Java Development Kit 5.0

Zu neuen Ufern

Das nächste JDK soll Java modernisieren. Ob das Ergebnis den Versionssprung auf 5.0 rechtfertigt oder ob er sich als Marketing erweist, verrät dieser Ausblick auf die bevorstehende Java-Release. Bernhard Bablok

Die Firma Sun hatte schon immer ein Händchen dafür, Java-Entwickler zu verwirren. Hieß die zweite Major-Release noch Java 1.1, trug die nächste Version den Namen Java 2, Version 1.2. Die Unterteilung in Standard Edition (J2SE) und Enterprise Edition (J2EE) führte zu weiteren parallel gepflegten Versionen. Die folgenden Releases 1.3 und 1.4 firmierten trotzdem unter Java 2 und auch die erste Beta der aktuellen Release hieß Version 1.5.

Für diesen Artikel dient ein mit Version 1.5 Beta 2 ausgezeichnetes JDK als Grundlage. Kurz danach folgte der Versionsschwenk zu 5.0 [1]. Eine Version höher wäre durchaus auch sinnvoll gewesen, handelt es sich doch um die sechste Major-Release. Wichtiger als Versionsnummern sind aber die Inhalte, und hier bietet das JDK 5.0 einiges Neue. Die Sprache hat sich an einigen Stellen geändert, neue Klassen sind hinzugekommen

Die gute Nachricht für alle JVM-Autoren: Der Bytecode ändert sich nicht, alle neuen Sprachkonstrukte werden mit dem bestehenden Befehlssatz emuliert. Trotzdem laufen 5.0-Programme nicht unbedingt auf älteren JVMs, wenn sie von neuen Klassen abhängen.

Vom JCP zu JSRs

Sun bastelt schon lange nicht mehr alleine an Java, auch wenn die Sprache (noch) nicht freigegeben ist. Der so genannte Java Community Process (JCP) installiert Arbeitsgruppen zu den vorgeschlagenen Java Specification Requests (JSR), die letztlich ihren Weg in den Sprachstandard oder in die Standardklassen finden – falls sie nicht abgelehnt werden. Die genaue Zahl der JSRs, die in die neue Version eingeflossen sind, ist schwer zu bestimmen, da es eine Reihe von Sammel-JSRs gibt, zum Beispiel den JSR-176: J2SE 1.5 Release Content [2]. Insgesamt dürften es 15 bis 20 Änderungen sein.

Der Artikel geht aber nicht auf jedes Detail ein, sondern konzentriert sich auf die Highlights. Die Änderungen lassen sich grob in drei Kategorien einordnen: Änderungen an der Sprache, an den Bibliotheken sowie jene an den Tools. Soviel schon vorweg: Bei Letzteren hat sich Sun einen Fauxpas erlaubt und einem Tool den Namen »apt« gegeben, obwohl das Paketwerkzeug unter Debian schon so heißt. Über den eigenen Tellerrand hinaussehen war noch nie eine Stärke großer Firmen.

Installieren und einstellen

Sowohl das JDK als auch die zugehörige Dokumentation sind nach dem Abnicken einer entsprechenden Lizenzvereinbarung als ungefähr 45 MByte große Pakete von [1] herunterzuladen. Die Dokumentation kommt im schlichten Zip-Format auf den Rechner und kann an beliebiger Stelle ausgepackt werden. Das JDK für Linux gibt es als sich selbst entpackendes Shell-Skript. Vorsicht: Das Toplevel-Verzeichnis ist »docs«, es empfiehlt sich also, das Paket in einem eigenen Verzeichnis zu entpacken. Es enthält aber neben der Software nur noch einmal die Lizenz.

Nach deren Bestätigung entpackt das Skript ein RPM-Archiv und installiert es gleich unter »/usr/java/jdk1.5.0«. Das RPM selbst liegt anschließend im aktuellen Verzeichnis und kann auf weiteren Rechnern installiert werden. Insgesamt sind 132 MByte für das JDK notwendig und 252 MByte für die Dokumentation. Wer auf die Quelldateien sowie die Demos und Beispiele verzichten kann, spart beim JDK 31 MByte.

Um die Betaversion zu nutzen, genügt es in der Regel, das neue Verzeichnis »/usr/java/jdk1.5.0/bin« vorne an den »PATH« zu stellen und die Variable

»JAVA_HOME« entsprechend zu setzen. Suse-Nutzer müssen etwas mehr tun, denn Suse definiert eine ganze Reihe von Java-relevanten Umgebungsvariablen (einfach in einer Kommandozeile »echo $J« eingeben und die Tabulator-Taste drücken). Entweder, man setzt die Variablen von Hand oder schreibt sie in die Datei »/etc/java/java5.conf« (siehe Listing 1). Jetzt genügt bei Suse ein einfaches

```
. setJava --version 5.0
```

um die Variablen in der aktuellen Shell (beziehungsweise in einem Skript) zu setzen. Ebenso funktioniert:

```
setDefaultJava --version 5.0
```

Dieses Kommando, als Root ausgeführt, setzt den symbolischen Link von »/usr/lib/java« auf das passende Verzeichnis. Wie vieles andere bei Suse, sieht auch der Umgang mit Java zuerst kompliziert aus, um sich dann als durchdacht und praktisch zu erweisen.

Generics statt Templates

Nachdem jetzt das neue JDK zur Verfügung steht, geht es gleich hinein in die schöne neue Java-Welt. Ein seit langem gewünschtes Feature, insbesondere der C++-Fraktion, waren Templates. Die gibt es jetzt auch in Java, wo sie allerdings Generics heißen. Im Gegensatz zu C++, bei dem Prä-Compiler und Compiler für jede Template-Instanz eine eigene Klasse generieren, gibt es unter Java nur eine Klasse mit erweiterter Typdeklaration (siehe Listing 2).

Bisher lässt sich eine solche Anwendung über eine Objektliste mit passenden Type Casts programmieren. Der mit »javap -c« disassemblierte Bytecode in Listing 3 zeigt, dass der Compiler genau dies macht – der erzeugte Code kennt keine Generics mehr, nur die normale »ArrayList«-Klasse und die Interface-Methoden von »List« sind zu finden. Entwickler sparen sich also lästige Casts und auch »ClassCastErrors« gehören damit der Vergangenheit an.

Leider verkomplizieren Generics die Sprache und man muss bei manchen Konstrukten aufpassen. So erbt »List<Integer>« eben nicht von »List<Number>«, obwohl »Integer« von »Number« abgeleitet ist. Dafür gibt es die neuen Sprachkonstrukte »List<? extends Number>« und »List<? super Integer>«, die fast schon perlianisch anmuten.

Einfache Schleifen, Autoboxing, Aufzählungen

Listing 2 zeigt in den Zeilen 43 und 44 eine weitere nützliche Erweiterung. Der Enhanced for Loop vereinfacht die Formulierung von Schleifen, was sich insbesondere bei mehrfach geschachtelten Algorithmen positiv auf die Lesbarkeit auswirkt. Wieder ist es der Compiler, der dieses Konstrukt in normalen Code umsetzt. Die Zeilen 65, 72 und 81 in Listing 3 beschreiben die verwendete Iteratorklasse mit ihren Methoden »hasNext()« und »next()«. Die Zeilen 41, 43 und 44 in Listing 2 demonstrieren, wie nützlich das Autoboxing ist. Damit ist das automatische Ein- und Auspacken von Wrapperobjekten für primitive Typen gemeint, im Beispiel »Integer« zu »int« und umgekehrt.

Enumerations (Enums) helfen beim Typsicheren Programmieren. Bisher musste man sie emulieren, siehe das Beispiel in Listing 4. Der Trick bestand darin, einen privaten Konstruktor zusammen mit einer Reihe von finalen Objekten der Klasse zu definieren. Jetzt gibt es dafür das einfache Schlüsselwort »enum« wie im Beispiel »public enum Farbe {...}«. Java-Enums sind mächtiger als ihre C- und C++-Verwandten, denn es handelt sich um spezielle Klassen.

Objekte dieser Klassen lassen sich zwar nicht erzeugen und auch Subklassen sich nicht ableiten, man kann aber Enums mit eigenen Methoden versehen. Der Java-Compiler wandelt Enums tatsächlich in normale Klassen um, die von der neuen Klasse »java.lang.Enum« erben. Damit können Programme mit Enums natürlich nicht mehr unter alten Java-Runtimes laufen, da dort die Klasse nicht bekannt ist.

Das neue Konstrukt »import static« erspart viel Tipparbeit. Mit ihm sind alle statischen Methoden und Konstanten ohne Klassenpräfix verfügbar. Zum Beispiel schreibt man bei »java.lang.Math« einfach »sin(PI/2.0)« statt »Math.sin(Math.PI/2.0)«.

Anwendungssysteme benötigen heute eine Vielzahl von Zusatzinformationen über die Softwareobjekte, die sie verarbeiten. Leider lassen sich diese nicht vollständig über Reflection [4] abfragen, daher gibt es für den Entwickler eine Reihe von Zusatzdateien zu pflegen, etwa Interface-Beschreibungen oder Deployment-Deskriptoren.

Metadaten

Diese Dateien enthalten teilweise redundante Informationen, sie sind also entweder identisch zu den Infos des Programmcodes oder sie sind falsch. Die Pflege solcher verstreuter Informationen ist aufwändig und sie konsistent zu halten ist ein ständiges Problem. Durch so genannte Annotations lassen sie sich bei Java 5.0 innerhalb der Klassendatei pflegen. Annotations fangen mit @ an (etwa »@Copyright« oder »@Remote«) und

Listing 1: »java5.conf« unter Suse

```
01 # Configuration for JDK 1.5.0 aka 5.0
02
03 Priority: 20
04
05 Vendor  :  Sun
06 Version :  5.0
07 Devel   :  True
08
09 JAVA_BINDIR = /usr/java/jdk1.5.0/bin
10 JAVA_ROOT   = /usr/java/jdk1.5.0
11 JAVA_HOME   = /usr/java/jdk1.5.0
12 JRE_HOME    = /usr/java/jdk1.5.0/jre
13 JDK_HOME    = /usr/java/jdk1.5.0
14 SDK_HOME    = /usr/java/jdk1.5.0
15
16 JAVA_LINK   = /usr/java/jdk1.5.0
```

Listing 2: »ListDemo.java«

```
22 import java.util.*;
23
31 public class ListDemo{
32
35   public static void main(String[] args) {
36     List<Integer> list = new ArrayList<Integer>();
37
38     list.add(new Integer(1));
39     list.add(new Integer(2));
40     list.add(new Integer(3));
41     list.add(4);
42
43     for (int i: list)
44       System.out.println("i = " + i);
45   }
46 }
```

werden im normalen Programmablauf ignoriert. Mit dem erwähnten Programm »apt« (Annotation Processing Tool) ist es ähnlich wie bei Doclets möglich, die Annotations zu verarbeiten und aus dem Code heraus weitere Dateien zu erzeugen. Annotations werden sich sicherlich schnell durchsetzen, aber auch die Portierbarkeit von einer Entwicklungsumgebung zur nächsten wegen fehlender Standards erschweren.

Noch mehr Klassen

Andere Neuerungen in Java 5 betreffen die Klassenbibliothek, die wieder deutlich gewachsen ist. Neue Klassen verwenden bedeutet natürlich Verzicht auf Abwärtskompatibilität – ein zweischneidiges Schwert insbesondere für Softwarehersteller. Denn leider ist eine neue JRE-Version nicht für jedes Betriebssystem ausreichend schnell und vor allem fehlerfrei verfügbar. „Compile once, run everywhere“ ist so nur ein Traum.

Trotzdem lohnt sich ein Blick auf die neuen Klassen, das eine oder andere Goodie beschleunigt vielleicht den Umstieg. An erster Stelle stehen für alle, deren Muttersprache C ist, natürlich die Klasse »java.util.Formatter« und ihre versteckte Inkarnation in der statischen Methode »String.format()«. Endlich lassen sich Ausgaben und Strings so bequem formatieren wie mit »printf()« in C. Als Nebeneffekt beherrscht Java jetzt auch Methoden mit einer variablen Anzahl von Argumenten.

Die Klasse »Formatter« ähnelt »printf()«, bietet aber mehr Möglichkeiten. Sie verarbeitet die von C bekannten Formatstrings, bringt aber schon Mechanismen zur Lokalisierung mit. Stimmen die übergebenen Daten nicht mit den verlangten Formaten überein, löst die Methode eine Exception aus.

Zwei Drittel der langen Klassendokumentation bestehen aus einer Zusammenfassung der Funktionalität samt anschließender genauer Definition des jeweiligen Verhaltens. Für die eigentliche Methodendokumentation reicht das restliche Drittel aus.

Prozesse, Threads, Umgebungen

Unter Linux ist es guter Stil, Applikationen nicht monolithisch aufzubauen, sondern bestehende Programme zu nutzen. So verwenden viele GUI-Programme vorhandene Kommandozeilentools. Solche grafischen Oberflächen mit Java zu bauen war zwar schon bisher möglich, aber nur unter systembedingten Einschränkungen.

Mit den neuen Klassen in Java 5 wird vieles einfacher. Mit dem »ProcessBuilder« kann man ein Kommando mit Argumenten, Arbeitsverzeichnis und Environment vorbereiten und dann wiederholt ausführen. Die Abfrage der Systemumgebung mit »System.getenv()«, seit JDK 1.1 als deprecated (wörtlich: abgelehnt) gekennzeichnet, ist jetzt wieder

Listing 3: »ListDemo«-Bytecode

```
00 new   #2; //class java/util/ArrayList
03 dup
04 invokespecial  #3; //Method java/util/ArrayList."<init>":()V
07 astore_1
08 aload_1
09 new   #4; //class java/lang/Integer
12 dup
13 iconst_1
14 invokespecial  #5; //Method java/lang/Integer."<init>":(I)V
17 invokeinterface   #6,  2; //InterfaceMethod java/util/List.add:
    (Ljava/lang/Object;)Z
22 pop
23 aload_1
24 new   #4; //class java/lang/Integer
27 dup
28 iconst_2
29 invokespecial  #5; //Method java/lang/Integer."<init>":(I)V
32 invokeinterface   #6,  2; //InterfaceMethod java/util/List.add:
    (Ljava/lang/Object;)Z
37 pop
38 aload_1
39 new   #4; //class java/lang/Integer
42 dup
43 iconst_3
44 invokespecial  #5; //Method java/lang/Integer."<init>":(I)V
47 invokeinterface   #6,  2; //InterfaceMethod java/util/List.add:
    (Ljava/lang/Object;)Z
52 pop
53 aload_1
54 iconst_4
55 invokestatic   #7; //Method java/lang/Integer.valueOf:(I)Ljava/lang/
    Integer;
58 invokeinterface   #6,  2; //InterfaceMethod java/util/List.add:
    (Ljava/lang/Object;)Z
63 pop
64 aload_1
65 invokeinterface   #8,  1; //InterfaceMethod java/util/List.iterator:
    ()Ljava/util/Iterator;
70 astore_2
71 aload_2
72 invokeinterface   #9,  1; //InterfaceMethod java/util/
    Iterator.hasNext:()Z
77 ifeq  121
80 aload_2
81 invokeinterface   #10,  1; //InterfaceMethod
    java/util/Iterator.next:()Ljava/lang/Object;
86 checkcast   #4; //class java/lang/Integer
89 invokevirtual  #11; //Method java/lang/Integer.intValue:()I
92 istore_3
93 getstatic   #12; //Field java/lang/System.out:Ljava/io/PrintStream;
96 new   #13; //class java/lang/StringBuilder
99 dup
100 invokespecial  #14; //Method java/lang/StringBuilder."<init>":()V
103 ldc   #15; //String i =
105 invokevirtual  #16; //Method java/lang/StringBuilder.append:
     (Ljava/lang/String;)Ljava/lang/StringBuilder;
108 iload_3
109 invokevirtual  #17; //Method java/lang/StringBuilder.append:(I)
     Ljava/lang/StringBuilder;
112 invokevirtual  #18; //Method java/lang/StringBuilder.toString:()
     Ljava/lang/String;
115 invokevirtual  #19; //Method java/io/PrintStream.println:
     (Ljava/lang/String;)V
118 goto  71
121 return
```

erlaubt und wurde sogar noch durch eine etwas überladene Version ergänzt, die alle Umgebungsvariablen in einer Map zurückgibt.

Das neue JDK vereinfach auch das Leben mit Threads deutlich. Die Klasse »java.util.concurrent« ist ein kleines Framework mit allem, was dazu erforderlich ist: Thread-Pools, Queues, Zeitsteuerung und Synchronisierung des Zugriffs auf gemeinsam genutzte Ressourcen. Gewissermaßen als Zugabe gibt es Thread-sichere Collections. Wohin dies alles zielt, ist klar. Auch wenn es sich um Version 5.0 der Java-Standard-Edition handelt, sind die Erweiterungen offensichtlich für künftige Enterprise-Editionen wichtig. In dieselbe Richtung weisen Neuerungen beim Monitoring und Management (Stichwort JMX) – doch das führt an dieser Stelle zu weit.

An der Oberfläche

Eine grundsätzliche Änderung, die Neuimplementierung XAWT des AWT, wird für viele kaum sichtbar sein, da es nur noch wenige AWT-Anwendungen gibt. Trotzdem ist es ein wichtiger Schritt gerade für Linux, denn jetzt hängt das JDK nicht länger von Motif und der XT-Bibliothek ab. Das neue Toolkit, XToolkit genannt, soll auch deutlich schneller sein. Bei Swing-Anwendungen wirkt sich das aber nur wenig aus. Auf einem Rechner mit 1,5 GHz war kein Unterschied zu erkennen.

Man könnte dies als Beweis der alten Weisheit ansehen, dass geistige oder finanzielle Investitionen in schnellere Programme nicht lohnen, da demnächst ohnehin schnellere Hardware zur Verfügung steht. Wichtiger ist aber die Aussicht, dass XToolkit künftig durch Windowmanager-konforme Versionen ersetzt werden könnte. Das wäre ein neues Kapitel beim homogenen Look & Feel für Java-Anwendungen.

In den Bereichen GUI, Java 2D und Sound gibt es noch viele weitere Änderungen, die aber insgesamt nicht sehr spektakulär sind. Swing kommt mit einem neuen Look & Feel unter dem Namen Ocean, aber auch hier sind die Unterschiede marginal (siehe Abbildungen 1 und 2). Mit Synth steht eine weitere Look & Feel-Variante zur Verfügung, die sich ohne aufwändige Programmierung anpassen lässt.

Kaum spürbar schneller

Wie bei jeder JDK-Version, soll auch bei dieser die Performance deutlich steigen. XAWT wurde in diesem Zusammenhang schon erwähnt. Auch die VM-Implementation bringt Neues unter dem Namen Class Data Sharing. Das bedeutet, dass die Core-Klassen in einem eigenen Format unter »$JRE_HOME/lib/i386/client/classes.jsa« liegen. Diese Datei blendet der Loader beim Start schreibgeschützt in den Adressraum ein.

Mehrere JVMs nutzen die Datei gemeinsam, wodurch sie insgesamt weniger Speicher verbrauchen. Der Effekt ist umso größer, je weniger Spezialklassen eine Anwendung nutzt. Entsprechende Untersuchungen dazu haben ergeben, dass der Effekt zwar messbar (zirka 3 MByte RSS-Größe), aber auf aktuellen Geräten mit gutem Speicherausbau subjektiv nicht spürbar ist.

Auch das Feintuning des Garbage-Kollektors verbessert die Performance. Das ist aber nur mit der Server-VM möglich und soll das Verhältnis von Durchsatz und Speicherverbrauch steuern.

finally{}

Java wird immer mächtiger – und komplizierter. Die reinen Sprachänderungen sind noch überschaubar, doch ist bei jeder neuen Version mehr Zeit für die API-Docs aufzuwenden. Wo der Linux-Kernel es immer wieder schafft, sich von altem Ballast zu befreien, geht Sun mit Java den Weg der Kompatibilität. Selbst Methoden, die seit Jahren deprecated sind, gibt es in der neuesten Version immer noch. Den Bytecode-Befehlssatz ändern ist tabu, deshalb werden moderne Konstrukte für die alte JVM übersetzt. Trotz eines zwiespältigen Gefühls: Die neue Version bietet Entwicklern viel Neues und zeigt, dass Java eine lebendige Sprache ist. *(ofr)* ■

Infos

[1] J2SE 5.0 Beta: [http://java.sun.com/j2se/1.5.0/]
[2] Java Community Process: [http://www.jcp.org/en/jsr/detail?id=176]
[3] Bernhard Bablok, „Spieglein, Spieglein, Das Java-Reflection-API": Linux-Magazin 03/04, S. 110
[4] Listings dieses Coffee-Shops: [http://www.linux-magazin.de/Service/Listings/2004/10/Coffeeshop]

Der Autor

Bernhard Bablok arbeitet bei der AGIS mbH als Anwendungsentwickler. Wenn er nicht Musik hört, mit dem Radl oder zu Fuß unterwegs ist, beschäftigt er sich mit Themen rund um die Objektorientierung. Er ist unter [coffee-shop@bablokb.de] zu erreichen.

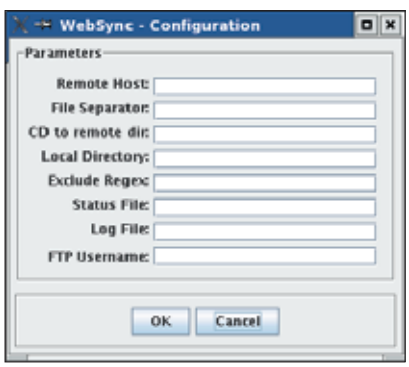

Abbildung 1: Eine Anwendung in dem neuen Look & Feel namens Ocean.

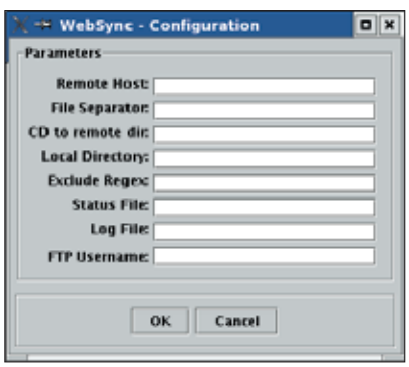

Abbildung 2: In der alten Look & Feel-Version Metal sehen insbesondere Buttons anders aus.

Listing 4: »EnumDemo.java«

```
31 public class EnumDemo {
32
33   private int iValue;
34
41   private EnumDemo(int i) {
42     iValue = i;
43   }
44
51   public static final EnumDemo RED   = new EnumDemo(1);
52   public static final EnumDemo WHITE = new EnumDemo(2);
53   public static final EnumDemo BLUE  = new EnumDemo(3);
54 }
```