Objektorientierte Programmierung

Kapitel 20: Generische Programmierung

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/oop18/

Inhalt

1. Einleitung
2. Generische Typen
3. Typ-Beschränkungen
4. Generische Methoden
5. Wildcards

Einleitung und Motivation (1)

- Methoden (Prozeduren, Funktionen) bieten die Möglichkeit, ähnliche Folgen von Statements durch eine einmal definierte Methode mit Parametern zu ersetzen.

  Man abstrahiert also von den kleinen Unterschieden in diesen Folgen von Statements und erkennt, dass es — davon abgesehen — eigentlich die gleiche Statementfolge ist.

- Auf höherer Ebene, bei den Klassen, stellt sich das gleiche Problem: Z.B. ist eine Klasse für Listen von ganzen Zahlen sehr ähnlich zu einer Klasse für Listen von `Person`-Objekten.

- Es ist ein allgemeines Prinzip guten Programmierstils, dass man jede Entscheidung, jeden Algorithmus, jede Idee nur an einer Stelle im Programm (einmal) aufschreiben sollte.

  Deswegen z.B. auch Konstanten für Arraygrößen etc.

Einleitung und Motivation (2)

- Programmieren durch Kopieren und Modifizieren (Copy&Paste Programmierung) ist meist schlechter Stil. Man hat dann mehrere, fast identische Programmteile, d.h.:
  - Mehr Tippaufwand,
  - mehr zu lesen und zu verstehen für neue Teammitglieder,

    Man muss jetzt ja auch erst einmal verstehen, dass es bis auf die kleinen Änderungen der gleiche Programmcode ist.

  - die Denkstrukturen bei der Entwicklung des Programms entsprechen nicht den Strukturen im Programmtext,
  - Änderungen werden aufwendig und führen leicht zu Inkonsistenzen.

    Manche Kopien werden bei der Änderung vergessen.

Einleitung und Motivation (3)

- Während Methoden Parameter für Werte haben, sollen jetzt Klassen mit Parametern für Typen eingeführt werden.
- Damit ist es möglich, wiederverwendbare Datenstrukturen zu definieren.

  Wiederverwendbarkeit von Programmcode: wichtiges Ziel der Objektorientierung.

- Z.B. `List<T>` für Listen mit einem beliebigen Element-Typ `T`.
- Für eine konkrete Anwendung kann man für den Typ-Parameter `T` einen beliebigen Referenztyp einsetzen, z.B. könnte man so eine Variable deklarieren, in der man eine Liste von Personen verwalten kann:

```
List<Person> l = new List<Person>();
```

  In Java kann man für die Typ-Parameter keine primitiven Typen einsetzen, aber Wrapper-Klassen: `List<Integer>`. In C++ ginge auch `int`.

Einleitung und Motivation (4)

- Klassen mit Parametern heißen auch “generische Klassen”, bzw. “generische Typen” (inkl. entsprechenden Interfaces).
- Das Wort “generisch” wird wohl verwendet, weil man aus dem allgemeinen Konzept, z.B. `List<T>`, viele spezielle Klassen, z.B. `List<Person>` und `List<Integer>` erzeugen kann.

  Für Java stimmt das so nicht ganz, weil, wie später erläutert wird, es tatsächlich nur eine Klasse ist. Das Wort “generisch” für allgemeine Datenstrukturen, die für viele konkrete Typen benutzt werden können, ist aber älter als Java. In C++ würde man tatsächlich aus einer allgemeinen Schablone (“Template”) viele konkrete Klassen erzeugen können.

- Es ist also ein wesentliches Kennzeichen, von den konkreten Element-Typen zu abstrahieren, und auf einer höheren Ebene zu programmieren.

Einleitung und Motivation (5)

- Typ-Parameter sind besonders für “Collection-Typen” nützlich, d.h. Datenstrukturen, die eine Sammlung von Werten eines anderen Datentyps verwalten, z.B. Menge oder Liste.

  Diese Typen werden manchmal auch Container-Typen genannt.

- Ein Array ist auch eine Art von Collection-Typ, der in die Sprache eingebaut ist.
- Bei Arrays kann man einen beliebigen Element-Typ wählen. Dies möchte man auch für eigene Collection-Typen haben.
- Damit werden die Collection-Typen zu einer Art “Typ-Konstruktor”: Man steckt einen Element-Typ `T` hinein und bekommt einen komplexeren Typ, z.B. `List<T>`.

  So wie man aus dem Typ `T` auch den Array-Typ `T[]` aufbauen kann.

Einleitung und Motivation (6)

- Wie bekannt, kommt Java mit einer großen Bibliothek vordefinierter Klassen mit vielen nützlichen Methoden.

  [http://docs.oracle.com/javase/7/docs/technotes/guides/collections/]

- Insbesondere gibt es auch recht viele Collection-Typen im Paket `java.util`. Z.B. ist die Klasse `LinkedList<T>` eine Implementierung des Interfaces `List<T>`.

  [http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html]

- Als Programmier-Anfänger wird man diese generischen Typen benutzen, aber vermutlich eher selten eigene definieren.

- Man versteht aber ein Konzept (wie Collection-Typen) besser, und kann sie besser (zielgerichteter, bewusster, effizienter) anwenden, wenn man sie auch selbst implementieren könnte.

  Siehe Vorlesung "Datenstrukturen und effiziente Algorithmen I" im Sommer.

Alternative: Object (1)

- Typ-Parameter gibt es in Java erst seit Version 5 (1.5).
- Eine Bibliotheksklasse `Vector` gab es schon in Java 1.0, das `Collection`-Interface mit vielen Collection-Klassen (z.B. `LinkedList`) wurde in Java 1.2 eingeführt.
- Bevor es Typ-Parameter gab, hat man mit der Klasse `Object` gearbeitet, z.B. war die Liste dann fest eine Liste von Elementen vom Typ `Object`.
- Da `Object` ein Obertyp von allen Referenztypen ist, und primitive Typen über Wrapper-Klassen auch passend gemacht werden können, ist `Object` als Element-Typ allgemein genug.

In eine Liste von `Object`-Elementen kann man beliebige Dinge speichern.

Alternative: `Object` (2)

- Die Übergabe von Werten an Methoden der Klasse (z.B. zum Anhängen neuer Elemente) ist auch kein Problem.

  Für Objekte beliebiger Klassen geschieht der "Up-Cast" automatisch, für Werte primitiver Typen ist Auto-Boxing eine Lösung (gibt es allerdings auch erst seit Java 5, die Wrapper-Klassen aber von Anfang an).

- Unangenehm sind Methoden, die ein Element der Liste liefern: Ihr Ergebnis-Typ ist `Object`, man muss den "Down-Cast" (z.B. nach `Person`) jedes Mal explizit schreiben.

- Riskant ist, dass man in die Liste wirklich beliebige Objekte einfügen kann, auch ganz unterschiedliche Dinge.
  Man merkt es erst beim "Down-Cast", wenn man ein Objekt aus der Liste entnehmen will: `ClassCastException`.

Alternative: `Object` (3)

- Das Problem mit dieser Lösung ist also, dass der Compiler nicht weiß, dass es sich tatsächlich um eine Liste von `Person`-Objekten handeln soll.

  Es ist ja als Liste von `Object` deklariert.

- Daher kann er keinen Schutz vor möglichen Fehlern geben.

- Die Fehler werden erst zur Laufzeit bemerkt, und zwar erst, wenn das fehlerhafte Objekt aus der Liste entnommen wird, nicht gleich bei der Einfügung.

  Der Fehler ist eventuell auch abhängig von den Eingabedaten und tritt gar nicht bei jedem Test auf. Er wird also nicht sicher gefunden, und wenn, gestaltet sich die Fehlersuche recht aufwändig.

  Ein Ausweg ist, sich eine Klasse `PersonList` zu definieren, die intern `List` verwendet, und alle Casts durchführt. Nach außen stimmt die Typ-Struktur dann, aber man dupliziert doch mindestens die Methoden-Köpfe.

Aktuelle Lösung in Java (1)

- Um die parametrisierten Klassen in Java richtig zu verstehen, muss man wissen, dass es intern bis heute so funktioniert (mit allgemeinstem Typ für den Parameter, meist `Object`).

  Die Java-Entwickler standen ja auch vor dem Problem, dass es schon viel Java-Code gab, der mit den alten Collections programmiert war.

  Man wollte auch keinen radikalen Bruch, sondern eine schrittweise Migration zu der neuen, sichereren Lösung erlauben.

- In Java gibt es also intern nur eine Implementierung für eine parametrisierte Liste, egal wie viele Typen eingesetzt werden.

  In der Sprache C++ werden die Typ-Parameter so implementiert, dass der Programmcode im Prinzip für jeden eingesetzten Typ dupliziert wird. Da man dort z.B. auch `int` und `double` mit unterschiedlichen Speichergrößen einsetzen kann, sind die Größen der entstehenden Objekte unterschiedlich (daher kann nicht der gleiche Maschinencode verwendet werden).

  Java ist in diesem Punkt deutlich anders.

Aktuelle Lösung in Java (2)

- Die Typ-Parameter erlauben dem Compiler aber eine bessere Prüfung des Programmcodes zur Fehler-Erkennung (und man muss die Casts nicht explizit schreiben).
- Es gibt in Java aber tatsächlich nur eine Klasse "`List`", und nicht Klassen `List<Person>` und `List<Integer>`.

```
List<Person>  x = new List<Person>();
List<Integer> y = new List<Integer>();
if(x.getClass() == y.getClass()) // das ist wahr!
    System.out.println("Gleiche Klasse");
```

- Wenn man die Typ-Parameter eines parametrisierten Typs weglässt, erhält man den zugehörigen "Raw Type".

  `List` wäre also der "Raw Type" von `List<Person>` und auch von `List<Integer>`.

Aktuelle Lösung in Java (3)

- Zur Laufzeit wird mit den “Raw Types” gearbeitet (also die alte Lösung mit dem allgemeinsten Typ `Object`).
- Die Typ-Parameter sind nur zur Compile-Zeit wichtig.
- Selbstverständlich werden bei der Typ-Prüfung durch den Compiler die vollständigen Typen verwendet.
- Wenn z.B. `x` eine `List<Person>` ist, und `y` eine `List<Integer>`, dann führt die Zuweisung

  ```
  x = y;
  ```

  zu einem “incompatible types”-Fehler.

  Das ist ja gerade der Vorteil gegenüber der alten Lösung, wo alles nur `List` war (mit `Object`-Elementen): Da wäre diese Zuweisung möglich gewesen. Wenn es auch heute noch intern (zur Laufzeit) Objekte der gleichen Klasse sind, sorgt der Compiler dafür, dass solche Zuweisungen nicht geschehen können.

Aktuelle Lösung in Java (4)

- Weil zur Laufzeit die Information über den Parameter-Typ `T` nicht zur Verfügung steht, gibt es Einschränkungen in seiner Verwendung (`T` ist nicht ganz wie eine Klasse zu benutzen).
- Zum Beispiel kann man in den Methoden einer generischen Klasse `List<T>` kein Objekt vom Typ `T` anlegen:
  - Objekte sind unterschiedlich groß (je nachdem, wie viele Attribute von welchen Datentypen sie enthalten).

    Man braucht den exakten Typ, und die richtige Menge Hauptspeicher zu reservieren, und den richtigen Konstruktor aufzurufen.
  - Außerdem ist in jedem Objekt vermerkt, welchen Typ es hat.

    Um überschriebenen Methoden die richtige Variante aufzurufen.
- Referenzen auf `T` sind dagegen kein Problem, z.B. kann man in `List<T>` ein Objekt vom Typ `Node<T>` erzeugen (s.u.).

Aktuelle Lösung in Java (5)

- Diese Lösung hat also einige Einschränkungen, aber auch Vorteile:
  - Bei Java braucht der Anwender einer generischen Klasse nicht den Java Quellcode.

    Bei der Lösung in C++ (Templates) ist das anders: Hier muss der Compiler im Prinzip den Quellcode für jede Instanziierung einer parametrisierten Klasse mit einem neuen Typ neu übersetzen.

  - Bei der Anwendung einer generischen Klasse für einen neuen Parameter-Typ können in dieser Klasse keine neuen Fehler vom Compiler gemeldet werden.

    Bei C++ findet mindestens ein Teil der Übersetzung erst statt, wenn das Template instanziiert wird. Das führt dann gelegentlich zu schwer verständlichen Fehlermeldungen.

  - Fließender Übergang von altem Code!

Inhalt

1. Einleitung
2. Generische Typen
3. Typ-Beschränkungen
4. Generische Methoden
5. Wildcards

Grundlagen, Erstes Beispiel (1)

- Als erstes Beispiel soll eine generische Klasse `Container<T>` definiert werden, die ein Attribut von wählbarem Typ `T` hat:

```
 (1)  class Container<T> {
 (2)      private T attr;
 (3)      Container(T a) {
 (4)          attr = a;
 (5)      }
 (6)
 (7)      T getAttr() {
 (8)          return attr;
 (9)      }
(10)
(11)      void setAttr(T a) {
(12)          attr = a;
(13)      }
(14)  }
```

Grundlagen, Erstes Beispiel (2)

- Eine generische Klassendefinition unterscheidet sich von einer normalen Klassendefinition also dadurch, dass nach dem Klassennamen in spitzen Klammern `<...>` (Größer- und Kleinerzeichen) Typ-Parameter angegeben werden.

- Die Typ-Parameter sind beliebige Bezeichner, aber es ist Konvention, einzelne Großbuchstaben zu verwenden, z.B. `T`.

  Für den Element-Typ einer Liste etc. ist "`E`" üblich. Für Datenstrukturen, die eine Abbildung von "Schlüsseln" auf Daten implementieren, z.B. von Kundennummern auf Kundendaten, ist "`K`" (von engl. "key") für den Definitionsbereich und "`V`" (von engl. "value") für den Wertebereich üblich.

- Eine generische Klasse kann mehrere Typ-Parameter haben, die durch Kommata getrennt werden:

```
class C<T1, T2, ..., Tn> { ... }
```

Grundlagen, Erstes Beispiel (3)

- Mit gewissen Einschränkungen, die unten genau erläutert werden, können die Typ-Parameter (im Beispiel `T`) in der Klassendefinition wie ein Typ verwendet werden.

  Wie oben schon erwähnt, ist der Grund für die Einschränkungen, dass es intern tatsächlich nur eine Klasse gibt, in der `T` durch `Object` ersetzt ist.

- Z.B. können Typ-Parameter verwendet werden als
  - Argument- und Resultattyp von (nicht-statischen) Methoden,
  - Argument-Typ des Konstruktors,
  - Typ von lokalen Variablen und Instanzvariablen (Attributen).

- Beachte: Der Name des Konstruktors ist der einfache Klassenname, d.h. dort wird nicht `<...>` angehängt.

  Das hängt wieder damit zusammen, dass es eigentlich nur eine Klasse ist.

Grundlagen, Erstes Beispiel (4)

- So wie man beim Methoden-Aufruf für die formalen Parameter konkrete Argument-Werte angibt, so kann man aus der generischen Klasse eine konkrete "parametrisierte Klasse" erzeugen, indem man für den Parameter `T` einen konkreten Referenztyp einsetzt:

```
Container<Integer> x = new Container<Integer>(1);
```

- In C++ erzeugt der Compiler eine spezialisierte Version der generischen Klasse, indem er den Typ-Parameter `T` durch den Argument-Typ `Integer` ersetzt.
- In Java funktioniert es intern nicht so, aber für das Verständnis der korrekten Typisierung kann man es sich so vorstellen.

  Was Parameter ausmacht, ist schließlich immer, dass man etwas Konkretes dafür einsetzen kann — hier eben Typen für Typ-Parameter.

Grundlagen, Erstes Beispiel (5)

- Ersetzt man in der generischen Definition `T` durch `Integer` und `Container<T>` durch `ContainerInteger`, so ergibt sich:

```
 (1)  class ContainerInteger {
 (2)     private Integer attr;
 (3)     Container(Integer a) {
 (4)        attr = a;
 (5)     }
 (6)
 (7)     Integer getAttr() {
 (8)        return attr;
 (9)     }
(10)
(11)     void setAttr(Integer a) {
(12)        attr = a;
(13)     }
(14)  }
```

Grundlagen, Erstes Beispiel (6)

- Das Ergebnis der Ersetzung muss korrekt typisiert sein, egal welchen Referenztyp man für den Typ-Parameter einsetzt.

  Das wäre in C++ anders: Dort müssen nur die tatsächlich eingesetzten Typen zu korrektem Code führen. Deswegen muss C++ jedes Mal neu übersetzen, Java übersetzt die generische Klasse dagegen nur ein Mal.

- Man kann also, wenn `x` eine Variable vom Typ `T` ist, nur die in `Object` definierten Methoden für `x` aufrufen (diese hätte ja auch jeder andere Referenztyp).

  Bei Bedarf gibt es aber die beschränkten Typ-Variablen (s.u.). Dann kann man nur Untertypen eines bestimmten Typs einsetzen, und entsprechend die für diesen Typ definierten Methoden in der generischen Klasse verwenden.

- In Java kann man keine primitiven Typen für die Typ-Parameter einsetzen.

  Wegen der Wrapper-Klassen ist das keine besondere Einschränkung.

Grundlagen, Erstes Beispiel (7)

- Beispiel-Anwendung der generischen Klasse mit zwei unterschiedlichen konkreten Typen:

```
 (1)  class ContainerTest {
 (2)      static public void main(String[] args){
 (3)          Container<Integer> x =
 (4)                  new Container<Integer>(1);
 (5)          Container<String> y =
 (6)                  new Container<String>("abc");
 (7)          int i = x.getAttr();
 (8)          String s = y.getAttr();
 (9)          x.setAttr(2);
(10)          y.setAttr("def");
(11)          System.out.println(x.getAttr());
(12)          System.out.println(y.getAttr());
(13)      }
(14)  }
```

Interne Funktionsweise: Roher Typ (1)

- Intern wird einfach `Object` für den Typ-Parameter eingesetzt:

```
 (1)  class Container {
 (2)     private Object attr;
 (3)     Container(Object a) {
 (4)        attr = a;
 (5)     }
 (6)
 (7)     Object getAttr() {
 (8)        return attr;
 (9)     }
(10)
(11)     void setAttr(Object a) {
(12)        attr = a;
(13)     }
(14)  }
```

Interne Funktionsweise: Roher Typ (2)

- Die Klasse `Container` (ohne Typ-Parameter) gibt es wirklich, es ist der sogenannte "Rohe Typ" (engl. "Raw Type"), der zu dem generischen Typ `Container<T>` gehört.
- Dies wurde vorgesehen, um eine nachträgliche "Generifizierung" der Bibliothek zu erlauben:
  - Als die generischen Typen eingeführt wurden, gab es schon viele Programme, die z.B. `List` (aus `java.util`) benutzten.
  - Die Bibliothek wurde umgestellt, so dass sie nun den generischen Typ `List<T>` definiert.
  - Alte Programme können aber weiter `List` benutzen.

    Sie bekommen dann aber eine Warnung (die man bei Bedarf abschalten kann). Genauer führt nicht die Verwendung des "Raw Types" selbst zur Warnung, aber alle hinsichtlich der Typ-Prüfung unsicheren Operationen.

Interne Funktionsweise: Roher Typ (3)

- Wenn (wie im Testprogramm) `x` ein `Container<Integer>` ist, und `y` ein `Container<String>`, würde die Zuweisung

  ```
  x = y;
  ```

  natürlich einen Typfehler liefern ("incompatible types").

- Unter Verwendung des "rohen Typs" kann man genau dies aber erreichen:

  ```
  Container z = x; // ok
  y = z; // Gibt eine "unchecked" Warnung!
  ```

- Bei Warnungen wird im Unterschied zu Fehlermeldungen Code erzeugt, man kann das Programm also ausführen.

  Warnungen kann man abschalten, Fehlermeldungen nicht. Allerdings sollte man die Warnung erst genau verstehen, bevor man sie abschaltet: Man ist dann selber dafür verantwortlich, dass es trotz Warnung korrekt ist. Andernfalls wird man viel Zeit mit der Fehlersuche verlieren.

Interne Funktionsweise: Roher Typ (4)

- Der Compiler kann also nicht sicher sein, dass seine Information über den tatsächlichen Argument-Typ stimmt.
- Deswegen fügt er überall Casts ein, wo die Zuweisung mit `Object` sonst inkorrekt wäre (also die "Down-Casts").
- Z.B. wird die Zeile (8) aus dem Testprogramm intern zu

```
(8)         String s = (String) y.getAttr();
```

  "`javap -s ContainerTest`" zeigt den entsprechenden "`checkcast`"-Befehl an.

- Solange man die "Raw Types" nicht verwendet, kann man hier keine "`ClassCastException`" bekommen.

  Der Laufzeit-Test ist dann überflüssig. So wird aber auch in unübersichtlichen Situationen (mit "Raw Types") die Integrität der virtuellen Maschine geschützt: Man kann niemals auf nicht existierende Attribute zugreifen (sonst könnte man mit ungültigen Offsets beliebig in den Speicher schreiben).

Einschränkungen (1)

- Wie oben schon erwähnt, gibt es Einschränkungen bei der Verwendung eines Typ-Parameters `T` (weil die genaue Typ-Information zur Laufzeit nicht zur Verfügung steht):
  - Man kann kein Objekt vom Typ `T` erzeugen, d.h. `new T(...)` ist verboten.

    Neben den oben (Folie 15) schon genannten Gründen kommt noch hinzu, dass man für den Typ-Parameter auch einen Interface-Typ einsetzen könnte. Dafür könnte man sicher nicht `new` benutzen. Allerdings gilt die Einschränkung auch, wenn man den Parameter mit einer Oberklasse beschränkt (s.u.), und so Interface-Typen ausschließt.

  - Man kann auch kein Array vom Typ `T[]` erzeugen.

    Hier wäre ein Interface-Typ kein Problem. Der Grund ist vielmehr, dass im Array der Element-Typ `T` vermerkt werden muss, um ggf. eine `ArrayStoreException` erzeugen zu können.

Einschränkungen (2)

- Einschränkungen in der Verwendung von Typ-Parametern:
  - Weil es nur eine Klasse gibt, kann der Parameter-Typ `T` nicht als Typ von statischen Attributen verwendet werden.

    Es gibt die Variable ja nur ein Mal, und nicht eine eigene Version für jeden Wert des Parameter-Typs `T`. Das wäre verwirrend für viele Programmierer, die sich `List<Person>` und `List<Integer>` als unterschiedliche Klassen denken, und möglichst selten merken sollen, dass es nicht so ist. Wenn man versucht, eine statische Variable mit dem Parameter-Typ `T` anzulegen, bekommt man die Fehlermeldung "non-static class T cannot be referenced from a static context".

  - Entsprechend kann `T` auch nicht als Argument- oder Resultat-Typ von statischen Methoden verwendet werden.

    Allerdings können statische Methoden eigene Typ-Parameter haben (s.u.), dann ist diese Einschränkung nicht so schwer.

Einschränkungen (3)

- Einschränkungen in der Verwendung von Typ-Parametern:
  - Ein Cast `(T) obj` kann nicht wirklich überwacht werden, man bekommt dann eine "unchecked cast" Warnung.

    Genauer bekommt man zuerst den Hinweis ("Note"), man möchte es doch bitte mit "`javac -Xlint:unchecked Datei.java`" übersetzen (d.h. diese Warnung anschalten). Die Warnung ist insofern wichtig, als normalerweise bei "Down Casts" zur Laufzeit überprüft wird, ob das Objekt auch wirklich zur gewünschten Klasse gehört, und andernfalls die "`ClassCastException`" erzeugt wird. Das geschieht hier nicht, der Compiler verläßt sich auf den Programmierer. Dadurch kann ein Attribut vom Parameter-Typ `T` zur Laufzeit auch ein Objekt enthalten, das gar nicht vom Typ `T` ist. Man kann es ausdrucken oder mit `getClass()` den Typ abfragen, und so bemerken, dass der Typ nicht stimmt. Intern wird `T` durch `Object` ersetzt, und da passt es. Speichert man das Objekt aber in eine Variable, die explizit den gewünschten Typ hat, findet ein überprüfter Cast statt, und es gibt die `ClassCastException`.

Einschränkungen (4)

- Einschränkungen in der Verwendung von Typ-Parametern:
  - Mit `instanceof` kann man nicht auf den Parameter-Typ `T` oder den generischen Typ `Container<T>` testen.

    Das gibt keine Warnung, sondern einen echten Fehler, denn der Compiler kann dafür einfach keinen Maschinencode erzeugen: Er hat zur Laufzeit nicht die Information, was der Typ `T` ist.

  - Eine Methode kann nicht für verschiedene Instanziierungen des gleichen generischen Typs überladen werden.

    Intern ist es ja der gleiche Typ.

  - Von einem parametrisierten Typ, z.B. `Container<Integer>`, als Oberklasse erbt man Methoden mit `Object`-Parametern.

    Der Compiler erzeugt automatisch "Brückenmethoden", so dass das Überschreiben wie erwartet funktioniert. Allerdings werden manche Typfehler dann erst zur Laufzeit erkannt, und nicht vom Compiler.

Beispiel: Verkettete Liste (1)

- Es soll jetzt eine Liste `List<T>` für einen beliebigen Element-Typ `T` programmiert werden.
- Dazu muss man zuerst die Schnittstelle klären: Welche Methoden soll die Liste anbieten?
- Zur Vereinfachung werden hier nur zwei Methoden betrachtet:
  - `add(e)`, um einen Wert `e` vom Typ `T` in die Liste einzutragen (hinten, also am Ende der Liste),
  - `printAll()`, um die ganze Liste auszudrucken.

    Dabei wird implizit die Methode `toString()` des Element-Typs `T` benutzt.
- Normalerweise würde man mindestens noch eine Möglichkeit zum Durchlauf durch die Liste bzw. zum Zugriff auf einzelne Elemente brauchen.

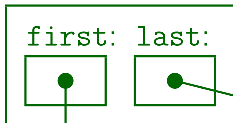
Beispiel: Verkettete Liste (2)

- Die Liste soll als verkettete Liste implementiert werden, d.h. sie besteht aus einzelnen Knoten, die
  - jeweils ein Element der Liste speichern, und
  - eine Referenz auf das jeweils nächste Element enthalten (`null` am Ende der Liste).
- Es wird daher noch eine zweite generische Klasse `Node<T>` für die Knoten der Liste benötigt:
  - Dem Konstruktor wird der zu speichernde Wert übergeben,

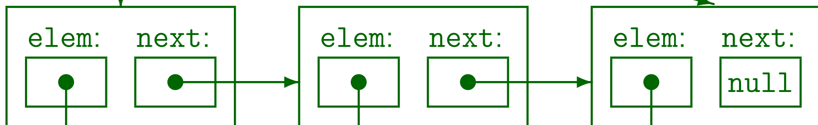
    Der Verkettungs-Zeiger wird zunächst auf `null` gesetzt.
  - Es gibt Lesezugriff auf beide Komponenten (`get`-Methoden).
  - Schreibzugriff nur auf den Verkettungszeiger (`set`-Methode).

Beispiel: Verkettete Liste (3)

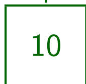
List<T>:

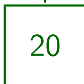


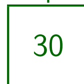
Node<T>:



T:







Beispiel: Verkettete Liste (4)

```
(1)  class Node<T> {
(2)
(3)      // Attribute:
(4)      private T elem;
(5)      private Node<T> next;
(6)
(7)      // Konstruktor:
(8)      Node(T e) {
(9)          elem = e;
(10)         next = null;
(11)     }
(12)
(13)     // Lese-Zugriff fuer Element:
(14)     T getElem() {
(15)         return elem;
(16)     }
(17)
```

Beispiel: Verkettete Liste (5)

```
(18)      // Lese-Zugriff fuer Verkettungszeiger:
(19)      Node<T> getNext() {
(20)         return next;
(21)      }
(22)
(23)      // Schreib-Zugriff fuer Verkettungszeiger:
(24)      void setNext(Node<T> n) {
(25)         next = n;
(26)      }
(27)   }
```

- Selbstverständlich kann auch der parametrisierte Typ selbst benutzt werden (für Attribute, und als Argument- und Resultat-Typ nicht statischer Methoden).

Beispiel: Verkettete Liste (6)

```
 (1)  class List<T> {
 (2)
 (3)      // Attribute:
 (4)      private Node<T> first; // Erster Knoten
 (5)      private Node<T> last;  // Letzter Knoten
 (6)
 (7)      // Konstruktor:
 (8)      public List() {
 (9)          // Leere Liste enthaelt keine Knoten:
(10)          first = null;
(11)          last = null;
(12)      }
(13)
```

- Selbstverständlich kann man auch andere generische Klassen verwenden, und z.B. den Typ-Parameter weiterreichen.

Beispiel: Verkettete Liste (7)

```
(14)     // Eintrag an die Liste anhaengen:
(15)     public void add(T e) {
(16)        Node<T> n = new Node<T>(e);
(17)        if(first == null) { // Liste ist leer
(18)           assert last == null;
(19)           first = n;
(20)           last = n;
(21)        }
(22)        else { // Liste ist nicht leer
(23)           assert last != null;
(24)           last.setNext(n);
(25)           last = n;
(26)        }
(27)     }
```

- Hier wird u.a. ein Objekt einer generischen Klasse erzeugt (ein neuer Knoten).

Beispiel: Verkettete Liste (8)

```
(28)
(29)      // Ganze Liste ausgeben:
(30)      public void printAll() {
(31)         for(Node<T> n = first; n != null;
(32)                n = n.getNext())
(33)             System.out.println(n.getElem());
(34)      }
(35)  }
```

- Diese Schleife ist typisch zum Durchlaufen einer verketteten Liste.
- Bei der Ausgabe wird die `toString()`-Methode des Element-Typs benutzt.

  Weil die Methode in `Object` definiert ist, hat jeder mögliche Parameter-Typ so eine Methode.

Beispiel: Verkettete Liste (9)

```
 (1)  class ListTest {
 (2)
 (3)     public static void main(String[] args) {
 (4)        List<Integer> x = new List<Integer>();
 (5)        x.add(10);
 (6)        x.add(20);
 (7)        x.add(30);
 (8)        x.printAll();
 (9)     }
(10)  }
```

- Es wird eine konkrete Liste mit `Integer`-Elementen angelegt.
- Durch das Auto-Boxing kann man der `add(...)`-Methode auch ganze Zahlen übergeben, sie werden automatisch in `Integer`-Objekte umgewandelt.

Beispiel mit Interface (1)

- Es gibt viele Arten, die oben angegebene Funktionalität einer Liste zu implementieren.
- Z.B. könnte man intern auch ein Array verwenden.
- Daher bietet es sich an, die Schnittstelle von der Implementierung zu unterscheiden, und die Funktionalität in einem generischen Interface zu definieren:

```
(1)  interface List<T> {
(2)
(3)      // Eintrag an die Liste anhaengen:
(4)      void add(T e);
(5)
(6)      // Ganze Liste ausgeben:
(7)      void printAll();
(8)  }
```

Beispiel mit Interface (2)

- Die Klasse mit der obigen Implementierung als verkettete Liste soll dann `LinkedList<T>` heißen:

```
(1)  class LinkedList<T> implements List<T> {
(2)
(3)      ... // Programmcode wie oben
(4)      ... // Konstruktor muss natuerlich
(5)      ... // LinkedList heissen
```

- Im Testprogramm ändert sich nur die Zeile mit der Objekt-Erzeugung:

```
(4)  List<Integer> x = new LinkedList<Integer>();
```

- Nun ist also nur in dieser einen Zeile festgelegt, dass die Implementierung einer Liste als verkettete Liste benutzt wird.

Listen-Implementierung mit Array (1)

- Will man die Liste mit einem Array implementieren, besteht das Problem, dass man kein Array vom Parametertyp (also `T[]`) anlegen kann.

  Im Array muss der Typ des Arrays gespeichert werden, dieser steht aber zur Laufzeit nicht zur Verfügung.

- Man benutzt daher ein Array vom Typ `Object[]`.

  Bei Attributen wird der Parameter-Typ `T` ja auch intern durch `Object` ersetzt.

- Mit den hier betrachteten Methoden ergibt das kein Problem, denn der Cast von `T` nach `Object` ist ein "Up-Cast".

- Bei einer realen Listen-Implementierung muss man aber auch ein Array-Element von `Object` nach `T` casten.

  Z.B. bei einer Methode `get(int i)`, die das `i`-te Element der Liste zurückgibt. Diese Methode hat natürlich den Ergebnis-Typ `T`.

Listen-Implementierung mit Array (2)

- Wie oben erläutert, führt der Cast von `Object` nach `T` zu einer "unchecked" Warnung, weil zur Laufzeit der Typ des Objektes nicht an dieser Stelle geprüft werden kann.

  Er wird später geprüft, wenn das Objekt z.B. in einer Variable gespeichert wird, die nicht mehr mit `T`, sondern explizit mit dem für den Parameter eingesetzten Typ deklariert ist (z.B. `Integer`).

- Diese Warnung kann man durch Voranstellen der Annotation

  ```
  @SuppressWarnings("unchecked")
  ```

  abschalten.

  Man kann es z.B. vor die ganze Methodendeklaration schreiben.

- Es gibt keine schöne Lösung für dieses Problem.

  So auch im Quellcode der openjdk Bibliothek für [`java.util.ArrayList`].

- Den Anwender der Klasse betrifft das nicht weiter.

Listen-Implementierung mit Array (3)

```
(1)  class ArrayList<T> implements List<T> {
(2)
(3)      // Array zur Speicherung der Elemente:
(4)      private Object[] arr;
(5)
(6)      // Aktuelle Anzahl Elemente im Array:
(7)      private int numElems;
(8)
(9)      // Konstruktor:
(10)     public ArrayList() {
(11)         // Initiale Kapazitaet: 10
(12)         arr = new Object[10];
(13)         numElems = 0;
(14)     }
(15)
```

Listen-Implementierung mit Array (4)

```
(16)     // Eintrag an die Liste anhaengen:
(17)     public void add(T e) {
(18)
(19)         // Falls noetig, groesseres Array
(20)         // anfordern und umkopieren:
(21)         if(arr.length == numElems) {
(22)             int newLength = arr.length * 2;
(23)             Object[] newArr =
(24)                 new Object[newLength];
(25)             for(int i = 0; i < arr.length; i++)
(26)                 newArr[i] = arr[i];
(27)             arr = newArr;
(28)         }
(29)
(30)         // Nun Element eintragen:
(31)         arr[numElems++] = e;
(32)     }
```

Listen-Implementierung mit Array (5)

```
(33)
(34)     // Ganze Liste ausgeben:
(35)     public void printAll() {
(36)         for(int i = 0; i < numElems; i++)
(37)             System.out.println(arr[i]);
(38)     }
(39)
(40)     // i-tes Element der Liste liefern:
(41)     @SuppressWarnings("unchecked")
(42)     public T get(int i) {
(43)         if(i < 0 || i >= numElems)
(44)             throw new
(45)                 IndexOutOfBoundsException();
(46)         // "unchecked" Warnung unterdrueckt:
(47)         return (T) arr[i];
(48)     }
(49) }
```

Inhalt

1 Einleitung

2 Generische Typen

3 Typ-Beschränkungen

4 Generische Methoden

5 Wildcards

Typ-Beschränkungen für Parameter (1)

- Manchmal sollen für den Typ-Parameter nicht beliebige Typen eingesetzt werden können, weil man eine gewisse Funktionalität von dem Typ benötigt.
- Wenn man z.B. sortierte Listen programmieren will, braucht man, dass der Element-Typ eine Anordnung hat, also die Methode `compareTo(...)` zur Verfügung stellt.
- Man kann daher den Typ-Parameter einschränken auf Subtypen von Klassen oder Interfaces.

```
class SortedList<T extends Comparable<T>> { ...
```

  Optimal wäre (s.u.): `class SortedList<T extends Comparable<? super T>>`

- Das Schlüsselwort `extends` wird benutzt, egal ob es sich um eine Klasse oder ein Interface handelt.

Typ-Beschränkungen für Parameter (2)

- Nun wird intern `Comparable` statt `Object` für Variablen und Parameter des Typ-Parameters benutzt.
- Damit kann man die Methode `compareTo(...)` aufrufen für Objekte vom Typ `T`.
- Es ist auch möglich, mehrere Obertypen nach `extends` anzugeben:

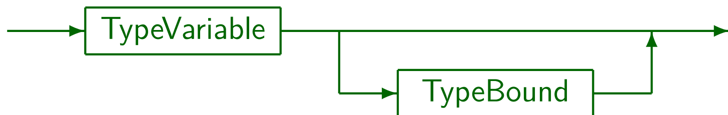
  ```
  class C<T extends T1 & T2 & T3> { ... }
  ```

  Nur eine der Schranken darf eine Klasse sein, und diese muss zuerst angegeben werden (T1).
  Die übrigen Schranken sind dann Interfaces.

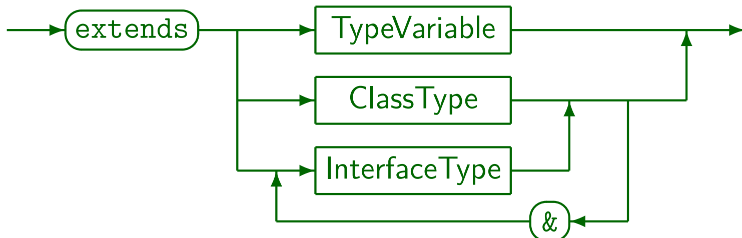
  So könnte das System eine Klasse konstruieren, die die Anforderungen an den Parameter beschreibt: Sie hat T1 als Oberklasse und implementiert T2 und T3. Tatsächlich wird aber intern T1 für den Parameter T eingesetzt.

Syntax: Deklaration von Parametern

- TypeParameter:



- TypeBound:



Inhalt

1. Einleitung
2. Generische Typen
3. Typ-Beschränkungen
4. Generische Methoden
5. Wildcards

Methoden mit Typ-Parametern (1)

- Es ist auch möglich, generische Methoden-Deklarationen zu schreiben. Insbesondere kann man auch Typ-Parameter für statische Methoden einführen:

```
public static <T extends Comparable<T>>
              T max(T x, T y) {
                  if(x.compareTo(y) >= 0)
                      return x;
                  else
                      return y;
              }
```

- Der Typ-Parameter muss vor dem Ergebnis-Typ der Methode in <...> deklariert werden.

  Auch Konstruktoren können auf diese Art generisch gemacht werden, dort steht das <...> entsprechend direkt vor dem Namen des Konstruktors.

Methoden mit Typ-Parametern (2)

- Typ-Parameter für statische Methoden sind interessant, weil die bisher besprochenen Typ-Parameter für Klassen in statischen Methoden nicht benutzt werden können.

  Man kann aber Typ-Parameter selbstverständlich auch für normale Methoden und Konstruktoren benutzen.

- Wenn man in einer generischen Klassen-Deklaration mit Typ-Parameter `T` einen Typ-Parameter `T` für eine Methode einführt, wird der äußere Typ-Parameter verschattet.

  Es ist eine Stilfrage, ob die zwei gleich benannten Parameter hier eher verwirren, oder unterstreichen, dass man den gleichen Typ einsetzen will.

Methoden mit Typ-Parametern (3)

- Beim Aufruf einer solchen generischen Methode kann man einen konkreten Typ angeben:

```
int m = MyClass.<Integer>max(1,2);
```

- Das ist relativ umständlich. Deswegen kann der Compiler in den meisten Fällen selbst einen passenden Typ bestimmen, der für den Typ-Parameter einzusetzen ist.
- Oben kann man `<...>` weglassen. Entsprechend geht auch:

```
System.out.println(MyClass.max(1,2));
```

- Es wird der spezifischste Typ genommen, der möglich ist, in diesen Beispielen also `Integer`.

  Der primitive Typ `int` geht natürlich nicht, weil für die Parameter nur Referenztypen eingesetzt werden können. Es findet dann Autoboxing statt.

Methoden mit Typ-Parametern (4)

- Diese automatische Bestimmung von konkreten Typen für Typ-Parameter heißt Typ-Inferenz (engl. “Type Inference”).
- Es werden dabei nicht nur die Typen der Argumente berücksichtigt, sondern auch der Ziel-Typ (“Target Type”).
- Z.B. ist folgende Methodendeklaration möglich:

```
static <T> T nil() { return null; }
```

- Beim Methodenaufruf

```
Integer iref = MyClass.nil();
```

  erkennt der Compiler aus der linken Seite der Zuweisung, dass für `T` hier `Integer` eingesetzt werden muss.

  Dies ist aber begrenzt. Wenn z.B. die Methode `f` ein Argument vom Typ `Integer` hat, führt der Aufruf `f(MyClass.nil())` zu einer Fehlermeldung, weil für `T` einfach `Object` eingesetzt wird. Bei Java 8 soll dies funktionieren.

Diamant-Operator

- Seit Java 7 ist Folgendes möglich:

```
List<Integer> l = new ArrayList<>();
```

- Hier kann der Typ-Parameter für den Konstruktor-Aufruf auch aus der linken Seite geschlossen werden.
- Vor Java 7 musste man den Typ-Parameter zwei Mal angeben:

```
List<Integer> l = new ArrayList<Integer>();
```

- “`<>`” heißt “Diamant” oder “Diamant-Operator”.
- Wenn man den Diamant auch weglässt, bekommt man den “rohen Typ”, und damit eine “unchecked” Warnung:

```
List<Integer> l = new ArrayList();
```

Inhalt

1. Einleitung
2. Generische Typen
3. Typ-Beschränkungen
4. Generische Methoden
5. Wildcards

Wildcards (1)

- Subtyp-Beziehungen übertragen sich nicht auf die generischen Klassen:
  - Z.B. ist `List<Integer>` kein Subtyp von `List<Number>`, obwohl `Integer` ein Subklasse von `Number` ist.
  - Würden solche Subtyp-Beziehung gelten, würde folgender Fehler vom Compiler nicht bemerkt:

```
LinkedList<Number> x = new LinkedList<Integer>();
    // Diese Zuweisung gibt einen Typfehler!
x.add(new Double(1.5));
```

    Nun würde ein `Double`-Objekt in einer `Integer`-Liste stehen.

    Bei Arrays gilt die entsprechende Subtyp-Beziehung, und da kann man ein `Integer[]`-Array an eine `Number[]`-Variable zuweisen. Dann muss aber jede Eintragung eines Wertes in das Array (entsprechend der zweiten Anweisung) zur Laufzeit geprüft werden. Bei generischen Klassen fehlt aber diese Typ-Information zur Laufzeit.

Wildcards (2)

- Wenn man jetzt eine Methode schreiben will, die allgemein Listen von Zahlen aufsummiert, ist das ein Problem.
- Deklariert man sie als

  ```
  static double sum(List<Number> list)
  ```

  kann man nicht eine `List<Integer>` übergeben.
- Als Lösung wurden "Wildcards" eingeführt:

  ```
  static double sum(List<? extends Number> list)
  ```

- Nun kann man einen Parameter vom Typ `List<T>` übergeben, wobei `T` eine Unterklasse von `Number` ist.
- Z.B. sind `List<Integer>` und `List<Double>` beides Untertypen von `List<? extends Number>`.

  Wie oben erläutert, sind sie dagegen nicht Untertypen von `List<Number>`.

Wildcards (3)

- Eine unbeschränkte Wildcard steht für einen beliebigen Typ, z.B. ist `List<?>` eine Liste von irgendeinem (unbekannten) Typ.

  Dies ist sehr ähnlich zum "rohen Typ" `List`. Aber jetzt weiß der Compiler, dass man die neuen Konstrukte für generische Typen verwendet, und gibt keine "unchecked" Warnungen mehr, sondern ggf. Fehlermeldungen.

- In eine Liste vom Typ `List<? extends T>` kann man kein Element einfügen, weil der tatsächliche Element-Typ ja ein beliebiger (unbekannter) Subtyp von `T` sein könnte.

  Z.B. kann man in eine Liste vom Typ `List<? extends Number>` kein `Integer`-Objekt einfügen, weil es ja auch eine Liste vom Typ `List<Double>` sein könnte.

- Man kann aber Elemente der Liste abfragen, und in eine Variable vom Typ `T` speichern.

Wildcards (4)

- Es gibt auch die Beschränkung nach unten: `List<? super T>` ist eine Liste von irgendeiner Oberklasse von `T`.

  Man kann nur eine von beiden Beschränkungen gleichzeitig verwenden.

- In eine solche Liste kann man ein Objekt vom Typ `T` einfügen: Der tatsächliche Elementtyp der Liste ist ja allgemeiner.

  Eine Unterklasse von `T` geht natürlich auch.

- Man kann nun aber kein Objekt mehr entnehmen.

  Bzw. nur als `Object`.

- Wenn man `compareTo` für den Elementtyp `T` braucht, ist die richtige Deklaration:

  ```
  List<T extends Comparable<? super T>>
  ```

  Es schadet ja nichts, wenn die Methode `compareTo` auch für allgemeinere Typen deklariert ist als der Typ `T`, den man braucht.

Wildcards (5)

- Wenn `U` ein Untertyp von `O` ist, ist auch `List<? extends U>` ein Untertyp von `List<? extends O>`.

  Jeder konkrete Typ `T`, für den `T extends U` gilt, gilt auch `T extends O`.

  Daher sind die möglichen Element-Typen von `List<? extends U>` eine Teilmenge der Elementtypen von `List<? extents O>`.

- Wenn sich die Subklassen-Beziehung vom Eingabetyp auf den Ergebnistyp überträgt, heißt der Typkonstruktor covariant.
- `List<? super U>` ist ein Obertyp von `List<? super O>`.
- Erhält man die umgekehrte Subklassen-Beziehung für den Ergebnis-Typ, heißt der Typkonstruktor contravariant.
- Zwischen `List<U>` und `List<O>` besteht gar keine Subklassen-Beziehung. Dieser Typkonstruktor ist invariant.

Wildcards (6)

- Manchmal muss man eine explizite Typ-Variable für eine Wildcard einführen.

  Jedes Vorkommen von "?" ist potentiell ein anderer Typ. Wenn man z.B. eine Variable vom Element-Typ eines `List<?>`-Parameters deklarieren will, geht das nicht direkt. Mit folgendem Trick aber doch.

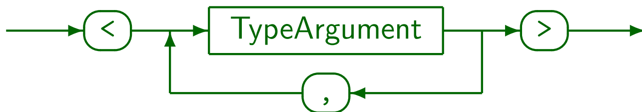
- Dazu kann man eine generische Hilfsmethode aufrufen, die eine neue Variable für die Wildcard einführt. Wenn z.B.

  ```
  static <T> void helper(List<T> list) { ... }
  ```

  deklariert ist, kann man diese Methode auch mit Argument vom Typ `List<?>` aufrufen.

Syntax: Angabe von Argumenten

- TypeArguments:



- TypeArgument:

- Wildcard: