Deze zomer zal de 1.5-versie van de Java 2 Standard Edition beschikbaar komen. J2SE 1.5 zal ondersteuning bieden voor generieke types en methods. Deze nieuwe feature van de Java-taal, die bekend is onder de naam Java Generics, is een belangrijke aanvulling op de kern van de taal. In dit artikel (het eerste uit een tweedelige serie) geven Angelika Langer en Klaus Kreft een overzicht van deze nieuwe feature. Zij baseerden zich daarbij op de laatste draft van de specificatie, gepubliceerd in juli 2003, en de bèta release van J2SE 1.5, die in februari 2004 beschikbaar kwam.[1]

thema

Java Generics: een introductie

Nieuwe features maken Java krachtiger

Laten we beginnen met een voorbeeld om het doel en de voordelen van generics toe te lichten. Stel dat ons programma een aantal input files moet verwerken. Hiertoe zou een lijst met bestandsnamen van de input files aangemaakt moeten worden. In de huidige niet-generieke Java-versie zou zo'n lijst van het type `List` zijn. Uit het type object kun je niet afleiden wat de lijst precies bevat. Je kunt zelfs niet vaststellen welk type objecten er in de lijst te vinden zijn; het zou van alles kunnen zijn.

Het type zelf verschaft weinig informatie. In generiek Java kun je de lijst met bestandsnamen met input files vastleggen met behulp van het type `List<InputFileName>`. Je zou de string die de bestandsnaam representeert kunnen inpakken in een class `InputFileName`, en vervolgens dat type gebruiken om het geparameteriseerde type `List<InputFileName>` te gebruiken. Je kunt dan door slechts te kijken naar het type object zien dat dit een lijst met input file bestandsnamen is.

Een naam als `List<InputFileName>` biedt veel voordelen. Het is niet alleen voor programmeurs gemakkelijker om Java source code te begrijpen, het stelt ook de compiler in staat die informatie te gebruiken en nuttige checks op typen uit te voeren. De compiler kan uit de type-informatie `List<InputFileName>` afleiden dat het object een lijst van input file bestandsnamen is, en de compiler weet ook zeker dat alleen objecten van het type `InputFileName` toegevoegd worden aan de lijst. Iedere poging een 'vreemd' element toe te voegen zou door de compiler worden geweigerd met een compile-time error message.

Wanneer een object uit deze lijst gekozen wordt, weet je dus gegarandeerd dat het object een input file bestandsnaam is. Er kan namelijk niets anders worden toegevoegd aan een `List<InputFileName>`. Ontelbare lelijke type casts, die we in niet-generieke Java nu nog dagelijks moeten uitvoeren, zijn in Java Generics niet meer nodig. We hoeven de compiler niet meer door middel van een cast te vertellen dat met het object van de lijst een input file name wordt bedoeld. De compiler weet namelijk al wat het is. Ook dit draagt bij aan de duidelijkheid en leesbaarheid van generieke Java source code.

Het grootste voordeel van Java Generics is zijn expressieve kracht. De bedoeling van Java Generics is om Java source code leesbaarder en begrijpelijker te maken en daardoor bij te dragen aan een hogere kwaliteit van de Java source code. Java Generics stelt bovendien de compiler in staat om vroege type checks uit te voeren (in tegenstelling tot runtime checks in een later stadium) die helpt om fouten in een zo vroeg mogelijk stadium op te sporen.

GESCHIEDENIS Het idee om generics toe te voegen aan Java is bijna zo oud als de taal zelf. Het specificatie-

1 Noch de besproken specificatie, noch de compiler implementatie zijn definitieve versies. Het is dus mogelijk dat er kleine verschillen zijn tussen de drafts en de definitieve versies, hoewel het onwaarschijnlijk is dat de in dit artikel besproken features nog zullen veranderen.

verzoek voor Java Generics is JSR 14 (zie /JSR14/) en het was één van de eerste, zoals blijkt uit het lage nummer. Al in 1999 werd het goedgekeurd. Direct na het ontstaan van de Java programmeertaal begonnen mensen te onderzoeken hoe de taal verbeterd kon worden door het toevoegen van geparameteriseerde types en methods ter versterking van het type system van de taal. Er vond research plaats (zie bijvoorbeeld /BRA/) en er werden verscheidene prototypen gebouwd (waaronder Pizza en NextGen). De resultaten en inzichten van deze vroege experimenten werden opgenomen in de specificatie van de huidige Java Generics.

Waar komt het idee voor Generics oorspronkelijk vandaan? De taal Java is geïnspireerd door andere programmeertalen. Oorspronkelijk werd Java beïnvloed door Smalltalk en C/C++. Deze talen waren erg populair op het moment dat Java werd uitgevonden. Op dit moment wordt Java duidelijk beïnvloed door C#. Generieke types en methods bestaan in vele talen: Ada heeft Generics, C++ heeft templates, en C# heeft Generics. Het gemeenschappelijke doel van deze kenmerken is om de respectievelijke type systemen van de talen te versterken.

Java is een hybride programmeertaal die elementen van sterke en zwakke typering in zich draagt. Zwakke typering betekent dat typen weinig informatie dragen en niet buitengewoon belangrijk zijn. Een voorbeeld van een Java-voorganger met zwakke typering is Smalltalk. Sterke typering daarentegen betekent dat een type juist heel veel informatie over objecten bevat. Sterke typering stelt de compiler in staat om zo vroeg mogelijk verscheidene checks uit te voeren, bijvoorbeeld om te controleren of twee objecten onderling compatibel zijn. Kunnen ze aan elkaar toegewezen worden? In een sterk getypeerde taal zal de compiler zo'n vraag onmiddellijk beantwoorden en direct een foutmelding afgeven. In een zwak getypeerde taal kan de compiler dat niet vaststellen. Het antwoord wordt pas duidelijk tijdens runtime wanneer de runtime systemen een runtime check uitvoeren en een exception melden. Java's sterk getypeerde voorganger is C++. Java bevat elementen van beide voorgangers en heeft dus ook kenmerken van zowel sterke als zwakke typering. Java Generics heeft als doel de sterke typering in de taal te verstevigen. Juist hier kunnen programmeurs voordeel halen uit de nieuwe feature.

> Het grootste voordeel van Java Generics is zijn expressieve kracht: de source code wordt leesbaarder en begrijpelijker

GEDETAILLEERD In alle talen vloeit de behoefte aan generieke types voort uit de implementatie en het gebruik van verzamelingen, zoals die in het Java collection framework. Doorgaans is de implementatie van een verzameling objecten onafhankelijk van het type objecten dat zich in de verzameling bevindt. Het heeft daarom geen zin om dezelfde datastructuur telkens opnieuw te her-implementeren, alleen maar omdat het mogelijk verschillende type elementen zou kunnen bevatten. In plaats daarvan is het de bedoeling om een enkele implementatie van de verzameling uit te voeren en die vervolgens te gebruiken om elementen van verschillende types vast te houden. Met andere woorden, in plaats van het implementeren van een class `IntList` en `StringList` voor het vasthouden van respectievelijk integraal waarden en strings, willen we één generieke implementatie `List` die in beide gevallen kan worden gebruikt.

In niet-generieke Java wordt dit soort generiek programmeren totnogtoe gerealiseerd door middel van `Object` referenties: een generieke lijst wordt geïmplementeerd als een verzameling `Object` referenties. Aangezien `Object` de superclass van alle classes is, kan de lijst met `Object`-referenties ook referenties bevatten naar elk ander type object. Alle collection classes in de Java platform library's gebruiken deze programmeertechniek om genericiteit te bereiken.

Een neveneffect van dit idioom is, dat we geen verzamelingen kunnen maken van waarden van een primitief type, zoals een lijst van integraalwaarden van het type `int`, omdat de primitieve types geen subclasses vormen van `Object`. Dat is nog geen grote restrictie omdat ieder primitief type een corresponderend referentie type heeft. In zo'n geval converteren we `ints` naar `Integers` voordat ze in een verzameling worden opgeslagen - een conversie die bekend staat als boxing en die ondersteund zal worden als een geautomatiseerde conversie (autoboxing) in JDK 1.5 (zie /BOX/).

Een Java-verzameling is erg flexibel; referenties naar elk type object kunnen erin worden opgeslagen. De verzameling hoeft zelfs niet eens homogeen te zijn (i.e. objecten van hetzelfde type te bevatten), maar het kan net zo goed heterogeen zijn, ofwel een mix van objecten van verschillende typen bevatten. Het gebruik van generieke Java verzamelingen is rechttoe rechtaan. Elementen worden toegevoegd door een referentie naar dat element toe te voegen aan de verzameling. Iedere

keer wanneer we een object uit een verzameling halen, ontvangen we een `Object` referentie. Voordat we de onttrokken elementen effectief kunnen gebruiken, moeten we de type-informatie over het element opnieuw opslaan.

Hiertoe stoppen we de teruggegeven `Object` referentie in het zogenaamde type van het element, zoals in het volgende voorbeeld:

```
LinkedList list = new LinkedList();
list.add(new Integer(0));
Integer i = (Integer) list.get(0);
```

We moeten de `Object` referentie, teruggegeven van method `get()`, casten naar type `Integer`. Deze cast is veilig omdat hij is gechecked in runtime. Als we zouden proberen om het te casten naar een type dat anders is dan dat van het geëxtraheerde element, dan zou een `ClassCastException` ontstaan, zoals in het onderstaande voorbeeld:

```
String s = (String) list.get(0); // is in orde
```

tijdens compile-time, maar mislukt tijdens runtime met een `ClassCastException`.

Het gebrek aan informatie over het element type in een verzameling en de resulterende behoefte aan ontelbare casts op alle plaatsen waar elementen worden geëxtraheerd uit een verzameling is de primaire drijfveer voor het toevoegen van geparameteriseerde typen aan de Java programmeertaal. Het idee is om de typen uit de verzameling te aan te vullen met informatie over het soort elementen dat zij bevatten.

In plaats van iedere verzameling te behandelen als een verzameling `Object` referenties, zouden we een onderscheid moeten maken tussen verzamelingen met referenties naar integers en verzamelingen met referenties naar strings. Een collection type zou een geparameteriseerd (of generiek) type worden dat een type parameter heeft, dat het soort element zou specificeren. Met een generieke lijst zou het hiervoor genoemde voorbeeld er als volgt uitzien:

```
LinkedList<Integer> list = new LinkedList
<Integer>();
list.add(new Integer(0));
Integer i = list.get(0);
```

Merk op, dat de `get()` method van een generieke lijst een referentie naar een object van een specifiek type retourneert, in ons voorbeeld van het type `Integer`, in welk geval de cast van `Object` naar `Integer` niet langer meer nodig is.

Ook zou het gebruik van het geëxtraheerde element - als dit van een ander type zou zijn - al tijdens het compileren onderschept worden, in plaats van tijdens runtime. Zo zou in het volgende voorbeeld niet gecompileerd kunnen worden:

```
String s = list.get(0); // compile-time
error
```

Op deze wijze verhoogt Java Generics de expressieve kracht en de type-veiligheid van de taal door middel van vroege statische checks, in plaats van late dynamische checks.

Java Generics voorziet niet alleen in geparameteriseerde collection types zoals gebruikt in het voorbeeld, het stelt ons ook in staat zelf generieke types te implementeren. Om te zien hoe we deze nieuwe feature voor onze eigen Java-programma's kunnen gebruiken, zullen we Java Generics grondiger gaan verkennen. In het navolgende zullen we kort kijken naar de syntax van de definitie van generieke types en naar andere taalkenmerken die met Java Generics te maken hebben.

GENERIEKE TYPES Codevoorbeeld 1 laat een voorbeeld zien van de definitie van verscheidene generieke types. De sample code toont ook schetsmatig een geparameteriseerde verzameling genaamd `LinkedList<A>`, diens superinterface `Collection<A>` en diens iterator type `Iterator<A>`. De typen in dit voorbeeld zijn geïnspireerd door de verzameling classes uit de library van het Java platform in package `java.util`.

```
interface Collection<A>  {
  public void add (A x);
  public Iterator<A> iterator ();
}

interface Iterator<A> {
  public A next ();
  public boolean hasNext ();
}

class NoSuchElementException extends RuntimeException f{}

class LinkedList<A> implements Collection<A> {
  protected class Node {
     A elt;
     Node next = null;
     Node (A elt) { this.elt = elt;  }
  }

  protected Node head = null, tail = null;
  public LinkedList () {}
  public void add (A elt) {
     if (head == null) {
        head = new Node(elt);
        tail = head;
```

```
    } else {
       tail.next = new Node(elt);
       tail = tail.next;
    }
  }
  public Iterator<A> iterator () {
    return new Iterator<A> () {
       protected Node ptr = head;
       public boolean hasNext () { return ptr != null; }
       public A next () {
          if (ptr != null) {
             A elt = ptr.elt;
             ptr = ptr.next;
             return elt;
          } else {
             throw new NoSuchElementException ();
          }
       }
    };
  }
}

final class Test {
  public static void main (String[ ] args) {
    LinkedList<String> ys = new LinkedList<String>();
    ys.add("zero"); ys.add("one");
    String y = ys.iterator().next();
  }
}
```

CODEVOORBEELD 1. Voorbeelden van geparameteriseerde typen

Geparameteriseerde typen hebben type parameters. In ons voorbeeld hebben zij precies één parameter, namelijk `A`. In het algemeen kan een geparameteriseerd type een onderhandelbaar aantal parameters bevatten. In ons voorbeeld staat parameter `A` voor het type van de elementen in de verzameling. Een parameter als `A` wordt ook wel een *type variabele* genoemd. Type variabelen kunnen worden beschouwd als plaatsvervangers, die later vervangen worden door een concreet type. Bijvoorbeeld, wanneer een instantiëring van het generieke type, zoals `LinkedList<String>`, wordt toegepast, zal `A` worden vervangen door `String`.

Verderop in dit artikel zullen we constateren dat er restricties zijn in het gebruik van type variabelen. We zullen constateren dat een type variabele niet zomaar kan worden gebruikt als een type. De analogie met een "plaatsvervanger voor een type" is niet volledig correct, maar slechts een benadering van wat een type variabele is. Voorlopig beschouwen we de type variabele echter nog even als een plaatsvervanger voor een type - het type elementen uit de verzameling in ons voorbeeld.

BOUNDS Voor de implementatie van een generieke lijst zoals in ons voorbeeld, is het nooit nodig een beroep te doen op enige method van het element type. Een lijst gebruikt slechts referenties naar de elementen, maar heeft nooit echt toegang tot die elementen. De lijst hoeft daarom niets te weten over het element type. Niet alle geparameteriseerde typen hebben zulke elementaire requirements voor hun element typen.

Stel dat we een hash-gebaseerde verzameling zouden willen implementeren, zoals een hash tabel. Een hash-gebaseerde verzameling moet de hash codes van de entry's calculeren. Echter, het element type is onbekend in de implementatie van een geparameteriseerde hash tabel. Slechts de type variabele die het element type vertegenwoordigt, is beschikbaar. Codevoorbeeld 2 laat een fragment zien van de implementatie van een geparameteriseerde hash tabel. Het is een geparameteriseerde class die twee type parameters heeft voor het key type en het associated value type.

```
public class Hashtable<Key, Data> {
  ...
  private static class Entry<Key, Data> {
     Key key;
     Data value;
     int hash;
     Entry<Key, Data> next;
     ...
  }
  private Entry<Key,Data>[] table;
  ...
  public Data get(Key key) {
     int hash = key.hashCode();
        for (Entry<Key,Data> e = table[hash &
        hashMask]; e != null; e = e.next) {
        if ((e.hash == hash) && e.key.equals
        (key)) {
           return e.value;
        }
     }
     return null;
  }
}
```

CODEVOORBEELD 2. Een geparameteriseerd type: een hash tabel

Zoals we kunnen zien, verplaatst de implementatie van de hash tabel niet alleen referenties naar de entry's, maar moet ook methods van de key type aanroepen, namelijk `hashCode()` en `equals()`. Beide methoden worden gedefinieerd in de class `Object`. De implementatie van de hash tabel vereist dus, dat de type variabelen `Key` en `Data` vervangen worden door concrete types die subtypes zijn van `Object`. Verderop in dit artikel zullen we zien dat dit altijd gegarandeerd is, omdat primitieve types verboden zijn als type argumenten voor Generics. Een concreet type dat een type variabele vervangt moet een referentie type zijn. Om deze reden kunnen we rustig aannemen dat de key type de vereiste methods heeft.

Wat nu als het nodig is om methods aan te roepen

die niet zijn gedefinieerd in de class `Object`? Overweeg dan de implementatie van een tree-based verzameling. Tree-based verzamelingen vereisen net als een `TreeMap` een sorteer-volgorde voor de elementen, die in die verzameling zitten. Element types kunnen in de sorteervolgorde voorzien door middel van de `compareTo()` method, die gedefinieerd is in de `Comparable` interface. De implementatie van een tree-based verzameling zou daarom de `compareTo()` method van het element type kunnen aanroepen. Codevoorbeeld 3 is een eerste poging tot een implementatie van een geparameteriseerde `TreeMap` verzameling.

```
public interface Comparable<T> {
  public int compareTo(T arg) ;
}

public class TreeMap<Key,Data>
{
     private static class Entry<K,V> {
        K key;
        V value;
        Entry<K,V> left = null;
        Entry<K,V> right = null;
        Entry<K,V> parent;
  }
  private transient Entry<Key,Data> root =
  null;
  ...

  private Entry< Key,Data > getEntry(Key key)
  {
     Entry<Key,Data> p = root;
     Key k = key;
     while (p != null) {
           int cmp =
           ((Comparable<Key>)k).compareTo
           (p.key);
        if (cmp == 0)
              return p;
        else if (cmp < 0)
              p = p.left;
        else
              p = p.right;
     }
     return null;
  }
  public boolean containsKey(Key key) {
        return getEntry(key) != null;
  }
  ...
}
```

CODEVOORBEELD 3. Implementatie van een tree-based verzameling - *zonder* bounds

De geparameteriseerde class `TreeMap` heeft twee type parameters `Key` en `Data`. Er worden geen requirements opgelegd aan één van deze type variabelen. Met deze implementatie zouden we een `TreeMap<X,Y>` kunnen aanmaken, zelfs als de key type `X` de `Comparable<X>` interface niet geïmplementeerd zou hebben en geen `compareTo()` method zou hebben. De aanroep van `compareTo()`, of (om precies te zijn) de cast van het key object naar het type `Comparable<Key>` voor het onvergelijkbare key type `X`, zou dan tijdens runtime mislukken met de melding `ClassCastException`.

```
public final class X { ... }
public final class Y { ... }

public final class Test {
  public static void main(String[] argv) {
     TreeMap<X,Y> tree = new TreeMap<X,Y>(); // compiles,
     although X is not Comparable
     ... add elements to the map ...
     X x = ... some key ...;
     tree.containsKey(x);  // fails runtime with a
     ClassCastException
  }
}
```

CODEVOORBEELD 4. Gebruik van de tree-based verzameling - *zonder* bounds

Om een vroege compile-time check mogelijk te maken, heeft Java Generics de taalfeature bounds: type variabelen van een geparameteriseerd type kan één of meerdere bounds hebben. Bounds zijn interfaces of superclasses die een type variabele moet implementeren of toevoegen. Als een geparameteriseerd type geïnstantieerd is met een concreet type argument dat niet de vereiste interface(s) of de vereiste superclass implementeert, dan zal de compiler deze overtreding van de requirements opmerken en een foutmelding afgeven.

In ons voorbeeld, zouden we de requirement zodanig kunnen formuleren dat de key type van onze `TreeMap` de interface `Comparable<Key>` moet implementeren door een bound voor de type variabele Key te specificeren. De aangepaste implementatie van `TreeMap` is te zien in Codevoorbeeld 5.

```
public class TreeMap<Key extends Comparable<Key>,Data> {
     static class Entry<K,V> {
        K key;
        V value;
        Entry<K,V> left = null;
        Entry<K,V> right = null;
        Entry<K,V> parent;
     }
     private transient Entry<Key,Data> root = null;
     ...
private Entry< Key,Data > getEntry(Key key) {
           Entry<Key,Data> p = root;
        Key k = key;
```

```
        while (p != null) {
              int cmp = k.compareTo(p.key);
           if (cmp == 0)
              return p;
           else if (cmp < 0)
              p = p.left;
           else
              p = p.right;
        }
        return null;
  }
  public boolean containsKey(Key key) {
        return getEntry(key) != null;
  }
  ...
}
```

CODEVOORBEELD 5. Implementatie van tree-based collection - *met* bounds

Nu doen we een poging een key type te gebruiken dat niet de `Comparable` interface implementeert, die afgewezen zal worden door de compiler, zoals in Codevoorbeeld 6.

Het primaire doel van bounds is om vroege compile-time checks mogelijk te maken.

- Methods van een type variabele *zonder bounds* kunnen alleen benaderd worden door de type variabele vorm te geven als een type dat de gewenste methods benoemt; zo'n cast zou tijdens runtime mislukken als het concrete type niet de gewenste methods heeft.
- Methods van een type variabele *met bounds* zijn direct toegankelijk (zonder enige casts) en de compiler zou al tijdens compile-time detecteren wanneer het concrete type niet de gewenste methods heeft.

Enkele aanvullende syntax details: een type variabele kan verschillende bounds bevatten. De syntax is: `TypeVariable implements` $\texttt{Bound}_1$ `&` $\texttt{Bound}_2$ `& ... &` $\texttt{Bound}_n$ Een voorbeeld:

```
final class Pair<A extends Comparable<A>
& Cloneable<A>,
           B extends Comparable<B> &
           Cloneable<B>>
     implements Comparable<Pair<A,B>>,
     Cloneable<Pair<A,B>> { ... }
```

Zoals dit voorbeeld duidelijk maakt, kunnen type variabelen verschijnen in hun bounds. Bijvoorbeeld: de type variabele A wordt gebruikt als type argument voor de geparameteriseerde interface `Comparable`, waarvan de instantiatie `Comparable<A>` een bound is van `A`. Er is een beperking gesteld aan bounds die instantiaties zijn van een geparameteriseerde interface: de verschillende bounds mogen geen instantiaties zijn van dezelfde geparameteriseerde interface. Het volgende is niet toegestaan:

```
class SomeType<T implements Comparable<T>
& Comparable<String> & Comparable<String
Buffer>>
{ ... }
```

Deze restrictie vloeit voor uit de manier waarop Java Generics worden geïmplementeerd. Verderop in dit artikel zullen we hier nader op ingaan. Classes kunnen ook bounds zijn. Het concrete type argument moet dan een subclass zijn van de bounding class of het mag dezelfde class zijn als de bounding class. Zelfs final classes zijn toegestaan als bounds. Bounding classes verschaffen, net als interfaces, toegang tot niet-statische methods die het concrete type argument overerft van zijn superclass. Bounding classes geven geen toegang tot constructors en statische methods. De bounding superclass moet in een lijst met bounds verschijnen als de eerste bound. De syntax voor de specificatie van bounds is dus: `TypeVariable implements Superclass &` $\texttt{Interface}_1$ `&` $\texttt{Interface}_2$ `& ... &` $\texttt{Interface}_n$

```
public final class X { ... }
public final class Y { ... }

public final class Test {
  public static void main(String[] argv) {
     TreeMap<X,Y> tree = new TreeMap<X,Y>();
     // compile-time error: type parameter X
     is not within its bound
     ... add elements to the map ...
     X x = ... some key ...;
     tree.containsKey(x);
  }
}
```

CODEVOORBEELD 6. Gebruik van de tree-based verzameling - *met* bounds

TAAL FEATURES In het tweede en laatste deel van deze serie, dat in de volgende editie van Java Magazine zal verschijnen, zullen we kort ingaan op een paar andere belangrijke taalfeatures van Java Generics: geparameteriseerde methods en wildcard instantiaties. Verder zullen we een aantal onderliggende principes van Java Generics onderzoeken, in het bijzonder de vertaling van geparameteriseerde types en methods naar Java bytecode.

Angelika Langer is onafhankelijk trainer en consultant, gespecialiseerd in objectgeoriënteerde software ontwikkeling in C++ en Java (e-mail: langer@camelot.de). Klaus Kreft is senior consultant en software architect voor Siemens Business Services (e-mail: klaus.kreft@siemens.com).