thema

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 het tweede en laatste deel van deze serie vervolgen Angelika Langer en Klaus Kreft hun bespreking van deze nieuwe feature. Zij baseerden zich daarbij op de laatste draft van de specificatie (zie /SPC), gepubliceerd in juli 2003, en de bèta release van J2SE 1.5 (zie /JSR15/), die in februari 2004 beschikbaar kwam.1

Java Generics: een introductie (2)

Nieuwe features maken Java krachtiger

In het eerste deel uit deze serie (zie Java Magazine 2004 nr. 1) bespraken we een paar belangrijke taalfeatures die met Java Generics te maken hebben: geparameteriseerde types, geparameteriseerde methods en gebounde types. We zullen in dit laatste deel kort ingaan op een paar andere belangrijke taalfeatures: generieke methods en wildcard instantiaties. De rest van dit artikel gebruiken we om een aantal onderliggende principes van Java Generics te onderzoeken, in het bijzonder de vertaling van geparameteriseerde types en methods naar Java bytecode.

GENERIEKE METHODS Niet alleen types kunnen worden geparameteriseerd. Naast generieke classes en interfaces, kunnen we generieke methods definiëren. Statische en niet-statische methods, evenals constructors kunnen geparameteriseerd worden op ongeveer dezelfde manier zoals we types geparameteriseerd hebben eerder in dit artikel. De syntax is enigszins anders, zie daarvoor Codevoorbeeld 1. Alles wat in dit artikel geschreven is over type variabelen van geparameteriseerde types kan op precies dezelfde manier worden toegepast op type variabelen van geparameteriseerde methods. Codevoorbeeld 1 laat een voorbeeld zien van een geparameteriseerde statische method `max()`.

Geparameteriseerde methods worden aangeroepen als reguliere niet-generieke methods. De type parameters worden afgeleid uit de context van de aanroep. In ons voorbeeld roept de compiler automatisch `max<Byte>()` aan. Het type inferentie algoritme is aanzienlijk complexer dan dit eenvoudige voorbeeld suggereert. Een uitputtende verhandeling over type inferentie valt buiten de scope van dit artikel.

WILDCARD INSTANTIATIES Voor de volledigheid zullen we nu kort het onderwerp wildcards aanstippen. Tot zover hebben we geparameteriseerde types geïnstantieerd door een concreet type te gebruiken dat de parameter van het type in de instantiatie vervangt. Daarnaast kunnen ook zogenaamde wildcards gebruikt worden om een geparameteriseerd type te instantiëren. Een wildcard instantiatie ziet er zo uit:

```
List<? extends Number> ref = new
LinkedList<Integer>();
```

In dit statement is `List<? extends Number>` een wildcard instantiatie, terwijl `LinkedList<Integer>` een reguliere instantiatie is.

Er zijn drie typen wildcards: "`? extends Type`", "`? super Type`" en "`?`". Iedere wildcard duidt een familie van types aan. "`? extends Number`" bijvoorbeeld is de familie van subtypes van het type Number, "`? super Integer`" is de familie van supertypes van type Integer, en "`?`" is de set van alle types. Hiermee corresponderend staat de wildcard instantiatie van een geparameteriseerd

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.

type voor een set instantiaties; e.g. `List<? extends Number>` verwijst naar de set van instantiaties van `List` voor types die subtypes zijn van `Number`.

Wildcard instantiaties kunnen gebruikt worden voor de benoeming van referentie variabelen, maar ze kunnen niet gebruikt worden voor het aanmaken van objecten. Toch kunnen referentie variabelen van een wildcard instantiatie type verwijzen naar een object van een compatibel type. Met compatibel worden in deze context de concrete instantiaties bedoeld uit de familie van instantiaties aangemeld door de wildcard instantiatie. Tot op zekere hoogte is dit vergelijkbaar met interfaces.

We kunnen namelijk ook geen objecten aanmaken van een interface type, maar een variabele van een interface type kan wel verwijzen naar een object van een compatibel type, waarbij "compatibel" in dit geval betekent: een type dat de interface implementeert. Evenzo kunnen we geen objecten van een wildcard instantiatie type creëren, maar een variabele van het wildcard instantiatie type kan wel verwijzen naar een object van een compatibel type, waarbij "compatibel" in dat geval een type aanduidt uit de corresponderende familie van instantiaties.

De toegang tot een object via een referentie variabele van een wildcard instantiatie type is beperkt. Via een wildcard instantiatie met "`extends`" moeten we geen methods aanroepen die argumenten neemt van het type dat door de wildcard wordt gerepresenteerd. Hier ziet u daarvan een voorbeeld:

```
List<? extends Number> list = new Linked-
List<Integer>();
list.add(new  Integer(25));  //  compile-
time error
```

De `add()` method van type `List` neemt een argument van het element type, welke de type parameter is van het geparameteriseerde `List` type. Via een wildcard instantiatie zoals `List<?` extends Number> is het niet toegestaan om de `add()` method aan te roepen. Vergelijkbare beperkingen zijn van toepassing op wildcards met "`super`": methods waarbij het return type hetzelfde type is als die de wildcard representeert, zijn verboden. Voor referentie variabelen met een "`?`" wildcard gelden beide restricties.

Dit korte overzicht van wildcard instantiaties is verre van volledig; een uitputtende behandeling van dit onderwerp valt buiten de scope van dit artikel. In de praktijk duiken wildcard instantiaties meestal op als argument of return types in method declaraties, en slechts zelden in de declaraties van variabelen. De meest nuttige wildcard is de "`extends`" wildcard. Voorbeelden van het gebruik van deze wildcard zijn te vinden in de J2SE 1.5 platform library's; een voorbeeld is de

```
interface Comparable<A> {
  public int compareTo (A that);
}

final class Byte implements Comparable<Byte> {
  private byte value;
  public Byte (byte value) { this.value = value; }
  public byte byteValue () { return value; }
  public int compareTo (Byte that) {
     return this.value - that.value;
  }
}

class Collections {
  public static <A extends Comparable<A>> A max
(Collection<A> xs) {
     Iterator<A> xi = xs.iterator();
     A w = xi.next();
     while (xi.hasNext()) {
        A x = xi.next();
        if (w.compareTo(x) < 0) w = x;
     }
     return w;
  }
}

final class Test {
  public static void main (String[ ] args) {
     LinkedList<Byte> byteList = new LinkedList<Byte>();
     byteList.add(new Byte((byte)0));
     byteList.add(new Byte((byte)1));
     Byte y = Collections.max(byteList);
  }
}
```

CODEVOORBEELD 1. Een geparameteriseerde method max().

method `boolean addAll(Collection<? extends ElementType> c)` van de class `java.util.List`. Hiermee is het mogelijk elementen toe te voegen aan een `List` van element type `ElementType`, waar de elementen uit een verzameling elementen van het subtype van `ElementType` worden gehaald.

PRINCIPES We hebben nu alle belangrijke taalfeatures besproken die met Java Generics te maken hebben. De rest van dit artikel gebruiken we om een aantal onderliggende principes van Java Generics te onderzoeken, in het bijzonder de vertaling van geparameteriseerde types en methods naar Java bytecode. Hoewel dit nogal technisch klinkt en vooral de aandacht vraagt van de bouwer van de compiler, draagt een beter begrip van deze principes wel degelijk bij aan een beter begrip van een aantal minder voor de hand liggende effecten van Java Generics.

IMPLEMENTATIE VAN JAVA GENERICS Hoe worden Java Generics geïmplementeerd? Wat doet de Java compiler met onze Java sourcecode die definities en

gebruiksvoorschriften bevat van geparameteriseerde types en methods? Welnu, zoals gebruikelijk vertaalt de Java compiler de Java source code naar Java byte code. In het hierna volgende, proberen we een kijkje te nemen onder de motorkap van het compilatieproces om de effecten en neveneffecten van Java Generics beter te begrijpen.

```
interface Collection {
  public void add (Object x);
  public Iterator iterator ();
}
interface Iterator {
  public Object next ();
 public boolean hasNext ();
}
class NoSuchElementException extends RuntimeException {}

class LinkedList implements Collection {
  protected class Node {
     Object elt;
     Node next = null;
     Node (Object elt) { this.elt = elt; }
  }
  protected Node head = null, tail = null;
  public LinkedList () {}
  public void add (Object elt) {
     if (head == null) {
        head = new Node(elt);
        tail = head;
     } else {
        tail.next = new Node(elt);
        tail = tail.next;
     }
  }
  public Iterator iterator () {
     return new Iterator () {
      protected Node ptr = head;
      public boolean hasNext () { return ptr != null; }
      public Object next () {
         if (ptr != null) {
            Object elt = ptr.elt;
            ptr = ptr.next;
            return elt;
         } else {
            throw new NoSuchElementException ();
         }
      }
     };
  }
}
final class Test {
  public static void main (String[ ] args) {
     LinkedList ys = new LinkedList();
     ys.add("zero"); ys.add("one");
     String y = (String)ys.iterator().next();
  }
}
```

CODEVOORBEELD 2. Geparameteriseerde types na type erasure

VERTALING VAN GENERICS Een compiler die een geparameteriseerd type of method moet vertalen (in welke taal dan ook), heeft in principe twee keuzemogelijkheden:

- *Code specialisatie*. De compiler genereert een nieuwe representatie voor iedere instantiatie van een geparameteriseerd type of method. Een compiler kan bijvoorbeeld code genereren voor een lijst met integers en daarnaast nog andere code voor een lijst met strings.
- *Code sharing*. De compiler genereert code voor slechts één representatie van een geparameteriseerd type of method en mapt alle concrete instantiaties van het generieke type of method naar een unieke representatie, en voert waar nodig type checks en type conversies uit.

Code specialisatie is de benadering die C++ gebruikt voor zijn templates. De C++ compiler genereert executable code voor iedere instantiatie van een template. Het nadeel van code specialisatie van generieke types is het risico dat de code veel te omvangrijk wordt. Zo zou een lijst integers en een lijst strings in de executable code gerepresenteerd kunnen worden als twee implementaties van twee verschillende types. Dit is vooral verspillend in het geval de elementen in een verzameling referenties (of pointers) zijn, omdat alle referenties (of pointers) van dezelfde omvang zijn en intern dezelfde representatie hebben. Het genereren van grotendeels identieke code voor een lijst referenties naar integers en een lijst referenties naar strings is eigenlijk niet nodig. Beide lijsten zouden intern kunnen worden gerepresenteerd door een lijst referenties naar ieder type of object. De compiler hoeft dan slechts een paar casts toe te voegen wanneer deze referenties heen en weer gaan tussen een generiek type of methode. Aangezien in Java de meeste types referentie types zijn, ligt het voor de hand dat Java uiteindelijk code sharing zal kiezen als de techniek voor vertaling van generieke types en methods.[2]

Een nadeel van code sharing is dat er problemen ontstaan wanneer primitieve types gebruikt worden als parameters of generieke types of methods. Waarden van een primitieve type zijn van een andere omvang en vereisen dat verschillende code gegenereerd wordt voor een lijst `int` en een lijst `double` bijvoorbeeld. Het is niet uitvoerbaar om beide lijsten naar een enkele lijstimplementatie te mappen. Er zijn een aantal oplossingen voor dit probleem:

- *Geen primitieve types*. Primitieve types worden verboden als type arguments van geparameteriseerde types

2 Overigens gebruikt C# beide vertaaltechnieken voor zijn generieke types: codespecialisatie voor de value types en code sharing voor de referentie types.

```
interface Comparable {
  public int compareTo (Object that);
}

final class Byte implements Comparable {
  private byte value;
  public Byte (byte value) { this.value =
  value; }
  public byte byteValue () { return value; }
  public int compareTo (Byte that) {
     return this.value - that.value;
  }
  public int compareTo (Object that) {
     return this.compareTo((Byte)that);
  }
}

class Collections {
  public static Comparable max (Collection
  xs) {
     Iterator xi = xs.iterator();
     Comparable w = (Comparable)xi.next();
     while (xi.hasNext()) {
        Comparable x = (Comparable)xi.next();
        if (w.compareTo(x) < 0) w = x;
     }
     return w;
  }
}

final class Test {
  public static void main (String[ ] args) {
     LinkedList ys = new LinkedList();
     ys.add(new Byte((byte)0)); ys.add(new
Byte((byte)1));
     Byte y = (Byte)Collections.max(ys);
  }
}
```

CODEVOORBEELD 3. Geparameteriseerde method na type erasure

of methods; de compiler wijst dus instantiaties als `List<int>` af.

- *Boxing.* Waarden van primitieve types worden automatisch geboxed zodat referenties naar de geboxte primitieve value intern gebruikt worden. Boxing is het proces waarbij een primitieve waarde wordt ingepakt in zijn corresponderende referentie type, en unboxing is het omgekeerde daarvan (zie /BOX/.) Boxing heeft vanzelfsprekend een negatief effect op de performance vanwege de extra box en unbox operaties.

Java Generics gebruikt de eerste benadering en beperkt de instantiatie van Generics tot referentie types. Vandaar dat een `LinkedList<int>` in Java verboden is.[3]

TYPE ERASURE We gaan nu kijken naar de details van de code sharing implementatie van Java Generics. De belangrijkste vraag daarbij is, op welke manier de Java compiler verschillende instantiaties mapt van een geparameteriseerd type of methode naar één representatie van het type of de methode?

De vertaaltechniek die wordt gebruikt door de Java compiler kan worden omschreven als een vertaling van generieke Java source code terug naar reguliere Java code. Deze vertaaltechniek wordt *type erasure* genoemd: de compiler verwijdert alle voorkomende type variabelen en vervangt deze door hun bound, of het type `Object`, voor het geval er geen bound gespecificeerd is. Zo zouden de instantiaties `LinkedList<Integer>` en een `LinkedList<String>` uit ons vorige voorbeeld (zie Codevoorbeeld 1) kunnen worden vertaald naar een `LinkedList<Object>`, of kortweg `LinkedList`, en de methods `max<Integer>()` en `max<String>()` (uit Codevoorbeeld 7) zouden vertaald kunnen worden naar `max<Comparable>()`. Naast het verwijderen van alle type variabelen en vervanging daarvan door de bijbehorende bound, voegt de compiler op bepaalde plaatsen een paar casts toe en indien nodig zogenaamde bridge methods.

De vertaling van generieke Java code naar reguliere Java code is met opzet door de Java-ontwerpers gekozen. Een essentiële requirement voor alle nieuwe taalfeatures in Java 1.5 is hun compatibiliteit met eerdere versies van Java. In het bijzonder is vereist dat een pre-1.5 Java virtual machine in staat moet zijn om 1.5 Java code uit te voeren. Dit is slechts te bereiken wanneer de byte code uit een 1.5 Java source eruit ziet als reguliere byte code uit pre-1.5 Java code. Type erasure komt aan deze requirement tegemoet: na de type erasure is er geen verschil meer tussen een geparameteriseerde en een reguliere type of method.

Omwille van de duidelijkheid omschreven we de type erasure als een vertaling van generieke Java code naar reguliere niet-generieke Java code. Dat is eigenlijk niet helemaal juist; de vertaalslag loopt van generieke Java code rechtstreeks naar Java byte code. Desondanks zullen we het type erasure proces voor het hierna volgende aanduiden als een vertaling van generieke Java naar niet-generieke Java.

Codevoorbeeld 2 illustreert de vertaling door type erasure, waarbij gebruik is gemaakt van ons eerdere voorbeeld van generieke types (zie codevoorbeeld 1 uit het eerste artikel van deze serie).

Zoals in de code te zien is, zijn alle voorkomende type variabelen `A` vervangen door type `Object`. De

3 In C++ en C# zijn primitieve types toegestaan als type arguments omdat deze talen codespecialisatie gebruiken (in C++ voor alle instantiaties en in C# in ieder geval voor instantiaties op primitieve types).

Referenties

/JDK15/
Java 2 SDK, Standard Edition 1.5.0 Beta 1 (build 32c)
http://java.sun.com/j2se/1.5/index.html

/JSR14/
Adding Generics to the Java Programming Language
Java Specification Request
http://jcp.org/en/jsr/detail?id=14

/SPC/
Adding Generics to the Java Programming Language
Public Draft Specification, Version 2.0, July 2003
Beschikbaar als onderdeel van een experimentele, vroege release van Java Generics
http://developer.java.sun.com/developer/earlyAccess/adding_generics/index.html

/BOX/
Autoboxing support for the Java Programming Language
Draft Specification
http://jcp.org/aboutJava/communityprocess/jsr/tiger/autoboxing.html

/BRA/
Making the future safe for the past: Adding Genericity to the java programming language Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler
Proc. OOPSLA'98
http://lamp.epfl.ch/~odersky/papers/oopsla98.ps.gz
http://citeseer.nj.nec.com/bracha98making.html

implementatie van onze generieke verzameling komt nu exact overeen met een implementatie die de traditionele Java-techniek voor genericiteit toepast, namelijk een implementatie in termen van `Object` referenties. De voorbeeldcode laat ook een automatisch toegevoegde cast zien: in de `main()` method, waar een gelinkte lijst met strings gebruikt wordt, voegde de compiler een cast toe van `Object` naar `String`.

Codevoorbeeld 3 laat de type erasure zien van onze geparameteriseerde `max()` method uit Codevoorbeeld 1.

Ook hier zijn alle voorkomende type variabelen vervangen door ofwel type `Object` (in de `Comparable` interface) ofwel door de bound (type `Comparable` in method `max()`). Opnieuw zien we ook de ingevoegde cast van `Object` naar `Byte` in de `main()` method waar de generieke method wordt aangeroepen voor een verzameling van `Bytes`. Tenslotte zien we een voorbeeld van een bridge method in class `Byte`.

De compiler voegt bridge methods toe in subclasses om ervoor te zorgen dat het tijdelijk opheffen correct functioneert. In dit voorbeeld implementeert class `Byte` de interface `Comparable<Byte>` en moet daarom de `compareTo()` method van de superinterface opheffen. De compiler vertaalt de `compareTo()` method van de generieke interface `Comparable<A>` naar een method die een Object neemt, en vertaalt de `compareTo()` method in de class `Byte` naar een method die een *Byte* neemt. Na deze vertaling is method `Byte.compareTo(Byte)` niet langer meer een opheffende versie van method `Comparable<Byte>.compareTo(Object)`, omdat de twee methodes verschillende signatures hebben als neveneffect van translation by erasure. Om het opheffen mogelijk te maken, voegt de compiler een bridge method toe aan de subclass. De bridge method heeft dezelfde signature als de method van de superclass die moet worden opgeheven en delegeert naar de andere methods in de afgeleide class die het resultaat was van translation by erasure.

SAMENVATTING In dit artikel gaven we een overzicht van alle belangrijke taalkenmerken die te maken hebben met geparameteriseerde types en methods. Uiteraard kan een behoorlijk complexe taalfeature als Java Generics binnen het bestek van een artikel als dit nooit volledig behandeld worden. Om Java Generics op een betrouwbare en effectieve manier te kunnen gebruiken, moet de programmeur nog veel meer details begrijpen en zich eigen maken. De belangrijkste moeilijkheden bij het gebruik en begrip van Java Generics vloeien voort uit het type erasure vertaalproces, waarmee de compiler alle voorkomende type parameters kan weglaten. Dat heeft nogal wat verrassende effecten. Om er één te noemen: array's van geparameteriseerde types zijn verboden in Java. Dat betekent dus dat `Comparable<String>[]` een illegaal type is, terwijl `Comparable[]` is toegestaan. Dat is op zijn minst verbazingwekkend en in de praktijk nogal hinderlijk. Daarom kunnen array's het beste vermeden worden en bij voorkeur vervangen worden door verzamelingen, indien het element type een geparameteriseerd type blijkt te zijn. Het is noodzakelijk om deze en vele andere tips en technieken eerst grondig te bestuderen, voordat de nieuwe taal feature voluit benut kan worden. Ondanks de hier en daar nog wat ruwe kantjes, voegt Java Generics een aanzienlijke expressieve kracht toe aan de Java programmeertaal. Onze eigen ervaring is: wanneer je eenmaal een tijdje met Generics hebt gewerkt, zul je ze erg missen als je weer moet overschakelen naar niet-generiek Java met z'n niet gespecificeerde types en ontelbare casts en runtime checks.

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).