aktuellen Typparameter anhand des Rückgabetypen und dem Typ der Referenz, der der Rückgabewert zugewiesen wird, hergeleitet. Einige Methoden haben jedoch keinen Rückgabetyp. Das folgende Beispiel zeigt eine solche Methode:

```
 1 public static <T> void addAll(List<T> list, T...ts) {
 2    for (T current : ts) list.add(current);
 3 }

10 List<String> list = new ArrayList<String>();
11 addAll(list, "eins", "zwei", "drei");
```

In diesem Fall leitet der Übersetzer den aktuellen Typparameter aus den Parametern her, mit der `addAll(...)` aufgerufen wird. Für den Aufruf in Zeile 11 ist der aktuelle Typparameter `String`.

Der Argumentenrückschluss kann auch mit parametrisierten Jokertypen umgehen. Das folgende Beispiel zeigt ein paar Aufrufe der generischen Methode `emptyList()` der Klasse `Collections`:

```
1 List<? extends Number> list1 = Collections.emptyList();
2 List<? super Integer>  list2 = Collections.emptyList();
3 List<? extends Thread> list3 = Collections.emptyList();
4 List<?>                list4 = Collections.emptyList();
```

Einige Übersetzer, wie die erste Version des Sun 1.5 Übersetzers[1], geben bei der Verwendung von parametrisierten Jokertypen aus, mit welchem Typ ein formaler Typparameter belegt wurde. Die Ausgabe ist jedoch (besonders bei größeren Klassen) nicht besonders aussagekräftig. Die Ausgabe des gerade gezeigten Beispiels ist die folgende:

```
hibounds=java.lang.Object
hibounds=java.lang.Object
hibounds=java.lang.Object
hibounds=java.lang.Object
```

Der formale Typparameter wurde in allen Fällen durch `Object` belegt. Durch welchen Typ ein formaler Typparameter belegt wird, hängt von seiner Bindung ab. Im Unterschied zu generischen Typen können generische Methoden nicht mit Jokerzeichen instanziiert werden. Obwohl der Argumentenrückschluss auch mit Jokerzeichen umgehen kann, sind Jokerzeichen als aktueller Typparameter unzulässig. Beispiele:

```
1 Collections.<?>emptyList();                  // unzulässig
2 Collections.<? extends Number>emptyList();   // unzulässig
3 Collections.<? super Integer>emptyList();    // unzulässig
```

[1] JDK 1.5.0; verfügbar unter http://java.sun.com/products/archive/j2se/5.0/index.html

6 Formale Typparameter

In diesem Kapitel geht es um formale Typparameter. Insbesondere werden die Unterschiede zu Java-Typen beschrieben. Formale Typparameter sind das wesentliche Merkmal generischer Einheiten. Eine generische Einheit hat mindestens einen formalen Typparameter. Mit ihrer Hilfe lassen sich typunabhängige und zugleich statisch typsichere Einheiten implementieren.

Definition Ein formaler Typparameter ist ein Platzhalter für einen beliebigen Referenztyp.

Formale Typparameter sind Platzhalter und sollten nicht mit Java-Typen, wie Klassen oder Schnittstellen, verwechselt werden. In vielen Fällen können formale Typparameter zwar wie Java-Typen benutzt werden, in einigen Fällen (auf Grund ihrer hohen Abstraktion) jedoch nicht.

6.1 Sichtbarkeit

Die Sichtbarkeit eines formalen Typparameters ist vom Kontext abhängig, in dem er vereinbart wurde. Hier werden generische Klasse, generische Schnittstelle und generische Methode unterschieden:

- *Klasse:* Ein im Klassenkopf vereinbarter formaler Typparameter ist im gesamten nicht-statischen Kontext der Klasse sichtbar. Zu diesem Kontext gehören auch innere nicht-statische Klassen. Innere Schnittstellen und Aufzählungstypen gehören jedoch nicht zu diesem Kontext.

- *Schnittstelle:* Ein im Schnittstellenkopf vereinbarter formaler Typparameter kann nur in Methodenvereinbarungen benutzt werden. In Schnittstellen vereinbarte Variablen und geschachtelte Typen sind immer statisch. Die formalen Typparameter der umgebenen Schnittstelle sind dort daher nicht sichtbar.

- *Generische Methode:* Formale Typparameter, die im Kopf einer generischen Methode vereinbart wurden, sind innerhalb der gesamten Methode sichtbar. Nicht-statische generische Methoden haben zudem Zugriff auf die formalen Typparameter ihrer umgebenen Klasse.

6.2 Gemeinsamkeiten und Unterschiede zu Java-Typen

Formale Typparameter sind zwar Platzhalter für Java-Typen, dürfen jedoch nicht überall wie Java-Typen benutzt werden. In den folgenden Fällen dürfen formale Typparameter wie oder anstelle von Java-Typen benutzt werden:

- Parametertyp oder Rückgabetyp von Methoden, Parametertyp von Konstruktoren

- Typ einer Variable oder einer lokalen Referenz
  Beispiel:
  ```
  public class Twins<E> {
        private E first;
  }
  ```

- Zieltyp einer Typumwandlung
  Beispiel:
  ```
  T myType = (T) object;
  ```

- Expliziter aktueller Typparameter beim Aufruf einer generischen Methode
  Beispiel:
  ```
  java.util.Collections.<T>emptyList()
  ```

- Aktueller Typparameter anderer parametrisierter Typen
  Beispiel:
  ```
  class Twins<T> {
        private List<T> list;
  }
  ```

  Hier wird der formale Typparameter `T` als aktueller Typparameter von `List` benutzt.

- **`throws`**-Klausel
  Beispiel:
  ```
  public interface Connection<E extends Exception> {
        public void init() throws E;
  }
  ```

In vielen anderen Fällen dürfen formale Typparameter nicht anstelle von Java-Typen benutzt werden. Einer dieser Fälle ist der **`new`**-Ausdruck. Jede Objekterzeugung ist an einen Konstruktor gekoppelt. Um ein Objekt des Java-Typen zu erzeugen, für den der formale Typparameter steht, benötigt man Zugriff auf einen seiner Konstruktoren. Von einigen Java-Typen, wie Schnittstellen oder Aufzählungstypen kann jedoch kein Objekt erzeugt werden. Diese Typen besitzen auch keine Konstruktoren. Da ein formaler Typparameter für einen beliebigen Java-Typ steht, kann nicht sichergestellt werden, dass dieser Typ einen Konstruktor besitzt. Aus diesem Grund kann ein formaler Typparameter nicht in einem **`new`**-Ausdruck benutzt werden.

Ein weiterer Fall, in dem formale Typparameter unzulässig sind, ist als Komponententyp bei der Erzeugung einer Reihung. Formale Typparameter dürfen zwar als Komponententyp einer Reihungsreferenz, jedoch nicht als Komponententyp eines Reihungsobjektes verwendet werden. Beispiel:

```
1 T[] array1 = null;        // zulässig
2 T[] array2 = new T[10];   // unzulässig
```

Eine mögliche Ursache dafür könnte die Typentfernung sein. Formale Typparameter werden bei der Typentfernung durch ihre linke Bindung[1] ersetzt (oder durch `Object`, falls sie ungebunden sind). Wäre es möglich, eine Reihung zu erzeugen, deren Komponententyp ein parametrisierter Typ ist, hätte das folgende Auswirkungen.

Die Klasse `Twins` vor der Typentfernung:

```
 1 public class Twins<E> {
 2
 3    private E first;
 4    private E second;
 5
 6    public E[] toArray() {
 7          return new E[] { this.first, this.second }; // unzulässig
 8    }
 9    // ...
10 }
```

Die Erzeugung der Reihung in Zeile 7 ist unzulässig. Wäre sie jedoch möglich, sähe der Code nach der Typentfernung wie folgt aus:

```
 1 public class Twins {
 2
 3    private Object first;
 4    private Object second;
 5
 6    public Object[] toArray() {
 7          return new Object[] { this.first, this.second };
 8    }
 9    // ...
10 }
```

Die Methode `toArray()` liefert nun eine Reihung mit dem Komponententyp `Object`. Das ist jedoch nicht das, was gewollt war. Der benutzende Code macht das Problem deutlich:

```
10 Twins<String> twins = new Twins<String>();
11 String[] array = twins.toArray(); // incompatible Reihungen > Exception
```

Da die Klasse `Twins` mit `String` instanziiert wurde, sollte man annehmen, dass `toArray()` eine Reihung mit dem Komponententyp `String` liefert. Das ist jedoch nicht der Fall. Die zurückgelieferte Reihung enthält zwar Objekte vom Typ `String`, der Komponententyp ist jedoch `Object`. Die Zuweisung in Zeile 11 würde deshalb mit einer `ClassCastException` fehlschlagen. Für einen

[1] Mehr zu Bindungen im Kapitel 0

solchen Fall hätte man zwar eine Hilfsroutine integrieren können, diese würde aber zu Performanceeinbußen gegenüber der Benutzung nicht-generischer Typen führen.

Eine weitere Einschränkung von formalen Typparametern gegenüber Java-Typen ist die Benutzung im statischen Kontext. Formale Typparameter dürfen nicht im statischen Kontext benutzt werden. Diese Einschränkung bezieht sich jedoch nur auf formale Typparameter, die im Kopf eines Typen vereinbart wurden. Formale Typparameter, die im Kopf einer generischen Methode vereinbart wurden, sind selbstverständlich innerhalb der Methode sichtbar, auch wenn diese statisch ist.

Wie zuvor beschrieben, ist es möglich formale Typparameter in der **throws**-Klausel zu benutzen. In der **catch**-Klausel sind formale Typparameter jedoch unzulässig. Das folgende Beispiel zeigt warum:

```
1 <E extends Exception> void method() {
2     try { /* irgendetwas */ }
3     catch (E e) {}                // unzulässig
4     catch (IOException e) {}
5     catch (Exception e) {}
6 }
```

Die Benutzung des formalen Typparameters `E` in Zeile 3 ist unzulässig. Wäre sie zulässig, sähe der Code nach der Typentfernung wie folgt aus:

```
1 void method() {
2     try { /* irgendetwas */ }
3     catch(Exception e) {}
4     catch(IOException e) {}
5     catch(Exception e) {}
6 }
```

Der formale Typparameter `E` wurde bei der Typentfernung durch `Exception` ersetzt. Das führt dazu, dass die **catch**-Klausel in Zeile 3 alle Ausnahmen abfängt. Die beiden nachfolgenden **catch**-Klauseln wären wirkungslos.

Formale Typparameter dürfen auch nicht in einem **class**-Literal oder einem **instanceof**-Ausdruck benutzt werden. Der Grund dafür liegt wieder in der Typentfernung. Es dürfen ausschließlich Java-Typen in einem **class**-Literal oder einem **instanceof**-Ausdruck benutzt werden, die einen Laufzeittyp-Repräsentanten besitzen.

Weiter ist es unzulässig, einen formalen Typparameter als Obertyp eines anderen Typen zu benutzen. Bei der Erweiterung eines Typen muss dieser statisch feststehen. Zudem steht ein formaler Typparameter für einen beliebigen Java-Typ also auch für eine Schnittstelle oder einen Aufzählungstypen. Diese Typen können jedoch nicht erweitert werden. Beispiel:

```
1 class MyClass extends T {} // unzulässig
```

6.3 Gebundene formale Typparameter

Bei der Vereinbarung einer generischen Einheit ist der Typ, für den ein formaler Typparameter steht, unbekannt. Aus diesem Grund sind die Zugriffsmöglichkeiten auf einen formalen Typparameter sehr eingeschränkt. In manchen Fällen benötigt man jedoch mehr Informationen über den Typ, für den der formale Typparameter steht. Das folgende Beispiel zeigt einen solchen Fall. Die Methode `getSmaller()` der Klasse `Twins` gibt den nach der natürlichen Ordnung kleineren Zwilling zurück.

```
1 public class Twins<T> {
2     private T first;
3     private T second;
4
5     public T getSmaller() {
6           // ?!?
7     }
8     // ...
9 }
```

Die beiden beliebigen Typen `first` und `second` können nicht ohne weiteres geordnet werden. Für einen solchen Fall gibt es die Möglichkeit, den formalen Typparameter an einen Java-Typ zu binden. Die Bindung erfolgt über das Schlüsselwort **`extends`**. Beispiel:

```
1 public class Twins<T extends Comparable<T>> {}
```

Der formale Typparameter `T` wird in diesem Beispiel an die Schnittstelle `Comparable` gebunden. Ein gültiger aktueller Typparameter für `T` muss die Schnittstelle `Comparable` implementieren oder erweitern. Einen formalen Typparameter mit Bindung nennt man *gebundener formaler Typparameter* (engl. wildcard bounded type parameter). Obwohl der formale Typparameter in diesem Beispiel an eine Schnittstelle gebunden wurde, muss das Schlüsselwort **`extends`** benutzt werden. Dank der Bindung können die beiden Typen `first` und `second` über die Schnittstelle `Comparable` geordnet werden.

Ordnen von beliebigen, typgleichen Werten über `Comparable`:

```
1 public class Twins<T extends Comparable<T>> {
2     private T first;
3     private T second;
4
5     public T getSmaller() {
6         if (this.first.compareTo(this.second) < 0) return this.first;
7         else return this.second;
8     }
9     // ...
10 }
```

6.3.1 Zugriffsmöglichkeiten

Die Bindung schränkt den Satz möglicher aktuellen Typparameter ein, ermöglicht aber den Zugriff auf alle öffentlichen Elemente des beliebigen Typen, für den der formale Typparameter steht. Zu diesen öffentlichen Elementen gehören:

- (statische) Variablen
- (statische) Methoden
- (statische) geschachtelte Typen

Auf Konstruktoren (egal mit welcher Sichtbarkeit), Variablen, Methoden oder geschachtelten Typen mit eingeschränkter Sichtbarkeit (**private**, **default**, **protected**) erhält man keinen Zugriff. Der folgende Quellcode demonstriert die Zugriffsmöglichkeiten:

```
1 public class SomeClass {
2
3    public String publicField                = "SomeClass";
4    public static String PUBLIC_STATIC_FIELD = "SomeClass";
5
6    public String someMethod() { return "SomeClass"; }
7    public static String someStaticMethod() { return "SomeClass"; }
8
9    public class SomeNestedClass {}
10   public static class SomeNestedStaticClass {}
11 }
```

Die Klasse `Twins` wird im folgenden Beispiel um die Methode `accessDemo()` erweitert. Der formale Typparameter `E` wird an die Klasse `SomeClass` gebunden. Die Methode `accessDemo()` zeigt alle durch die Bindung möglichen Zugriffe.

```
1 public class Twins<E extends SomeClass> {
2
3    private E first;
4
5    // ...
6
7    private void accessDemo() {
8        // Variablen
9        String publicField       = this.first.publicField;
10       String publicStaticField = E.PUBLIC_STATIC_FIELD;
11
12       // Methoden
13       String methodeReturn       = this.first.someMethod();
14       String staticMethodeReturn = E.someStaticMethod();
15
16       // geschachtelte Typen
17       E.SomeNestedClass type;
18       E.SomeNestedStaticClass staticType;
19   }
20
21   // ...
22 }
```

Ersetzt (überdeckt oder überschreibt) ein Untertyp Attribute, statische Methoden oder geschachtelte Typen seiner Obertypen, hat dies keine Auswirkung auf die Erreichbarkeit. Nur bei nicht-statischen Methoden bezieht sich die Erreichbarkeit auf die im Untertyp vereinbarte Methode. Das folgende Beispiel zeigt, welche Auswirkungen das Ersetzen von Elementen auf die Erreichbarkeit hat. Die Klasse `SomeSubClass` erweitert die Klasse `SomeClass` und ersetzt alle Elemente der Oberklasse:

```
1 public class SomeSubClass extends SomeClass {
2
3    public String publicField = "SomeSubClass";
4    public static String PUBLIC_STATIC_FIELD = "SomeSubClass";
5
6    public String someMethod() { return "SomeSubClass";  }
7    public static String someStaticMethod() { return "SomeSubClass";  }
8
9    public class SomeNestedClass {}
10   public static class SomeNestedStaticClass {}
11 }
```

Wird die Klasse `Twins` mit `SomeSubClass` als aktuellem Typparameter instanziiert (Beispiel: `Twins<SomeSubClass>`), wird einzig die nicht-statische Methode `someMethod()` aus `SomeSubClass` aufgerufen. Alle anderen Zugriffe beziehen sich auf die Elemente der Oberklasse `SomeClass`.

6.3.2 Gültige Bindungstypen

Ein formaler Typparameter kann an eine Klasse oder einen Aufzählungstyp, und beliebig viele Schnittstellen gebunden werden. Verschiedene Bindungstypen werden durch `&` getrennt. Die Syntax ist wie folgt:

```
Syntax TYPE_PARAMETER extends (Klasse | Aufzählungstyp) [& Schnittstelle]*
       TYPE_PARAMETER extends [Schnittstelle] [& Schnittstelle]
```

Beispiele:

```
 1 public class Twins<T extends Number> {}
 2 public class Twins<T extends Serializable> {}      // Schnittstelle
 3 public class Twins<T extends Comparable<T>> {}     // gen. Schnittstelle
 4 public class Twins<T extends Thread.State> {}      // Aufzählungstyp
 5 public class Twins<T extends Map.Entry<?, ?>> {}   // innerer Typ
 6 public class Twins<T extends ArrayList> {}         // Rohtyp
 7 public class Twins<T extends ArrayList<String>> {}
 8 public class Twins<T extends ArrayList<? extends Number>> {}
 9 public class Twins<T extends ArrayList<? super Number>> {}
10 public class Twins<T extends Number & Comparable<T>> {}
```

Das Schlüsselwort **`extends`** ist fester Bestandteil der Bindung und muss auch benutzt werden, wenn ein formaler Typparameter ausschließlich an Schnittstellen gebunden wird. Primitive Typen und Reihungstypen sind als Bindungstypen unzulässig. Alle anderen Typen, wie:

- nicht-statische innere Typen
- Rohtypen
- parametrisierte Typen (konkrete, gebundene und ungebundene parametrisierte Typen)

dürfen als Bindungstyp benutzt werden. Ein Typ darf innerhalb einer Bindung maximal einmal auftreten. Das gilt auch für parametrisierte Typen. Es ist unzulässig, zwei verschiedene Instanzen eines generischen Typen innerhalb einer Bindung zu benutzen. Beispiel:

```
1 class Twins<T extends Comparable<T> & Comparable<String>> {}// unzulässig
```

Diese Einschränkung hängt mit der Typentfernung zusammen. Mehrere verschiedene Instanzen eines generischen Typen hätten nach der Typentfernung mehrere identische Methodensignaturen zur Folge. Die Bindung eines formalen Typparameters wird bei der Typentfernung berücksichtigt. Ein formaler Typparameter wird bei der Typentfernung durch seine linke Bindung (ersten Bindungstyp) ersetzt. Alle anderen Bindungstypen bleiben unberücksichtigt. Im folgenden Beispiel ist der linke Bindungstyp `Number`:

```
1 public class Twins<T extends Number & Comparable<T>> {
2     private T first;
3     // ...
4 }
```

`Twins` nach der Typentfernung:

```
1 public class Twins {
2     private Number first;
3     // ...
4 }
```

Neben Java-Typen können auch formale Typparameter als Bindung eines anderen formalen Typparameters benutzt werden.

Beispiel:

```
1 public class Pair<T, S extends T> {}
```

In der Klasse `Pair` werden zwei formale Typparameter vereinbart. Der zweite formale Typparameter `S` wird an den ersten formalen Typparameter `T` gebunden. Gültige aktuelle Typparameter für `S` müssen so vom selben Typ oder Untertyp des aktuellen Typparameters von `T` sein. Beispiele:

```
1 Pair<Number, Number>  pair1 = new Pair<Number, Number>();
2 Pair<Number, Integer> pair2 = new Pair<Number, Integer>();
3 Pair<Integer, Number> pair3 = new Pair<Integer, Number>(); // unzulässig
```

Bevor ein formaler Typparameter als Bindungstyp benutzt werden kann, muss er vereinbart sein. Das folgende Beispiel zeigt eine unzulässige Bindung. Der formale Typparameter `S` wird als Bindungstyp für `T` benutzt, ohne selber vorher vereinbart worden zu sein.

```
1 public class Pair<T extends S, S> {} // unzulässig
```

7 Aktuelle Typparameter

Dieses Kapitel befasst sich mit aktuellen Typparametern (engl. type arguments). Wie im Kapitel 6 beschrieben, hat eine generische Einheit mindestens einen formalen Typparameter. Bei der Instanziierung eines generischen Typen werden alle formalen Typparameter mit aktuellen Typparametern belegt. Ein formaler Typparameter ist also im Grunde ein Platzhalter, der für eine Menge von aktuellen Typparametern steht.

Definition Ein aktueller Typparameter ist ein Referenztyp oder ein Jokerzeichen, der bei der Instanziierung eines generischen Typen einen formalen Typparameter ersetzt.

Das folgende Beispiel zeigt eine parametrisierte Instanz der Klasse `Twins`:

```
1 public class Twins<E> {
2     // ...
3 }
4
5 public class Example {
6     public static void main(String[] args) {
7           Twins<String> twins = new Twins<String>();
8     }
9 }
```

Der formale Typparamater `E` wird in Zeile 7 mit dem aktuellen Typparameter `String` belegt. Eine solche Instanz nennt man *parametrisierter Typ* (engl. parameterized type). Entsprechendes gilt für generische Methoden, die explizit mit aktuellen Typparametern aufgerufen werden. Eine solche Methode nennt man *parametrisierte Methode* (engl. parameterized method).

7.1 Gültige Typen

Generische Typen können mit Referenztypen oder Jokerzeichen instanziiert werden. Generische Methoden können hingegen nur mit Referenztypen instanziiert werden. Primitive Typen sind unzulässig. Beispiele:

```
1 Twins<String>
2 Twins<Thread.State>
3 Twins<Comparable<String>>
4 Twins<Class<? extends Number>>
5 Twins<T>
6 Twins<int[]>
7 Twins<int> // unzulässig
```

Wie in den Beispielen zu sehen, sind auch geschachtelte Typen (siehe Zeile 2), parametrisierte Typen (siehe Zeile 3) und formale Typparameter (siehe Zeile 5) zulässig. Primitive Typen (siehe Zeile 7) sind dagegen unzulässig. Reihungen mit primitiven Komponenten (Zeile 6) sind dagegen zulässig. Wurde ein generischer Typ ausschließlich mit Java-Typen instanziiert (Zeile 1), spricht man von einem *konkreten parametrisierten Typen* (engl. concrete parameterized type). Entsprechendes gilt für generische Methoden:

```
1 Collections.<String>emptyList();
2 Collections.<Thread.State>emptyList();
3 Collections.<Comparable<String>>emptyList();
4 Collections.<Class<? extends Number>>emptyList();
5 Collections.<T>emptyList();
6 Collections.<int[]>emptyList();
7 Collections.<int>emptyList();     // unzulässig; kein Referenztyp
```

7.2 Jokerbindung

Generische Typen können nicht nur mit Referenztypen, sondern auch mit Jokerzeichen instanziiert werden. Jokerzeichen stehen im Gegensatz zu Referenztypen nicht für genau einen Typ, sondern für eine Typfamilie.

Definition Ein Jokerzeichen ist ein syntaktisches Konstrukt, das für eine Typfamilie steht.

Einen mit Jokerzeichen instanziierten generischen Typ nennt man *parametrisierter Jokertyp* (engl. wildcard parameterized type) Es werden drei Jokerzeichen unterschieden:

Das Jokerzeichen `?`

Das sogenannte ungebundene Jokerzeichen `?` steht für die Familie aller Referenztypen. Ungebunden nennt man es deshalb, weil es an keinen speziellen Typ und auch keine spezielle Typfamilie gebunden ist. Beispiele dafür sind:

```
1 Twins<?>
2 Pair<String, ?>
3 Pair<?, Integer>
4 Pair<?, ?>
```

Wenn alle aktuellen Typparameter ungebunden sind, spricht man von einem *ungebundenen parametrisierten Jokertyp* (engl. unbounded wildcard parameterized type). Ein solcher Typ ist kompatibel zu allen Instanzen des entsprechenden generischen Typen. Beispiele:

```
1 Twins<?> twins1 = new Twins<Object>();
2 Twins<?> twins2 = new Twins<String>();
3 Twins<?> twins3 = new Twins<Comparable<Number>>();
4 Twins<?> twins4 = new Twins<Thread.State>();
5
6 Pair<?, ?> pair1 = new Pair<String, String>();
7 Pair<?, ?> pair2 = new Pair<String, Integer>();
8 Pair<?, ?> pair3 = new Pair<Object, Boolean>();
```

Das Jokerzeichen `?` schränkt den Zugriff auf den parametrisierten Typ stark ein. Näheres dazu wird im Kapitel 4.2.1 erläutert.

Das Jokerzeichen `? extends TYPE`

Das Jokerzeichen `? extends TYPE` ermöglicht eine sogenannte obere Bindung (engl. upper bound). Die Typfamilie umfasst dabei alle Untertypen von `TYPE` inklusive `TYPE`. Beispiele:

```
1 Twins<? extends Number>
2 Pair<? extends Number, String>
3 Pair<Integer, ? extends Number >
4 Pair<? extends Number, ? extends Cloneable>
```

Die generische Instanz von `Twins` (Zeile 1) ist zu allen generischen Instanzen von `Twins` kompatibel, die den formalen Typparameter mit `Number` oder Untertypen von `Number` belegt haben. Beispiele:

```
1 Twins<? extends Number> myTwins;
2 myTwins = new Twins<Number>();
3 myTwins = new Twins<Integer>();
4 myTwins = new Twins<Double>();
5 myTwins = new Twins<Object>(); // inkompatibel > Fehler bei Übersetzung
```

Das Jokerzeichen `? super TYPE`

Das Jokerzeichen `? super TYPE` steht für eine untere Bindung (engl. lower bound). Die Typfamilie umfasst alle Obertypen von `TYPE` inklusive `TYPE`. Beispiele:

```
1 Twins<? super Number>
2 Pair<? super Integer, String>
3 Pair<Cloneable, ? super Integer>
4 Pair<? super Integer, ? super Boolean>
```

Die generische Instanz von `Twins` (Zeile 1) ist zu allen generischen Instanzen von `Twins` kompatibel, die den formalen Typparameter mit `Integer` oder Obertypen von `Integer` (wie zum Beispiel `Number`) belegt haben. Beispiele:

```
1 Twins<? super Number> myTwins;
2 myTwins = new Twins<Number>();
3 myTwins = new Twins<Serializable>(); // Number erweitert Serializable
4 myTwins = new Twins<Object>();
5 myTwins = new Twins<Integer>(); // inkompatibel > Fehler bei Übersetzung
```

Anders als formale Typparameter, können aktuelle Typparameter nur eine Bindung haben. Konstrukte wie `? extends Comparable<String> & Serializable` sind nicht erlaubt. Als Bindungstypen sind folgende Typen zulässig:

- (geschachtelte, innere, parametrisierte) Klassen
- (parametrisierte) Schnittstellen
- Aufzählungstypen
- Reihungen

Primitive Typen sind dagegen als Bindungstyp unzulässig. Beispiele:

```
1 Twins<? extends Integer>                // nicht-generische Klasse
2 Twins<? extends ArrayList<Number>>      // parametrisierte Klasse
3 Twins<? extends Pair<String, ?>>        // Bindung d. aktuellen Typparam.
4 Twins<? extends Cloneable>              // nicht-generische Schnittstelle
5 Twins<? extends Comparable<String>>     // parametrisierte Schnittstelle
6 Twins<? extends Thread.State>           // (geschachtelter) Aufzählungstyp
7 Twins<? extends int[]>                  // Reihung (primitiven Typen)
8 Twins<? extends Long[]>                 // Reihung (Referenztypen)
9 Twins<? extends int>                    // primitiver Typ > Unzulässig
```

8 Modellimplementierung II

Nachdem es in dem Kapitel Modellimplementierung I um die Grundlagen der Modellimplementierung ging, geht es in diesem Kapitel um die Feinheiten der Typentfernung und den damit verbundenen Techniken.

8.1 Brückenmethoden

Brückenmethoden (engl. bridge methods) sind immer dann nötig, wenn ein Typ eine parametrisierte Schnittstelle implementiert oder eine parametrisierte Klasse erweitert und deren Methoden überschreibt. Wie zuvor gezeigt, werden durch die Typentfernung Methodensignaturen verändert. Dadurch kann es vorkommen, dass eine im Untertyp überschriebene Methode nicht mehr die vom Obertyp geforderte Signatur aufweist. Das folgende Beispiel zeigt einen solchen Fall. Mit Hilfe der generischen Schnittstelle `Comparable` können typgleiche Objekte verglichen werden. Dazu muss der Typ der Objekte, die man vergleichen möchte, `Comparable` erweitern, beziehungsweise implementieren:

```
1 public interface Comparable<T> {
2     public int compareTo(T other);
3 }
```

Die Klasse `Point` implementiert die parametrisierte Schnittstelle `Comparable` mit dem aktuellen Typparameter `Point`. Die Methode `compareTo(T)` muss deshalb in der Klasse `Point` mit dem aktuellen Typparameter `Point` überschrieben werden:

```
1 public class Point implements Comparable<Point> {
2
3     public int compareTo(Point other) {
4           // Vergleich
5     }
6     // ...
7 }
```

Bei der Übersetzung der Schnittstelle `Comparable` und der damit verbundenen Typentfernung wird die Methodensignatur der Methode `compareTo(T)` verändert. Der formale Typparameter `T` wird durch `Object` ersetzt. Der folgende Quellcode zeigt die Schnittstelle `Comparable` nach der Typentfernung:

```
1 public interface Comparable {
2     public int compareTo(Object other);
3 }
```

Die in der Klasse `Point` überschriebene Methode entspricht nun nicht mehr der geforderten Signatur. Um diese Unstimmigkeit zu bereinigen, fügt der Übersetzer eine Brückenmethode ein. Die Brückenmethode hat die im Obertyp geforderte Signatur und macht nichts anderes, als die eigentliche überschriebene Methode aufzurufen.

`Point` nach der Typentfernung:

```
 1 public class Point implements Comparable {
 2
 3    public int compareTo(Point other) {
 4          // Vergleich
 5    }
 6
 7    public int compareTo(Object other) { // Brückenmethode
 8          return compareTo((Point) other);
 9    }
10    // ...
11 }
```

8.2 Seiteneffekte

Die Änderungen von Variablentypen, Konstruktoren- und Methodensignaturen hat einige Seiteneffekte zur Folge. Ein Seiteneffekt ist der Signaturkonflikt bei Methodenüberlagerung. Er tritt auf, wenn eine Methode dieselbe Signatur aufweist, wie eine mit einem formalen Typparameter arbeitende Methode nach der Typentfernung. Das folgende Beispiel zeigt einen solchen Konflikt:

```
 1 public class Twins<T> {
 2
 3    public void someMethod(T obj) {
 4          // ...
 5    }
 6
 7    public void someMethod(Object obj) {
 8          // ...
 9    }
10 }
```

In diesem Beispiel ist der formale Typparameter `T` ungebunden, das heißt er wird bei der Typentfernung durch `Object` ersetzt. Die Methode `someMethd(T)` in den Zeilen 3 bis 5 hätte nach der Typentfernung dieselbe Signatur, wie die Methode `someMethod(Object)` in den Zeilen 7 bis 9. Das folgende Beispiel zeigt einen ähnlichen Fall, bei dem jedoch kein Konflikt vorliegt und der daher zulässig ist. Der formale Typparameter `T` ist nun an die Klasse `Number` gebunden.

```
 1 public class Twins<T extends Number> {
 2
 3    public void someMethod(T obj) {
 4          // ...
 5    }
 6
 7    public void someMethod(Object obj) {
 8          // ...
 9    }
10 }
```

Der formale Typparameter `T` wird bei der Typentfernung durch `Number` ersetzt. Die beiden Methodensignaturen sind deshalb auch nach der Typentfernung verschieden:

```
 1 public class Twins {
 2
 3    public void someMethod(Number obj) {
 4          // ...
 5    }
 6
 7    public void someMethod(Object obj) {
 8          // ...
 9    }
10 }
```

9 Praktische Tipps

Nachdem es in den vorherigen Kapiteln um den Umgang mit generischen Einheiten und der Modellimplementierung ging, soll dieses Kapitel nützliche und interessante Praxisbeispiele darbieten. Insbesondere geht es in diesem Kapitel darum, die Vorteile und Eigenheiten von generischen Einheiten auszunutzen und auf mögliche Implementierungsfehler hinzuweisen. Die Beispiele 9.1, 9.2, 9.4, 9.5, 9.6, 9.7 und 9.8 beruhen auf Beispielen des Fragen-und-Antworten Katalogs (Java Generics FAQ) von Angelika Langer. Die Beispiele 9.2 und 9.8 beruhen auf Beispielen aus [bloch2008].

9.1 Ungebundene parametrisierte Jokertypen als Rückgabetyp von Methoden

Ungebundene parametrisierte Jokertypen sollten als Rückgabetyp von Methoden vermieden werden. Das Jokerzeichen `?` steht für die Familie aller Java-Typen und erlaubt daher nur einen sehr eingeschränkten Zugriff. Beispiel:

```
1 public static Twins<?> someMethod(Twins<?> twins) {
2     // ...
3 }

10 Twins<Integer> ints = new Twins<Integer>();
11 Twins<?> result     = someMethod(ints);
12
13 result.setFirst(new Integer(10));      // Übersetzungsfehler
```

Die Methode `someMethod(Twins<?>)` gibt ein Objekt vom Typ `Twins<?>` zurück. `Twins<?>` ist ein ungebundene parametrisierter Typ. Der Zugriff auf einen solchen Typ ist sehr eingeschränkt. Deutlich wird dies, wenn man Methoden benutzt, die einen formalen Typparameter als Parameter verlangen. Ist dieser formale Typparameter durch `?` belegt, akzeptiert die Methode ausschließlich `null` als Parameter. Andere Typen, wie in Zeile 13, werden nicht akzeptiert. Eine mögliche Lösung für dieses Problem wäre eine generische Methode:

```
1 public static <T> Twins<T> someGenericMethod(Twins<T> twins) {
2     // ...
3 }

10 Twins<Integer> ints = new Twins<Integer>();
11 Twins<?> result     = someGenericMethod(ints);
12
13 result.setFirst(new Integer(10));
```

9.2 Mischen von Rohtypen und parametrisierten Typen

Im Kapitel 2.2.1 wurde die so genannte Rohtyp-Instanziierung erläutert. Rohtypen sind ausschließlich auf Grund der Abwärtskompatibilität zu älteren (nicht-generischen) Java-Code zulässig. Durch ihre Verwendung büßt man die positiven Eigenschaften generischer Einheiten (zum Beispiel die statische Typsicherheit) ein. Aus diesem Grund sollte man immer parametrisierte Typen anstelle von Rohtypen benutzen. In einigen Fällen ist man jedoch gezwungen, Rohtypen zu benutzen, zum Beispiel wenn man ältere APIs oder Frameworks benutzt. Absolut vermeiden sollte man das Vermischen von Rohtypen und parametrisierten Typen. Dieses Vermischen ist potentiell gefährlich, wie das folgende Beispiel zeigt:

```
1 Twins<String> strings = new Twins<String>();
2 Twins raw             = strings;
3
4 raw.setFirst(new Integer(10));
5 String str = strings.getFirst(); // ClassCastException
```

In Zeile 2 wird einer Rohtyp-Referenz ein parametrisiertes Objekt zugewiesen. Auf Grund der Abwärtskompatibilität ist das zulässig. In den Zeilen 4 und 5 führt das jedoch zu einem schwerwiegenden Fehler. Dem mit `String` parametrisierten Objekt wird über die Rohtyp-Referenz ein `Integer`-Objekt hinzugefügt. Anders als bei Reihungen findet hier keine Laufzeitprüfung statt. In Zeile 5 führt das zu einer `ClassCastException`. Das Beispiel zeigt, welche trügerische Sicherheit hinter parametrisierten Typen steckt. Parametrisierte Typen sind nur so lange statisch typsicher, wie sie nicht mit Rohtypen vermischt werden.

Eine weitere Gefahr, die durch das Vermischen von Rohtypen und parametrisierten Typen auftreten kann, ist die sogenannte Haldenunreinheit (engl. heap pollution). Eine Haldenunreinheit tritt auf, wenn eine parametrisierte Referenz auf ein Objekt zeigt, das nicht vom selben parametrisierten Typ ist. Das folgende Beispiel zeigt eine solche Unreinheit. Die Zuweisung in Zeile 2 führt zu einer Warnung von Seiten des Übersetzers, ist jedoch auf Grund von Abwärtskompatibilität zulässig.

```
1 Twins raw = new Twins<String>();
2 Twins<Integer> ints = raw;
3
4 Integer int1 = ints.getFirst(); // ClassCastException
```

Eine solche Unreinheit kann auch bei ungeprüften Typumwandlungen (engl. unchecked casts) auftreten. Das folgende Beispiel zeigt eine solche Typumwandlung:

```
1 Object obj = new Twins<String>();
2 Twins<Integer> twins = (Twins<Integer>) obj; // ungeprüfte Typumwandlung
```

Generell ist man am besten beraten, Rohtypen im eigenen Code konsequent nicht zu benutzen (außer in **`class`**-Literalen und **`instanceof`**-Ausdrücken). Parametrisierte Typen sind in jeder Hinsicht eine bessere und vor allem sicherere Wahl.

9.3 Verdeckung von Java-Typen durch formale Typparameter

Der Java-Konvention nach sollen formale Typparameter mit Großbuchstaben benannt werden. An diese Konvention sollte man sich strikt halten. Formale Typparameter können jedoch auch mit Kleinbuchstaben oder beliebig langen Bezeichnern benannt werden. Bei unglücklicher Benennung kann das jedoch zu einer Verdeckung von Java-Typen führen. Beispiele:

```
1 class Twins<t> {}
2 class Twins<TYPE> {}
3 class Twins<String> {}
```

Alle drei Vereinbarungen sind zulässig. Die Vereinbarung in Zeile 3 deutet jedoch schon auf die Gefahr hin, die hinter einer unglücklichen Benennung steckt. Der formale Typparameter in Zeile 3 hat denselben Namen wie ein Java-Typ. Das führt dazu, dass dieser Java-Typ innerhalb des Sichtbarkeitsbereiches des formalen Typparameters verdeckt wird. Das folgende Beispiel zeigt die Auswirkung einer solchen Typverdeckung:

```
 1 public class Twins<String> {
 2
 3    public void someMethod() {
 4          String str = "eine Zeichenkette";   // Übersetzungsfehler
 5    }
 6
 7    public String toString() {                // Übersetzungsfehler
 8          // ...
 9    }
10    // ...
11 }
```

Die Methode `toString()` der Klasse `Object` wird in diesem Beispiel nicht überschrieben. Die Methodensignatur entspricht nicht der Signatur, die in der Oberklasse gefordert wird. Die Folge ist ein Übersetzungsfehler. Zeile 4 zeigt einen weiteren Übersetzungsfehler. Auf den ersten Blick wirkt die Zuweisung in Zeile 4 korrekt. Bei `String` handelt es sich hier jedoch nicht um die Klasse `String`, sondern um den formalen Typparameter `String`.

Der Übersetzer meldet bei einer Typ-Verdeckung eine Warnung. Diese Warnung sollte auf keinen Fall ignoriert werden. Auf verdeckte Java-Typen kann zwar weiterhin vollqualifiziert zugegriffen werden (Beispiel: `java.lang.String str = "eine Zeichenkette"`), viel besser ist es jedoch, Typverdeckungen zu vermeiden und formale Typparameter ausschließlich mit einem Großbuchstaben zu benennen.

9.4 Besonderheiten bei Methodenüberlagerung

Der Aufruf einer überladenen Methode aus einer generischen Methode kann zu überraschenden Ergebnissen führen. Das folgende Beispiel zeigt einen solchen Fall:

```
 1 public static void method(String str) {
 2    System.out.println("method(String)");
 3 }
 4
 5 public static void method(Object obj) {
 6    System.out.println("method(Object)");
 7 }
 8
 9 public static <T> void genericMethod(T t) {
10    method(t);
11 }
12
13 public static void main(String[] args) {
14    genericMethod("eine Zeichenkette");
15 }
```

Die generische Methode `genericMethod(T)` (Zeile 9 – 11) wird aus der `main(String[])`-Methode (Zeile 13 – 15) mit dem aktuellen Typparameter `String` aufgerufen. `genericMethod(T)` ruft wiederum die überladene Methode `method(...)` auf. In diesem Beispiel ist der formale Typparameter `T` mit `String` belegt. Daher sollte man annehmen, dass in Zeile 10 `method(String)` aufgerufen wird. Wenn man die Anwendung ausführt, erscheint jedoch folgende Ausgabe auf der Konsole:

```
method(Object)
```

Die Ausgabe zeigt, dass `method(Object)`, also die überladene Methode mit `Object` als Parameter aufgerufen wurde. Das liegt daran, dass der formale Typparameter `T` der Methode

`genericMethod(T)` bei der Typentfernung durch `Object` ersetzt wurde. Bindet man den formalen Typparameter `T`, wie im folgenden Quellcodeausschnitt, an den Typ `String`, wird `T` bei der Typentfernung durch `String` ersetzt.

```
 8 // ...
 9 public static <T extends String> void genericMethod(T t) {
10    method(t);
11 }
12 // ...
```

Ausgabe der Anwendung nach der Bindung des formalen Typparameters an `String`:

```
method(String)
```

Die Ausgabe und somit der Programmablauf hat sich durch die Bindung des formalen Typparameters `T` entscheidend geändert. Es ist also Vorsicht geboten, wenn man eine überladene Methode aus einer generischen Methode aufruft, und als Parameter ein Objekt vom Typ eines formalen Typparameters benutzt. Welche überladene Version aufgerufen wird, hängt von der Bindung der formalen Typparameter ab.

9.5 Implementierung verschiedener Instanzen einer generischen Schnittstelle

Eine Klasse darf nicht verschiedene Instanzen einer generischen Schnittstelle implementieren. Wie für nicht-generische Schnittstellen gilt hier: Eine Klasse darf jede Schnittstelle maximal einmal implementieren. Der Grund für diese Einschränkung ist die Typentfernung. Bei der Typentfernung werden verschiedene Instanzen einer generischen Schnittstelle auf denselben Typ, den Rohtypen abgebildet. Beispiel (vor Typentfernung):

```
1 public class Twins<T>
2     implements Comparable<T>, Comparable<String> { // unzulässig
3
4     public int compareTo(T o) { /* Vergleich */ }
5
6     public int compareTo(String o) { /* Vergleich */ }
7 }
```

Der vorherige Code lässt sich nicht kompilieren. Ließe er sich jedoch kompilieren, sähe der Code nach der Typentfernung wie folgt aus:

```
 1 public class Twins
 2    implements Comparable, Comparable {
 3
 4    public int compareTo(Object o) { /* Vergleich */ }
 5
 6    public int compareTo(String o) { /* Vergleich */ }
 8
 9    public int compareTo(Object o) { return compareTo((String) o); }
10 }
```

Die Schnittstelle `Comparable` würde nach der Typentfernung zweimal implementiert werden. Das ist jedoch unzulässig. Der Generierungsmechanismus für Brückenmethoden kann diese Situation nicht handhaben. Als Resultat gäbe es mehrere Methoden mit derselben Signatur (siehe Zeile 4 und 9).

Ähnliches gilt auch für Untertypen (Klassen oder Schnittstellen). Ein Typ muss dieselben Instanzen einer generischen Schnittselle implementieren, beziehungsweise erweitern, wie seine Obertypen. Welche Instanz einer generischen Schnittstelle von einem Typ implementiert, beziehungsweise erweitert werden muss, wird in den Obertypen festgelegt. Beispiel:

```
1 public interface SomeInterface extends Comparable<SomeInterface> {}
```

Die Schnittstelle `SomeInterface` erweitert hier die Instanz `Comparable<SomeInterface>` der generischen Schnittstelle `Comparable`. Klassen, die `SomeInterface` implementieren, implementieren automatisch `Comparable<SomeInterface>` und können selber keine anderen Instanzen von `Comparable` implementieren. Beispiel:

```
1 public class SomeClass
2     implements SomeInterface, Comparable<SomeClass> {} // unzulässig
```

Entsprechendes gilt auch für Schnittstellen. Schnittstellen, die `SomeInterface` erweitern, erweitern automatisch `Comparable<SomeInterface>` und können selber keine anderen Instanzen von `Comparable` erweitern.

9.6 Object.equals(Object) in einer generischen Klasse überschreiben

Das folgende Beispiel zeigt, wie man die Methode `equals(Object)` der Klasse `Object` in einer generischen Klasse überschreibt.

```
 1 public class Twins<T> {
 2
 3    private T first;
 4    private T second;
 5
 6    public boolean equals(Object other) {
 7          if(other == null)                          return false;
 8          if(this == other)                          return true;
 9          if(this.getClass() != other.getClass())    return false;
10
11          Twins<?> that = (Twins<?>) other;
12
13          return this.first.equals(that.first) &&
14                 this.second.equals(that.second);
15    }
16    // ...
17 }
```

Mit Hilfe eines ungebundenen parametrisierten Jokertypen können Objekte eines generischen Typen innerhalb der `equals(Object)`-Methode verglichen werden. Auf den ersten Blick sollte man vermuten, eine Typumwandlung zu `Twins<T>` wäre eine bessere Alternative. Beispiel:

```
11          Twins<T> otherTwins = (Twins<T>) other;
```

Typumwandlungen zu parametrisierten Typen können zur Laufzeit fehlschlagen und sind daher statisch nicht typsicher (mit Ausnahme von ungebundenen parametrisierten Typen). Die Typumwandlung in Zeile 11 nach `Twins<T>` führt zu einer Warnung (engl. unchecked warning). Durch die Verwendung von `Twins<?>` kann man die Warnung beseitigen.

9.7 Überschreibung von Object.clone() in einer generischen Klasse

Das folgende Beispiel zeigt, anhand der generischen Klasse `Twins`, wie man die Methode `clone()` aus der Klasse `Object` in einer generischen Klasse überschreibt:

```
 1 public class Twins<T> implements Cloneable {
 2
 3    private T first;
 4    private T second;
 5
 6    public Twins<T> clone() {
 7          Twins<T> clone = null;
 8          try {
 9                clone = (Twins<T>) super.clone(); // Warnung: ungeprüft
10          } catch (CloneNotSupportedException e) {
11                throw new InternalError();
12          }
13          try {
14                Class<?> clzz = this.first.getClass();
15                Method   meth = clzz.getMethod("clone", new Class[0]);
16                Object   dupl = meth.invoke(this.first, new Object[0]);
17                clone.first = (T) dupl; // Warnung: ungeprüft
18          } catch (Exception e) {
19                // Ausnahmebehandlung
20          }
21         try {
22                Class<?> clzz = this.second.getClass();
23                Method   meth = clzz.getMethod("clone", new Class[0]);
24                Object   dupl = meth.invoke(this.second, new Object[0]);
25                clone.second = (T) dupl; // Warnung: ungeprüft
26          } catch (Exception e) {
27                // Ausnahmebehandlung
28          }
29          return clone;
30    }
31    // ...
32 }
```

Seit Java 1.5 ist es möglich, dass eine überschreibende Methode einen anderen Rückgabetyp hat, als die Originalmethode im Obertyp. Der Rückgabetyp muss dabei jedoch Untertyp des Rückgabetypen der Originalmethode sein. Gerade bei Methoden wie `clone()` ist diese Änderung sinnvoll. Die beiden Variablen der Klasse `Twins` (`first` und `second`) sind vom Typ eines formalen Typparameters. Um von ihnen eine Kopie erzeugen zu können, muss das Reflection API benutzt werden. Die abgefangenen Ausnahmen in den Zeilen 18 und 26 wurden aus Platzgründen nicht behandelt. In diesem Beispiel gibt es drei **`try`**-Blöcke. Man hätte die Methode natürlich auch mit einem **`try`**-Block implementieren können, aber auf diese Weise kann man differenziert auf mögliche Ausnahmen reagieren.

Sehr ärgerlich sind die Warnungen in den Zeilen 9, 17 und 25. In Zeile 9 wird die `clone()`-Methode der Oberklasse aufgerufen. In diesem Beispiel ist das `Object`. Die Typumwandlung ist nötig, um den nötigen Zugriff auf den Klon zu erhalten. Wenn die `clone()`-Methode der Oberklasse ein Objekt mit ausreichendem Zugriff zurück liefern würde, wäre an dieser Stelle keine Typumwandlung nötig. Eine Typumwandlung nach `Twins<?>` würde zwar die Warnung verschwinden lassen, gäbe jedoch nicht den nötigen Zugriff auf das erhaltene Objekt. Um `clone()` für Variablen aufrufen zu können, die vom Typ eines formalen Typparameters sind, ist das Reflection API nötig. Zur Erinnerung: Die Zugriffsmöglichkeiten auf einen formalen Typparameter ohne Bindung beschränken sich auf die

Methoden von `Object`. Die `clone()`-Methode von `Object` hat einen eingeschränkten Zugriff (**`protected`**). Aus diesem Grund muss sie über das Reflection API aufgerufen werden.

9.8 „unchecked warnings“ beseitigen

Im Zusammenhang mit der Einführung generischer Einheiten in Java, wurde der Übersetzer um zahlreiche Warnungen erweitert. Diese Warnungen (engl. unchecked warning) haben die Aufgabe, den Programmierer darauf hinzuweisen, dass der betreffende Codeabschnitt nicht statisch typsicher ist und zur Laufzeit zu einer `ClassCastException` führen kann. Viele dieser Warnungen können leicht beseitigt werden.

In einigen Fällen, besonders wenn man Frameworks oder APIs benutzt, die mit Rohtypen arbeiten, können diese Warnungen nicht immer entfernt werden. Für solche Fälle gibt es die Annotation `@SuppressWarnings("unchecked")`. Mit ihr können sogenannte „unchecked warnings“ ausgeblendet werden. Das sollte man jedoch nur dann tun, wenn man sich sicher ist, das der bemängelte Codeabschnitt wirklich typsicher ist. Je nachdem, wo die `@SuppressWarnings(...)`-Annotation vereinbart wird, hat sie einen verschieden großen Auswirkungsbereich. Der kleinste Auswirkungsbereich ist vor einer Variablenvereinbarung. Dabei werden nur Warnungen ausgeblendet, die von der nachfolgenden Vereinbarung verursacht werden. Die `@SuppressWarnings(...)`-Annotation kann auch vor Methoden- oder Klassenvereinbarungen vereinbart werden. Dabei werden alle Warnungen innerhalb des entsprechenden Elementes ausgeblendet. Man sollte immer die kleinstmögliche Ebene wählen, um andere Warnungen nicht unbeabsichtigt auszublenden. Auf Klassenebene oder vor langen Methoden, beziehungsweise Konstruktoren sollte man die Annotation nicht verwenden, da ansonsten alle Warnungen innerhalb des Elementes ausgeblendet werden. `@SuppressWarnings(...)` darf nicht vor einer **`return`**-Anweisung stehen, da dies keine Vereinbarung ist. In einem solchen Fall sollte man den Rückgabewert einer Variablen zuweisen und diese zurückgeben. Neben der Annotation empfiehlt es sich, einen Kommentar hinzuzufügen, der erklärt, warum der entsprechende Codeabschnitt typsicher ist. Das ist besonders für eine spätere Refaktorisierung des Codes hilfreich.

9.8.1 Ungeprüfte Konvertierung (engl. unchecked conversion warning)

Diese Warnung wird beim Mischen von Rohtypen und parametrisierten Typen gemeldet. Der Übersetzer weist darauf hin, dass es unsicher ist, Rohtypen mit parametrisierten Typen zu vermischen.

Beispiel (ungeprüfte Konvertierung):

```
1 Twins<String> list = new Twins(); // unchecked conversion
```

Die Warnung kann sehr leicht beseitigt werden. Der Übersetzer gibt bereits den Lösungsweg vor. Er schlägt vor, anstelle des Rohtypen einen parametrisierten Typen (in diesem Fall vom Typ `Twins<String>`) zu benutzen. Lösung:

```
1 Twins<String> list = new Twins<String>();
```

9.8.2 Ungeprüfter Methodenaufruf (engl. unchecked call warning)

Diese Warnung wird gemeldet, wenn der Übersetzer die Typsicherheit eines Methodenaufrufes nicht sicherstellen kann. Das ist immer dann der Fall, wenn eine Methode einer Rohtyp-Instanz aufgerufen wird, die als Parameter ein Objekt vom Typ eines formalen Typparameters verlangt. Beispiel:

```
1 Twins list = new Twins();
2
3 list.setFirst(new Integer()); // unchecked call
```

Die Methode `setFirst(T)` der generischen Klasse `Twins` erwartet ein Objekt vom Typ `T`. Bei einer Rohtyp-Instanz ist das `Object`. Dadurch kann die statische Typsicherheit nicht gewährleistet werden. Die Warnung kann durch die Verwendung von parametrisierten Typen beseitigt werden. Lösung:

```
1 Twins<Integer> list = new Twins<Integer>();
2
3 list.setFirst(new Integer());
```

9.8.3 Ungeprüfter generischer Methodenaufruf (engl. unchecked method invokation warning)

Diese Warnung wird gemeldet, wenn eine generische Methode, die als Parameter einen parametrisierten Typ verlangt, mit einem Rohtyp aufgerufen wird. Beispiel:

```
1 ArrayList list = new ArrayList();
2
3 Collections.sort(list); // unchecked invocation
```

Die Warnung kann man durch die Verwendung eines parametrisierten Typen beseitigen. Lösung:

```
1 ArrayList<String> list = new ArrayList<String>();
2
3 Collections.sort(list);
```

9.8.4 Ungeprüfte Typumwandlung (engl. unchecked cast warning)

Diese Warnung wird bei einer Typumwandlung zu einem parametrisierten Typen gemeldet. Beispiel:

```
 1 public class Twins<T> {
 2
 3    // ...
 4
 5    public boolean equals(Object other) {
 6          // ...
 7          Twins<T> that = (Twins<T>) other; // unchecked cast
 8          // ...
 9    }
10 }
```

Die Warnung kann durch einen ungebundenen parametrisierten Jokertyp beseitigt werden. Eine Typumwandlung zu einem ungebundenen parametrisierten Jokertyp ist immer typsicher. Lösung:

```
7           Twins<?> that = (Twins<?>) other;
```

In einigen Situationen ist diese Lösung jedoch nicht anwendbar. Der Zugriff auf einen ungebundenen parametrisierten Jokertyp ist sehr eingeschränkt. Diese Lösung ist deshalb nur dann anwendbar, wenn die Zugriffsmöglichkeiten auf einen ungebunden parametrisierten Jokertyp ausreichen.

9.8.5 Ungeprüfte Erzeugung einer Reihung (engl. unchecked generic array creation warning)

Diese Warnung tritt in Verbindung von vararg-Methoden und generischen Einheiten auf. Die Parameter einer vararg-Methode werden vom Übersetzer zu einer Reihung transformiert. Es ist jedoch verboten, eine Reihung mit einem parametrisierten Typ als Komponententyp zu erzeugen. Trotzdem meldet der Übersetzer keinen Fehler, sondern eine Warnung. Beispiel:

```
1 public static <T> void varargMethod(T...ts) {
2     for(T current : ts) {
3           // ...
4     }
5 }

10 varargMethod(new Integer(10), new Integer(20));
11 varargMethod(new Twins<String>(), new Twins<String>()); // ungeprüft
```

In Zeile 10 wird die vararg-Methode `varargMethod(T...)` mit zwei `Integer`-Objekten aufgerufen. `Integer` ist keine generische Klasse, daher wird an dieser Stelle keine Warnung gemeldet. In Zeile 11 wird `varargMethod(T...)` mit dem generischen Typ `Twins` aufgerufen. An dieser Stelle meldet der Übersetzer eine ungeprüfte Reihungserzeugung. Warum diese Zeile nicht typsicher ist, zeigt sich, wenn man sich den Code nach der Typentfernung ansieht:

```
1 public static void varargMethod(Object[] ts) {
2     for(int i = 0; i < ts.length; i++) {
3           // ...
4     }
5 }

10 varargMethod(new Integer[] { new Integer(10), new Integer(20) });
11 varargMethod(new Twins[] { new Twins(), new Twins() }); // unsicher
```

Bei der Typentfernung wird der parametrisierte Typ `Twins<String>` durch seinen Rohtyp `Twins` ersetzt. Die einzelnen Parameter werden der Methode als Reihung übergeben. Sofern innerhalb von `varargMethod(T...)` nur lesend auf die Reihung zugegriffen wird, gibt es keine Probleme. Problematisch wird es jedoch, wenn auf die Reihung schreibend zugegriffen wird. Das folgende Beispiel verdeutlicht dies:

```
1 public static Twins<String>[] doSomething(Twins<String>...twArr) {
2     Object[] oArr = twArr;
3     oArr[0] = new Twins<String>("first", "second");
4     oArr[1] = new Twins<Integer>(new Integer(10), new Integer(20));
5
6     return twArr;
7 }

10 Twins<String>[] twArr = doSomething(  new Twins<String>(),
11                                       new Twins<String>());
12
13 for(Twins<String> current : twArr) {
14    String first = current.getFirst(); // ClassCastException
15    // ...
16 }
```

In Zeile 3 und 4 der Methode `doSomething(Twins<String>[])` wird schreibend auf den Parameter zugegriffen. In die Reihung mit dem Komponententyp `Twins<String>` wird ein Objekt vom Typ `Twins<Integer>` geschrieben. Man sollte annehmen, dass die Zuweisung zu einer `ArrayStoreException` führt, was aber nicht passiert. Nach der Typentfernung ist der Komponententyp der Reihung nicht mehr `Twins<String>` sondern `Twins`. Erst in Zeile 14 des benutzenden Codes fällt der Fehler auf.

Der folgende Codeausschnitt zeigt eine weitere potentielle Gefahr, die von der Kombination aus generischen Typen und vararg-Methoden ausgeht:

```
1 public static <T> T[] sort(T first, T second) {
2     return varargSort(first, second);
3 }
4
5 public static <T> T[] varargSort(T...twArr) {
6     // sortieren
7     return twArr;
8 }

10 Integer[] intArr = sort(new Integer(21), new Integer(3));
                                                    // ClassCastException
```

Der Aufruf von `sort(T, T)` schlägt im benutzenden Code (Zeile 10) mit einer `ClassCastException` fehl. Auch hier lässt sich der Fehler leicht erkennen, wenn man sich den Code nach der Typentfernung ansieht:

```
1 public static Object[] sort(Object first, Object second) {
2     return varargSort(new Object[] { first, second });
3 }
4
5 public static Object[] varargSort(Object[] twArr) {
6     // sortieren
7     return twArr;
8 }

10 Integer[] intArr = (Integer[]) sort(new Integer(21), new Integer(3));
```

In Zeile 10 des benutzenden Codes wird `sort(Object, Object)` mit `Integer` als Parameter aufgerufen. `sort(Object, Object)` verpackt die beiden `Integer`-Objekte in ein `Object[]` und ruft damit `varargSort(Object[])` auf. `varargSort(Object[])` und `sort(Object, Object)` geben Reihungen mit dem Komponententyp `Object` zurück. Der Übersetzer fügt deshalb in den benutzenden Code eine Typumwandlung nach `Integer[]` ein. Diese schlägt jedoch mit einer `ClassCastException` fehl, da eine Reihung mit dem Komponententyp `Object` zu einer Reihung mit dem Komponententyp `Integer` inkompatibel ist.

Die beiden Beispiele zeigen, dass die Kombination vararg-Methode und parametrisierter Typ potentiell gefährlich ist. Es empfiehlt sich daher, parametrisierte Typen nicht in Verbindung mit vararg-Methoden zu verwenden.

9.9 Unterschied zwischen Twins<?> und Twins<Object>

Das ungebundene Jokerzeichen `?` wurde bereits im Kapitel 4.2.1 vorgestellt. Es steht für die Familie aller Java-Typen. `Twins<?>` repräsentiert die Familie aller Instanzen der generischen Klasse `Twins`, wie zum Beispiel `Twins<String>`, `Twins<Integer>`, `Twins<Object>`. Der eigentliche Nutzen des Jokerzeichens `?` ist der, dass man sich nicht auf einen konkreten Typ festlegen muss. Bei der Instanz `Twins<Object>` hat man sich hingegen für einen Typ entschieden, nämlich `Object`. `Twins<Object>` repräsentiert ein typgleiches Paar, dessen Objekte vom Typ `Object` (oder Untertypen von `Object`) sind.

`Twins<?>` ist als parametrisierter Jokertyp nur als Typ einer Referenz zulässig. Eine solche Referenz kann jede Instanz des entsprechenden Typen referenzieren. Beispiel:

```
1 Twins<?> twins = new Twins<String>();
2 twins = new Twins<Integer>();
3 twins = new Twins(); // Warnung
```

Die ungebundene Instanz der Klasse `Twins` in Zeile 1 steht für ein homogenes Paar typgleicher Objekte. Ganz anders ist das bei der Instanz `Twins<Object>`, das für ein heterogenes Paar typgleicher Objekte steht.

10 Glossar

Dieses Kapitel enthält alle im Tutorial eingeführten deutschen und englischen Fachbegriffe aus dem Bereich generische Einheiten in Java.

argument type inference

siehe *Argumentenrückschluss*

Argumentenrückschluss

Bei einem Argumentenrückschluss (engl. argument type inference) leitet der Übersetzer den Typ der aktuellen Typparameter einer generischen Methode anhand des Aufrufkontextes her. Beispiel:

```
public static final <T> List<T> emptyList() {}

List<String> list = Collections.emptyList();
```

Der Übersetzer erkennt in diesem Beispiel, dass der Rückgabewert der Methode einer Referenz vom Typ `List<String>` zugewiesen wird. Daraus leitet er her, dass die Methode mit dem aktuellen Typparameter `String` aufgerufen werden muss.

Aktueller Typparameter

Ein aktueller Typparameter ist ein Referenztyp oder ein Jokerzeichen, der bei der Instanziierung eines generischen Typen einen formalen Typparameter ersetzt. Beispiele:

```
List<String>            // String ist der aktuelle Typparameter
List<?>                 // ungebunden
List<? extends Number>  // obere Bindung (Number und Unterklassen)
List<? super Integer>   // untere Bindung (Integer und Obertypen)
```

Bindung

Bei einer Bindung wird ein formaler Typparameter an Typen gebunden. Eine Bindung schränkt den Satz möglicher aktueller Typparameter ein, ermöglicht aber den Zugriff auf alle öffentlichen Elemente der Bindungstypen, für die der formale Typparameter steht. Bei einer generischen Instanziierung stehen zwei Bindungsarten zur Verfügung:

- obere Bindung (engl. upper bound)
- untere Bindung (engl. lower bound)

Bei der Vereinbarung eines formalen Typparameters steht hingegen nur die obere Bindung zur Verfügung.

bounded wildcard parameterized type

siehe *gebundener parametrisierter Jokertyp*

bridge method

siehe *Brückenmethode*

Brückenmethode

Brückenmethoden (engl. bridge methods) sind immer dann nötig, wenn ein Typ eine parametrisierte Schnittstelle implementiert oder eine parametrisierte Klasse erweitert und deren Methoden überschreibt. Bei der Typentfernung werden Methodensignaturen geändert. Daher kann es vorkommen, dass eine im Untertyp überschriebene Methode nicht mehr die im Obertyp geforderte Signatur aufweist. In einem solchen Fall wird eine Brückenmethode eingefügt. Diese Methode hat die vom Obertyp geforderte Signatur und ruft die eigentliche überschriebene Methode auf.

concrete parameterized type

siehe *konkreter parametrisierter Typ*

formaler Typparameter

Ein formaler Typparameter ist ein Platzhalter für einen beliebigen Referenztyp.

Beispiel (`E` ist der formale Typparameter):

```
public interface List<E> {}
```

gebundener formaler Typparameter

Ein gebundener formaler Typparameter ist an mindestens einen Typ gebunden. Eine Bindung hat folgende Auswirkungen auf den formalen Typparameter:

- Ermöglicht den Zugriff auf alle öffentlichen Elemente des Bindungstypen (mit Ausnahme von Konstruktoren)
- Schränkt den Satz möglicher aktueller Typparameter ein

Beispiele:

```
T extends Comparable<T>
T extends Number & Comparable<T>
```

gebundener parametrisierter Jokertyp

Instanziierung eines generischen Typ, bei der alle aktuellen Typparameter durch die Verwendung des Jokerzeichens `? extends TYPE` oder des Jokerzeichens `? super TYPE` an Typfamilien gebunden sind.

Beispiele:

```
Twins<? extends Number>
List<? super Integer>
Pair<? extends Number, ? super Integer>
```

generische Einheit

Eine generische Einheit ist (in Java) ein generischer Typ oder eine generische Methode. Generische Einheiten sind mit formalen Typparametern parametrisiert.

Beispiel für einen generischen Typ:

```
public interface List<E> {}
```

Beispiel für eine generische Methode:

```
public static <T> void copy(List<T> src, List<? extends T> dest) {}
```

generische Methode

Eine generische Methode ist eine Methode mit eigenen formalen Typparametern. Beispiele:

```
1 public static final <T> List<T> emptyList() { /* ... */ }
2 public static <T> void addAll(List<T> list, T...ts) { /* ... */ }
```

Aufruf einer generischen Methode:

```
1 List<String> list1 = Collections.<String>emptyList();
2 List<String> list2 = Collections.emptyList(); // Arg.-rückschluss
```

generischer Typ

Ein generischer Typ ist (in Java) ein Referenztyp mit formalen Typparametern. Mit Ausnahme von Aufzählungstypen, anonymen inneren Klassen und Ausnahmeklassen (Erweiterungen von `java.lang.Throwable`) können alle Java-Referenztypen generisch sein. Beispiele:

```
1 interface Comparable<T> {
2     public int compareTo(T other);
3 }
```

```
1 class Pair<T, S> {
2     // ...
3 }
```

Jokerzeichen

Ein Jokerzeichen ist ein syntaktisches Konstrukt, das für eine Typfamilie steht. Jokerzeichen können als aktuelle Typparameter bei der Instanziierung eines generischen Typen benutzt werden. In einem **new**-Ausdruck oder der Instanziierung einer generischen Methode dürfen Jokerzeichen jedoch nicht benutzt werden. Beispiele:

```
1 new ArrayList<? extends Number>()          // unzulässig
2 Collections.<? extends Number>emptyList(); // unzulässig
```

Es werden drei Jokerzeichen unterschieden:

- `?`
  ungebundenes Jokerzeichen, steht für die Familie aller Referenztypen

- `? extends TYPE`
  Steht für eine obere Bindung, das heißt alle Untertypen von `TYPE` inklusive `TYPE`

- `? super TYPE`
  Steht für eine untere Bindung, das heißt alle Obertypen von `TYPE` inklusive `TYPE`

konkreter parametrisierter Typ

Einen generischen Typ, der ausschließlich mit Java-Typen instanziiert ist, nennt man *konkreter parametrisierter Typ* (engl. concrete parameterized type). Beispiele:

```
Twins<Integer>
List<String>
Pair<String, java.util.Date>
```

linke Bindung

Unter linker Bindung versteht man den ersten Bindungstypen eines formalen Typparameters. Ein formaler Typparameter wird bei der Typentfernung durch seine linke Bindung ersetzt. Alle anderen Bindungstypen bleiben unberücksichtigt. Beispiele:

```
1 public class Pair<T extends Comparable<T>> {}
2 public class Twins<T extends Number & Comparable<T>> {}
```

In Zeile 1 ist `Comparable<T>` die linke Bindung, in Zeile 2 ist es `Number`.

obere Bindung

Bei einer oberen Bindung (engl. upper bound) wird ein formaler Typparamater mit Hilfe des Jokerzeichens `? extends TYPE` an eine Typfamilie gebunden. Die Typfamilie umfasst dabei alle Untertypen von `TYPE` inklusive `TYPE`. Beispiele:

```
1 Twins<? extends Number>
2 Pair<? extends Number, String>
3 Pair<Integer, ? extends Number >
4 Pair<? extends Number, ? extends Cloneable>
```

parametrisierte Methode

Eine parametrisierte Methode (engl. parameterized method) ist eine mit aktuellen Typparametern instanziierte generische Methode. Beispiel:

```
1 Collections.<String>emptyList()
```

parametrisierter Jokertyp

Ein parametrisierter Jokertyp ist eine Instanz eines generischen Typen, bei der mindestens ein aktueller Typparameter ein Jokerzeichen ist (engl. wildcard parameterized type). Beispiele:

```
Twins<?>
Pair<String, ? extends Number>
Pair<? super Integer, String>
```

Siehe auch *gebundener parametrisierter Jokertyp* und *ungebundener parametrisierter Jokertyp*.

parametrisierter Typ

Ein parametrisierter Typ (engl. parameterized type) ist ein mit aktuellen Typparametern instanziierter generischer Typ. Beispiele:

```
Twins<String>
ArrayList<Integer>
Pair<String, ?>
List<?>
```

parameterized method

siehe *parametrisierte Methode*

parameterized type

siehe *parametrisierter Typ*

raw type

siehe *Rohtyp*

Rohtyp

Ein Rohtyp ist eine Instanz eines generischen Typen ohne explizite aktuelle Typparameter. Alle formalen Typparameter des generischen Typen werden implizit mit `Object` belegt. Rohtyp-Instanziierungen sind nur aus Kompatibilitätsgründen zu früheren nicht-generischen Java-Versionen erlaubt. Beispiele:

```
Twins
Comparable
ArrayList
```

type argument

siehe *aktueller Typparameter*

type erasure

siehe *Typentfernung*

type parameter

siehe *formaler Typparameter*

Typentfernung

Die Typentfernung (engl. type erasure) wird während der Übersetzung von Java-Quellcode durchgeführt. Bei der Typentfernung werden nahezu alle[1] generischen Informationen aus dem Code entfernt. Das Ergebnis der Typentfernung ist ein nicht-generischer Code.

unbounded wildcard

siehe *ungebundenes Jokerzeichen*

unbounded wildcard parameterized type

siehe *ungebundener parametrisierter Jokertyp*

ungebundener parametrisierter Jokertyp

Ein ungebundener parametrisierter Jokertyp ist eine Instanz eines generischen Typen, bei der alle aktuellen Typparameter durch die Verwendung des Jokerzeichen `?` ungebunden sind. Beispiele:

```
Twins<?>
List<?>
Pair<?, ?>
```

ungebundenes Jokerzeichen

Das Jokerzeichen `?` steht für die Familie aller Referenztypen und ist daher ungebunden. Beispiele:

```
Twins<?>
List<?>
Pair<?, ?>
```

[1] Der Übersetzer fügt Kommentare mit generischen Informationen hinzu. Diese Kommentare werden von einigen Decompilern benutzt, um den die entfernten Typparameter wieder herzuleiten.

untere Bindung

Bei einer unteren Bindung (engl. lower bound) wird ein formaler Typparamater mit Hilfe des Jokerzeichens `? super TYPE` an eine Typfamilie gebunden. Die Typfamilie umfasst dabei alle Obertypen von `TYPE` inklusive `TYPE`. Beispiele:

```
1 Twins<? super Integer>
2 Pair<? super Integer, String>
3 Pair<Cloneable, ? super Integer>
4 Pair<? super Integer, ? super Boolean>
```

wildcard

siehe *Jokerzeichen*

wildcard bounded type parameter

siehe *gebundener formaler Typparameter*

wildcard parameterized type

siehe *parametrisierter Jokertyp*

11 Anhang

Die folgende Tabelle zeigt die Unterschiede der verschiedenen Instanzen eines generischen Typen bezüglich ihrer Verwendbarkeit:

	Typ einer Referenz	`new`-Ausdruck	Obertyp	`class`-Literal	`instanceof`-Ausdruck
Rohtyp	ja	ja	ja	ja	ja
konkreter parametrisierter Typ	ja	ja	ja	nein	nein
parametrisierter Jokertyp	ja	nein	nein	nein	nein, ja wenn alle Typparameter ungebunden sind

Tabelle 1 Verwendbarkeit verschiedener Instanzen

Quellenverzeichnis

Selbständige Bücher und Schriften

[grude2005] Prof. Dr. Ulrich Grude, 2005, Java ist eine Sprache, Vieweg Verlag, ISBN-10: 3528059141, ISBN-13: 978-3528059149

[bloch2008] Joshua Bloch, 2008, Effective Java Second Edition, Prentice Hall PTR, ISBN-10: 0321356683, ISBN-13: 978-0321356680

Internetquellen

Java Generics FAQ http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html

Sun Generics Tutorial http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

Wikipedia http://www.wikipedia.de/

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich während der Ausarbeitung dieser Masterarbeit unterstützt haben. Ich möchte mich vor allem bei meinen Eltern bedanken, die mich immer unterstützt haben und ohne die dieses Studium nicht möglich gewesen wäre. Besonders möchte ich mich bei Herrn Prof. Dr. Ulrich Grude bedanken, der sich viel Zeit für mich genommen und mir kompetent zur Seite gestanden hat. Letztlich möchte ich mich noch bei Frau Angelika Langer bedanken, die mit ihrem FAQ eine hervorragende Grundlage für meine Arbeit geschaffen hat und mir in einigen Fragen kompetente und ausführliche Antworten gegeben hat.

Ein recht herzlicher Dank geht auch an Fabian Seidel und Gerald Drescher die meine Arbeit korrektur gelesen haben.

Mehr als zwei Jahre sind vergangen seit ich diese Arbeit verfasst habe. Nun habe ich endlich die Zeit gefunden mein Werk zu überarbeiten. Das wäre jedoch nicht ohne die Mithilfe einiger fleißiger Menschen möglich gewesen. Ich möchte mich daher nochmals bei Prof. Dr. Ulrich Grude bedanken, der mir seine Notizen und Anmerkungen zu der ersten Version zur Verfügung gestellt hat. Ebenso möchte ich mich bei Svetlana Grinman bedanken, die meine Arbeit korrektur gelesen hat.