MailTo@Bastie.de

Stand: 22.Nov.2001

Sebastian Ritter - Bastie

Make Java

Performance

All rights reserved. Copyright © 2001, Bastie – Sebastian Ritter. Der Autor erkennt ausdrücklich alle Rechte Dritter an. Dies gilt einschließlich aber nicht beschränkt auf Marken und Markennamen. Die Verwendung dieser ist nicht als Geltungmachung von Rechten auszulegen.

Make Java

Performance

Teil 0 **Vorwort** *2*

Motivation und Ziel dieses Dokumentes *2*

Motivation für die Erstellung 2

Inhalt 2

Fundstellen 2

Ziel dieses Dokuments 3

Unterstützung 3

Historie 3

Allgemeines *5*

Glauben ist etwas Grundsätzliches! – Falsches?!? 5

Java und seine Versionen 5

Begriffsdefinitionen 6

Verwendete Abkürzungen 6

Teil 1 **Allgemeines** *8*

Warum ist Java langsamer 8

Gute Gründe nicht zu optimieren 9

Gründe zu optimieren 9

Wann sollten Sie optimieren? 10

Wo sollten Sie optimieren? 11

Was sollten Sie optimieren 11

Wann sind Sie fertig mit der Optimierung? 12

Teil 2 **Performance managen** *13*

Planung *13*

Performance Experten 13

Performance Spezifikation 14

Performance in Analyse und Design 14

Performance testen 15
Performance messen 15

Teil 3 **System und Umgebung** *16*

Vor- und Nachteile *16*
Die richtige Java Virtual Machine [JVM] *16*
Bytecode Interpreter 17
Just-In-Time Compiler 17
Hot Spot Compiler 18
Thread-Umsetzung 19
Gleitpunktdarstellung in der JVM 19
Hardware *20*
Prozessoren 20
Grafikausgabe 20
Gleitpunktdarstellung 20
Native Übersetzung *20*
Speicher *21*
Tools *22*
javac 22
Alternative Compiler 23
serialver 24
jar 24
Obsfucators 24
Profiler 25
Externe Daten *25*
Grafiken 25
Datenbanken *26*
Datenbanktreiber 26
Netzwerk *28*
Applets 29
Internet Explorer 29

Teil 4 **Analyse und Performance** *31*

Nebenläufigkeit *31*
Asynchrone Methodenaufrufe 32
Wertebereich *34*

Teil 5 **Design und Performance** *35*

Vor- und Nachteile *35*

Datentypen *35*
Primitive und Referenzen **36**
Sammlungen *37*
Synchronisiete Sammlungen **37**
Gegenüberstellung von Sammlungen **38**
Versteckte Sammlungen **38**
Erweiterungen *38*
Cache **39**
Objektpool **39**
Netzwerkdesign *39*
Callback **40**
Methodenaufruf **41**
Entwurfsmuster *42*
Fliegengewicht Muster **42**
Fassade Muster **43**
Null Pattern **44**
Process-Entity Muster **45**
Singleton Muster **45**
Sichtbarkeiten *46*
Inlining und Sichtbarkeit **46**
Vererbungshierarchie *47*
Antidesign *48*
Direkter Variablenzugriff **48**
Vermischung von Model und View **48**
Teil 6 **Implementierung** *50*
Vor- und Nachteile *50*
Grundwissen **50**
Optimierung für bestimmte Anwendungsbereiche **57**
Teil 7 **Implementierung mit Java** *59*
Eigener Profiler **59**
Java Virtuelle Machine **60**
Variablen / Datentypen und Operatoren **61**
Zeichenketten **64**
Primitive Datentypen **64**
Sammlungen – Container für Datentypen **65**
Synchronisierte Sammlungen **68**
Methoden **69**

Objekte **71**
Plattformübergreifend native Implementierung **76**
Exceptions **76**
Ein- und Ausgabe **77**
Benutzeroberflächen **91**
AWT **93**
Swing **95**
Grafiken **98**
Threads **101**
Datenbankzugriff **103**
Netzwerk **104**
Mathematik **105**

Teil 8 ***Implementierung mit Perl*** *106*

Perlversionen **106**
Eigener Profiler **106**
Inlining **108**
Operatoren und logische Ausdrücke **109**

Anhang I ***Vorgehen*** *110*

vor / bei der Codierung ***110***
nach der Codierung ***110***
bei Web-Anwendungen ***111***

Anhang II ***Quelltextmuster*** *112*

Objektpool **112**

Anhang III ***Small FAQ*** *114*

Anhang IV ***Datenbank-Internaas*** *116*

Anlagen ***Quellen und Literatur*** *117*

Make Java

Performance

Meinungen zu diesem Dokument

Gefunden unter Dokuwelt.de, die Version 2.1 betreffend von Bruno Persi: „Wow!! Echt genial Doku!! Zu einem Thema wo man nur sehr wenig dazu findet(auf Deutsch)“

Gepostet im Performance-Forum unter Bastie.de, die Version 2.1 betreffend von Bernhard Bauder: „Super Arbeit! Danke für deine Hilfe mit der Page. Bernhard“

TEIL 0
Vorwort

Inzwischen ist der Titel vielleicht nicht mehr ganz so passend, da erstmals auch eine andere Programmiersprache – Perl – betrachtet wird. Der Hauptteil dieses Dokumentes beschäftigt sich jedoch weiterhin mit der Sprache Java.

Motivation und Ziel dieses Dokumentes

Motivation für die Erstellung

Da Java nicht gerade als hoch performante Sprache gilt beschäftigt man sich automatisch früher oder später mit diesem Thema etwas näher und irgendwann fing ich dann an alles zum Thema Java und Performance im allgemeinen zu sammeln. Zu diesem Zeitpunkt entstanden einige Folien, in denen ich dieses Wissen festhalten wollte. In der Newsgroup habe ich dann irgendwann einmal jemanden angeboten ihm diese zuzusenden. Ich hatte natürlich nicht mit dem Echo gerechnet, so dass ich innerhalb von zwei Tagen mehr als 20 Anfragen hatte. Ich entschloss mich die Folien auf meine Webseite zu setzen, wo diese bzw. dieses Dokument inzwischen über 10.000[1] mal abgerufen wurde. Irgendwann sind es über einhundert Folien und somit ließ leider die Übersichtlichkeit nach. Aus diesem Grund habe ich schließlich dieses Dokument erstellt.

Inhalt

Dieses Dokument stellt Möglichkeiten und Ansätze dar, wie Sie die Performance Ihrer Anwendungen erhöhen können. Inzwischen gibt der weitaus überwiegende Anteil Informationen wieder, die sich nicht auf eine bestimmte Programmiersprache beziehen. Dies ist jedoch eine historische Gegebenheit und wird sich vielleicht auch wieder ändern.

Fundstellen

Die jeweils aktuellste Version dieses Dokuments können Sie unter http://www.Bastie.de finden. Fragen, Rückmeldungen, Anregungen und / oder Wünsche sind willkommen und sollten per elektronischer Nachricht an mailto:MailTo@Bastie.de, Betreff: „MJP“ gesendet werden.

[1] Stand 01.11.2001

Dieses Dokument ist grundsätzlich frei und kostenlos. Links auf dieses Dokument und die Internetpräsenz sind ausdrücklich erwünscht.

Ziel dieses Dokuments

Die wesentlichen Ziele des Dokumentes bestehen aus den folgenden Punkten:

1. Mein erworbenes Wissen über Java, Perl und Performance so festzuhalten, dass ich es auch nach längerer Pause schnell wieder reaktivieren kann.

2. Ein Nachschlagewerk zu sein, um schnell Lösungen zu finden oder zu erstellen.

3. Deutschsprachig zu sein.

Durch die Zielsetzung ergeben sich mitunter Wiederholungen. Dies ist bedingt durch die Tatsache, dass es sich auch um ein Nachschlagewerk handeln soll. Mir persönlich ist es nicht sehr angenehm, wenn in einem an sich abgeschlossenen Teil ständig Verweise auf Textpassagen anderer Teile zu finden sind.

Unterstützung

Es gibt mehrere und ganz verschiedene Möglichkeiten, wie Sie "Make Java - Performance" unterstützen können.

Privat spenden

Diese Möglichkeit ist geeignet, dieses Angebot auf einfache Art und Weise zu unterstützen. Überweisen Sie an das folgende Konto einen Betrag Ihrer Wahl[1]:

S. Ritter
Deutsche Bank 24 Berlin
BLZ: 100 700 24
Kto: 288225661

Inhalte beitragen

Sind Sie der Meinung es gibt wichtige Punkte die bisher nicht in diesem Dokument behandelt wurden? Dann steht Ihnen auch die Möglichkeit ohne finanzielle Mittel dieses Projekt zu unterstützen zur Verfügung. Sie können Anregung zu diesem Punkt geben oder auch einen eigenen Artikel verfassen.

Historie

Dieses Dokument ist aus den unter den selben Namen veröffentlichten Folien hervorgegangen. Daher werde ich die Historie übernehmen und weiterführen.

[1] Keine Sorge jeder noch so kleine oder große Betrag findet meine Dankbarkeit.

Version	vom	Bemerkung
1.00	?	Der erste Wurf mit den Teilen Optimieren, Java optimieren – Allgemein, System und Umgebung, Design, Implementation.
1.01	?	Teile Das Vorgehen, Erweiterungen zu Zeichenketten, Klassen, Containerklassen
1.02	?	Teil Implementation – Codierung : Threads ergänzt, System und Umgebung ergänzt, Ergebnisse zu transient eingearbeitet
1.10	?	Teil Java ist langsamer - Warum, Erläuterungen zur GC aufgenommen, Objekte erzeugen, Objekte verwerfen, Ergänzungen zu Containerklassen (Hashtable), Methoden (getter), Vergleichen
1.11	?	Objekte sichern
1.12	?	Wann sind Sie mit der Optimierung fertig?, 2. Objektpoolbeispiel, Streams, BNO, Component, Überarbeitungen und Ergänzungen u.a. im Aufbau
1.13	02.09.2000	Attribute, Java ist langsamer – Warum?, mehr JVM Wissen, Wahrheit – Sein oder Schein, Low-Priority-Threads, Exceptions, Außerdem wurde mehr Wert auf den Speicherverbrauch zur Laufzeit gelegt, List, HashMap, System und Umgebung – Netzwerk, Design – Netzwerk(CallBack), dynamisches Laden, System und Umgebung - Tools
2.00 beta	17.10.2000	wesentliche Erweiterung zu Datenbanken und Netzwerk
2.00	12.12.2000	Totale Umgestaltung: keine Folien mehr, sondern Fließtext. Wesentliche Erweiterungen zu Streams, Objekte sichern und Datenbank, Anfang eines kleinen FAQ. Und etwas Swing ist auch schon mit drin.
2.10	01.05.2001	Neuer Teil „Performance managen“, Allgemeines zu Performanceoptimierungen als eigenen Teil aufgenommen, Process Entity Muster, Ergänzungen zu System und Umgebung, Tools – serialver, Klarere Trennung zwischen Allgemeiner und Java Implementierung, Implementierung: ObjectOutputStream, Swing-Models, Icon und Tilesets
2.20	22.11.2001	Löschen von JTextComponent, Null Pattern, Singleton Muster, Synchronisieren synchronisierter Methoden, Zählrichtung in Schleifen, Speicherbedarf Innerer Klassen, Schlüsselwort strictfp, Teil 8 Implementierung mit Perl begonnen

Besonderer Dank für diese Version geht an

- Bruno Persi, für seinen Hinweis zum In- und Dekrementtieren,
- Die de.comp.lang.iso-c++ Newsgroup für die Hinweise zum NullPattern

Allgemeines

Glauben ist etwas Grundsätzliches! – Falsches?!?

Dieses Dokument enthält (fast) ausschließlich Hinweise und Tipps, welche ich getestet habe. Allerdings bedeutet testen hier auf der Windows-Plattform (Windows NT 4 und Windows 98). Ein kleines Zitat aus der deutschen Newsgroup soll verdeutlichen, was ich hiermit zum Ausdruck bringen möchte:

> *„Uebrigens treffen Performancebehauptungen nicht fuer alle Plattformen gleichemassen zu. So sollte man z.B. unter FreeBSD oder Linux (Blackdown jdk1.2) einen Vector mittels der Methode toArray umwandeln. Unter Windows war die Methode um den Faktor 2 langsamer, dafuer das Umkopieren in einer Schleife etwas besser.* [1]

Es bietet sich somit immer an die hier gemachten Aussagen zumindest kritisch zu hinterfragen. Das Dokument ist nicht der Weisheit letzter Schluss und gerade neuere Java-Versionen können zu neuen Ergebnissen führen. Irgendwo, schon wieder lässt mich mein Gedächnis im Stich, hab ich mal einen schönen Satz gelesen, welcher auch für dieses Dokument gilt: „Rechtschreib- und Grammatikfehler sind gewollt und dienen der Erhöhung der Aufmerksamkeit beim Leser.“ Na ja, oder so ähnlich…

Java und seine Versionen

Grundsätzlich spreche ich hier nicht von Javaversionen sondern von den Versionen des JDK[2]. Das Dokument beschäftigt sich mit den verschiedensten Version. Die Beispiele wurden unter folgenden JRE getestet:

Java Runtime Envoirement	Hersteller	Testplattform
JRE 1.1.6	Borland	WinNT
JRE 1.1.7	Sun	WinNT
JRE 1.1.8	Sun	Win98
JRE 1.2.0	Sun	Win98
JRE 1.2.2	Sun	Win98
JRE 1.3.0	Sun	WinNT, Win98
JRE 1.3.0	IBM	Win98

[1] Wie gesagt ein Zitat also macht mich nicht für diese Rechtschreibfehler verantwortlich! ☺

[2] JDK 1.0 bis 1.1.8 werden allgemein als Java 1 bezeichnet, während JDK 1.2 bis 1.3.x als Java 2 bezeichnet werden. Wirkliche Sprünge gab es insbesondere zwischen JDK 1.0 und JDK 1.1 bzw. JDK 1.1 und JDK 1.2. Die anderen Versionen waren meist nur Bugfixes und enthielten kleinere (meist sinnvolle) Ergänzungen. 1.3.0 ist die derzeitige Test-JVM.

JRE 1.3.1	Sun	Win98, Win2000
JView	Microsoft	Win98

Obwohl Java 2 (also JDK 1.2 ff.) sich in der Stand Alone Anwendungsentwicklung auf den drei direkt von Sun unterstützten Plattformen durchgesetzt hat, bleibt Java 1 weiterhin ein wichtiger Bestandteil. Dies liegt nicht zuletzt an der Tatsache das Microsoft im Internet Explorer bisher nur Java 1 direkt unterstützt. Das PlugIn von Sun schafft zwar Abhilfe, muss jedoch vom Benutzer vorher installiert werden. Aus diesem Grund wird auch immer wieder auf Java 1 verwiesen. Wer Applets entwickelt muss sich somit fast immer auch mit Java 1 auseinander setzten

Ein weiterer Grund liegt in den unterstützten Plattformen. Nicht jede Plattform verfügt über eine Java 2 Umsetzung und wiederum nicht jede Plattform hat einen JIT Compiler. Also werden auch ein paar Aussagen zu diesen Themen gemacht werden.

Begriffsdefinitionen

Eine Anwendung im hier verwendeten Sinn ist jeder Bytecode, der in der Lage ist eine Aufgabe in der ihm vorherbestimmten Weise durchzuführen.

Daraus folgt, das Applets, Applikationen, Beans und Servlets für mich diese Anforderungen erfüllen. Im Zweifel kann ich darunter auch noch ein neues „Look & Feel“ u.ä. verstehen. Ich will mich dabei nicht zu sehr einschränken.

Verwendete Abkürzungen

Abkürzung	Bedeutung	Erläuterungen
BNO	Benutzeroberfläche	häufig auch als GUI bezeichnet
GUI	Graphic User Interface	Benutzeroberfläche
JDK	Java Development Kit	Java Klassen zur Entwicklung
JIT	Just In Time Compiler	Optimierung für die JVM
JRE	Java Runtime Environment	Ausführungsumgebung für Javaanwendungen
JVM	Java Virtuelle Maschine	Der Übersetzer des Bytecode in Maschinensprache
LAN	Local Area Network	Lokales Netzwerk. Ein LAN ist im allgemeinen ein besonders schnelles Netzwerk, auf welches von außen nur beschränkt zugegriffen werden kann.
MVC	Model View Controller	Ein Konzept zur Trennung von Anwendung und BNO
OO	Objektorientierung	Ein Konzept der Anwendungsentwicklung

OOA	Objektorientierte Analyse	Fachanalyse, welche bereits den OO-Gedanken umsetzt
OOD	Objektorientiertes Design	Modell, welches die Fachanalyse mit Rücksicht auf die verwendete Programmiersprache für die Implementierung vorbereitet
OOI	Objektorientierte Implementation	gleich wieder vergessen!
UML	Unified Modelling Language	Beschreibungssprache für objektorientierte Systeme.

TEIL 1
Allgemeines

Warum ist Java langsamer

Da Sie dieses Dokument lesen sind Sie entweder sehr wissbegierig oder wollen Ihre Anwendungen performanter machen (Entwickeln / Implementieren / Ablaufen lassen). In beiden Fällen ist es wichtig zuerst zu wissen, warum Java Bytecode langsamer ist als viele andere ausführbaren Anwendungen.

Java ist eine objektorientierte Sprache. Objektorientierung führt leider meist zu Anwendungen, welche nicht ganz so performant sind. Dafür sind (gut) objektorientierte Anwendungen meist besser wartbar[1]. Ein OO-Objekt muss nun einmal nach den Informationen gefragt, diese können Ihm nicht einfach entrissen werden. Der zweite wichtige Punkt ist, dass die Anwendung erst zur Laufzeit in Maschinencode übersetzt wird. Dies ermöglicht zwar die Anwendung auf jeder geeigneten JRE ausgeführt werden kann (also plattformübergreifend) jedoch dort jeweils zuerst übersetzt werden muss. Bei älteren JRE kommt noch das Fehlen von JIT Compilern bzw. des Hot Spot Compilers hinzu.

Ein weiterer Grund liegt in der Art wie Java konzipiert wurde. Java sollte eine sichere Sprache werden. Dies führt u.a. dazu, dass bei jedem Objektzugriff zur Laufzeit geprüft wird, ob eine gültige Referenz vorliegt. Sonst kommt die geliebte `NullPointerException`. Zugriffe auf Arrays und Strings werden ebenfalls zur Laufzeit geprüft. Der Zugriff auf ein Element außerhalb des Gültigkeitsbereich führt auch hier zu einer `Exception` (z.B. `ArrayIndexOutOfBoundsException`). Typanpassungen oder auch Casts werden zur Laufzeit geprüft. Neben der `Exception` die hier auftreten kann, bedeutet dies auch, dass für jedes Objekt im Speicher noch Metainformationen vorhanden sind. Referenzen auf Methoden und Attribute erfolgen (meist) über den Namen. Dies erfordert zur Laufzeit eine Prüfung auf deren Vorhandensein und führt ggf. zu einer `NoSuch...Exception`.

Es gibt jedoch auch Vorteile. Sofern sich die Systemarchitektur ändert, haben Sie keinerlei Auswirkungen auf den Bytecode zu fürchten. Anders ausgedrückt wollen Sie Ihren Bytecode auf einem anderen Betriebssystem ausführen benötigen Sie lediglich eine entsprechende JRE. Eine C-Anwendung müssten Sie zumindest neu übersetzen. Und

[1] Dies ist eine Meinung die heutzutage allgemein vertreten wird. Obwohl mir die Erfahrung in nicht objektorientierten Projekten fehlt, sollte m.E. gleiches auch bei guten nicht objektorientierten Anwendungen gelten. Gut bedeutet dabei in Design, Quelltext, Umsetzung, Dokumentation und was sonst einem noch so einfällt.

bei Assembler haben Sie beispielsweise Probleme, wenn Sie Ihre Anwendung auf einen anderen Prozessortyp[1] portieren müssen.

Gute Gründe nicht zu optimieren

Es gibt sehr gute und triftige Gründe nicht zu optimieren. Meist steht der Punkt Optimierung der Anwendung erst dann an, wenn das Programm fehlerfrei lauffähig ist. Bei kleinen Anwendungen ist dies auch ein probates Mittel, da die Machbarkeit hier im Vordergrund steht. Dies führt jedoch zu einigen Gründen, warum Sie nicht optimieren sollten.

1. Bei der Optimierung Ihrer Anwendung kann viel Zeit und Geld investiert werden, ohne dass es zu spürbaren Veränderungen kommt.

2. Optimierungen können zu schlechter lesbaren und wartbaren Quelltext führen.

3. Optimierungen können zu neuen Fehlern in Ihrer Anwendung führen.

4. Optimierungen sind meist Abhängig von Compilern, Ausführungsplattform und anderen Teilen der Anwendungsumgebung.

5. Der / Jeder Objektorientierungsgedanke geht bei der Optimierung verloren.

Keiner dieser Punkte muss Ihre Anwendung betreffen; Sie sollten jedoch vorher abwägen, ob es sinnvoll ist eine Optimierung durchzuführen, wenn die Anwendung in einer Art und Weise lauffähig ist die keinerlei (Performance)–Problem hervorruft.

Gründe zu optimieren

Wenn Sie Ihre Anwendung optimieren müssen dann liegt es meist an diesen Punkten

1. Ihre Anwendung benötigt zu viele Ressourcen:

    1.1. Speicher

    1.2. Prozessorkapazität

    1.3. Bandbreite

2. Ihre Anwendung findet aufgrund dieses Problems keine Akzeptanz.

[1] Hier ist natürlich nicht die Intel-Architektur gemeint, sondern das portieren einer Anwendung z.B. auf den C-64 oder Amiga oder den Palm.

3. Ihre Anwendung erfüllt nicht die Anforderungen des Auftraggebers.

4. Ihre Anwendung ist ohne Optimierung nicht sinnvoll lauffähig.

Bei jedem dieser Punkte ist dann zuerst einmal Analyse zu betreiben, wo der eigentliche Schwachpunkt liegt und wie dieser aus dem Weg geräumt werden kann. Dieses Dokument teilt sich dabei in die drei wesentlichen Teile auf, in welchen eine Optimierung sinnvoll ist: System und Umgebung, Design und Implementierung.

Wann sollten Sie optimieren?

Im Abschnitt „Gründe zu optimieren“ sind bereits wesentliche Punkte die für das Optimieren aufgezählt worden. Wenn wir diesen Abschnitt mit dem Abschnitt „Gute Gründe nicht zu optimieren“ vergleichen lassen sich schon einige Grundregeln für den Zeitpunkt einer Optimierung festlegen.

Performanceoptimierungen sollten nur dann durchgeführt werden, wenn es zu einem Ressourcenengpass, mangelnder Akzeptanz (durch Benutzer oder Auftraggeber) oder zu keinem sinnvoll lauffähigen Produkt kommt.

Ein weiteres Ziel ist sollte es sein Optimierungsmöglichkeiten außerhalb der Anwendungserstellung zu nutzen. Um diese jedoch wirksam testen zu können, ist zumindest ein funktionierender Prototyp Voraussetzung. Auf jeden Fall sollten nötige Performanceoptimierungen vor der eigentlichen Anwendungsauslieferung erfolgen.

Die Performanceoptimierung sollte schließlich zuerst auf das Design zielen bevor die Implementierung in das Visier rückt. Dies hat einige weitere Vorteile. Ein gutes und performantes Design erhöht die Wartbarkeit einer Anwendung erheblich. Bei Performancegewinnen durch Änderung des Design entsteht kein „Trashcode“. Natürlich ist hier insbesondere darauf zu achten, das die Designoptimierung nach Möglichkeit nicht zu einem schlechten Design führen sollte. Der Einbau eines Cache oder Objektpools in das Design ist sicherlich positiv zu bewerten, während der direkte Zugriff auf Variablen ebenso wie die Auflösung der Trennung von Anwendung und Oberfläche (Stichwort MVC) eher in den Teil schlechtes Design fällt[1].

Bleib zum Schluss nur noch die Feststellung, dass in der Analysephase keine Performanceüberlegungen durchgeführt werden. Hier geht es um die fachlichen Anforderungen die Ihre Anwendung bewältigen muss, nicht deren Umsetzung.[2]

---

[1] Im Vorgriff sei nur soviel gesagt, dass es sich bei jedem dieser Punkte um Möglichkeiten handelt dir Performance der Anwendung zu erhöhen.

[2] Hier wird auch keine Annahme darüber getroffen, welche Implementierungssprache gewählt wird. Es sollte daher ggf. auch Mehrfachvererbung eingesetzt werden. Sofern man dies strikt auslegt gibt es auch keine Variablentypen wie `boolean` oder String sondern nur Typen ähnlich Wahrheitswert, Zeichenkette der Länge 12 u.ä.

Wo sollten Sie optimieren?

Es gibt drei wesentliche Ansatzpunkte, um Ihre Anwendung performanter zu gestalten. Sie können Optimierungen in der Anwendungs- / Systemumgebung vornehmen, im Design und in der Implementation. Jeder dieser Punkte hat seine Vor- und Nachteile und Daseinsberechtigung. Grob kann man diese Teile wie folgt umreißen:

Optimieren der Anwendungs- bzw. Systemumgebung (Laufzeitumgebung) bedeutet (meist) ohne direkten Eingriff in die Anwendung. Der Eingriff kann z.B. beim Wechsel der Datenbank nötig sein. Optimierungen werden dabei speziell auf die jeweilige Umgebung abgestimmt und sind nur selten auf andere Systeme übertragbar. Beispiele wären der Einsatz einer anderen JRE oder Datenbank oder auch das Erhöhen des Umgebungsspeichers für die Java Virtuelle Maschine.

Optimieren des Designs bedeutet fast immer einen grundsätzlichen Umbau der Anwendungsstruktur. Dies zieht somit auch Implementierungsarbeiten nach sich. Der Vorteil ist jedoch, das der Quelltext zu einem hohen Anteil eins zu eins in die neue Struktur übernommen werden kann (Copy & Paste) oder nur geringe Anpassungen nötig sind. So kann der Einbau eines Cache innerhalb einer Klasse auch ganz ohne Änderung der Abhängigen Strukturen erfolgen. Hier bietet sich auch der Einsatz von bestimmten Entwurfsmusterns an.

Die Optimierung in der Implementation (hier ist nicht die Folgearbeit nach Designoptimierung gemeint) soll schließlich schnelleren Quelltext erzeugen, ohne dass dieser nicht mehr wartbar wird. Dies bedeutet meist Optimierung des Algorithmus.

Obwohl ich hier bewusst von einigen Optimierungsmöglichkeiten, wie direkten Zugriff auf Objektvariablen, abrate, werden auch diese Möglichkeiten im Laufe dieses Dokumentes zur Vollständigkeit näher besprochen.

Was sollten Sie optimieren

Wo und Was greift natürlich sehr ineinander. Zuerst also noch einmal „Wo sollten Sie optimieren?“ – in der Anwendungsumgebung, in dem Design und in der Implementierung. Während Sie sich bei der Anwendungsumgebung und im Design keinerlei Grenzen zu stecken brauchen, ist bei der Implementierung, also dem Quelltext, etwas mehr Vordenken gefordert.

Die 80 / 20 Regel

Eine Feststellung die sich in den Jahren seit den ersten Programmzeilen herauskristallisiert hat, ist, dass 80% der eigentlichen Aufgabe einer Anwendung in etwa 20% des Quelltextes erledigt wird. Typisches Beispiel dafür ist eine Schleife. Schleifen, insbesondere häufig durchlaufende, enthalten in sich Quelltext der während der Anwendung wesentlich häufiger ausgeführt wird. Es ist daher sinnvoller diesen Teil zu optimieren, als den umliegenden Quelltext. Das Problem welches sich dann meist stellt, wie findet man diese 20% Quelltext. Wie schon erwähnt sind Schleifen immer besonders scharf

zu betrachten. Des weiteren hilft der Einsatz von Profilern. Profiler sind Anwendungen, welche in der Lage sind die Performanceeigenschaften andere Anwendungen zu messen. Dank Sun steht Ihnen mit dem JDK ein einfacher Profiler schon zur Verfügung.

Wann sind Sie fertig mit der Optimierung?

Ein Ende der Optimierungsarbeiten ist grundsätzlich nie bzw. genau dann erreicht, wenn Ihre Gründe zu optimieren (vgl. Seite 9) nicht mehr bestehen. Selbst dann kann jedoch wieder ein Optimierungsbedarf neu entstehen[1]. Je nach entwickelter Anwendung gibt es jedoch noch Unterschiede zu beachten.

Applets und Servlets

sind kritische Anwendungen. Hier ist die eigentliche Anwendung die es zu überprüfen gilt die Präsenz im Netz (Intranet, Internet oder was auch immer). Diese muss mittels geeigneter Werkzeuge (Profiler) einem ständigen Überprüfungsprozess unterliegen. Hierbei kann es sich dann ergeben, dass eine bis dahin gute Javaanwendung plötzlich zum Performanceproblem wird.

Applikationen

sind hingegen relativ leicht zu handhaben. Hier gilt der Grundsatz: Wenn die Gründe wegfallen, kann die Optimierung als abgeschlossen angesehen werden.

Beans

sind die wahre Herausforderung. Ein Bean kann nicht schnell genug sein. Beans sind wie auch Applets und Servlets Teil einer größeren Anwendung. Und hier liegt das Problem. Ein Bean, welches des Flaschenhals in Sachen Performance ist, wird auf Dauer keine Verbreitung finden. Hier liegt das Problem in der Abhängigkeit anderer Entwickler von Ihrer Anwendung. Sollte sich Ihr Bean also als unperformant herausstellen wird sich irgendwann einer dieser Fachleute hinsetzen und eine eigene Entwicklung mit gleichem Ziel vorantreiben.

Neben diesen grundsätzlichen Anmerkungen gilt natürlich, dass Aufwand und Nutzen in einem gesunden Verhältnis stehen sollten.

[1] Stellen Sie sich nur vor, dass aus Kostengründen der Umstieg auf eine andere nicht so schnelle Datenbank bzw. ein anderes Betriebssystem in Ihrer Firma beschlossen worden ist.

TEIL 2
Performance managen

Dieser Teil beschäftigt sich mit Maßnahmen die Sie als Verantwortlicher ergreifen können, um Performanceprobleme von Beginn an bei Ihren Projekt in den Griff bekommen zu können.

Planung

Wie bereits erwähnt ist eine der wesentlichsten Voraussetzung diejenige, dass Sie die Performance Ihrer Anwendung optimieren müssen. Im Vorfeld ist dies nicht immer abzuschätzen, trotzdem bzw. gerade deshalb sollten Sie eine Planung aufstellen. Planen Sie auch für die Performance finanzielle und personelle Ressourcen sowie Zeit ein. Dabei ist es günstiger die Performance bereits von Beginn an in Ihre Planung mit einzubeziehen.

Performance Experten

Versuchen Sie bei mittelgroßen Projekten zumindest zwei Programmierer so ein, dass diese sich zu Performance Experten entwickeln können. Die Ausbildung von internen Mitarbeiter ist dabei üblicherweise preiswerter als externe Speziallisten. Diese können Sie zu Rat ziehen, wenn sich Ihr Projekt bereits in der Abschlussphase befindet. Insbesondere das Internet bietet zahlreiche Informationsquellen, die Ihre Experten mit geringen Kosten nutzen können. Sie sollten jedoch gerade bei mit der Materie wenig vertrauten Mitarbeitern ausreichend Zeit einplanen. Neben dieses „kostenlosen" Quellen sollten Sie ebenfalls finanzielle Mittel für Bücher, Magazine und Tools sowie Zeit für die Suche im Internet und die Evaluierung der Tools berücksichtigen[1]. Gerade für die Suche im Internet sollten Sie genügend Zeit einplanen. Meine Erfahrung hat gezeigt, dass erst nach etwa ½ Jahr die Suche relativ flott von der Hand geht. Erheblicher Aufwand ist jedoch in die tatsächliche Bearbeitung zu stecken. Neben dem reinen Durcharbeiten der Informationen, welche fast ausschließlich in englischer Sprache vorliegen, benötigen Ihre Experten vor allem Zeit zum Testen der gemachten Aussagen. Diese sind leider nicht immer nicht ganz richtig[2].

[1] Natürlich können Sie sämtliche Informationen im Internet beziehen. Der zeitliche Aufwand ist dafür jedoch enorm.
[2] Auch dieses Dokument erhebt nicht den Anspruch auf Fehlerlosigkeit. Dies ist bei der raschen Fortentwicklung von Java kaum möglich.

Performance Spezifikation

Ein weiterer wichtiger Punkt ist die Anforderungen an die Performance Ihrer Anwendung durch den Kunden festzuhalten. Herzu bietet sich das Pflichtenheft an. Dabei sollten Sie möglichst genau werden und die Festlegungen für jede Anwendungsschicht getrennt vornehmen. Hierbei sollten Sie keine Verrechnungen untereinander zulassen. Zur frühen Absicherung Ihrer Spezifikationen sollten Sie einen Prototypen aufsetzen, mit welchem die grundsätzliche Machbarkeit auch der Performance geprüft wird.

Getrennte Performance für Schichten

Die getrennten Performancefestlegungen für Ihre Anwendungsschichten ermöglichen Ihnen neben der Prüfung jeder einzelnen Schicht vor allem gezielt die Performance Ihrer Anwendung zu verbessern. Ein Beispiel zur Verdeutlichung. Ihr Projekt soll der Einfachheit halber aus drei Schichten bestehen. Zum einem die Datenbank, die Anwendung an sich und die Präsentationsschicht (BNO). Die Performance jeder Ihrer Schichten wurde festgelegt. Natürlich besteht untereinander eine Abhängigkeit die nicht zu unterschätzen ist. Falls Sie nun feststellen, dass sowohl die Datenbank- als auch die Anwendungsschicht die gesetzten Performanceanforderungen erfüllen, müssen Sie lediglich die Präsentationsschicht und somit die Oberfläche optimieren. Dies kann durch einen geänderten Ansatz z.B. dynamisch generierte HTML Dokumente statt Applets geschehen. Sie können und sollten auch noch einen Schritt weiter gehen. So könnten Sie unterscheiden zwischen der Performance der Präsentationsschicht im LAN und der über Ihrer Internetpräsenz.

Aber warum darf keine Verrechnung geschehen? Dies lässt sich wiederum an einem Beispiel gut darstellen. Sie haben wieder die drei o.g. Schichten. Obwohl Ihre Anwendungs- und Präsentationsschicht die Anforderungen nicht erfüllt, können Sie Ihr Projekt dank einer besonders schnellen Datenbankanbindung erfolgreich beenden. Leider beschließt Ihr Kunde gerade das Datenbanksystem zu wechseln …

Performance in Analyse und Design

Bereits in der Analyse sollten Sie die Performance Ihres Projektes berücksichtigen. Hierbei machen sich sowohl die globalen als auch die lokalen Entscheidungen bemerkbar. Der Wertebereich einer wesentlichen Variablen kann hier ebenso von Bedeutung sein, wie die Entscheidung einer Client Server Architektur. Da die Analyse die fachlichen Anforderungen festlegt benötigen Sie hier zwar keinen Performanceexperten, Sie sollten jedoch mindestens einen Designer dazu auserkoren. Ein Aufwand von ca. 10% sollten Sie dabei einplanen. Jeder Designer muss sich jedoch über die Auswirkungen seiner Entscheidungen auch im Hinblick auf die Performance im Klaren sein. Dies kann z.B. auch die Überlegung sein, wie häufig und wichtig Abfragen über große Datenmenge sind.

Ein weiterer wichtiger Punkt ist das Testen von „3rd Party Tools“. Dies sollten Sie auf keinen Fall vernachlässigen, da Sie auf die Performance dieser Zulieferungen nur selten Einfluss haben.

Performance testen

Das Testen nimmt einen relativ großen Stellenwert ein. Ihre Spezifikationen müssen in geeigneter Weise in eine Testbasis umgesetzt werden, die noch genauer darstellt, welche Anforderungen erfüllt werden müssen. Der Performancetest ist dabei nachrangig gegenüber der fachlichen Funktionalität sollte jedoch von Beginn an in Ihr Testverfahren mit einbezogen werden. Dies erfordert unter Umständen auch den Aufbau eines eigenen Testsystems. Dabei muss die Anzahl der Benutzer, die Netzwerklast und der Zugriff von und zu externen Ressourcen ebenso beachtet werden, wie die Geschwindigkeit bei einem „Ein Personen Ein System Lokalen Zugriff". Simulationen können sich als besonders geeignet für derartige Tests erweisen.

Die Tests müssen wiederholbar und nachvollziehbar sein. Die Ergebnisse dieser Tests müssen sich in geeigneter Weise in den geprüften Teilen Ihrer Anwendung nachvollziehen lassen. Dies kann durch externe Dokumente ebenso wie durch ein Versionierungstool oder einen Kommentar innerhalb einer Quelltextdatei geschehen.

Performance messen

Für die Messung der Performance gibt es während der Entwicklungszeit genügend Tools (Profiler), zumal dieses Thema mehr und mehr Aufmerksamkeit gewinnt. Für die Performancemessung zur Laufzeit können Sie jedoch eigene Wege beschreiten. Hier bietet sich das Einziehen einer weiteren Schicht in Ihre Anwendung an, welche die Protokollierung der Performance zur Laufzeit vornehmen kann. Dies kann mit relativ wenig Aufwand realisiert werden und ermöglicht einer permanente Überwachung, wie Sie bei Internetpräsenzen bereits üblich ist.

TEIL 3
System und Umgebung

Nun wollen wir uns dem ersten Teil der Optimierung von System und Umgebung widmen. Die Aussagen werden dabei auch andere Teile dieses Dokuments streifen. Damit Sie nicht unnötig sich durch das Dokument suchen müssen, wird jeder Punkt jedoch so ausführlich wie hier notwendig betrachtet.

Vor- und Nachteile

Der wesentliche Vorteil einer Optimierung von System und Umgebung liegt in der Tatsache begründet, dass die Optimierung keine oder nur sich nur gering auf die Implementierung und das Design Ihrer Anwendung auswirkt. Leider ist dies nicht ganz so optimal wie es erst einmal den Anschein hat. Eine andere Datenbank kann schon erhebliche Auswirkungen auf die Implementierung haben. selbst ein Wechsel des Datenbanktreibertyps kann uns nicht zwingend davor schützen.

Der große Nachteil der folgenden Optimierungen besteht in der Tatsache, dass die Optimierungen spezifisch für das jeweilige System vorgenommen werden müssen. Haben Sie nur 64 MB Arbeitsspeicher können / sollten Sie den Umgebungsspeicher für die JVM nicht auf 128 MB setzen. Für Ihre privaten Tools lassen sich jedoch hier kaum Einschränkungen machen, sofern diese nur auf einem Rechner laufen.

Die richtige Java Virtual Machine [JVM]

Das JVM nicht gleich JVM ist, ist allgemein bekannt. Dies liegt natürlich auch an den JIT Compiler und der neueren Version dem Hot Spot. Was aber bedeutet die richtige JVM zu wählen.

Die richtige JVM bedeutet vielmehr die richtige JRE und bei der Entwicklung das richtige JDK zu benutzen. Neben der Quelle Sun Microsystem ist IBM die große Alternative für verschiedene Plattformen. Es gibt jedoch auch weitere Anbieter und ein Vergleich lohnt sich. So bedenken sind jedoch einige wichtige Punkte. Sun Microsystem setzt den Standart und ist immer aktuell. Die Alternativen ermöglichen meist eine schnellere Ablaufgeschwindigkeit, liegen jedoch nach einer neuen Version des JDKs einige Monate im Rückstand. Beachten Sie auch, dass Sie niemanden vorschreiben können, welche JVM er benutzen soll. Ziel sollte somit stets sein, dass Ihre Anwendung zumindest unter der Sun JVM performant genug

läuft. Ein weiterer wichtiger Punkt ist, dass nicht jede Alternative das gesamte Klassenspektrum unterstützt. So gibt es beispielsweise spezielle JVMs für den Serverbereich, welche weder AWT noch Swing unterstützen. Der Einsatz der IBM Implementation der JVM bringt auf meinem System Geschwindigkeitsvorteile von bis zu knapp 200%.

Ein weiterer wichtiger Punkt ist die JVM in den Internetbrowsern. Hier ist die Geschwindigkeit nicht nur vom jeweiligen Browser, sondern auch vom dahinterliegenden Betriebssystem abhängig. Im Internet lassen sich dazu stets einige Vergleiche finden:

Betriebssystem / Browser	Geschwindigkeit
Win 2000 / IE 5.01	72.944
Win 98 / IE 5.01	72.590
Win 98 / Navigator 4.73	60.565
Win 2000 / Navigator 4.73	60.123
Win 2000 / Opera 4 Beta 6	57.736
Win 98 / Opera 4 Beta 6	57.294
Win NT 4 / IE 5.01	49.160
Win NT 4 / Navigator 4.73	44.873
Win NT 4 / Opera 4 Beta 6	37.754
Suse Linux 6.4 / Navigator 4.73	8.795

Tabelle 1 - Testwerte von zd-Bench 1.5 Quelle: http://www.zdnet.de

Bytecode Interpreter

Der Urahn der JVM arbeitete mit dem Bytecode Interpreter und auch die heutigen Virtuellen Maschinen lassen sich immer noch in den Interpretermodus umschalten. Dieser Modus erleichtert insbesondere die Fehlersuche und wird daher von Debuggern gern verwendet.

Der Bytecode Interpreter arbeitet die einzelnen Befehle im Javabytecode wie bei einer Scriptsprache ab. Der jeweilige Befehl wird eingelesen in Maschinencode umgesetzt und ausgeführt. Da dies für jeden Befehl auch bei nochmaligem Aufruf neu geschieht, hält sich die Ausführungsgeschwindigkeit in Grenzen. Sie ermöglichen keine Nutzung der Maschinenarchitektur, wie Cache und Sprungvorhersage. Bytecode Interpreter sind jedoch die Mindestumsetzung einer JVM und daher auf allen Javaplattformen vorhanden.

Bytecode Interpreter waren die ersten JVM und sind daher auch weit verbreitet. Heutzutage ist jedoch grundsätzlich zumindest mit JIT Compilern zu arbeiten, um auch den gestiegenen Performancebewusstsein gerecht zu werden.

Just-In-Time Compiler

Just In Time [JIT] Compiler sind die Nachfolger der Bytecode Interpreter. Sie übersetzen den gesamten Bytecode vor Ausführung in Maschinencode. Wenn dies geschehen ist wird lediglich auf den Maschinencode zurückgegriffen. Durch die Übersetzung des Bytecode

erreichen JIT Compiler eine wesentlich höhere Ausführungsgeschwindigkeit. Ziel der JIT Compiler ist es dabei den Maschinencode möglichst schnell, nicht jedoch möglichst schnellen Maschinencode zu erstellen. JIT Compiler übersetzen grundsätzlich den gesamten zur Verfügung[1] stehenden Bytecode. Dies hat zur Folge, dass JIT Compiler einen sehr hohen Speicherplatzbedarf haben.

JIT Compiler waren für die Ausführungsgeschwindigkeit von Javaanwendungen ein enormer Sprung. Sie können jedoch nicht den einmal erzeugten Maschinencode weiter optimieren. Hier kommt die nächste Generation der HotSpot Compiler zum tragen.

Hot Spot Compiler

HotSpot Compiler sind erweiterte JIT Compiler. Sie haben zusätzlich zum JIT Compiler die Möglichkeit den Maschinencode zu Überwachen und ggf. eine optimierte Neuübersetzung des Maschinencode zu veranlassen. In diesem Zusammenhang arbeitet der HotSpot Compiler wie ein Profiler, welcher die Auswertung und Änderung des geprüften Anwendungsteils übernimmt. Zudem hat der HotSpot Compiler eine altersbasierende Garbage Collection. D.h. während kurzlebige Objekte schneller aus dem Speicher entfernt werden bleiben länger und langlebige Objekte im Speicher. Dies ermöglich dem HotSpot Compiler derartige Objekte schneller zu erzeugen. Durch die Tatsache, dass der HotSpot Compiler den bereits erstellten Bytecode ständig überprüft und ggf. neu übersetzt, ist er statischen / nativen Compilern in diesem Punkt überlegen.

Sun Microsystems hat den HotSpot Compiler in zwei Versionen entwickelt, die Client-JVM und die Server-JVM. Bei der Clientversion steht weiterhin die Geschwindigkeit der Übersetzung und, anders als beim JIT Compiler, des Starten der Anwendung im Vordergrund. Die Client-JVM startet hierbei die Javaanwendung wie ein Bytecodeinterpreter und übersetzt während der Ausführung der Anwendung die zeitkritischen Anwendungsteile. Auch hier gilt, dass der Maschinencode zuerst nicht nach Performancegesichtspunkten entsteht. Erst durch das integrierte Profiling entsteht ein schneller Maschinencode. Anders jedoch die Servervariante. Hier wird der zeitkritische Bytecode bereits vor dem Starten der eigentlichen Anwendung in optimierten Maschinencode übersetzt. Erst dann wird die Anwendung gestartet. Dies ist sinnvoll, da Serveranwendung normalerweise eine wesentliche längere Ausführungsdauer haben als Stand Alone Anwendungen oder Applets.

Es lässt sich letztendlich folgendes feststellen:

Bytecode Interpreter sind langsam, jedoch stets vorhanden. Sie benötigen die geringsten Systemressourcen, da Sie jeweils nur den aktuellen Befehl übersetzen.

JIT Compiler sind Standard uns sollten nach Möglichkeit verwendet werden. Sie erreichen hohe Ausführungsgeschwindigkeiten, fordern jedoch für diese auch hohe Systemressourcen.

[1] Das Einbinden von weiteren Klassen ist immer noch möglich z.B. mit

HotSpot Compiler vereinen die Vorteile von Bytecode Interpreter (sofortiger Start, wenig Systemressourcenanspruch) mit denen von JIT Compilern (schnelle Ausführungsgeschwindigkeit). Sie haben Vorteile gegenüber statischem / nativem Compilieren und sollten die erste Wahl sein. Sie sind jedoch nicht auf allen Plattformen verfügbar.

Thread-Umsetzung

Wenn Sie ein Mehrprozessorsystem Ihr Eigen nennen, müssen Sie noch einen weiteren wichtigen Punkt beachten: Die Umsetzung der Threads in das Betriebssystem.

Java ermöglicht mit seiner Klasse `Thread` und dem Interface `Runnable` nebenläufige Programmteile zu erstellen. Dies allein kann schon zur subjektiven Performancesteigerung führen. Sofern Sie ein System mit mehreren Prozessoren haben, ist die Umsetzung dieser Threads auf das Betriebssystem wichtig. Unterschieden werden dabei drei Typen.

Der Typ „Many to One“ bedeutet, dass alle Java Threads in einem Betriebsystem Thread[1] umgesetzt werden. Dies bedeutet auch, dass Sie nur einen Ihrer Prozessoren nutzen können und sollte daher bei Mehrprozessorsystemen keine Verwendung finden.

Der zweite Typ „One to One“ legt für jeden Java Thread zwingend einen Betriebssystem Thread an. Obwohl dies auf den ersten Blick vorteilhaft scheint, müssen Sie beachten, dass das Erzeugen eines Threads weder in Java noch auf Betriebssystemebene ein performanter und schneller Vorgang ist. Viele kleine Threads können hier Ihre Anwendung letztlich mehr behindern als beschleunigen.

Der dritte Typ „Many to Many“ ermöglicht schließlich der JVM zu entscheiden, ob für einen Java Thread auch ein Betriebssystem Thread angelegt werden soll. Dieser Typ ist generell für Mehrprozessorsystem zu empfehlen.

Gleitpunktdarstellung in der JVM

Mit dem JDK 1.2 hat Sun Microsystem sich dem Problem der erweiterten Darstellung von Gleitpunktdatentypen gewidmet. Die in Java standardmäßig unterstützten Datentypen `float` und `double` besitzen eine Genauigkeit von 32 Bit bzw. 64 Bit. In der IEEE 754 werden jedoch auch erweiterte Darstellungen mit mindestens 43 Bit bzw. 79 Bit vorgesehen. Bei Hardware, die diese Spezifikationen unterstützen kann durch die Wahl einer diese Darstellung unterstützenden Java Virtual Machine die Geschwindigkeit der Rechenoperationen erhöht werden. Eine Java Virtual Machine, die diese Spezifikationen unterstützt ist nicht mehr gezwungen bei den

---

`Class.forName()`.

[1] Hier spricht man meist von Prozess.

einzelnen Rechenoperationen die Genauigkeit auf den der jeweiligen Datentypen herabzurechen[1].

Hardware

Prozessoren

Inzwischen gibt es verschiedene Hersteller, welche sich mit der Herstellung von Hardware beschäftigen, die den Java Bytecode direkt ausführen können oder für diesen zumindest optimiert sind. Aufgrund der relativ hohen Kosten rentiert sich der Einsatz dieser jedoch nur selten. Meist ist es kostengünstiger und effektiver seine Finanzen in die Verbessung des Systems allgemein zu investieren (Prozessor, Netzbandbreite, etc.). Dies führt neben der Beschleunigung von Javaanwendungen auch zu einer besseren Performance der nativen Anwendungen, wie beispielsweise das Betriebssystem.

Grafikausgabe

Ein bisschen Hintergrundwissen noch zur grafischen Ausgabe. Natürlich ist Java nicht direkt unabhängig vom Betriebssystem. Mit einer guten DirectX- und OpenGL-Grafikkarte kann man Swing und den Java2/3D-APIs unter Windowssystemen gut beschleunigen.

Gleitpunktdarstellung

Sofern Ihre Hardware die erweiterte Darstellung von Gleitkommadatentypen[1] gem. IEEE 754 unterstützt kann insbesondere bei umfangreichen Rechenoperationen ein Geschwindigkeitsvorteil durch die Wahl der richtigen Java Virtual Machine erreicht werden.

Native Übersetzung

Auch wenn ich selbst kein Freund von nativen / statischen[2] Compilern bin[3], sollen diese nicht fehlen. Native Compiler übersetzen den Javaquelltext direkt in Maschinencode. Die zweite Möglichkeit ist das Postcompiling. Hierbei wird der Java Bytecode in Maschinencode übersetzt. Dieses Verfahren wird für Serveranwendungen häufig verwendet, da somit eine plattformunabhängige Anwendung erstellt wird, welche nur ggf. für bestimmte Server in den grds. schnelleren Maschinencode übersetzt werden kann. Der Vorteil der nativen Übersetzung liegt u.a. in der Geschwindigkeit. Native Anwendungen sind (sollten) schneller als Bytecode Interpreter und JIT Compiler. Der Start einer nativen Anwendung erfolgt immer schneller als der Start

[1] Dies führt jedoch dazu, dass auf unterschiedlichen JVMs unterschiedliche Rechenergebnisse entstehen. Um dies zu umgehen wurde das Schlüsselwort `strictfp` aufgenommen.

[2] native und statisch wird in diesem Zusammenhang für die gleiche Semantik verwendet. Ziel ist es jeweils eine Anwendung unabhängig von einer JVM lauffähig zu machen.

[3] Falls mir jemand einen Assembler oder C / C++ Kurs sponsert wäre ich dankbar!

über eine JVM. Hier entfällt bereits das Übersetzten des Bytecode in Maschinencode. Nativer Compiler können sich mit der Übersetzung beliebig viel Zeit lassen, da diese nur einmal anfällt. Dadurch ist es möglich besonders schnellen Maschinencode zu erstellen.

Es gibt jedoch auch Nachteile. Der Maschinencode ist jeweils Betriebssystemabhängig und damit geht eine der wichtigsten Eigenschaften von Java, die Plattformunabhängigkeit, verloren. Außerdem ist das nachträgliche Einbinden von Javaklassen nicht mehr möglich. Hier gilt das Prinzip alles oder nichts. Der HotSpot Compiler greift die nativen Compiler noch mit der Möglichkeit des Ändern des Maschinencode zur Laufzeit an. Hier sind nativer Compiler klar unterlegen. Der Maschinencode wird immer aufgrund der jetzigen und wahrscheinlichen Systemkonfiguration zu Laufzeit übersetzt und kann Änderungen somit nicht berücksichtigen. Es gibt zahlreiche Projekte und Firmen, welche sich die native Übersetzung zum Ziel gemacht haben, so dass Sie mit einer Suche im Internet schnell Erfolg haben werden.

Speicher

Die wahrscheinlich einfachste Möglichkeit eine Geschwindigkeitssteigerung zu erreichen ist die Erhöhung des Umgebungsspeichers für die JVM. Natürlich leider die Performance in Bezug auf Speicherverbrauch darunter aber dies meist nur bei Servern ein Problem.

Tastsächlich läuft die JVM nicht schneller sondern ein Teil, die Garbage Collection, muss seltener auf Hochtouren laufen. Dazu sollten wir uns kurz mit der Garbage Collection beschäftigen. Java hat eine automatische Speicherbereinigung[1]. Das bedeutet als Entwickler müssen Sie sich eigentlich keine Sorge um die Müllverwertung, das Entfernen von überflüssigen Objekten machen. Diesen Mechanismus nimmt die Garbage Collection war. Die meisten Garbage Collections arbeiten als Low Priority Thread im Hintergrund. Sofern kein Speicher mehr vorhanden ist, arbeitet diese jedoch mit höherer Priorität als Ihre Anwendung. Dies bedeutet, dass wenn Ihre Anwendung häufig Speichermangel hat, die Garbage Collection die Geschwindigkeit deutlich absinken lässt. Die Erhöhung des Umgebungsspeichers kann dies hinauszögern. Eine andere Möglichkeit ist die Garbage Collection anzuweisen mit den Speicher nur dann zu bereinigen, wenn kein Speicher mehr vorhanden ist. Dies bietet sich insbesondere bei kurzlebigen Anwendungen an.

Dieser Vorteil ist jedoch relativ. Wenn die Garbage Collection auf Grund von Speichermangel läuft, prüft Sie stets alle Objekte im Speicher, ob diese freigegeben werden können. Durch geeignete Implementierung kann dies jedoch vermieden werden. Objekte die eine Referenz auf `null` haben, benötigen keine Prüfung und werden daher übersprungen. Sofern Sie Objekte definitiv nicht mehr benötigen

[1] Zumindest bei Javaobjekten. Bei Einbindung von nativen Bibliotheken sollten Sie selbst darauf achten auch den Speicher wieder freizugeben.

sollten Sie daher die Referenz auf `null` setzen. Weitere probate Mittel ist der Einsatz von Objektpools und Cacheverfahren.

Die Parameter für die JVM sind leider je nach verwendeter JVM und Version unterschiedlich. Zum Prüfen stellt die JVM von Sun den Parameter `-verbose gc` bereit. Die JVM gibt Ihnen somit den jeweiligen Speicherverbrauch an, so dass Sie leicht feststellen können, ob die Zuweisung von mehr Arbeitsspeicher sinnvoll ist. Der Parameter `-noasngc` ermöglicht es Ihnen bei der Sun JVM schließlich die Arbeit der Garbage Collection im Hintergrund auszuschalten.

Tools

Das JDK wird bereits mit einigen Tools ausgeliefert. Mit dem richtigen Einsatz der verschiedenen Tools lassen sich bereits Performancegewinne erzielen.

javac

Der Compiler javac ermöglicht uns bereits einfache Optimierungsmöglichkeiten. Da wir den Quelltext hier nicht mehr verändern müssen, sind Optimierungen durch den Compiler Quelltextmanipulationen vorzuziehen.

Automatische Optimierung

Mit Hilfe des Parameters „`-o`“ wird der Optimierungsmodus bei der Übersetzung in Bytecode eingeschaltet. Dieser kann sowohl Geschwindigkeitsgewinne als auch Gewinne in der Größe des Bytecode bringen. Der Compiler unterstützt dabei unter anderem das Methodeninlining und entfernt nicht erreichbaren Quelltext[1]. Inlining bedeutet, dass private Methoden und final Attribute nicht aufgerufen bzw. deren Wert abgefragt wird, sondern dass der Bytecode bzw. Wert direkt an die verwendeten Stellen kopiert werden. Dies führt zwar zu mehr Bytecode, erhöht jedoch auch die Geschwindigkeit, da der Overhead eines Methodenaufrufs nicht zum Tragen kommt. Ein weiterer Vorteil liegt an der Tatsache, dass Sie nicht per Hand das Inlining durchführen. Dadurch bleibt die Wartbarkeit des Quelltextes erhalten – Sie müssen spätere Änderungen weiter nur an einer Stelle vornehmen. Das automatische Inlining findet jedoch nur bei relativ kurzen Methoden statt.

Es werden jedoch nicht alle bekannten Optimierungsmöglichkeiten genutzt. Beispielsweise werden überflüssige Kopieranweisungen, wie

```
// hier stand noch etwas

a = h;
o = h;
a = o;
```

[1] Nicht erreichbarer Quelltext (engl. Dead Code) ist Quelltext der aus der umgebenden Semantik heraus nie ausgeführt werden kann. Typische

```
// hier folgt noch etwas
```

oder auch überflüssige Zuweisungen, wie

```
// hier stand noch etwas

u = u++;
u = 25;

// hier folgt noch etwas
```

bisher nicht optimiert[1].

Debug Informationen

Der Compiler „javac“ speichert neben dem reinen Bytecode auch noch Metadaten in der Klassendatei. Hierbei werden u.a. die Zeilennummern und Variablennamen gesichert. Dies führt natürlich zu entsprechend größeren Bytecodedateien. Sie können jedoch dem Abhilfe schaffen. Durch Verwendung des Parameters „`-g:none`“ können Sie diese Metadaten entfernen. Nach einem abgeschlossenen Debugging können Sie somit den Bytecode weiter verkleinern.

Die Klasse World können Sie mit den Parametern des Javacompilers so wesentlich Verkleinern.

```
class World{
  public static void main (String [] arg){
    if (true){
      System.out.println (“Hello World”);
    }
  }
}
```

Parameter	Größe des Bytecode	Relativer Zeitverbrauch für die Compilierung
	415 Bytes	880
-g:none -O	335 Bytes	940

Tabelle 2 - Optimierungswerte des Standard Javacompilers

Alternative Compiler

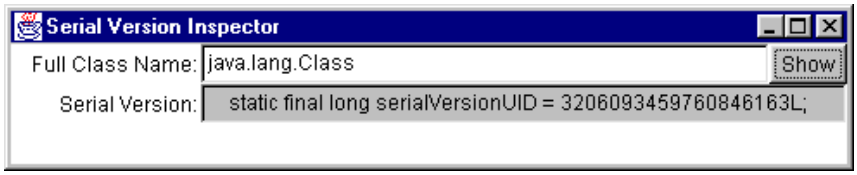
Alternative Compiler bieten gerade in bezug auf die Performance weitere Steigerungsmöglichkeiten. Sie analysieren den Quelltext auf weitere Punkte und optimieren so den Bytecode. Auch Obsfucators führen nicht selten noch Performanceoptimierungen durch. Einer der bekanntesten Compiler ist „`jikes`“. Wie immer haben die Alternativen jedoch das Problem der Aktualität. Alle im Dokument gemachten Aussagen beziehen sich daher auf den Standardjavacompiler.

Beispiele sind Debugausgaben, welche mit Hilfe einer statischen Konstante aktiviert oder deaktiviert werden können.

[1] Dies ist natürlich dank der Nebenläufigkeit (manchmal auch als Parallelität bezeichnet) nicht einfach zu realisieren.

serialver

Mit Hilfe des Tools „`serialver`“ (Serial Version Inspector), welches mit dem JDK ausgeliefert wird, können Sie die Serialversion einer Klasse berechnen. Das direkte Einfügen dieser in serialisierbare Klassen bringt einen geringen Geschwindigkeitsvorteil. Mit Hilfe des Startparameters ‚-show’ können Sie die grafische Oberfläche des Tools starten. Durch Eingabe des qualifizierten Klassennamen erhalten Sie dann die `serialVersionUID`, welche Sie lediglich noch in Ihre Klasse kopieren müssen.



Statt mit der grafischen Oberfläche zu arbeiten können Sie auch direkt den qualifizierten Klassennamen als Startparameter übergeben.

jar

Mit dem jar Packer hat Sun einen weiteren Schritt unternommen, um die Performance zu erhöhen. Gerade bei Netzwerkverbindungen mit niedriger Bandbreite bietet sich die Verwendung von jar Paketen an. Dies ist insbesondere in Verbindung mit Applets vorteilhaft. jar selbst ist ein Packer, welcher auf dem zip Algorithmus beruht. Zusätzlich kann ein jar Paket noch Metainformationen, wie zum Beispiel die Startklasse enthalten. Der Einsatz der jar Pakete hat sich inzwischen auch bei Javaanwendungen durchgesetzt. Dies hat nicht zuletzt etwas mit der Benutzerfreundlichkeit zu tun. Mit ein paar kleinen Handgriffen kann man jar Dateien auch unter Windows zur selbststartende Datei machen.

Obsfucators

Die eigentliche Aufgabe von Obsfucator Tools ist es den Bytecode vor unberechtigtem Einsehen und Dekompilierung zu schützen. Als nützlicher Nebeneffekt wird der Bytecode meist noch wesentlich verkleinert. Realisiert wird dieses, in dem Methoden und Variablennamen durch kurze Zeichenketten ersetzt werden. So kann aus einem „`gibAlter(Person p)`“ schnell ein „`d(P a)`“ werden. Dabei erkennen die Tools, ob es sich um eine externe Klasse oder um eine mit zu optimierende Klasse handelt. Einige Obsfucatoren nehmen außerdem noch Bytecodeänderungen zur Optimierung der Ausführungsgeschwindigkeit vor. Hier ist wie immer natürlich darauf zu achten, dass auch 100% Pure Java Bytecode erstellt wird. Wie vieles hat sich auch hier schon die Open Source Gemeinde an die Arbeit gemacht. Der Obsfucator RetroGuard kann z.B. die jar Datei „swingall.jar“ von 2.420.388 Byte auf 1.737.944 Byte drücken, was etwa 30 % ausmacht[1].

Obsfucators eignen sich jedoch nur, wenn Sie ein fertiges Produkt ausliefern. Wollen Sie ein Bean oder eine Klassenbibliothek anbieten,

[1] Sie sollen natürlich nicht die Java Pakete entsprechend behandeln. Aber für einen Test bieten diese sich durchaus an.

werden Sie mit Klassennamen wie A und B wohl keinen Erfolg haben ☺.

Profiler

Damit Sie eine Anwendung performanter gestalten können, ist es nötig zu wissen, wo der Flaschenhals sich befindet. Profiler sind Anwendungen, welche Ihnen erlauben diese Stellen in Ihrer Anwendung zu finden. Sun Microsystems hat auch hier dem Entwickler bereits an Werkzeug an die Hand gegeben. Die JVM verfügt über einen internen Profiler, welchen Sie lediglich starten müssen. Natürlich hat sich auch hier der Parameter in den laufenden Versionen des JDK geändert.

JDK Version	Parameter
JDK 1.1.x final	-prof
JDK 1.2.x final	-Xrunhprof
JDK 1.3.x final	-Xrunhprof

Die Ausgabe erfolgt in der Datei „java.prof“ und kann mit einem beliebigen ASCII-Editor gelesen werden. Da die Ausgabe an sich jedoch eher kryptisch ist, sollten Sie sich ggf. eine kostenlose Anzeigeanwendung besorgen. Allgemein gern genannt werden hier die Anwendungen „`HyperProf`“ und „`ProfileViewer`“. Als weitere Alternative stehen auch kommerzielle Profiletools zur Verfügung.

Profilertools helfen jedoch nur, wenn Sie in der Lage sind das Design bzw. den Quelltext zu ändern; zur Vollständigkeit sind diese jedoch hier mit aufgeführt.

Externe Daten

Grafiken

Grafiken werden in den verschiedensten Varianten verwendet. U.a. als Teil der BNO oder auch in Spielen. Es gibt zwei wesentliche Arten von Grafiken. Bitmap Grafik ist eine Darstellungsform, wobei hier für jeden darzustellen Punkt die Informationen wie Farbe gesichert werden. Die andere Grafikart ist die Vektorgrafik. Diese kennt End- und Anfangspunkte bzw. Formeln, nach denen die Erstellung der Grafik geschieht sowie Farben oder auch Farbverläufe. Bitmapgrafiken werden hauptsächlich für die Verwaltung von Fotos verwendet. Vectorgrafiken haben den Vorteil, dass Sie wesentlich genauer und beliebig vergrößerbar sind. Schriftarten sind meist eine Art Vectorgrafik. Außerdem ist nochmals zu unterscheiden zwischen statischen Grafiken und nicht statischen Grafiken, den Animationen.

Java unterstützt standardmäßig die Bitmap Grafiktypen ‚GIF‘ und ‚JPEG‘ und seit Java 2 auch ‚PNG‘. Die richtige Wahl des Grafiktyps ist dabei schon eine erste Möglichkeit den Speicherbedarf und somit ggf. auch die Netzwerklast zu minimieren. Ich bin kein Speziallist was Grafiken betrifft aber einige Eigenheiten sollten Ihnen bekannt sein. GIF Grafiken können maximal 256 Farben (8 Bit) darstellen. Für

(hochauflösende) Fotos sind Sie daher ungeeignet. Allerdings können GIF Grafiken einen transparenten Hintergrund haben und auch Animationen enthalten. JPEG Grafiken können 24 Bit (True Color) Bilder verwalten. Die Wahl des richtigen Datenformats kann hier die Größe dieser Dateien verringern. Prüfen Sie auch, ob Sie wirklich animierte GIF Grafiken verwenden wollen. Sollten Sie hier Änderungen vornehmen, können Sie den Speicherverbrauch Ihrer Anwendung zu Laufzeit und auch die Netzwerklast stark vermindern. Besonders das PNG Format ist zu empfehlen. Es verbindet die Vorteile von GIF (z.B. Transparenz) und JPEG (photogeeignet) und unterliegt dabei wie JPEG keinen einschränkenden Rechten.

Gerade für Icons und Infobildern bietet es sich an, das Datenformat Ihrer Grafiken zu prüfen. Hier können Sie noch einige Bytes herausholen. Stellen Sie sich auch die Frage, ob Sie wirklich animierte GIF Grafiken benötigen, da sich hier ein besonders großer Performancegewinn erreichen lässt.

Datenbanken

Der Bereich Datenbanken ist ein wichtiger Punkt um die Performance Ihrer Anwendung entscheidend zu beeinflussen. Hierbei müssen wir zusätzlich noch zwischen Datenbank und Datenbanktreiber unterscheiden, wobei wir letzteres zuerst betrachten wollen. Der Datenbankzugriff wird in Java mittels JDBC realisiert. Für den Zugriff über JDBC werden dabei vier Typen von JDBC Treibern unterschieden.

Datenbanktreiber

Typ 1 – Die JDBC ODBC Brücke

Die JDBC-ODBC-Brücke (kurz Brücke) wird meist zum Testen des Datenbankzugriffs verwendet. Sie hat jedoch, auch wenn Sun Microsystem die Brücke nur als geringes strategisches Ziel sieht, ihre Daseinsberechtigung. Der Vorteil den die Brücke zur Verfügung hat, liegt in der Verbreitung von ODBC. ODBC ist auf fast jedem Betriebssystem vorhanden. Da die Brücke lediglich einen ODBC-Syntax erzeugt und diesen an den ODBC-Treiber weitergibt, ermöglicht Ihnen die Brücke somit einen Datenbankzugriff auf einer Vielzahl von Betriebssystemen. Eine weiterer Vorteil ist die (eigentlich) völlige Unabhängigkeit vom jeweiligen Datenbanksystem. In der Implementierung wird kein Datenbank(treiber)spezifischer Quellcode erzeugt. Somit ist es uns jederzeit möglich die zugrundeliegende Datenbank auszutauschen. Hinzu kommt, dass die Performance in Bezug auf den Speicherverbrauch der eigentlichen Java Anwendung praktisch vernachlässigbar ist. Dies liegt an der Tatsache, dass das Paket `java.sql` Bestandteil der Standard Java API ist. Der gravierenste Nachteil ist jedoch die Performance in Bezug auf die Geschwindigkeit. Durch die häufigen Konvertierungen die bei einem Zugriff auf das Datenbanksystem nötig ist, wird die Geschwindigkeit unser Java Anwendung vermindert. Da dies zu einem extremen Flaschenhals führt, wird die Brücke heutzutage kaum bei professionellen Anwendungen verwendet. Leider erreicht jedoch kein anderer Typ die komplette Datenbankunabhängigkeit. Ein weiterer Nachteil ist das ODBC Management. Abgesehen von

Steuerungsprogrammen für Clients (z.B. MS Netmeeting) und ähnlichen Verfahren muss die ODBC Treibereinrichtung manuell und vor Ort durch einen mit entsprechenden Rechten ausgestatteten Benutzer - meist Administrator - durchgeführt werden. Für einen Prototyp bleibt die Brücke jedoch meist erste Wahl.

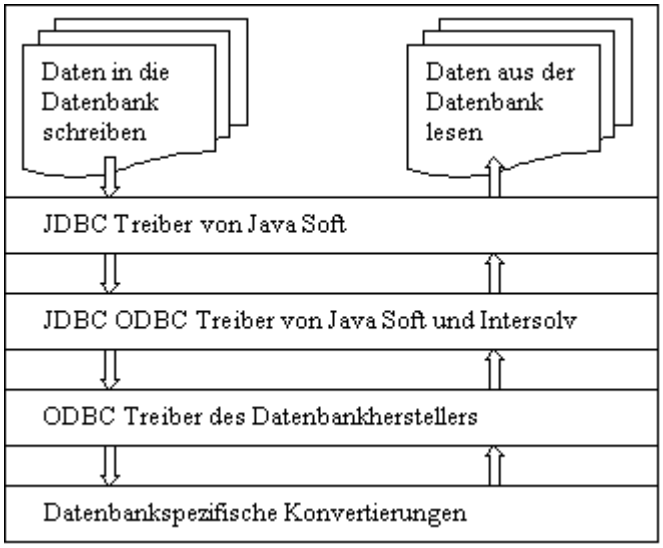


Abbildung 1 - Konvertierungen bei Datenbankzugriff über die JDBC ODBC Brücke

Typ 2 – Plattformeigene JDBC Treiber

Plattformeigene JDBC Treiber konvertieren die JDBC Aufrufe direkt in die Aufrufe der Client-API der Datenbank. Dabei werden die proprietären Datenbankschnittstellen verwendet. Dies macht Ihre Abfragen wesentlich schneller als mit Hilfe der Typ 1 – Die JDBC ODBC Brücke. Damit der direkte Zugriff auf die Schnittstelle Ihrer Datenbank jedoch möglich wird, ist es nötig eine kleine Brücken-DLL auf jedem Datenbankclient zu installieren. Mit Hilfe der Plattformeigenen JDBC Treiber wird es Ihnen jedoch möglich direkt auf die im Unternehmen vorhandene Mittelware des Datenbankmanagmentsystems zuzugreifen. Sie sollten sich bei einem Einsatz jedoch im klaren sein, dass Sie sich abhängig von dem jeweiligen Datenbankmanagmentsystem machen. Der Wechsel zu einer anderen Middleware / Datenbank zieht bei dieser Variante erhebliche Änderungen im Quelltext nach sich. Da außerdem eine native DLL installiert werden muss, büßen Sie weiterhin die Plattformunabhängigkeit ein; auch die Anbindung über Internet stellt Sie bei dieser Variante von größere Probleme.

Typ 3 – Universelle JDBC Treiber

Universelle JDBC Treiber konvertieren Ihre Datenbankaufrufe in ein datenbankunabhänigiges Netzwerkprotokoll, welches von dem Datenbankmanagmentsystem dann in das datenbankspezifische Protokoll umgewandelt wird. Im Vergleich zu JDBC ODBC Brücke entfällt somit je eine Konvertierung. Der Einsatz von Universellen JDBC Treibern ermöglicht Ihnen grundsätzlich eine völlige Unabhängigkeit von der Datenbank und Systemplattform. Aus diesem Grund können Sie diesen Treiber im Intra- / Internet genauso gut einsetzen wie bei

einer Stand Alone Anwendungen. Dabei weist diese Lösung eine wesentlich höhere Performance als JDBC ODBC Brücke auf und sollte dieser auch vorgezogen werden. Universelle JDBC Treiber werden inzwischen sowohl von den Datenbankherstellern als auch von unabhängigen Softwareanbietern angeboten. Dabei unterstützen auch kostenlose Datenbanken den JDBC Standart. Auf der Internetpräsenz von Sun finden Sie eine Liste der jeweiligen Anbieter. Auch bei den Datenbankanbietern können Sie die jeweils aktuelle Treiber finden. Ein wesentlicher Vorteil dieser Treiber ist, dass Sie eine weittestgehende Unabhängigkeit von der Plattform und dem Datenbankmanagmentsystem erreichen können.

Typ 4 – Direkte JDBC Treiber

Der letzte Typ der JDBC Treiber arbeitet schließlich direkt auf der Protokollebene der Datenbank. Durch diese Tatsache ist dieser Typ aus Performancesicht eindeutig zu bevorzugen, da ein Client direkt auf den Datenbankserver, ohne den Umweg über die Middleware, zugreifen kann. Da die Schnittstellen auf Datenbankebene nicht offengelegt werden, können derartige Treiber nur von den Datenbankherstellern selbst entwickelt werden. Da dies jedoch fast immer mit propietären Erweiterungen verbunden ist, ist die Datenbankunabhängigkeit leider nicht gegeben und unterstützen somit direkt die schnelleren datenbankspezifischen SQL Erweiterungen. Bei einem Datenbankwechsel sind hier ebenfalls größere Änderungen am Quelltext nötig.

Es bleibt eigentlich folgendes festzuhalten:

Die JDBC ODBC Brücke ist aus Geschwindigkeitssicht sehr schlecht ermöglicht jedoch einen Datenbankzugriff auf fast jeder Plattform. Für das Testen Ihrer Anwendung ist diese Lösung jedoch brauchbar.

Plattformeigene JDBC Treiber erreichen zwar eine gute Performance sind jedoch nicht Plattformunabhängig und daher im Zusammenspiel mit Java ein schlechte Alternative.

Universelle JDBC Treiber ermöglichen Ihnen bei weittestgehender Plattform- und Datenbankunabhängigkeit eine gute Performance und sind daher erste Wahl.

Direkte JDBC Treiber sind in Sachen Geschwindigkeit durch die anderen Treibertypen nicht zu schlagen. Negativ ist jedoch, dass eine relative hohe Abhängigkeit von der Datenbank die Folge ist.

Eine wirkliche Datenbankunabhängigkeit zu erreichen ist trotz JDBC leider bei weitem nicht so einfach wie sich jeder Entwickler erhofft hat. Hier ist es eine Frage Ihres guten Design die Datenbank soweit zu kapseln, dass ein Austausch dieser nicht zu enormen Quelltextänderungen führt.

Netzwerk

Java ist ein Programmiersprache für ein Netzwerk. Dies zeigt sich unter anderem an den zahlreichen Unterstützungen. Neben der Unterstützung der Soketabfragen exitstieren auch vorgefertigte APIs

für RMI, Corba oder auch das Versenden von eMails. Auch im Bereich der möglichen Javaanwendungen sehen Sie, mit Applets und Servlets, bereits die Ausrichtung von Java auf Netzwerkanwendungen. Gerade in diesem Bereich ist die Plattformunabhängigkeit von Java ein wesentlicher Vorteil. Netzwerkübertragung ist unter Java ein üblicher Datenweg und nimmt auch im generellen Entwicklungsgeschäft mehr und mehr Platz ein.

Technologische Ansätze bzw. Verbesserungen können im Ändern der Netzwerkstruktur z.B. von Token-Ring-Netzwerken[1] auf Ethernet-Netzwerk[2] oder Erhöhung der Bandbreite erfolgen. Eine Bandbreitenerhöhung ist natürlich nur sinnvoll, wenn die Banbreite auch der tatsächliche Flaschenhals ist.

Applets

Applets sind besonders im Internet oder für Datenbankabfragen weit verbreitet. Mit Einführung des Tools jar hat Sun Microsystems hier eine Möglichkeit zur Performanceverbesserung ermöglicht. Dazu wird das Applet inklusive benötigter Ressourcen (Grafiken, Musik,...) in eine jar Datei gepackt. Nachdem dies geschehen ist, müssen Sie lediglich den Aufruf in Ihrer Webseite ändern. Ein derartiger Aufruf könnte nun wie folgt aussehen:

```
<-- vorher:
   <applet code="App.class" heigth=200 width=200>
      <p>Sorry hier war ein Applet!</p>
   </applet>
//-->

<applet code="App.class" archive="a.jar" height=200
width=200>
   <p>Sorry hier war ein Applet!</p>
</applet>
```

Leider hält es auch hier jeder Browserhersteller etwas anders. Während Netscape jar und zip Archive unterstützt lässt der Internet Explorer lediglich cab Archive zu.

Internet Explorer

Eigentlich handelt es sich bei dem folgenden Tipp nicht tatsächlich um eine Möglichkeit die Netzwerkperformance zu erhöhen. Der Tipp ist jedoch insbesondere in Zusammenhang mit Applets zu gebrauchen, so dass er irgendwie hier hinpasst.

---

[1] Im Token-Ring wird ein Paket von einer Station in das Netzwerk (Ring) geschickt und jeder Client schaut nach, ob das Paket für Ihn bestimmt ist. Falls nicht schickt gibt er das Paket weiter. Da der Token solange nicht weitergegeben werden kann, kann nur ein Paket sich maximal im Netzwerk befinden. Die Zwischenschaltung von Routern und ähnlichen kann diese Auswirkungen zwar minimieren, das Paket muss jedoch im Zweifel genau (Anzahl der Clients - 1)-Clients passieren, ehe es aus dem Netz genommen wird und ein neues Token/Pakte in das Netz kommt.

[2] Im Ethernet werden die Pakete an den Empfänger adressiert. Dadurch horcht zwar jeder Client ebenfalls am Netzwerk, es können jedoch auch mehrere Pakete unterwegs sein.

Da es sich selbst in der JRE immer noch um einige Klassen handelt, wird diese normalerweise in Archiven ausgeliefert. Um die Zugriffsgeschwindigkeit zu erhöhen können Sie die dort enthaltenen Dateien in ein Verzeichnis „`root://java//classes`“ entpacken.

Benötigen Sie jedoch mehr Festplattenkapazität lässt sich auch der umgekehrte Weg gehen. Die Archive sind tatsächlich normalerweise nur eine Sammlung der Dateien ohne Komprimierung. Sollten Sie mehr Platz benötigen erstellen Sie die Archive mit einer beliebigen Komprimierung neu.

Sollten Sie Entwickler sein oder Applikationen starten müssen so gelten die vorgemachten Aussagen sinngemäß für die Datei „`classes.zip`“. Ggf. müssen Sie noch den CLASSPATH anpassen.

TEIL 4
Analyse und Performance

Bevor wir uns den Vor- und Nachteilen widmen noch ein kleiner wichtiger Punkt. Performanceüberlegungen haben, bis auf wenige Ausnahmen[1], erst ab der Designphase eine Berechtigung. In der Objektorientierten Analyse[2] sind derartige Überlegungen meist unnötig. Die Fachliche Analyse kann Ihnen aber vor unser Design einige Performancemöglichkeiten eröffnen, welche sonst nur unzureichend oder nicht erkannt werden.

Nebenläufigkeit

Nebenläufigkeit oft auch Parallelität genannt ist ein wichtiger Punkt, welcher in der Analyse zu klären ist. Ich werde hier den m.E. genaueren Begriff Nebenläufigkeit verwenden. Nebenläufigkeit zu bestimmen bedeutet festzulegen, welche Aufgaben nebeneinander durchgeführt werden können. Während bei Einprozessorsystemen meist nur eine subjektive Verbesserung der Geschwindigkeit zu verzeichnen ist[3], ist für Multiprozessorsysteme dies eine elementare Vorraussetzung zur Ausnutzung der Besonderheiten des Betriebssystems.

Nur Sie als Entwickler, welcher die Fachkenntnis hat, können festlegen, welche Abläufe tatsächlich nebenläufig stattfinden können. Mit der UML können Sie diese Nebenläufigkeiten mittels der Tätigkeitsdiagramme sehr gut darstellen. Dabei werden Anwendungsteile, welche nebenläufig sind zwischen einem Synchronisationsbalken dargestellt. Beachten Sie jedoch, dass überflüssige Nebenläufigkeit, selbst falls diese möglich ist, in der Praxis nicht verwendet werden. Ein weiteres Problem ist das Schneiden von Nebenläufigkeiten. Sehr kleine Nebenläufigkeiten sind zwar sehr

[1] Wer noch Make Java – Performance im Folienstil kennt, wird feststellen, dass dort noch die strikte Auffassung vertreten wurde, Performanceüberlegungen haben während der Analyse niemals eine Berechtigung. Diese These musste ich dann doch überdenken.

[2] In der Objektorientierung kann man grob drei Phasen unterscheiden. Objektorientierte Analyse [OOA], Objektorientiertes Design [OOD], Objektorientierte Implementierung [OOI]. OOA bedeutet dabei die fachlichen Anforderungen für die Anwendung zu klären. Hier gilt der Grundsatz der totalen Freiheit, so dass auch Mehrfachvererbung eingesetzt werden kann und sollte. OOD ist schließlich die Aufstellung eines Modells, wie die fachlichen Anforderungen durch ein sauberes programmiersprachenabhängiges Modell gelöst werden können. OOI gibt es nicht! Hier wird anhand des vorher entwickelten Designs die Codierung vorgenommen. Der OO-Gedanke sollte dabei schon vorher im Modell vorhanden sein.

[3] Die tatsächliche Geschwindigkeit kann durch den Overhead der Threaderstellung bzw. des Wechsel zwischen den Thrads sogar abnehmen.

Anwenderfreundlich aber werden selten genutzt. Zu große Nebenläufigkeiten dagegen werden den Anwender verärgern. Ein gesundes Mittelmass ist hier wichtig.

Beispiel: Während Sie das Finale der Fußballweltmeisterschaft im Fernsehen nachverfolgen, trinken Sie Ihr Bier. Oder während der Übertragung des Finale des Synchronschwimmen trinken Sie Ihr Glas Weißwein.

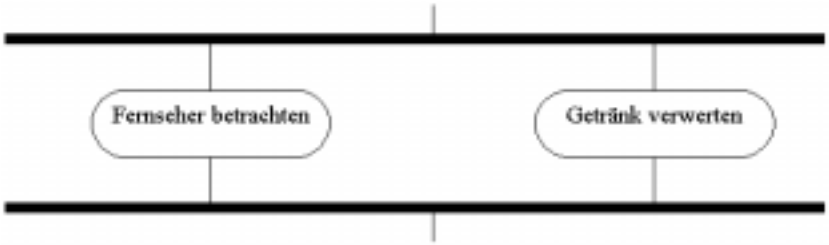


Abbildung 1 - UML Tätigkeitsmodell (Beispiel für Nebenläufigkeit)

Asynchrone Methodenaufrufe

Asynchrone Methodenaufrufe sind ebenfalls ein wichtiges Ergebnis der Fachanalyse. Hier bestimmen Sie, ob auf ein Ergebnis einer Tätigkeit gewartet werden soll, bevor mit der nächsten Tätigkeit fortgefahren werden kann. Dies ist insbesondere im Netzwerkbereich eine gute Möglichkeit die Performance zu erhöhen. Dort wird dieses Prinzip allgemein unter dem Begriff „Callback“ verwendet.

Wir modifizieren unser Beispiel etwas ab. Nun soll der Ehemann auch Herr im Haus sein[1] und die Ehefrau anweisen erstens zur Fußballübertragung umzuschalten und zweitens das Bier zu holen.

Dies könnte nun ein Tätigkeitsmodell zu unserer Erweiterung sein. Aus diesem kann man jedoch leider nicht entnehmen, ob hier asynchrone Methodenaufrufe möglich sind.

[1] Ein wirklich sehr seltener Fall.

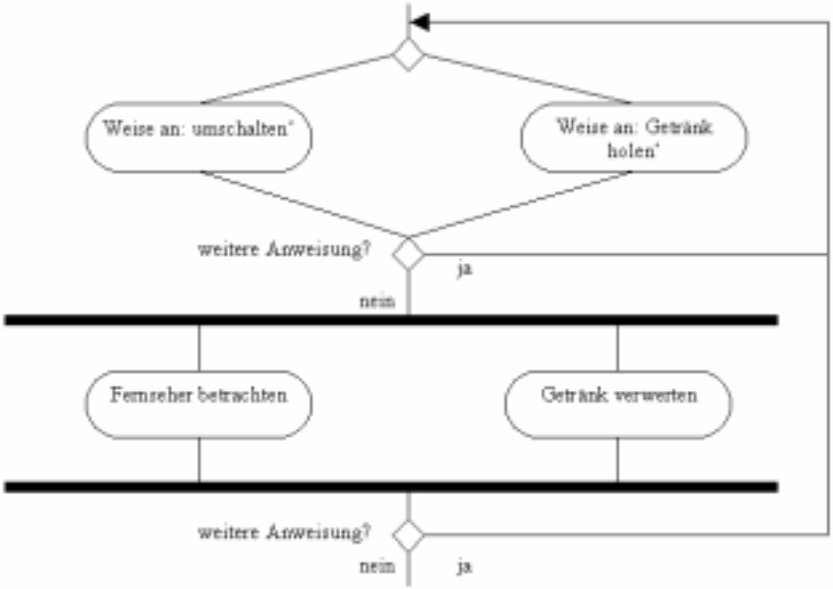
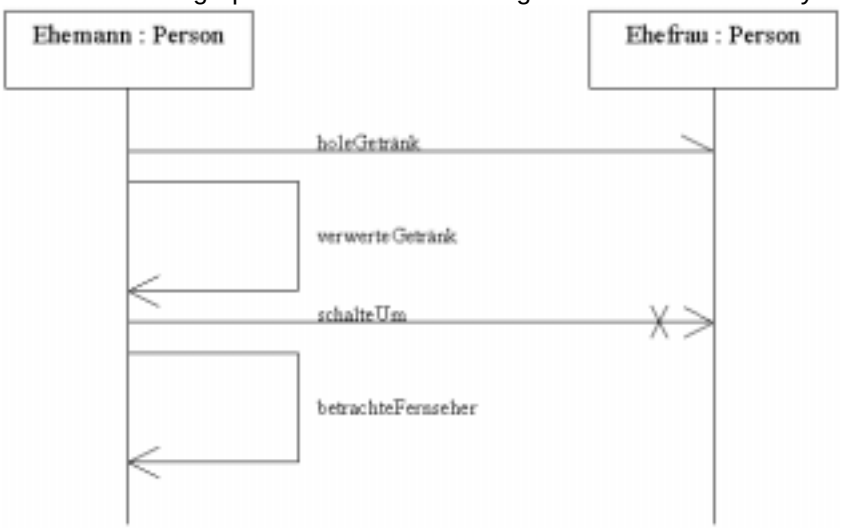


Abbildung 1 -UML Tätigkeitsmodell (Beispiel für asynchrone Methodenaufrufe)

In unserem Beispiel müsste der Ehemann zwar warten bis das Getränk geholt wurde, bevor er trinken kann, er könnte jedoch den Fernseher durchaus betrachten, bevor die Ehefrau umgeschaltet hat. Um dies darzustellen bieten sich die Interaktionsdiagramme an. Ein entsprechendes Diagramm könnte dann so aussehen:

Die Beschreibungssprache UML ist eine Möglichkeit in der OO Analyse



Grundvoraussetzungen für eine performante Anwendung zu legen. Im Rahmen des OO Designs werden Sie diese dann um die Implementationssprachspezifischen Optimierungen erweitern müssen.

Wertebereich

Die Festlegung des Wertebereichs ist eine wesentliche Aufgabe der OOA. Der Wertebereich gibt uns für das Design die benötigten Vorgaben, um den Typ einer Variablen festzulegen. Falls Sie sich schon mit Performance und Java beschäftigt haben, werden Sie eventuell zu der Ansicht kommen, der Typ `int` sei für Ganzzahlen zu bevorzugen, da er der schnellste Typ ist. Richtig! Sie sollten hier jedoch Analyse und Design besser trennen. Die Analyse legt den Wertebereich fest. Das Design bestimmt, welcher Typ den tatsächlichen Wert enthält. Ein Beispiel soll den Unterschied etwas verdeutlichen:

Wir nehmen das Standardbeispiel – eine Person. In unserem Beispiel wollen wir lediglich das Alter einer Person speichern.

In der Analysephase würden Sie eine Ressource Person entwickeln, welche Ihr Alter kennt. Das Alter soll eine Ganzzahl mit einem Wertebereich von 0 bis 200[1] sein. Die Analyse legt somit fest, dass wir eine Ganzzahl benötigen, die 200 verschiedene Werte aufnehmen kann. Im Design liegen jedoch andere Prämissen. Hier ist es sinnvoll für die Ausführungsgeschwindigkeit in Java den Typ zur Laufzeit als `int` festzulegen. Es kann jedoch sein, dass wir für das Sichern unserer Werte den Typ `short` bzw. `byte` verwenden wollen, da diese bereits entsprechende Werte aufnehmen können und weniger Speicherplatz benötigen.

Die Trennung von Analyse und Design ist hier für unsere Performanceüberlegungen nötig, da nur so beide Seiten , hier Speicherbedarf und Ausführungsgeschwindigkeit, betrachtet werden. Für die Ausführungsgeschwindigkeit wird dem schnelle primitive Typ `int` der Vorzug gegeben. Für die Sicherung wird jedoch der Typ `byte` verwendet, welcher weniger Speicherplatz benötigt.

[1] Dies sollte selbst für langlebige Personen ausreichen.

TEIL 5
Design und Performance

Design ist ein Thema, welches sich immer (wieder) großer Beliebtheit erfreut. Betrachtet werden dabei jedoch meist die Vor- und Nachteile in Bezug auf Wartbarkeit, Entwicklungszeit und Fehleranfälligkeit. Unbestritten haben Entwurfsmuster[1] Vorteile. Die entwickelte Anwendung baut auf einem bereits bekannten und erfolgreich eingesetzten Prinzip der Klassenmodellierung auf. Dadurch wird die Fehleranfälligkeit und Entwicklungszeit deutlich verringert, während die Wartbarkeit zunimmt. Nur selten fließen dabei jedoch Performancebetrachtungen ein. Diesem Thema werden wir im folgenden nachgehen.

Das Optimieren des Designs ist dabei Grundlage für eine optimal performante Anwendung. Ihre Designentscheidungen beeinflussen wesentlich die Performanceeigenschaften Ihrer Anwendung in bezug auf benötigte Bandbreite, Arbeitsspeicher und auch Geschwindigkeit. Eine performante Anwendung kann natürlich auch ohne das optimale Design entstehen und ein optimales performantes Design hat nichts mehr mit Objektorientierung zu tun, wie wir sehen werden. Trotzdem ist es sinnvoll zumindest einen Blick auf Performancemöglichkeiten im Design zu werfen.

Vor- und Nachteile

Der große Vorteil in einem performanten objektorientierten Design ist, dass Sie sich die Vorteile der Objektorientierung waren und gleichzeitig eine performante Anwendung entwickeln können. Objektorientierte Anwendungen gelten allgemein als wesentlich langsamer als Programme, welche mit herkömmlichen Programmiersprachen / bzw. -konzepten entwickelt wurden. Dies muss nicht sein. Leider müssen Sie dazu Ihr ansonsten gutes OOD nochmals betrachten und ggf. abändern. Zudem ist der Schritt von einem performanten objektorientierten Design zu einem performanten „Etwas“ nicht sehr groß. Hier heißt es Mäßigung üben. Auf einige der Stolperstellen werden wir zum Schluss des Teils noch einmal zurückkommen.

Datentypen

Im Design werden unter anderem die Datentypen festgelegt. Aus diesem Grund ist hier eine gewisse Grundkenntnis der JVM nötig, um performante Datentypen zuzuweisen.

[1] engl. Design Pattern

Die JVM ist stackorientiert, wobei dieser Stack 32 Bit[1] breit ist. Das bedeutet, dass 32 Bit breite Datentypen schneller verarbeitet werden können. Eine Fließkommazahl wird nicht so schnell verarbeitet wie Ganzzahlen. Weiterhin ist zu beachten, dass lokale Variablen wesentliche schneller verarbeitet werden können. Bei diesen ist zusätzlich zu beachten, ob es sich um Slotvariablen handelt. Die ersten 128 Bit, welche sich auf dem Slot befinden können mittels speziellem Code in der JVM besonders performant verarbeitet werden, so dass sich diese z.B. als Zählvariablen anbieten. Der Slot wird dabei nach einer bestimmten Reihenfolge gefüllt. Handelt es sich um eine Objektmethode so enthalten die ersten 32 Bit eine Referenz auf das Objekt. Die folgenden Bits werden zuerst mit den Parametern und dann mit lokalen Variablen aufgefüllt. Sammlungen sind dabei lediglich als Referenzen zu verstehen. Die Slotvariablen machen sich jedoch nur bei Java Interpretern deutlich bemerkbar.

Primitive und Referenzen

Da der Typ `int` und Referenzen jeweils 32 Bit breit sind, werden diese am schnellsten verarbeitet. Der Typ `float` kann als Fließkommazahlen nicht so performant verarbeitet werden. Die Typen `long` und `double` sind jeweils 64 Bit breit. Die verbleibenden primitiven Datentypen `boolean`, `byte`, `short` und `char` werden intern in den Typ `int`, für jede mit ihnen auszuführenden Operation, gecastet. Für die Operationen inkl. Laden und Sichern wird hier der selbe Bytecode verwendet, wie beim Typ `int` Nach Abschluss dieser werden sie zurück in den Ursprungstyp gecastet. Aus diesem Grund sind diese Typen, wenn es um Geschwindigkeit geht, zu vermeiden. Der benötigte Speicherbedarf entspricht jedoch dem Typen.

Als Designer obliegt es Ihnen somit zu prüfen, ob Sie einen Ganzzahltypen, auch wenn er als mögliche Wertbereiche beispielsweise nur 0..10 hat, als Javatyp `int` deklarieren. Dies kann zu erheblichen Geschwindigkeitsvorteilen führen. Natürlich sollen Sie die Fachanforderungen dabei nicht außer Acht lassen. Diese können in bezug auf andere Performancepunkte, wie Netzwerklast genutzt werden[2].

Besonderheiten in Arrays

Arrays haben eine etwas abweichende Speichernutzung. Datentypen nehmen auch hier grundsätzlich den Platz ihres Typs ein. Die Typen `short` und `char` werden dabei mit einer Breite von 16 Bit verwaltet. Verwenden Sie ein Array als Behälter für `boolean` oder `byte` Werte, so nehmen beide Typarten eine Bitbreite von 8 ein.

---

[1] Ich möchte allerdings nicht ausschliessen, dass sich das nicht mal ändern kann.

[2] Ein Beispiel dazu findet sich im Teil Implementierung zur Schnittstelle Externalizable, wo ein Datentyp zur Laufzeit als int deklariert, jedoch als short persistent gemacht wird.

Sammlungen

Sammlungen oder auch Collections sind einige der Klassen / Objekte, welche mit am häufigsten Verwendung finden. Da Sammlungen auch nur bestimmte Datentypen sind, welche andere Datentypen in primitiver Form oder als Referenzen beinhalten, müssen Sie einige Einzelheiten wissen, um einen geeigneten Datentyp als Designer auszuwählen.

Synchronisiete Sammlungen

Bis zum JDK 1.2 hatten Sie keine große Auswahl. Seit dem hat sich jedoch eine Menge getan. Es wurde der Typ `java.util.Collection` in Form eines Interfaces eingeführt. Auch die bisherigen Typen, wie `Vector` wurden entsprechend angepasst. Bis zum JDK 1.2 bedeutet die Verwendung von Sammlungen immer auch Synchronisation. Wollen Sie diese verhindern, so mussten Sie zwangsweise eigene Klassen verwenden. Den Quelltext dieser können Sie dabei von den synchronisierten Klassen übernehmen. Ab dem JDK 1.2 hat sich eine Wendung vollzogen. Sammlungen sind grds. nicht synchronisiert. Dies erhöht die Zugriffsgeschwindigkeit enorm. Lediglich die aus dem alten JDK Versionen übernommenen Klassen wurden dahingehend nicht verändert. So ist die Klasse `Vector` immer noch synchronisiert. Es wurden für diese jedoch entsprechende nicht synchronisierte Sammlungen eingeführt.

Es ist aber für viele Aufgaben durchaus sinnvoll bzw. nötig mit synchronisierten Sammlungen zu arbeiten. Dies stellt jedoch keine großen Anforderungen an die Implementierung dar. Mit Hilfe der Fabrikmethoden der Klasse `java.util.Collections` können Sie synchronisierte Sammlungen erzeugen.

```
// Beispiel synchronisieren einer TreeMap

TreeMap treeMap;
treeMap = (TreeMap)Collections.
        synchronizedSortedMap(new TreeMap());
```

Beim Aufruf einer Methode wird die Synchronisation durch den von Ihnen soeben erzeugten Wrapper bereit gestellt. Dieser leitet daraufhin den Aufruf an Ihre ursprüngliche Sammlung weiter. Die beiden Klassen Hashtable und Vector wurden zwar nachträglich an das Collection-Framework angepasst, sind jedoch weiterhin synchronisiert. Aber auch hier hat Sun Microsystems an Sie gedacht. In der Beschreibung zur Sammlung `ArrayList` ist daher folgendes zu lesen: „`This class is roughly equivalent to Vector, except that it is unsynchronized.`“ `ArrayList` ist somit der nicht synchronisierte `Vector`. In der Dokumentation zur Klasse `HashMap` finden wir schließlich folgenden Satz „`The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.`“

Gegenüberstellung von Sammlungen

nicht synchronisiert	synchroisiert
`HashSet`	
`TreeSet`	
`ArrayList`	`Vector`
`LinkedList`	`Stack`
`HashMap`	`Hashtable`
`WeakHashMap`	
`TreeMap`	

Sammlungen eignen sich grundsätzlich nur für Objekte. Sollten Sie eine große Anzahl von primitiven Datentypen sichern wollen ist ein Array die bessere Alternative.

Nicht jede Sammlung ist für jeden Zweck geeignet. Werden die Elemente einer Sammlung häufig durchlaufen, so ist die `LinkedList` mit einer besseren Performance ausgestattet als die `ArrayList`.

Versteckte Sammlungen

Das sich die Sammlungen nicht für primitive Datentypen eignen, liegt daran, dass eine Sammlung nur Objekte aufnehmen kann. Die Entwickler von Java wollten sich jedoch offenbar nicht dieser Einschränkung unterwerfen und haben sich daher eine Hashtable für `int` Werte geschrieben.

Die IntHashtable

Diese `Hashtable` ist versteckt im Paket `java.text`. Da Sie netterweise final deklariert ist, ist ein Aufruf leider nicht möglich. Hier hilft nur „Copy & Paste“. Die Verwendung dieser Klasse ist jedoch wesentlich performanter als das Verpacken von `int` Werten in `Integer`.

Erweiterungen

Es gibt Punkte, welche in der Analysephase keine oder nur geringe Beachtung verdienen, jedoch für eine performante Anwendung zu berücksichtigen sind.

Hierzu zählen insbesondere die Prüfung des Einbaus von Caches bzw. Objektpools.

Cache

Ein Cache ist einer sehr effiziente Möglichkeit die Geschwindigkeit einer Anwendung zu erhöhen. Ein Cache ist hier als halten einer Referenz auf ein Objekt einer Klasse zu verstehen. Immer wenn Sie ein Objekt dieser Klasse benötigen, prüfen Sie zuerst, ob bereits ein entsprechendes Objekt vorliegt. Falls dies nicht der Fall ist erzeugen Sie ein Objekt und sichern dies im Cache. Nun arbeiten Sie mit dem Objekt im Cache. Das Erzeugen von Objekten ist eine sehr zeitaufwendige Angelegenheit. Um ein Objekt zu erzeugen müssen die Konstruktoren der Klasse und aller Superklassen verarbeitet werden. Dies führt zu einem enormen Overhead, welcher durch einen Cache in der Anzahl verringert wird.

Ein Cache bietet sich für Netzwerkanwendungen an, da so die Netzwerkkommunikation minimiert werden kann. Objekte können lokal beim Client im Cache gehalten werden und müssen nicht bei erneutem Gebrauch wieder über das Netzwerk geladen werden. Bei Applets bietet sich diesen Vorgehen z.B. bei Multimediaobjekten wie Bilder und Musik an.

Objektpool

Ein Objektpool kann ebenfalls eine Anwendung wesentlich beschleunigen. In einem Objektpool befinden sich Objekte in einem definierten Ausgangszustand (anders Cache). Objekte, die sich in einem Objektpool befinden, werden aus diesem entliehen und nach der Verwendung wieder zurückgegeben. Der Objektpool ist dafür zuständig seine Objekte nach dem Gebrauch wieder in den vordefinierten Zustand zu bringen.

Für die Verbindung zu Datenbanken sind Objektpools gut zu verwenden. Eine Anwendungen / ein Objekt, welches eine Anbindung zur Datenbank benötigt entleiht sich ein Objekt, führt die nötigen Operationen aus und gibt es wieder an den Objektpool zurück.

Caches und Objektpools bieten eine gute Möglichkeit Ihre Anwendung zu beschleunigen.

Ein Cache sollten Sie verwenden, wenn der Zustand eines Objektes / einer Ressource für Ihre Anwendung nicht von Interesse ist.

Objektpools sollten Sie in Ihrer Anwendung verwenden, wenn Sie Objekte / Ressourcen mit einem definierten Zustand benötigen.

Netzwerkdesign

Auch für das Netzwerk müssen Sie als Designer bestimmte Überlegungen anstellen. Sobald Sie Daten über ein Netzwerk austauschen können bisher kleinere Performanceeinbussen sich deutlicher auswirken. Hier müssen Sie als Designer einige Punkte optimieren bzw. vorsehen, die während der Analyse vorbereitet wurden. Während bei lokalen (nicht verteilten) Anwendungen der Methodenaufruf keine relevanten Performanceengpässe auslöst (und

trotzdem optimiert werden kann), ist bei verteilten Anwendung ein Methodenaufruf anders zu betrachten. Bei der Kommunikation in lokalen Anwendungen besteht die Zeit, in welcher der Aufrufer blockiert wird, asynchronen Aufrufen lediglich aus der Zeit, in welcher die Methoden aufgerufen wird. Bei synchronisierten Aufrufen ist noch die Zeit der Ausführung der Methode hinzuzurechen. Bei verteilten Anwendungen kommen noch weiter Zeitfaktoren hinzu. Unter RMI müssen ggf. noch Objekte serialisiert werden. Der gleiche Effekt ist das Verpacken der Informationen, wie die Signatur und benötigten Daten bzw. Referenzen, sowie zusätzlich das Transformieren dieser in das IIOP Format unter CORBA. Hinzu kommt ebenfalls die benötigte Zeit für den Transfer über das Netzwerk.

Bei einer verteilten Anwendung wirken sich bestimmte Modellierungen wesentlich Performancekritischer aus, als bei lokalen Anwendungen. Eine enge Kopplung von Objekten bedeutet hier nicht nur viel Kommunikation zwischen den Objekten sondern auch eine höhere Netzwerkauslastung. Dies können Sie durch verschiedene technologische Ansätze bzw. Verbesserungen und durch ein gutes Design optimieren.

Callback

Wenn Sie sich bei der Analyse die Arbeit gemacht haben, um die asynchronen von den synchronen Methodenaufrufen zu trennen, werden Sie als Designer nun die Möglichkeit haben die Infrastruktur aufzubauen.
Callbacks sind eine (in CORBA bereits weit verbreitete) Möglichkeit die Netzwerklast zu verringern. Dabei wird dem Server eine Referenz auf den Client übergeben. Während ansonsten der Client in regelmäßigen Abständen den Server nach Neuigkeiten fragt[1], informiert nunmehr der

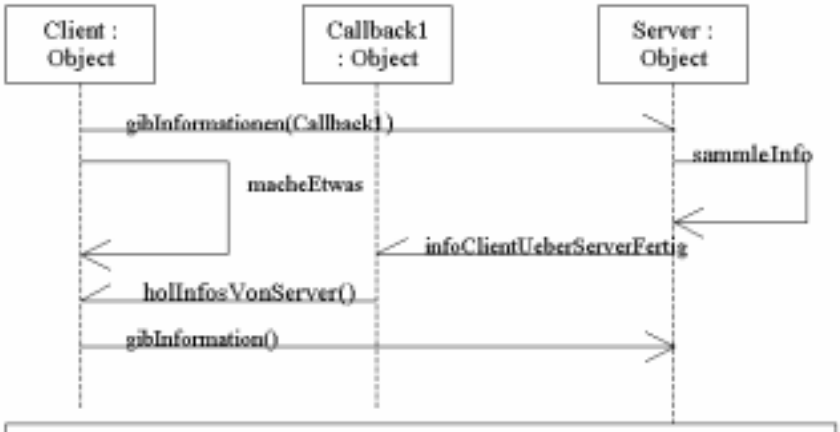


Abbildung 1 - Sequenzdiagramm für Callbackaufruf

Server den Client über Änderungen die für Ihn relevant sind. Optimalerweise verbindet man das Ganze noch mit dem Observer-Pattern (Beobachter-Muster) um den Server unabhängig vom jeweiligen Client zu machen.

[1] Viele Mailingprogramme arbeiten nach diesem Prinzip. Sofern Sie bei Ihrem Mailingprogramm einstellen können, wann bzw. nach welcher Zeitspanne es neue Nachrichten abrufen soll, können Sie auf ein derartiges Verfahren bauen.

Das Callback-Objekt und Client-Objekt befinden sich dabei lokal. Bei der Anfrage an den Server-Objekt wird dem Server das Callback-Objekt übergeben und durch den Client mit seiner weiteren Aufgabe fortgefahren. Ihr Server-Objekt führt nun ohne das Client-Objekt zu blockieren die aufgerufene Methode aus. Nachdem diese ausgeführt wurde benachrichtigt er das Callback-Objekt. Dieses ruft nunmehr auf dem Aufrufer eine entsprechende Methode auf. Die Vorteile sind dabei klar erkennbar. Während es normalerweise zu einem Blockieren des Aufrufer bis zur Abarbeitung der aufgerufenen Methode kommt kann dieser weiter arbeiten.

Natürlich muss man auch hier die bessere Netzwerkperformance mit einem mehr an Bytecode bezahlen. Dies dürfte jedoch bei den meisten Anwendungen – und den heutigen Rechnern – eher in den Hintergrund treten. Bei kurzen Antwortzeiten können Sie jedoch die Performance Ihrer Anwendung verschlechtern. In diesen Fällen kann der Overhead der Netzwerkübertragung nicht durch das nebenläufige weiterarbeiten des Client ausgeglichen werden. Noch besser ist es eine entsprechende Adapterklasse zu erstellen. Diese wird über das Netzwerk angesprochen und kann nun ohne den Netzwerkoverhead das lokale Objekt Meier ansprechen.

Um den Overhead durch die Netzwerkübertragung zu verringern, sollten Sie ein weiteres Augenmerk auf die Methodenaufrufe legen.

Methodenaufruf

Eine wichtige Optimierungsmöglichkeit bei allen Anwendungen, welche sich jedoch besonders bei verteilten Anwendungen bemerkbar machen, ist die Verwendung von asynchronen Methodenaufrufen (siehe Callback und Asynchrone Methodenaufrufe).

Eine weitere weit verbreitete Möglichkeit die Performance zu erhöhen, ist das Teilen der Ergebnismengen. Dabei werden zuerst die wahrscheinlich gewünschten Ergebnisse geliefert. Benötigt der Anwender / Aufrufer weitere Ergebnisse so werden diese auf Anfrage nachgeliefert. Ein Vorteil ist, dass während der Zusammenstellung des Ergebnisses bereits der Anwender ein Ergebnis sieht und subjektiv die Anwendung schneller wirkt. Sofern der Anwender bereits durch die ersten übermittelten Daten zufrieden gestellt werden kann, wird die Netzwerklast wesentlich verringert. Sofern Sie eine Informationssystem entwickeln werden Sie, ein derartiges Verfahren vermutlich verwenden. Dies ist einer der Gründe, weshalb Internetsuchmaschinen dieses Prinzip nutzen. Der Speicherverbrauch beim Aufrufer wird ebenfalls vermindert, da nur die gerade angeforderten Daten im System gehalten werden müssen. Bei der Umsetzung mittels Java sind u.a. noch andere Punkte ausschlaggebend. Stellen Sie sich eine Personenrecherche in einem umfangreichen System vor, die einige

hundert Ergebnisse liefert, die Sie optisch darstellen wollen. Hier kann der Aufbau der entsprechenden grafischen Komponente Ihr System ausbremsen.

Methodenaufrufe können noch weitere Probleme erzeugen. Eine große Anzahl von Methodenaufrufen in verteilten Objekten führt zu einer hohen Netzwerklast. Hierbei kann es insbesondere in Verbindung mit CORBA sinnvoll sein diese in einem Methodenaufruf zu binden. Hierbei bieten erweitern Sie die Schnittstelle Ihrer Klasse bzw. Ihres Subsystem derart, dass Sie für mehrere Operationsaufrufe eine gemeinsame Operation anbieten.

Der Vorteil liegt in der Senkung des Overheads bei einem Methodenaufruf. Da die Daten für das Senden über das Netzwerk zuerst verpackt werden müssen, können Sie hier Performancegewinne erreichen. Hier bieten sich gerade bei RMI weitere Möglichkeiten, die im Teil Implementierung Sicherung und Wiederherstellung von Objekten genauer betrachtet werden.

Das Vorgehen entspricht dem Fassade-Muster, da die Kommunikation hier durch eine kleinere einheitliche Schnittstelle verringert wird. Dies lässt sich neben der Erweiterung der Schnittstelle auch durch eine Verkleinerung erreichen, indem ein Objekttyp zum Übertragen verwendet wird, welcher die benötigten Informationen beinhaltet. Des selbe Ansatz lässt sich auch durch das Verpacken in XML Dokumente umsetzen.

Entwurfsmuster

Auch die sogenannten Entwurfsmuster können bei der Erstellung von performanten Anwendungen helfen. Wir haben im vorigen Abschnitt bereits das Callback Muster näher betrachtet.

Fliegengewicht[1] Muster

Zweck

Es sollen mit Hilfe des Musters Fliegengewicht große Mengen von Objekten optimal verwendet bzw. verwaltet werden.

Verwendungsgrund

Die durchgehende Verwendung von Objekten ist für Ihre Anwendung grundsätzlich vorteilhaft, die Implementierung führt jedoch zu Schwierigkeiten. Durch das Fliegengewicht Muster, können Sie ein Objekt in unterschiedlich vielen Kontexten verwenden. Dieses Fliegengewicht-Objekt enthält dabei die kontextunabhängigen Informationen und kann nicht von einem anderen Fliegengewicht-Objekt unterschieden werden. Zu diesem Fliegengewicht-Objekt

[1] engl. Flyweigth

existiert ein weiteres Objekt, welche die kontextabhängigen Informationen beinhaltet[1].

Vorteile

Das Anlegen von Objekten ist nicht gerade eine sehr performante Angelegenheit und kann mit Hilfe des Fliegengewicht Musters deutlich verringert werden. Dies führt auch in Bezug auf die GarbageCollection zu besseren Zeiten und kann Ihre Anwendung daher beschleunigen.

Nachteile

Mir sind keine Nachteile bekannt. Kennen Sie welche?

Fassade[2] Muster

Das Fassade Mustersermöglicht es sowohl die Anzahl der verteilten Objekte als die Anzahl der Methodenaufrufe an die verteilten Objekte zu verringern. Dieses Muster befasst sich mit dem Problem, dass ein Client viele Methoden auf verteilten Objekten aufruft, um seine Aufgabe zu erfüllen.

Ziel

Die Verwendung des Fassade Musters soll eine einheitliche Schnittstelle zu einem Subsystem bieten. Dabei ist es nicht Aufgabe des Fassade-Objektes Aufgaben selbst zu implementieren. Vielmehr soll die Fassade die Aufgaben an die Objekte des Subsystems weiterleiten. Diese Klassen kennen jedoch die Fassade-Objekt nicht.

Das Process-Entity Muster ermöglicht die Anzahl der Methodenaufrufe auf verteilten Objekten, sowie die Anzahl der verteilten Objekte selbst zu verringern.

Verwendungsgrund

Das Fassade Muster soll, als abstrakte Schnittstelle, die Kopplung zwischen einzelnen Subsystemen verringern. Zusätzlich soll mit Hilfe der Fassade es Fremdsubsystemen möglich sein, ohne Kenntnis über den Aufbau eines Subsystems dieses allgemein verwenden zu können.

Das Process-Entity Muster soll die Anzahl der verteilten Objekte und der Methodenaufrufe auf verteilten Objekten verringern. Zu diesem Zweck wird für eine Aufgabe ein Fassadenobjekt erstellt (Process), welches die Schnittstelle zur Bewältigung der Aufgabe anbietet. Dies ist das einzige verteilte Objekt. Dass Process Objekt übernimmt serverseitig die Aufgabe und delegiert diese an die (nun) lokalen Entity Objekte.

---

[1] Das Beispiel aus Entwurfsmuster (siehe Quellen) hat mich hier begeistert. Der Editor „Doc“ hat für ein Dokument mit 180.000 Zeichen lediglich 480 Zeichenobjekte angelegt.

[2] engl. Facade

Performancegründe

Für die Verwendung in Netzwerken, können Sie mittels des Fassade Musters die Anzahl der Operationsaufrufe verringern. Bei der Kommunikation über das Netzwerk werden, anstatt mehrere Operationen auf entfernten Objekten eines Subsystems, eine oder wenige Operation des entfernten Fassade-Objektes aufgerufen. Dieses delegiert nunmehr die einzelnen Aufgaben an die lokalen Klassen des Subsystems. Das Process-Entity Muster ist die Umsetzung des Fassade Muster speziell für Netzwerke und wird daher besonders in Zusammenhang mit CORBA und EJB verwendet.

Vorteile

Die Verwendung des Subsystems wird erleichtert, ohne dass Funktionalität verloren geht. Dies wird realisiert, da die Schnittstellen der Subsystemklassen nicht verkleinert werden. Außerdem wird die Netzwerkkommunikation verringert, wodurch der Overhead des entfernten Operationsaufrufes verringert wird. Die Kopplung zwischen den einzelnen Subsystemen kann verringert werden, wodurch die Wartbarkeit der Anwendung erhöht wird.

Nachteile

Es entsteht mehr Bytecode. Als Entwickler des Subsystems, welches die Fassade-Klasse enthält haben Sie mehr Arbeit, da Sie auch die Wartung dieser Klasse übernehmen müssen. Während die Netzwerkgeschwindigkeit steigt, sinkt jedoch durch die Delegation der Aufgabe durch die Fassade an die Objekte des Subsystems die Geschwindigkeit der Ausführung nach außen. Lokale Anwendungen, werden somit etwas langsamer als ohne das Fassade Muster. Dies ist jedoch lokal durch den direkten Zugang an die Objekte des Subsystems lösbar.

Null Pattern

Das Null Pattern kann, wie mit die de.comp.lang.iso-c++ Newsgroup bestätigt hat durchaus für einen Performancegewinn sorgen. Unter Java führt die Verwendung jedoch zu keinen Geschwindigkeitsgewinnen, da die JVM immer strikt gegen `null` prüft, bevor diese Methoden auf Objekten aufruft – Sie kennen doch die `NullPointerException`?

Ziel

Das Null Pattern soll vor allem unnötige `if then else` Verzweigungen vermeiden.

Verwendungsgrund

Damit Operationen / Methoden auf Objekten aufgerufen werden können müssen diese zwangsläufig erst einmal erstellt werden. Als Entwickler können Sie jedoch nicht zwangsläufig davon ausgehen, dass auch immer eine Instanz vorliegt. Das typische Vorgehen ist dabei die Prüfung der Referenz auf `null`. Dies führt dabei meist zu if then `else` Blöcken, die den Quelltext schlecht lesbar machen. Um dies zu vermeiden können in vielen Fällen auch Null Objekte angelegt

werden. Diese haben üblicherweise nur leere Methodenrümpfe und werfen keine `Exception`. In einigen Fällen ist es auch möglich dieses Muster mit dem Singleton Muster zu verknüpfen.

Performancegründe

Nicht jede Programmiersprache wird über eine eigene virtuelle Maschine gestartet welche prüft, ob eine Instanz vorliegt auf die die angesprochene Referenz verweist. Hierbei kann es u.U. schneller sein dafür zu sorgen, dass stets eine gültige Referenz vorliegt.[1]

Vorteile

Insbesondere die Lesbarkeit des Quelltextes nimmt zu. Daneben kann aber auch die Ausführungsgeschwindigkeit erhöht werden.

Nachteile

Der Speicherverbrauch erhöht sich durch die vermehrten Objekte. Außerdem müssen Sie beachten, dass im ggf. die Zeit zur Erstellung der Objekte den tatsächlichen Zeitgewinn wettmachen kann. Hier müssen Sie Tests machen.

Process-Entity Muster

Das Process-Entity Muster ist eine spezielle Ausprägung des Fassade Musters[2]. Dabei wird eine Adapterklasse bereitgestellt, die auf dem Server steht und als Ansprechpartner über das Netzwerk zu Verfügung steht. Objekte dieser Klasse delegieren die entfernten Aufrufe ihrer Methoden an die lokalen Objekte und vermindert somit den Netzwerkoverhead.

Singleton Muster

Das Singleton Muster hat den sekundären Zweck die Performance Ihrer Anwendung zu erhöhen.

Ziel

Ein Instanz einer Klasse soll zur Laufzeit nur einmal existieren.

Verwendungsgrund

In bestimmten Fällen ergibt es zur Laufzeit keinen Sinn mehr als eine Instanz einer Klasse zu erzeugen. In diesen Fällen wird das Singleton Muster verwendet um das mehrfache Erzeugen von Instanzen der Klasse zur Laufzeit zu verhindert. Das Singleton Muster ist dabei eine spezielle Art des Fabrikmusters (meist Methoden Fabrik), bei welchem stets das selbe Objekt an den Aufrufer zurückgegeben wird.

---

[1] Leider konnte ich bisher noch nicht nachvollziehen, warum einige C-Compiler das Null Pattern verwenden, um Schleifen besser umordnen zu können. Dies war jedoch eine Antwort, warum das Muster zu Performancegewinn führen kann.

[2] Siehe Fassade Muster

Performancegründe

Das Singletonmuster ermöglicht es unnötige Instanzen einer Klasse zu vermeiden. Es ist jedoch nicht unbedingt, auch wenn es der Name ausdrückt, auf eine Instanz beschränkt, sondern kann beliebig erweitert werden, um lediglich eine bestimmte Anzahl von Instanzen zuzulassen. Dies ermöglicht es Ihnen unnötige Speicherverschwendung zu unterbinden.

Vorteile

Der Speicherverbrauch der Anwendung kann bei der Objekterzeugung von bestimmten Klassen gezielt vermindert werden, um unnötige Objekterzeugung zu vermindert. Das Singleton wirkt außerdem wie ein Ein-Objekt-Cache und vermindert daher die Zugriffszeit auf dieses Objekt.

Nachteile

Das Singleton Muster ist ein Erzeugungsmuster. Sie sollten daher überlegen, ob Sie wirklich diese eine Instanz der Klasse benötigen oder direkt mit Klassenoperationen arbeiten können. Dies würde die Zeit für die Objekterstellung einsparen.

Sichtbarkeiten

Jetzt kommen wir in die Teile, die zwar die Performance erhöhen können, jedoch nicht unbedingt besseres Design darstellen.

Die Sichtbarkeiten von Referenzen, also auch der Beziehungen, und der primitiven Datentypen endgültig festzulegen, ist ebenfalls Ihre Aufgabe als Designer. Die Sichtbarkeit kann Ihre Anwendung beschleunigen, da die Zugriffsgeschwindigkeit u.a. von dieser Abhängig ist. Es gibt jedoch neben der Sichtbarkeit auch noch andere Optimierungsmöglichkeiten. Lokale Variablen sind am schnellsten, so dass der Einsatz des Memory Access Pattern hier zu Geschwindigkeitsvorteilen führen kann. Auch Klassenvariablen sind schneller als Variablen auf Objektebene. Dies führt jedoch schon wieder in die OO Abgründe der Performanceoptimierungen.

Lokale Variablen sind in der JVM am schnellsten, da Sie über den jeweiligen Stack abgearbeitet werden und keine Referenz auf das Objekt nötig ist. Außerdem hat die JVM spezielle Bytecodes für den Zugriff auf die ersten vier lokalen Variablen und Parameter (Slot).

Die Zugriffsgeschwindigkeit auf Variablen verringert sich wie folgt:

`static` ➢ `private` ➢ `final` ➢ `protected` ➢ `public`

Inlining und Sichtbarkeit

Das Inlining ist eine sehr gute Möglichkeit die Geschwindigkeit Ihrer Anwendung zu optimieren. Der Compiler `javac` von Sun kann bereits kleine Operationen und Variablen inlinen. Dazu wird die Compilierung mit dem Parameter `-o` durchgeführt. Damit das Inlining durch den

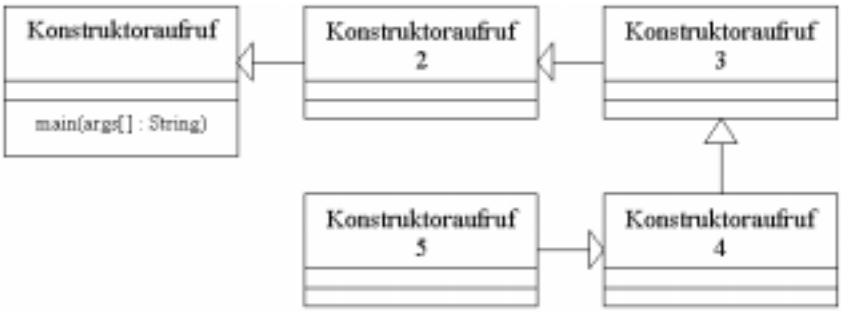
Compiler erfolgen kann, ist jedoch die Sichtbarkeit einzuschränken. Hierbei gilt, dass Variablen, die `private` sind, sowie `private` Methoden grundsätzlich optimiert werden können. Konstanten und innere Operationen (Schlüsselwort `final`) kann der Compiler ebenfalls optimieren.

Vererbungshierarchie

Die Vererbungshierarchie ist ebenfalls ein wichtiger Ansatzpunkt. Als Grundsatz gilt, dass eine flache Klassenhierarchie performanter ist. Natürlich ist dies eine Prüfung des Design und sollte von Ihnen nicht nur aufgrund von Performanceüberlegungen nicht oder unzureichend berücksichtigt werden. Es kann sich jedoch anbieten eine spezialisierte Klasse einer weiteren Vererbung vorzuziehen. Dies bleibt jedoch von Ihnen als Designer im einzelnen zu prüfen.

Insbesondere das Erzeugen von Objekten kann hier zu hohen Geschwindigkeitseinbussen führen. Beim Erzeugen eines Objekts wird üblicherweise der Konstruktor aufgerufen. Der Aufruf eines Konstruktors hat jedoch stets auch den Aufruf eines Konstruktors in jeder Superklasse zur Folge. Dies zieht entsprechende Folgen nach sich. Sinnvoll ist es daher ebenfalls möglichst wenig – besser noch gar keine Funktionalität in den Standardkonstruktor[1] zu legen.

Ein kleines Beispiel hierzu verdeutlicht die Aufrufkette, welche entsteht bei einer kleinen Vererbungshierarchie. Der Aufruf des Standardkonstruktor muss dabei nicht explizit angegeben werden.




```
package bastie.performance.vererbung;

public class Konstruktoraufruf extends
Konstruktoraufruf2{
  public Konstruktoraufruf() {
    System.out.println(Konstruktoraufruf.class);
  }

  public static void main(String [] args){
    Konstruktoraufruf k = new Konstruktoraufruf();
  }
}

class Konstruktoraufruf2 extends Konstruktoraufruf3{
  public Konstruktoraufruf2() {
```

[1] Dies ist ebenfalls beim Wiederherstellen von Objekten mittels der Schnittstelle Externizable zu beachten.

```
    System.out.println(Konstruktoraufruf2.class);
  }
}

class Konstruktoraufruf3 extends Konstruktoraufruf4{
  public Konstruktoraufruf3() {
    System.out.println(Konstruktoraufruf3.class);
  }
}

class Konstruktoraufruf4 extends Konstruktoraufruf5{
  public Konstruktoraufruf4() {
    System.out.println(Konstruktoraufruf4.class);
  }
}

class Konstruktoraufruf5 {
  public Konstruktoraufruf5() {
    System.out.println(Konstruktoraufruf5.class);
  }
}
```

Beim Start der Klasse Kontrukotoraufruf erhalten Sie nun diese Ausgabe:

```
class bastie.performance.vererbung.Konstruktoraufruf5
class bastie.performance.vererbung.Konstruktoraufruf4
class bastie.performance.vererbung.Konstruktoraufruf3
class bastie.performance.vererbung.Konstruktoraufruf2
class bastie.performance.vererbung.Konstruktoraufruf
```

Antidesign

Die Überschrift Antidesign ist vielleicht etwas hart, drückt jedoch gut aus, welche Betrachtungen wir im folgenden Anstellen.

Direkter Variablenzugriff

Der direkte Variablenzugriff ist eine Möglichkeit die letzten Geschwindigkeitsoptimierungen aus Ihrer Anwendung herauszukitzeln. Dabei rufen andere Objekte nicht mehr die Zugriffsoperationen auf, um die Informationen von einem bestimmten Objekt zu erfragen, sondern nehmen sich die Informationen direkt.

Dies widerspricht dem OO Gedanken der Kapselung derart, dass ich hier nicht näher darauf eingehen möchte. Da jedoch der Overhead des Operationsaufrufes hier wegfällt, ist diese Zugriffsart schneller. Sie müssen jedoch alle Referenzen / primitive Datentypen hierfür mit der Sichtbarkeit „`public`“ belegen.

Vermischung von Model und View

Hier gilt ähnliches wie bei dem direkten Variablenzugriff. Sofern Sie Model und View nicht trennen, sondern dies innerhalb eines einzigen Objektes gestalten erhalten Sie eine schnellere Anwendung. Hier entfällt neben dem Methodenaufruf (innerhalb einer Klasse, können Sie ja auch nach dem OO Ansatz direkt auf die Informationen

zugreifen) auf das Erstellen des zweiten Objektes und der Zugriff auf dieses.

Dies ist nicht so weit herbeigezogen, wie Sie vielleicht denken. Die `java.awt.List` hält, wie alle AWT Klassen auch die Daten in sich. Die `javax.swing.JList` hingegen trennt die Repräsentation von dem eigentlichen Modell (`javax.swing.ListModel`).

TEIL 6
Implementierung

Vor- und Nachteile

Implementation bedeutet zwangsweise das Arbeiten mit Typen, Objekten, Methoden, Varablen, etc. Ziel ist es Beispiele und (teilweise) Anleitungen zu geben wie die Performance durch gute Implementierung erhöht werden kann. Es ist nicht (mehr) Ziel wesentliche Designentscheidungen zu treffen oder auf Native Programmierung umzuschwenken. Performante Implementierungstechniken können jedoch Auswirkungen auf das Design haben

Die Optimierung der Performance innerhalb der Implementierung ist der Standardfall. Innerhalb der Implementierung können Sie die größten Performancegewinne erreichen. Sie müssen jedoch darauf achten, das kein Trashcode entsteht. Eine Änderung der Implementation zieht im Gegensatz zur Optimierung von System und Umgebung zudem immer auch ein Testen der Anwendung nach sich.

Sie haben bei der Implementierung gewisse Optimierungsmöglichkeiten, welche unabhängig von der verwendeten Programmiersprache sind bzw. für viele Programmiersprachen zutreffen.

Grundwissen

Es gibt viele Faustregeln, deren Beachtung Ihnen helfen können. Wichtig ist insbesondere der Algorithmus. Die Optimierung / Änderung eines Algorithmus kann Ihnen eine bessere Performance bringen als die Summe aller anderen Optimierungen zusammen. Hierbei ist natürlich zu beachten, wo Sie die Performance erhöhen wollen.

Anhand eines Beispiels werden wir hier ein bisschen Licht ins Dunkel bringen. Wir werden uns dazu der allseits beliebten Primzahlenberechnung widmen.

Zuerst einmal erstellen wir ein Quelltext, welcher die Aufgabe der Ermittlung und der Ausgabe der Primzahlen bis zu 100.000 funktional erledigt. Dies sollte immer der erste Schritt sein, einen Algorithmus zu entwickeln, welcher die Machbarkeit aufzeigt. Das Beispiel selbst wird für die Zeitmessung mit dem JDK 1.3 incl. Client HotSpot ausgeführt.

```
public void simplePrimzahlen(){
    int i = 4;
```

```
    boolean isPrim = true;
    while (i < 100001){
      for (int j = 2; j < i-1; j++){
        if (i%j == 0){
          isPrim = false;
        }
      }
      if (isPrim){
        System.out.println(" "+i);
      }
      else{
        isPrim = true;
      }
      i++;
    }
  }
```

Ziel aller Optimierungen soll es sein, den Algorithmus zu verbessern oder durch einen günstigeren zu ersetzen. Am günstigsten ist es stets den Algorithmus durch einen besseren zu ersetzen. Da wir jedoch hier das Grundwissen über eine performante Implementierung erweitern wollen, werden wir diesen Schritt später betrachten. Mit unserem Beispiel haben wir die volle Funktionalität abgedeckt. Wir brechen mit etwa 95.000 Millisekunden jedoch keine Geschwindigkeitsrekorde.

Redundanz vermeiden

Ein wichtiger Punkt der OO ist die Vermeidung von Redundanz. D.h. Variablen oder Methoden, welche die gleiche Aufgabe bzw. Inhalt haben sollen auch nur einmal im Modell vorkommen. Aus Performancesicht bedeutet dies jedoch auch, Ausdrücke, mit gleichem Ergebnis bzw. Methoden sollten nur so oft verwendet werden, wie diese auch wirklich benötigt werden. ziel ist es dabei die bekannte Werte nochmals zu berechnen bzw. Methoden unnötigerweise aufzurufen[1].

Auf den ersten Blick scheint unsere Anwendung jedoch keine Redundanz zu beinhalten. Ausdrücke wie

```
//vorher
...
z[0] = a * b + c * d;
z[1] = a * b + e * f;
...
```

welche wir zu

```
//nachher
...
double ab = a * b;
z[0] = ab + c * d;
```

[1] Wir wollen also den Overhead, der durch den Aufruf von Methoden entsteht verringern.

```
z[1] = ab + e * f;
...
```

optimieren könnten sind zuerst nicht erkenntlich. Jedoch erkennen wir eine Methode „`System.out.println(" "+i);`“, welche wir jedes Mal erneut aufrufen. Wir wollen uns zuerst dieser Methode zuwenden. Unter der Vorraussetzung, dass wir über genügend Arbeitsspeicher verfügen, bietet es sich an, unser Ergebnis nicht sofort auszugeben, sondern zu sammeln. Nach Beendigung unseres Algorithmus soll dann die Ausgabe erfolgen. Unser geänderter Quelltext sieht nun wie folgt aus:

```
public void simplePrimzahlen1(){
    int i = 4;
    boolean isPrim = true;
    String ergebnis = "";
    while (i < 100001){
      for (int j = 2; j < i-1; j++){
        if (i%j == 0){
          isPrim = false;
        }
      }
      if (isPrim){
        ergebnis = ergebnis+" "+i+"\n\r";
      }
      else{
        isPrim = true;
      }
      i++;
    }
    System.out.println(ergebnis);
  }
```

Neben dem Speicherverbrauch zu Laufzeit hat sich auch der Bytecode minimal vergrößert. Unsere Berechnung benötigt nunmehr jedoch nur noch etwa 77.000 ms. Dies ist bereits eine erhebliche Einsparung von etwa 20%. In unserer Anwendung ist jedoch noch mehr Redundanz versteckt. Hier gilt es die Schleifen zu optimieren.

Schleifen

Schleifen kann man auch als gute Bremsen bezeichnen. Ein Performanceleck in einer Schleife hat meist katastrophale Auswirkungen, da es bei jedem Schleifendurchlauf Ihre Anwendung bremst. Insbesondere innerhalb von Schleifen gilt daher möglichst alle Performancemöglichkeiten zu finden. Wir haben in unserer Primzahlenanwendung durch die Schleife noch eine weitere Redundanz. Innerhalb der `for`-Schleife berechnen wir für unseren Vergleich von `j` bei jedem Schleifendurchlauf `i-1`. Diese Berechnung wird allein bis i den Wert 103 erreicht, also 100 Durchläufe bereits 5050 mal durchgeführt. Es würden jedoch 100 Berechnungen ausreichen. Wenn i den Wert von 203 erreicht hat, hat unsere Anwendung bereits über 20000 statt 200 Berechnungen durchgeführt. Letztendlich führt dies zu etwa 5 Milliarden unnötigen Berechnungen. Dies bremst natürlich nicht nur eine Javaanwendung aus. Nach einer weiteren Optimierung sieht unser Quelltext nun etwa so aus:

```
public void simplePrimzahlen2(){
    int i = 4;
    boolean isPrim = true;
    int vergleich;
    String ergebnis = "";
    while (i < 100001){
      vergleich = i-1;
      for (int j = 2; j < vergleich; j++){
        if (i%j == 0){
          isPrim = false;
        }
      }
      if (isPrim){
        ergebnis = ergebnis+" "+i+"\n\r";
      }
      else{
        isPrim = true;
      }
      i++;
    }
    System.out.println(ergebnis);
  }
```

Wir haben die benötigte Zeit weiter verringern können. Beim Hot-Spot-Compiler macht sich diese Optimierungsmethode jedoch im Gegensatz zum Bytecode Interpreter nur gering bemerkbar – wir benötigen jetzt ca. 74.000 ms.

Je mehr Iterationen eine Schleife durchführt, desto höher ist der Overhead. Daher ist es sinnvoll Ihre Schleifen auszurollen, sofern die Anzahl der Iterationen bekannt ist. Dies führt aus OO Sicht natürlich zu dem unschönen Effekt, dass Sie Quelltext mit der gleichen Funktionalität mehrfach in Ihrer Klasse haben. Dies ist nicht sehr Wartungsfreundlich.

Weitere Optimierungsmöglichkeiten liegen bei den Zählvariablen versteckt. Sofern Sie Ihre Anwendung auch auf Bytecode Interpretern ausführen müssen, ist es günstiger Zählschleifen abwärts zu zählen. Das Abwärtszählen ist schneller, da für den Vergleich auf 0 ein spezieller Bytecode in der JVM vorhanden ist und der Vergleichswert nicht erst vom Methodenstack geholt werden muss. Ein JIT Compiler bzw. Hot Spot Compiler generiert Maschinencode, so dass ein Vergleich auf 0 hier keine Geschwindigkeitsgewinne bringt. Das verwenden von Slotvariablen als Zählvariable kann noch ein paar Prozente Geschwindigkeitsgewinn bringen. Innerhalb eine Methode werden die ersten 128 Bit durch Slotvariablen besetzt. Sofern es sich um eine Objektmethode handelt belegt jedoch die Referenz auf das Objekt bereits die ersten 32 Bit. Der verbleibende Speicher wird in der Reihenfolge der Deklaration Ihrer Variablen belegt. Auf diese 128 Bit hat die JVM, dank spezieller Operationen, einen besonders schnellen Zugriff. Es ist daher günstig auch die Zählvariable j frühzeitig zu deklarieren. Ein besonders wichtiger Punkt ist auch der Typ der Zählvariablen. Verwenden Sie für Schleifen nach Möglichkeit immer den primitiven Typ `int` als Methodenvariable für das Zählen innerhalb einer Schleife. Hierdurch erreichen Sie die beste Ausführungsgeschwindigkeit.

Bei JIT Compilern kann auch der freie Test zu einem Performancegewinn führen. Dabei verbindet der JIT Compiler den ermittelten Wert gleich mit dem Bindungsflag. Dadurch kann der direkte Vergleich unterbleiben. In manchen Fällen lässt man die Schleife auch besser auf einen Fehler laufen und wertet das Ergebnis mittels `try-catch-finally` Block aus. Gute Beispiele dafür sind große Arrays und Collections zu durchlaufen oder größere Dateien einzulesen. Hier wird absichtlich auf einen Fehler, wie `EOFException` gewartet. Beachten Sie dabei jedoch, dass bei kleineren Schleifen durch das Erzeugen der `Exception` die Ausführungsgeschwindigkeit verringert wird.

Zählschleifen

Die Zählschleife `for`[1] ist außerdem noch bzgl. des Gesichtspunktes Inkrementierung oder Dekrementierung zu Untersuchen. Grundsätzlich ist der Schleifentest auf 0 besonders schnell auch wenn bei den heutigen JVM s mit JIT und Hot Spot Compilern dies weniger ins Gewicht fällt. Daher sollten Zählschleifen abwärts ausgerichtet sein.

Algorithmus

Nachdem wir uns nun lange genug um Schleifenspezifische Optimierungen gekümmert haben, wollen wir uns jetzt dem Algorithmus vornehmen. Dies sollte stets Ihr erster Ansatzpunkt sein. Da die anderen Optimierungsmöglichkeiten jedoch in diesem Fall nicht mehr sinnvoll gewesen wären, haben wir den Punkt zurückgestellt.

Unser Algorithmus bzgl. der Primzahlenberechnung eine Katastrophe! Wir prüfen für jede Ganzzahl, ob es sich um eine Primzahl handelt, obwohl uns bekannt ist, dass nur ungerade Zahlen Primzahlen sein können. Auch das Prüfen per Modulo ist nicht optimiert. Anstatt die Prüfung nur mit Primzahlen kleiner als die Wurzel unserer zu prüfenden Zahl zu arbeiten, prüfen wir für alle Ganzzahlen kleiner unserer zu prüfenden Zahl. Dazu kommt, dass wir die Prüfung selbst dann fortführen, wenn wir bereits festgestellt haben, das es sich um keine Primzahl handelt. Hinsichtlich dieser Punkte werden wir nochmals eine letzte Optimierung unseres Quelltextes vornehmen.

```
  private IntHashtable ih;
  private int gaussPrimesLength;

public void gaussPrimzahlen(){
  ih = new IntHashtable (0);
  int pos = 0;
  int i = 2;
  StringBuffer buf = new StringBuffer();
  ih.put(pos,2);
  while (i<100001){
    i++;
    gaussPrimesLength = ih.size();
    if (this.isPrimzahlForGauss(i)){
      pos++;
      ih.put(pos,i);
      buf.append(i).append("\n\r");
```

[1] Was Sie nicht einschränkt auch do oder while als Zählschleife zu gebrauchen.

```
    }
  }
  System.out.println(buf.toString());
}

private final boolean isPrimzahlForGauss(int candidate){
  for (int i = 0; i<this.gaussPrimesLength;i++){
    if (candidate % this.ih.get(i) == 0){
      return false;
    }
  }
  return true;
}
```

Unsere jetzige Primzahlenberechnung[1] benötigt nur noch etwa 2.500 Millisekunden (entspricht 2,6% der Zeit des ersten Algorithmussees) und kann noch weiter optimiert werden. Für unser Beispiel soll uns diese Zeitspanne jedoch ausreichen. Mit einem guten Mathematikbuch können Sie auch andere Algorithmen ausprobieren. Es wird schnell deutlich dass die Optimierung des Algorithmus der entscheidende Punkt ist.

Konstante Ausdrücke

In diesem Abschnitt sind Variablen gemeint, welche ein konstanter Wert zugewiesen wird. Hier sind Optimierungen möglich, in dem Sie prüfen, ob unnötige Berechnungen mit diesen Variablen durchgeführt werden. Ziel sollte es sein konstante Ausdrücke ohne Berechnung zu definieren.

So sollte aus

```
//vorher

...
x = 10;
y = 4 *2398;
... // mehr, ohne dass x sich ändert
x = x + 15;
...
```

besser

```
//nachher

...
X = 10:
y = 4 *2398;
... // mehr, ohne dass x sich ändert
x = 25;
...
```

werden. Hier wird eine unnötige Operation (Addition) vermieden.

[1] Falls Sie diesen Quelltext testen wollen, müssen Sie sich in Ihr Package noch eine Kopie die Klasse `java.text.IntHashtable` anlegen.

Unerreichbaren Quelltext vermeiden

Unerreichbarer Quelltext[1] sollte vermieden bzw. entfernt[2] werden. Damit erhöhen Sie auch die Lesbarkeit Ihres Quelltextes. Der Javacompiler `javac` bietet mit dem Parameter `-o` eine gute Möglichkeit unerreichbaren Quelltext bei der Erstellung des Bytecode zu ignorieren.

```
// if-Schleife vorher:

boolean wahr = true;
if (wahr){
   x = 1;
}
else{
   x = 2;
}

// if-Schleife nachher:
```

Und eine `while(false)` Schleife fällt ebenfalls weg

```
// while-Schleife vorher:

while (false){

   x++;

}

// while-Schleife nachher ;-):
```

Quelltextverschiebung

Quelltextverschiebung ist ebenfalls ein gute Möglichkeit die Geschwindigkeit Ihrer Anwendung zu erhöhen. Insbesondere innerhalb von Schleifen kann sich diese Überlegung bemerkbar machen. Sofern Sie Quelltext aus der Schleife herausziehen können, muss dieser Teil nicht mehr für jede Iteration durchgeführt werden. Bedenken Sie jedoch, dass Sie hier die Semantik Ihrer Implementierung verändern. Gerade für das Auslösen bzw. werfen von Fehlermeldungen (`Exceptions`) kann sich die Quelltextverschiebung sowohl negativ, als auch Positiv bemerkbar machen. Das folgende Beispiel setzt den Dreisatz um. Dabei wird zuerst die Division durchgeführt, um die Division durch 0 möglichst früh abzufangen.

```
public double getDreisatz (int oben1, int unten1, int
oben2){
   double ergebnis = unten1 / oben1;
   ergebnis *= oben2;
   return ergebnis;
  }
```

[1] engl. dead code

[2] Für Testzwecke (Debugging) können Sie jedoch, derartige Konstrukte nutzen.

Optimierung für bestimmte Anwendungsbereiche

Eine hauptsächlich aus der Spielprogrammierung bekannte Möglichkeit basiert auf dem Prinzip der „Tiles“. Das Konzept soll es ermöglichen mit wenigen Grafiken komplexe Spielebenen[1] zu gestallten ohne gewaltige Grafikressourcen zu benötigen. Die Spielebenen werden dabei aus kleinen Grafiken, den Tiles, aufgebaut, welche alle in einer Datei gespeichert sind.

Abbildung 2 – Tilesbeispiel von http://www.kosmokleaner.de

Die obige Abbildung zeigt ein Beispiel einer solchen Grafikdatei, wie Sie in einem Spiel vorkommt. Grundsätzlich haben dabei die Grafiktiles die gleiche Größe, was jedoch nicht zwingend nötig ist. Ein Tile ist lediglich eine kleine Grafik mit fester Größe und entspricht damit also schon fast der Definition des Interface Icon der Swing-Bibliothek. Dieses eigentlich aus der Spielentwicklung bekannte Prinzip lässt sich jedoch auch in anderem Zusammenhang verwenden, so dass wir später in Zusammenhang mit Swing wieder auf diesen Punkt zurückkommen werden.

[1] meist wird der Begriff „Level“ verwendet.

Programmiersprachenunabhängige Optimierungen sind ein guter Ansatz. Sie erreichen somit eine schnelle Anwendung ohne von Implementation der JVM bzw. des Betriebssystems abhängig zu sein. Auch bei der Umsetzung in eine andere Programmiersprache haben Sie bereits einen performanten Ansatz.

Erster Schritt der Optimierung sollte stets das Überdenken des Algorithmussees sein. Danach sollten Sie Ihre Schleifen in Verbindung mit der Prüfung auf Redundanz und Quelltextverschiebung prüfen.

Erst nach dieser Optimierung sollten Sie in die programmiersprachenabhängige Optimierung einsteigen.

TEIL 7
Implementierung mit Java

Wir werden uns im folgenden den Javaspezifischen Optimierungen zuwenden. Wie in den anderen Abschnitten, liegt auch hier der Schwerpunkt auf die Beschleunigung der Ausführungsgeschwindigkeit. An geeigneter Stelle werden jedoch immer wieder Einwürfe zu den anderen Performanceeigenschaften einer Lösung fallen. Ein für Bytecode Compiler optimaler Code, kann mit JIT Compilern langsamer als der ursprüngliche Code sein. Einige Beispiele werden wir daher mit und ohne JIT testen, um dieses Problem aufzuzeigen.

Eigener Profiler

Bevor Sie Ihre Javaanwendung optimieren können müssen Sie erst einmal wissen, wo das Performanceleck liegt. Der Einsatz von Profilertools ist eine gute Möglichkeit. Eine andere Möglichkeit ist die Ausführungsgeschwindigkeit selbst zu messen.

Die können Sie gut umsetzen, indem Sie zwischen den Anweisungen die Zeit ausgeben. Mit der Operation `System.currentTimeMillis()` können Sie dies schnell durchführen. Hierbei machen Sie Ihr eigenes Profiling[1]. Der bedeutende Vorteil ist, dass keine weiteren Anwendungen (Profilertools) den Prozessor in Beschlag nehmen. Somit sind reale Zeitmessungen auf dem System möglich. Sie können jedoch nicht messen, wo innerhalb einer Fremdmethode eventuell Performanceprobleme liegen. Auch die Aufrufreihenfolge von Methoden können Sie nicht nachvollziehen.

Da diese beiden Punkte jedoch eine nicht zu unterschätzende Relevanz haben kann, ist der Einsatz eines Profilertools durchaus sinnvoll. Auch hier lässt uns Sun Microsystem nicht im Stich. Die JVM beinhaltet bereits einen Profiler, welchen Sie nur zu aktivieren brauchen.

JVM Version	Aufruf / Parameter
JVM 1.1.x final	java –prof
JVM 1.2.x final	java –Xrunhprof
JVM 1.3.x final	java –Xrunhprof

Tabelle 3 - Aufruf des Profiler der JVM

Die Standardausgabedatei heißt „`java.prof`“. Diese ist jedoch sehr kryptisch und in reiner Textform. Kommerzielle Profiler stellen die Ausgabe meist grafisch dar und erleichtern somit die Analyse. Glücklicherweise haben sich bereits einige Entwickler an die Arbeit

[1] Für Zeitmessungen verwende ich dies etwas abgewandelt, da es schnell und einfach umzusetzen ist. Zudem benötigt man keine anderen Tools.

gemacht und Tools umgesetzt, die unsere Profiledatei auch ohne Kryptografen lesbar machen. Sie finden diese Tools „`HyperProf`“ und „`ProfileViewer`“ im Internet.

Java Virtuelle Machine

Das Wissen über die Besonderheiten der JVM ist Grundlage für viele Optimierungsmöglichkeiten. Daher sollten Sie als Programmierer diesen Abschnitt genauer betrachten.

Zugriffsgeschwindigkeit auf Typen

Die JVM ist ein 32 Bit System. Aus diesem Grund sind Zugriffe auf Variablen mit einer Breite von 32 Bit am schnellsten. Somit ist der Zugriff auf Referenzen und `int` Variablen besonders schnell. `float` Variablen, sind Fliesskommazahlen und können nicht ganz so schnell verarbeitet werden. Die primitiven Datentypen unterscheidet man auch in Datentypen erster Klasse und Datentypen zweiter Klasse. Datentypen erster Klasse sind `int`, `long`, `float` und `double`. Für diese Datentypen liegt ein kompletter Satz an arithmetischen Funktionen vor, so dass Berechnung relativ zügig vonstatten gehen. Die Datentypen zweiter Klasse `boolean`, `byte`, `short` und `char` hingegen haben keine arithmetische Funktionen. Für Berechnungen werden diese innerhalb der JVM in den `int` Typ gecastet und nach Abschluss zurück umgewandelt. Dies führt zu erheblichen Zeitverzögerungen. (siehe auch Variablen / Datentypen und Operatoren).

Rechengeschwindigkeit für Gleitpunktdatentypen

`float` und `double` sind gemäß der Java Spezifikation Datentypen mit einer Genauigkeit von 32 Bit bzw. 64 Bit. In der IEEE 754 werden jedoch auch Gleitpunktdatentypen mit Genauigkeiten von mindestens 43 Bit bzw. 79 Bit vorgesehen. Seit dem JDK 1.2 können nunmehr Java Virtual Machines diese Genauigkeiten zur Beschleunigung der Rechenoperationen mit den beiden Datentypen nutzen. Sofern die Hardware höhere Genauigkeiten unterstützt unterbleibt das Umrechnen auf die Genauigkeit der Datentypen während der Rechenoperationen. Dies führt zu einer Geschwindigkeitserhöhung bei entsprechenden Berechnungen.

Die interne Darstellung der Datentypen verbleibt jedoch bei den 32 Bit für `float` bzw. 64 Bit für `double`. Die Änderung der Java Spezifikation hat außerdem dazu geführt, das in Abhängigkeit von der verwendeten Hardware und der Java Virtual Machine unterschiedliche Ergebnisse auftreten können. Um den Entwicklern eine Möglichkeit zu geben dies zu unterbinden wurde das Schlüsselwort `strictfp` eingeführt. Dieses kann auf Klassen und Methodendeklarationen angewendet werden und zwingt die JVM die alten Berechnungsverfahren anzuwenden – mit dem entsprechenden Geschwindigkeitsverlust.

Stackorientierter Prozessor der JVM

Der Prozessore der JVM ist stackorientiert und hat eine Breite von 32 Bit. Parameter und Rückgabewert von Methoden werden jeweils über dieses Stack gereicht. Lokale Variablen einer Methode können Sie

dabei mit den Registern der CPU vergleichen. Dies hat unter anderem zur Folge, dass die Erzeugung eines neuen Threads stets auch zur Erzeugung eines eigenen Stacks für diesen führt (Java Stack)[1].

Sobald Sie eine Methode aufrufen, wird für ein neuer Datensatz reserviert, in welchem der aktuelle Zustand gesichert wird (Methodenoverhead). Methoden können dabei Ihre eigene lokalen Variablen haben. Für den Zugriff auf die ersten 4 Slots (128 Bit) des Stack haben Ihre Methoden einen speziellen Bytecode. Daher kann durch gezielte Belegung der Slots die Geschwindigkeit innerhalb einer JVM (Bytecode Interpreter) erhöht werden. Bei den moderneren JIT und HotSpot Compilern führt dies jedoch nicht zu Gewinnen. Slots werden nach folgendem Schema belegt. Wenn es eine Objektmethode ist, wird der erste Slot mit einer Referenz auf das Objekt belegt (32 Bit). Werden Parameter übergeben so belegen diese die folgenden Slots. Danach belegen die lokalen Variablen in der Reihe Ihres Auftretens den Slot. Durch eine gezielte Deklaration der lokalen Variablen können Sie also weitere Performancegewinne erzielen. Dies bietet sich z.B. bei den Zählvariablen größerer Schleifen an. Beachten Sie: Ein Array wird immer mit der Referenz (32 Bit) übergeben, wobei sich nur die Zugriffsgeschwindigkeit auf das Array, nicht dessen Inhalt, erhöht. Die Typen double und long benötigen 2 Slots.

Eine weitere Steigerung ist in Verbindung mit dem Memory Access Pattern möglich.

JIT contra Interpreter

JIT und Hot Spot Compiler haben andere Ansprüche als ein Bytecode Interpreter an Sie, wenn es um die Optimierung der Performance geht. Für Bytecode Interpreter kann z.B. das gezielte Belegen der Slotvariablen Performancegewinne bringen. Auch der Zugriff auf die Variablen ist bei Bytecode Interpreter anders zu bewerten.

Variablen / Datentypen und Operatoren

Operatoren

Die richtige Verwendung der Operatoren von Java kann Ihre Anwendung beschleunigen. Wichtig ist jedoch, dass Sie die Semantik der verschiedenen Operatoren kennen. So kann das Shiften[2] in einigen Fällen die Division bzw. Multiplikation ersetzten und beschleunigen – jedoch nicht immer. Einige Grundsätze können Sie jedoch beachten.

Die Verwendung von verbundenen Zuweisungen kann durch den Compiler effizienter umgesetzt werden. Somit sollten Sie besser `x++;` als `x+=1;` als `x=x+1;` schreiben. Einige neuere Compiler erkennen jedoch bereits derartigen Quelltext und können diesen optimieren.

---

[1] In Abhängigkeit von der JVM werden auch noch native Threads bzw. Prozesse erzeugt.

2 Ein Shift schiebt die Bitfolge. Dies kann zum Beispiel zu Problemen bei negativen Zahlen führen. Shiften ist jedoch schneller. Aus x*2 wird somit x<<1 und aus x/4 wird x>>2.

Sie sollten nie die Typinformationen verlieren. Die Operationen `instanceof` und das Casten mittels `(type)` sind nicht sehr performant und überflüssig, wenn Sie den Typ des Objektes kennen[1].

Die Operatoren:
„++“, „--“, „+“, „-“, „~“, „!“, „(type)“, „“, „/“, „%“, „<<“, „>>“, „>>>“, „<“, „<=“, „>“, „>=“, „instanceof“, „==“, „!=“, „&“, „^“, „|“, „&&“, „||“, „?:“, „=“, „*=“, „/=“, „%=“, „+=“, „-=“, „<<=“, „>>=“, „>>>=“, „&=“, „^=“, „|=“, „new“*

Variablen bzw. Attribute

Die Initalisierung von Variablen ist in Java nicht stets notwendig. Klassen und Instanzvariablen werden automatisch initalisiert. Referenzen bekommen hierbei den Wert `null`; primitive Datentypen bekommen den Wert `0`. Auch die Inhalte eines Array werden automatisch initalisiert, sofern das Array bereits eine gültige Referenz ungleich `null` aufweist. In Abhängigkeit vom Java Compiler kann unnötiger Bytecode vermieden werden, indem Sie auf die Initalisierung mit den Standardwerten verzichten.

Falls Sie Optimierung um jeden Preis möchten, können Sie durch den direkten Attributzugriff weitere Geschwindigkeitsvorteile erreichen. Dies ist natürlich nur bedingt möglich. Bei Javabeans können Sie auf die Zugriffsmethoden nicht verzichten. Zudem ist Ihre Anwendung nicht mehr Objektorientiert und die Wartung einer solchen Anwendung auf Dauer vermutlich zum Scheitern verurteilt. Zum Testen bietet sich hier eine Schleife an. Die Geschwindigkeit des Attributzugriffs kann dabei etwa auf das Doppelte erhöht werden.

Auch die Sichtbarkeit von Variablen sollten Sie für einen effizienten Zugriff beachten.

Sichtbarkeit

Die Geschwindigkeitsgewinne / -verluste je nach Sichtbarkeit unterscheiden sich sehr je nach eingesetzter JVM. Daher können nur sehr wenige allgemeingültige Aussagen gemacht werden. Ein Quelltext zum Testen der Zugriffsgeschwindigkeiten finden Sie im Anhang. Wissen sollten Sie jedoch, das Methodenvariablen stets sehr schnell behandelt werden können. Aus diesem Grund existiert das Memory Access Pattern.

Memory Access Pattern

Das Memory Access Pattern wird häufig eingesetzt um, die Zugriffsgeschwindigkeit auf Variablen zu erhöhen. Es wird dabei nur innerhalb einer Methode verwendet. Ziel ist es den Wert einer Variablen innerhalb einer Methode schnell zu verändern und diesen nach Abschluss nach außen bereitzustellen. Besonders bei sehr zeitaufwendigen Operationen mit langsameren Variablentypen kann dies Geschwindigkeitsvorteile in Ihrer Anwendung bringen. Beachten

1 Leider bedeutet Abstraktion häufig, dass Sie mit diesen Operatoren arbeiten müssen. Ein gutes Beispiel ist der Vector. Sie können in diesen alles sichern. Der Zugriff auf Objekte des Vectors erfordert jedoch meist ein casten.

Sie jedoch, dass gerade bei Multithreading der Zugriff ggf. synchronisiert werden muss.

```
package bastie.performance.zugriffaufvariablen;

public class MemoryAccessPattern {
  private IntegerWrapper iw = new IntegerWrapper();
  private final int durchlauf = 10000000;

  public MemoryAccessPattern() {
  }

  public final void normalZugriff(){
    System.out.println("Normaler Zugriff");
    long start;
    start = System.currentTimeMillis();
      for (int i = 0; i < durchlauf; i++){
        iw.setValue(iw.getValue()+1);
      }
    System.out.println("Zeit =
"+(System.currentTimeMillis()-start)+"ms\n\r\n\rvalue =
"+this.iw.getValue());
  }

  public final void mapZugriff(){
    System.out.println("MemoryAccessPattern Zugriff");
    long start;
    start = System.currentTimeMillis();
    int j = iw.getValue();
    for (int i = 0; i < durchlauf; i++){
      j++;
    }
    this.iw.setValue(j);
    System.out.println("Zeit =
"+(System.currentTimeMillis()-start)+"ms\n\r\n\rvalue =
"+this.iw.getValue());
  }

  public final void syncmapZugriff(){
    System.out.println("Synchronisierter
MemoryAccessPattern Zugriff");
    long start;
    start = System.currentTimeMillis();
    synchronized (iw){
      int j = iw.getValue();
      for (int i = 0; i < durchlauf; i++){
        j++;
      }
      iw.setValue(j);
    }
    System.out.println("Zeit =
"+(System.currentTimeMillis()-start)+"ms\n\r\n\rvalue =
"+this.iw.getValue());
  }

  public static void main (String[] args){
    MemoryAccessPattern map = new MemoryAccessPattern();
    map.normalZugriff();
    map.mapZugriff();
    map.syncmapZugriff();
  }
}

class IntegerWrapper {
  private int value;
  public int getValue() {
    return value;
  }
```

```
  public void setValue(int newValue) {
    value = newValue;
  }
}
```

Das Verhältnis von Normalen – zu Memory Access Pattern – zu synchronisierten Memory Access Pattern Zugriff beträgt je nach System etwa 920 : 90 : 150 (JDK 1.3 Client HotSpot) bzw. 5650 : 1800 : 1830 (JDK 1.3 Interpreter Modus). Eine weitere sinnvolle Anwendung besteht darin primitive Datentypen in `int` zu casten, in der Methode mit diesem schnellen Datentyp zu arbeiten und dann erst das Rückcasten vorzunehmen.

Zeichenketten

Für die Verwaltung von Zeichenketten bieten sich grds. drei Möglichkeiten an: die Klasse `String`, die Klasse `Stringbuffer` und ein `char` Array. Die Klasse String ist final und besitzt nur Lesemethoden. Daher ist für die Arbeit mit veränderlichen Zeichenketten grundsätzlich die Klasse Stringbuffer zu bevorzugen. Diese verwaltet intern ein char Array, auf welchem sie die Lese- und Schreibmethoden ausführt. Die schnellste Möglichkeit ist schließlich das direkte arbeiten mit einem char Array ohne einen `Stringbuffer` als Wrapper zu verwenden. Die Arbeit mit der Klasse `Stringbuffer` ist jedoch wesentlich komfortabler als direkt auf ein char Array zuzugreifen.

Sie werden sicher schon gehört haben, dass für das Verketten von Zeichenketten die Nutzung der Klasse `Stringbuffer` schnelleren Bytecode erzeugt. Dies ist grundsätzlich richtig, sofern Sie mehr als zwei Zeichenketten verketten wollen. Die Klasse `Stringbuffer` können Sie dabei jedoch weiter optimieren, indem Sie die synchronisierten Methoden ersetzen.

Primitive Datentypen

Die JVM ist ein 32 Bit System. Aus diesem Grund sind Zugriffe auf Variablen mit einer Breite von 32 Bit am schnellsten. Somit ist der Zugriff auf Referenzen und `int` Variablen besonders schnell. `float` Variablen, sind Fliesskommazahlen und können nicht ganz so schnell verarbeitet werden. Die primitiven Datentypen unterscheidet man auch in Datentypen erster Klasse und Datentypen zweiter Klasse. Datentypen erster Klasse sind `int`, `long`, `float` und `double`. Für diese Datentypen liegt ein kompletter Satz an arithmetischen Funktionen vor, so dass Berechnungen relativ zügig vonstatten gehen. Die Datentypen zweiter Klasse `boolean`, `byte`, `short` und `char` hingegen haben keine arithmetische Funktionen. Für Berechnungen werden diese innerhalb der JVM in den `int` Typ gecastet und nach Abschluss zurück umgewandelt. Dies führt zu erheblichen Zeitverzögerungen.

Zahlensysteme

Die Umwandlung von einem Zahlensystem in ein anderes wird mit Hilfe der Klasse `Integer`in Java bereits unterstützt. Hierbei stellt die Klasse zum einem die Möglichkeit mit Hilfe der Methode `toString(wert, basis)`, sowie spezialisierte Methoden wie `toHexString(wert)` zur Verfügung. Die spezialisierten Methoden

bieten Ihnen dabei einen Geschwindigkeitsvorteil, da hier das schnellere Shiften bei einer Basis verwendet wird, welche selbst auf der Basis 2 berechnet werden kann. Somit benötigt `toHexString` (JDK 1.3) etwa 889 Zeiteinheiten zu 1.362 Zeiteinheiten bei `toString` zur Basis 16. Die Umwandlung eines primitiven Datentyp ist grundsätzlich stets mit den spezialisierten Methoden der Wrapperklassen vorzunehmen. So ist die Umwandlung einer Ganzzahl vom Typ `int` mit der statischen Methode `Integer.toString()` etwa doppelt so schnell wie ‚`""+int`'.

Sammlungen – Container für Datentypen

Sammlungen sind einige der Klassen / Objekte, welche mit am häufigsten Verwendung finden.

Sammlungen sollten Sie nach Möglichkeit immer auf Objekt- bzw. Klassenebene deklarieren und definieren. Wenn Sie eine Sammlung lediglich innerhalb einer Methode deklarieren / definieren, so wird bei jedem Methodenaufruf eine neue Initialisierung dieser vorgenommen.

Arrays

Ein Array hat gegenüber den Sammlungen des Collection Frameworks bestimmte Vor- und Nachteile. Arrays behalten die Typinformationen. Dadurch entfällt in Ihrer Anwendung das unperformante casten der Typen. Dies ist jedoch gleichzeitig eine Einschränkung. Ihr Array kann nur Objekte eines Typs aufnehmen. Außerdem muss die maximale Anzahl der Elemente Ihres Array bekannt sein.

Ob Sie ein Array verwenden können, um die Performance Ihrer Anwendung zu erhöhen hängt daher von verschiedenen Voraussetzungen ab. Wenn Sie die voraussichtliche größtmögliche Anzahl der Elemente kennen oder die Anzahl der Elemente keinen großen Schwankungen unterliegt ist ein Array performanter als eine andere Sammlung. Benötigen Sie eine Sammlung für Objekte verschiedener Typen, müssen Sie prüfen, ob es einen Typ (Superklasse, Interface) gibt, welcher allen Objekten gemeinsam ist. In diesen Fällen ist ein Array meist die performantere Alternative zu anderen Sammlungen. Ziel sollte es stets sein, dass alle (häufig) benötigten Informationen ohne einen Cast erreichbar bleiben.

Gerade wenn Sie häufig mit Arrays arbeiten sollten Sie sich genügend Zeit für die Optimierung dieser nehmen. Die Initialisierung eines Array kostet nicht nur Zeit, sondern vergrößert auch den Bytecode Ihrer Anwendung. Dies führt neben dem erhöhten Speicherplatzbedarf zur Laufzeit ggf. auch zu einer vermehrten Netzwerkbelastungen.

Eine Möglichkeit, wie Sie diesem Problem zu Leibe rücken können, ist das optionale Nachladen der benötigten Daten. Hier kann der Bytecode erheblich verringert werden. Auch die Netzwerklast sinkt, da Sie nur wirklich benötigte Daten über das Netz versenden müssen. Bei der Erstellung von Applets bietet sich diese Vorgehensweise besonders an, da die Bandbreite meist eingeschränkt und nicht vorhersehbar ist. Beachten Sie dabei, dass diese Daten nicht in der selben jar Datei liegen, wie Ihre Anwendung.

Ein anderer Ansatzpunkt, der als wesentliches Ziel nicht die Verkleinerung des Bytecode, sondern die Erhöhung der Ausführungsgeschwindigkeit zum Ziel hat, liegt in der Einschränkung der Sichtbarkeit. Wie jedes Objekt ist es stets sinnvoll ein Array als private final vorliegen zu haben. Durch die Optimierungen, die der Compiler somit an Ihrer Anwendung vornehmen kann, ist es möglich eine höhere Ausführungsgeschwindigkeit zu erreichen.

Arrays kopieren

Eine häufige Anwendung ist auch das Kopieren von Arrays. Dafür gibt es grundsätzlich drei verschiedene Möglichkeiten. Das Kopieren innerhalb einer Schleife. Das Klonen eines Arrays und das Verwenden der Methode `System.arraycopy()`. Die letzte Möglichkeit hat den Vorteil, dass es sich um eine nativ implementierte Methode handelt, welche für diese Aufgabe entsprechend optimiert ist. Sie sollten grds. diese Variante bevorzugen, da Sie sowohl bei Bytecode Interpretern als auch bei JIT- und HotSpot Compilern die schnellsten Ergebnisse liefert. Bei JIT- bzw. HotSpot Compilern ist der Unterschied zum Kopieren innerhalb einer Schleife jedoch nicht mehr so hoch.

```
// Array kopieren
System.arraycopy(array,0,klon,0,10);

// Array klonen
int [] klon = (int[]) array.clone();
```

Arrays durchlaufen

Das Durchlaufen von Arrays ist ebenfalls eine häufig auftretende Verwendung. Gerade bei großen Datenmengen und somit großen Arrays macht sich jedoch der Schleifenoverhead bemerkbar. Der Schleifenoverhead entsteht dabei u.a. durch die Gültigkeitsprüfungen (siehe auch Schleifen). Sie haben jedoch eine Alternative. Anstatt bei jedem Schleifendurchlauf zu prüfen, ob das Ende des Arrays erreicht wurde lassen Sie Ihre Anwendung bewusst auf einen Fehler laufen. Die erzeugte `ArrayIndexOutOfBoundsException` fangen Sie dann gekonnt ab. Gerade bei Bytecode Interpretern können Sie hier nicht zu unterschätzende Geschwindigkeitsvorteile erringen[1].

Hashtable

Die `Hashtable` ist eine schnelle Datenstruktur, deren Geschwindigkeit u.a. von Ihrem Füllfaktor abhängt. Bei Aufruf des Konstruktors können Sie, durch Angabe eines eigenen Füllfaktors die Geschwindigkeit und den Platzbedarf zur Laufzeit bestimmen. Der Standardwert unter Java ist 75% und sollte für eine schnelle Anwendung nicht überschritten werden. Wird der Füllfaktor einer `Hashtable` überschritten so findet ein rehashing statt. Dabei wird eine neue (größere) `Hashtable` angelegt. Die alte `Hashtable` wird in die neue überführt, wobei alle Hashcodes neu berechnet werden. Eine schnelle `Hashtable` setzt somit voraus, dass ein rehashing nur in begrenztem Umfang erfolgt.

[1] Falls Sie Lust haben, testen Sie für eine beliebige Schleife einmal, ab welcher Elementanzahl sich der Einsatz dieser Möglichkeit lohnt.

Die Identifizierung der einzelnen Objekte in einer `Hashtable` erfolgt über den Hashcode. Daher sollten Sie bei der Wahl des Schlüsseltyps vorsichtig sein. Während bei einer kleinen `Hashtable` der Schlüsseltyp nur geringen Einfluss auf die Performance hat, sollten Sie bei einer großen `Hashtable` als Schlüsseltyp nicht Objekte der Klasse `String` verwenden. Hier bieten sich eher Objekte der Klasse `Integer` an, da deren Hashcode schneller berechnet werden können.

Die `Hashtable` ist auch nach der Anpassung an das Collection-Framwork weiterhin zu einem großen Teil synchronisiert. U.a. gilt dies auch für die Methode `clear()`. Daher sollten Sie, wenn Sie einen Objektpool mit einer `Hashtable` aufbauen, diese Methode nicht verwenden. Hier bietet es sich statt dessen, eine neue `Hashtable` zu definieren[1].

Die IntHashtable

Das sich die Sammlungen nicht für primitive Datentypen eignen, liegt daran, dass eine Sammlung nur Objekte aufnehmen kann. Die Entwickler von Java wollten sich jedoch offenbar nicht dieser Einschränkung unterwerfen und haben sich daher eine `Hashtable` für `int` Werte geschrieben.

Diese Hashtabe ist versteckt im Paket `java.text`. Da Sie netterweise final deklariert ist, ist ein Aufruf leider nicht möglich. Hier hilft nur „Copy & Paste“. Die Verwendung dieser Klasse ist jedoch wesentlich performanter als das Verpacken von `int` Werten in `Integer`. In unseren Beispiel zur Berechnung von Primzahlen haben wir diese z.B. verwendet.

Die HashMap

Mit dem Collection Framework wurde u.a. auch die Klasse `HashMap` eingeführt. Sie kann als Ersatz für eine `Hashtable` verwendet werden, sofern keine Synchronisation benötigt wird. Durch die fehlende Synchronisation ist der häufige Zugriff auf eine `HashMap` bedeutend schneller. Auch bei der `Hashmap` ist jedoch wie bei der `Hashtable` die Wahl des richtigen Schlüsseltyps wichtig. Zudem ist die Klasse `HashMap` auch noch flexibler als `Hashtable`.

Der Vector, der Stack, die ArrayList und die LinkedList

Die Klasse `Vector` ist eine der flexibelsten Strukturen der Sammlungen. Sie ist bereits seit dem JDK 1.0 vorhanden und ist daher auch weiterhin synchronisiert. Auch hier bietet sich daher eine Prüfung an, ob einer derartige Synchronisation benötigt wird. Sofern dies nicht der Fall ist, verwenden Sie besser die Klassen `ArrayList` oder `LinkedList`. Sofern Sie auf Java 1 angewiesen sind, müssen Sie zwangsläufig sich eine eigene Klasse[2] basteln. Da die Klasse `Stack` eine Subklasse von Vector ist, gelten diese Ausführungen ebenfalls.

---

[1] Das bedeutet, dass wir eine besonders langsame Methode bei einer großen Hashtable durch den langsamen Konstruktoraufruf ersetzen ☺ - faszinierend nicht war!

[2] Wie war das? Kopieren geht über studieren; oder so ähnlich.

Die Klasse `ArrayList` ist als nicht synchronisierter Ersatz für die Klasse `Vector` mit dem Collection Framework aufgetaucht. Sie ist, wie auch die `LinkedList` nicht synchronisiert, so dass der Zugriff auf Elemente dieser Sammlungen schneller ist.

Synchronisierte Sammlungen

Bis zum JDK 1.2 hatten Sie keine große Auswahl. Seitdem hat sich jedoch eine Menge getan. Es wurde der Typ `java.util.Collection` in Form eines Interfaces eingeführt, auf welchem alle aktuellen Sammlungen beruhen. Auch die bisherigen Typen, wie `Vector` wurden entsprechend angepasst. Bis zum JDK 1.2 bedeutet die Verwendung von Sammlungen immer auch Synchronisation, welche Sie nur durch Erstellung eigener Klassen verhindern konnten. Seit dem JDK 1.2 ist jede Sammlung, bis auf `Vector` und `Hashtable` nicht synchronisiert.

Es ist aber für viele Aufgaben durchaus sinnvoll bzw. nötig mit synchronisierten Sammlungen zu arbeiten. Dies stellt jedoch keine großen Anforderungen an die Implementierung dar. Mit Hilfe der Fabrikmethoden der Klasse `java.util.Collections` können Sie synchronisierte Sammlungen erzeugen.

```
// Beispiel synchronisieren einer TreeMap

TreeMap treeMap;
treeMap = (TreeMap)Collections.
        synchronizedSortedMap(new TreeMap());
```

Beim Aufruf einer Methode wird die Synchronisation durch den von Ihnen soeben erzeugten Wrapper bereit gestellt. Dieser leitet daraufhin den Aufruf an Ihre ursprüngliche Sammlung weiter. Für eine nicht synchronisierte Version eines `Vector` können Sie die Klasse `ArrayList` verwenden. Die Klasse `HashMap` können Sie statt der synchronisierten `Hashtable` verwenden.

nicht synchronisiert	synchronisiert
HashSet	
TreeSet	
ArrayList	Vector
LinkedList	Stack
HashMap	Hashtable
WeakHashMap	
TreeMap	

Sammlungen eignen sich grundsätzlich nur für Objekte. Sollten Sie eine große Anzahl von primitiven Datentypen sichern wollen ist ein

Array die bessere Alternative, da das Verpacken der primitiven Datentypen in Objekte zu einem enormen Overhead führt.

Methoden

Zu jeder nicht abstrakten Klasse gehören auch Methoden[1] und im Sinne der Objektorientierung delegieren Methoden ihre Aufgaben an andere Methoden. Dies ist sinnvoll, wünschenswert und gut. Es kann jedoch zu einer Geschwindigkeitsfalle werden.

Ein Problem ist das verhindern von Doppelaufrufen. Da die Java Entwickler den Quelltext des JDK offengelegt haben, sollten Sie einen Blick in diesen öfter riskieren. Schauen Sie sich die Implementation von Methoden an, um unnötige Aufrufe zu vermeiden. Ein weiterer wichtige Punkt sind die benötigten Variablen. Der Zugriff auf diese sollte so schnell wie möglich sein. Hierzu sollten Sie ggf. vom Memory Access Pattern gebrauch machen. Ziel dieses ist es den Wert einer Variablen innerhalb einer Methode schnell zu verändern und diesen nach Abschluss nach außen bereitzustellen. Besonders bei sehr zeitaufwendigen Operationen mit langsameren Variablentypen kann dies Geschwindigkeitsvorteile in Ihrer Anwendung bringen. Beachten Sie jedoch, dass gerade bei Multithreading der Zugriff ggf. synchronisiert werden muss[2].

Es gibt zwei Arten von Methoden – Methoden die statisch aufgelöste werden können und solche, bei denen dies nicht möglich ist. Methoden die statisch aufgelöst werden können sind als `static`, `final` oder `private` deklariert sind bzw. Konstruktoren. Die anderen können erst zur Laufzeit aufgelöst werden, so dass der Compiler bei der Bytecodeerstellung keine Optimierungen vornehmen kann. Auch die Sichtbarkeit von Methoden beeinflussen deren Ausführungsgeschwindigkeit. Für die Geschwindigkeit kann daher grob von folgender Hierarchie ausgegangen werden:

➢`static` ➢`private` ➢`final` ➢`protected` ➢`public` Methoden bzw.

➢`static` ➢`final` ➢`instance` ➢`interface` ➢`synchronisierte` Methoden.

Häufig wird empfohlen Methoden, welche nicht überschrieben werden als final zu deklarieren. In der Implementierung sollte dies nicht mehr zur Debatte stehen; dies ist Aufgabe des Designs.

Methodenaufrufe optimieren

Inlining

Als quasi-goldene Regel gilt: „Nur eine nicht vorhandene Methode, ist eine gute Methode“. Umgesetzt heißt dies Inlining. Jede Methode hat einen Overhead, der u.a. durch Initalisierung ihres Methoden-Stack entsteht. JIT- und HotSpot-Compiler können die Auswirkungen zwar minimieren, jedoch noch nicht völlig wettmachen. Das Inlining,

[1] Falls nicht, sollten Sie Ihr Design noch mal überprüfen!
[2] Ein Beispiel finden Sie unter Variablen / Datentypen und Operatoren

bedeutet nichts anderes, als den Inhalt einer Methode, an den Platz ihres Aufrufes zu kopieren. Bei statisch auflösbaren Methoden kann der Javacompiler „`javac`“ dies bereits übernehmen. Dies hat den Vorteil, dass sowohl Design als auch Implementierung ‚sauber‘ bleiben und trotzdem eine höhere Ausführungsgeschwindigkeit erreicht werden kann. Mit dem Schalter „`-o`“ wird der Optimierungsmodus des Compilers eingeschaltet. Kapselmethoden[1] können nicht inline gesetzt werden. Sofern Sie die Geschwindigkeit weiter erhöhen wollen, können Sie auch manuelles Inlining vornehmen[2]. Gegen Inlining spricht jedoch, dass längerer Bytecode erzeugt wird.

Überladene Methoden nutzen

Mit der Möglichkeit überladene Methoden zu erstellen haben wir eine mächtiges Mittel zur Verfügung, um ein Paket für einen Programmierer gut nutzbar zu machen. Dies zusammen mit der Möglichkeit Aufgaben zu delegieren macht u.a. eine gute Klassenbibliothek aus. Gerade dieses Delegieren zu umgehen, kann jedoch die Geschwindigkeit erhöhen. Die Implementation der Methode `Exception.printStackTrace()` sieht beispielsweise so aus:

```
public void printStackTrace() {
  synchronized (System.err) {
    System.err.println(this);
    printStackTrace0(System.err);
  }
}
```

Wenn Sie sich den Quellcode näher betrachten stellen Sie evtl. fest, dass Sie auch selbst den Aufruf der Methode `Exception.printStackTrace();` mit einem Parameter `PrintStream System.err` vornehmen könnten und auch synchronisieren von `System.err` stellt keine Anforderung an Sie dar. Testen Sie selbst bei welchen Methoden dies auch sinnvoll sein kann[3]. Falls Sie nur in einem Thread arbeiten, werden Sie die Synchronisierung nicht einmal mehr umsetzen. Dies führt zu einem weiteren Punkt:

Methoden überschreiben / Eigene Methoden nutzen

Wir befassen uns jetzt mit einem weiteren elementaren Punkt der Objektorientierung der Polymorphie. Natürlich will ich jetzt nicht auf das gesamte OO Konzept eingehen, was hinter diesem Begriff steht, Sie können dieses Prinzip jedoch gut für Ihre Performanceoptimierungen verwenden. Polymorphie bedeutet letztendlich nichts anderes, als dass, wenn Sie eine ererbte Methode aufrufen, zuerst das Objekt nach der entsprechenden Methode durchsucht wird, dann die Superklasse, dann die SuperSuperklasse usw. Dies ist ein sehr schönes und brauchbares Konzept hat jedoch auch seine Nachteile, so dass bestimmte Methoden stets überschrieben werden sollten. Dies hat nicht nur Performancegründe

[1] getter und setter
[2] Was in der vollkommenden Durchführung eine Ein-Klassen-Anwendung mit nur der main()-Methode bedeuten ☹!
[3] Neben den Swing-Klassen sollten Sie auch mal die Konstruktoren untersuchen.

sondern ist vielfach einfach sinnvoll. Ein besonders verständliches Beispiel lässt sich an der Methode `Object.equals(Object);` zeigen. So kann bereits bei einer kleinen Vererbungsstruktur ein Vergleich optimiert werden:

```
class Frau extends Mensch{

  public boolean equals(Object o) {
    if (o instanceof Frau){
      super.equals(o);
    }
    return false;
  }

  public void gebähre(){
  }

// hier kommt noch mehr

}
```

Die andere Möglichkeit ist das Verwenden eigener Methoden. Die Methoden der Java Klassenbibliotheken sollen jeden denkbaren Fall abdecken. Dies führt zu vielen Prüfungen, welche in Ihrer Anwendung evtl. nicht oder nur selten benötigt werden. Hier bietet es sich an eine eigene Methode zu schreiben, welche diese Aufgabe erledigt. Dies ist ebenfalls sehr sinnvoll bei synchronisierten Methoden. Sollte die Anzahl der Methoden sehr hoch werden, so bietet es sich wahrscheinlich sogar an, eine eigene spezialisierte Klasse zu schreiben. Dies führt zugleich zu einer flacheren Vererbungshierarchie, wodurch die Performance weiter erhöht wird.

Synchronisierte Methoden

Sie sollten synchronisierte Methoden vermeiden. Selbst wenn Sie mit nur einem Thread arbeiten sind synchronisierte Methoden langsamer. Wenn Sie eine Synchronisation erzeugen, erstellt die JVM zuerst einen Monitor für diesen Teil. Will ein Thread nun den Programmteil ausführen so muss er zuerst sich beim Monitors anmelden (und ggf. warten). Bei jedem Aufruf / Verlassen eines solchen Programmteils wird der Monitor benachrichtigt, was entsprechende Verzögerungen mit sich bringt.

Objekte

Objekte sind der Hauptbestandteil Ihrer Anwendung, jeder OO Anwendung. Daher wollen wir nun einige der Performancemöglichkeiten in Zusammenhang mit Objekten betrachten. Wenn es Ihnen im Augenblick um das Sichern von Objekten geht, sollten Sie gleich in den Abschnitt Sicherung und Wiederherstellung von Objekten im Teil Ein- und Ausgabe gehen.

Erzeugung von Objekten

Java verfügt inzwischen über eine Unmenge von Möglichkeiten Objekte zu erzeugen. Zu diesen Möglichkeiten zählen der

- `new` Operator

- `Object.getClass().newInstance()`
- `Class.forName(‚package.Class‘).newInstance()`
- `object.clone()`
- mittels Reflection
- die Wiederherstellung von gesicherten Objekten.

Außerdem gibt es noch die Möglichkeit mit `ClassLoader` Klassen als Bytekette zu laden.

Der „`new`" Operator ist der gebräuchlichste und schnellste dieser Möglichkeiten. Da Sie mit diesem die Konstruktoren aufrufen lohnt es sich diese genauer zu betrachten. Konstruktoren sind besondere Methoden, welche nur von einem Thread gleichzeitig aufgerufen werden können. Die Instanztierung eines Objekt ist keine performante Sache. Auch die JIT Compiler konnten diesen Flaschenhals bisher nicht beseitigen. Es liegt an Ihnen dies durch geeignete Konstrukte auszugleichen. Wenn Sie eine Abstraktionsebene tiefer in Ihrer Klassenbibliothek gehen, so bedeutet dies immer auch den Aufruf von einem oder mehreren weiteren Konstruktoren, daher sind flache Vererbungshierarchien performanter. Allerdings ruft nicht nur der „`new`" Operator einen Konstruktor auf, sondern auch das Wiederherstellen über die Schnittstelle `Externalizable` führt zum Aufruf des Standardkonstruktor. Es ist daher sinnvoll Objekte wieder zu verwenden, statt neu zu erstellen (siehe Wiederverwendung von Objekten).

Einen wirklich performanten, schnellen Konstruktor zu erstellen führt zwangsläufig wieder zur Kollision mit dem Design, da Sie hier die Delegation einschränkend müssen. Wenn Sie beispielsweise ein 100 `JLabel` Objekte erzeugen, können Sie durch verwenden des überladenen Konstruktors (übergeben Sie „`Text`", `null`, `JLabel.LEFT`) schon einen Geschwindigkeitsgewinn zwischen 230% (JIT Compiler) bis 830% (Bytecode Interpreter)[1] zum Aufruf des Standardkonstruktors mit „`Text`" verbuchen.

Um die Erstellung eines Objektes zu beschleunigen sollten Sie auf unnötiges Initialisieren von Variablen verzichten. Objekt- und Klassenvariablen erhalten als Referenz grds. den Wert `null`, während primitive Datentypen mit `0` initialisiert werden.

Innere Klassen

Inner Klassen sind langsamer als andere Klassen. Hier kann Sie die Erzeugung eines Objektes bei Ausführung mit einem Bytecode Interpreter durchaus die doppelte Zeit kosten. Bei JIT- und HotSpot Compilern liegt hier jedoch keine große diese Differenz mehr vor. Um für Bytecode Interpreter gerüstet zu sein, bietet es sich jedoch ggf. an,

[1] Diese Ergebnisse rühren noch aus einem Test mit dem JDK 1.1.7 her.

Klassen mit einer auf das Paket begrenzten Sichtbarkeit zu erstellen[1]. Ein weiterer Punkt ist die Größe des Bytecodes. Innere Klassen führen zu wesentlich mehr Bytecode als die vergleichbare Variante einer Klasse mit der Sichtbarkeit auf Paketebene.

```
class WoIC {
  MyOuter outer;
}

class WiIC {
  class MyInner {
  }
}

class MyOuter {
}
```

Der Unterschied für diese Klassen liegt beim dem normalen Kompiliervorgang bei 216 Bytes.

Dynamisches Nachladen von Klassen

Das gezielte Nachladen von Klassen kann die Performance weiter anheben. Mittels `Class.forName()` können Sie zu einem beliebigen Zeitpunkt Klassen in Ihre Anwendung laden. Dies kann in einem Low-Priority-Thread sehr hilfreich sein, um im Hintergrund bereits weitere Anwendungsteile zu initialisieren. Auch in Applets können Sie Klassen laden. Somit ist es Ihnen möglich, Objekte erst dann zu laden, wenn Sie diese benötigen. Gerade für eine Verringerung der Netzwerklast bzw. dem schnellen starten von Applets ein gutes Konstrukt.

Konstante Klassen

Eine Eigenschaft einiger Standardklassen von Java ist deren Deklaration als `final`. Dies sind u.a. die Klasse `String` sowie die Wrapperklassen für die primitiven Datentypen `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer` und `Long`. Dieses Konstrukt kann jedoch zu Performanceeinbüssen führen. Bei der Klasse `String` wird daher meist die Verwendung der Klasse `StringBuffer` zum Verketten von Zeichenketten vorgeschlagen. Dieses Konstrukt ist jedoch nur dann schneller, wenn es sich um mehr als zwei Zeichenketten handelt. Bei lediglich zwei Zeichenketten ist die Verwendung des ‚+'-Operators schneller als die `append` Methode der Klasse `StringBuffer`. Auch die Wrapperklassen können zu Performanceeinbüssen führen. Dies ist insbesondere mit den grafischen Swing Komponenten zu beachten, da Sie für jeden Wert auch jeweils ein neues Objekt erstellen müssen.

Sicherung von Objekten

Das Sichern und Wiederherstellen wird ausführlich im Abschnitt Ein- und Ausgabe unter „Sicherung und Wiederherstellung von Objekten" behandelt, da die Sicherung stets auch ein Problem der Ein- und Ausgabe ist.

[1] Hier gilt natülich wiedereinmal die Einschränkung „sofern möglich".

Verwerfen von Objekten

Ihre Referenzen sollten Sie erst dann freigeben, wenn Sie diese nicht mehr benötigen. Neben den Performancekosten für das Anlegen eines neuen Objektes kommt ansonsten auch noch die für die GarbageCollection dazu. Das gezielte Verwerfen (`object = null;`) von nicht mehr benötigten Objekten kann die Performance erhöhen. Dies liegt an dem Verhalten der GarbageCollection. Diese überprüft alle Objekte, die sich im Arbeitsspeicher befinden und nicht null sind, ob diese noch benötigt werden. Das explizite Setzen verringert somit die Arbeit der GarbageCollection. Der Quelltextschnipsel

```
for (int i = 0; i< 10000000; i++){
   java.awt.Button b = new java.awt.Button("");
   b = null;
}
```

führt daher unter bestimmten Voraussetzungen zu einem schnelleren Bytecode als

```
for (int i = 0; i< 10000000; i++){
   java.awt.Button b = new java.awt.Button("");
}
```

Leider ist auch hier wieder der tatsächliche Geschwindigkeitsgewinn entscheiden von der verwendeten JVM abhängig. Daher bietet es sich an, auf das gezielte Verwerfen zu verzichten. Unter HotSpot Compilern führt dies zudem meist zu langsameren Bytecode; bei dem JDK 1.1.7 und sofern Sie die Hintergrundarbeit der GarbageCollection abschalten, können Sie einige Performancegewinne erzielen.

Wiederverwendung von Objekten

Sinnvoll ist es jedoch eigentlich Ihre Referenzen erst dann freizugeben, wenn Sie diese nicht mehr benötigen. Ein Wiederverwenden spart Ihre Zeit, da die GarbageCollection seltener arbeitet und der Overhead für das Anlegen von Objekten wegfällt. Es gibt dabei drei wesentliche Möglichkeiten: das Sammeln bestehender Objekte mit Hilfe eines Objektpools und / oder Caches, das Klonen bestehender Objekte und das Kopieren bestehender Objekte. Das Kopieren ist ein Sonderfall eines Arrays und wurde aus diesem Grund in den Abschnitt ‚Arrays kopieren' verlagert.

Klonen

Klonen ist das Anlegen einer Kopie mit den gleichen Eigenschaften, wie das Original. Klonen kann zu einer deutlichen Performanceverbesserung führen. Insbesondere unter einer JVM mit Bytecode Interpreter ist eine deutliche Beschleunigung spürbar. Damit Ihre Objekte klonbar werden, muss Ihre Klasse das Interface `java.lang.Cloneable` implementieren. Klonen Sie Objekte jedoch nur, wenn Sie ein Objekt mit genau des selben Eigenschaften neben Ihrem bisherigen Objekt benötigen. Falls Sie irgendwann einmal ein Objekt mit den selben Eigenschaften benötigen, ist hingegen der Cache die richtige Wahl.

Cache

Ein Cache ist ein Behälter in welchen Sie ein beliebiges Objekt ablegen können, um es zu späterer Zeit mit seinem jetzigen Zustand weiter zu verwenden. Sie könnten statt eines Cache auch Ihr Objekt sichern. Da Sie mit einem Cache das Objekt jedoch noch im Arbeitsspeicher haben, ist dies die bessere Alternative. Seit dem JDK 1.2 wurde dem Problem des Speicherüberlaufs durch die Klasse `SoftReference` wirksam begegnet, so dass dieses Hindernis weggefallen ist. Ein typisches Anwendungsbeispiel ist das sichern von Mediadaten im Cache. Bilder und Musik sind üblicherweise statische nicht änderbare Datenpakete und daher hervorragend für einen Cache geeignet. Immer wenn Sie ein Objekt dieses Typs benötigen, prüfen Sie zuerst, ob bereits ein entsprechendes Objekt vorliegt. Falls dies nicht der Fall ist erzeugen Sie ein Objekt und sichern dies im Cache. Nun arbeiten Sie mit dem Objekt im Cache. Ein Cache führt zwar zu mehr Bytecode und erhöht den Speicherverbrauch zur Laufzeit, kann jedoch in Ihrer Anwendung erheblich die Geschwindigkeit erhöhen und die Netzwerklast vermindern.

Ein Sonderfall ist der externe Cache. Hierbei werden nicht kritische, statische Daten auf den lokalen Rechner verlagert. Dies ist insbesonders bei einer geringen Netzwerkbandbreite sinnvoll und wird daher von allen üblichen Online-HTML-Browsern unterstützt.

Ein Cache bietet sich für Ihre Anwendung somit immer an, wenn Sie Daten haben, die Sie innerhalb Ihrer Anwendung nicht verändern. Sofern Sie jedoch genau dies vorhaben, verwenden Sie einen Objektpool.

Objektpool

Ein Objektpool[1] erlaubt es Ihnen Objekte aus einem Objektpool herauszunehmen, zu verändern und wieder in den Pool zu stellen. Auch hierbei wird der Overhead des Erzeugen von Objekten umgangen. Die Entnahme eines Objektes aus dem Objektpool liefert Ihnen dabei ein Objekt in einem bestimmten fachlichen Zustand. Sie können nun mit diesem Objekt beliebig arbeiten. Ähnlich wie bei der GarbageCollection müssen Sie sich nicht darum kümmern, was mit dem Objekt passiert, nachdem Sie es wieder dem Objektpool zur Verfügung gestellt haben. Der Objektpool selbst kümmert sich jedoch, um Ihr Objekt und stellt den definierten Anfangszustand wieder her.

Ein sehr häufiges Anwendungsgebiet sind Datenbankabfragen. Die eigentliche Datenbankverbindung wird dabei vorrätig gehalten und kann bei Bedarf abgerufen werden. Nach Verwendung gibt Ihre Anwendung die Datenbankverbindung zurück zum Objektpool.

Vergleichen von Objekten

Das Vergleichen von Objekten kann optimiert werden, in dem Sie zuerst der `Hashcode` der Objekte vergleichen.

[1] Ein Beispiel für den Aufbau eines Objektpools befindet sich im Anhang.

```
if (objectA.hashCode()==objectA.hashCode() &&
    objectA.equals(objectB))

// und weiter geht es
```

Hierbei wird nichts anderes gemacht, als was wir bereits im Abschnitt ‚Methoden überschreiben / Eigene Methoden nutzen' besprochen haben. Natürlich ist das Ergebnis bei identischen Objekten nicht so schnell verfügbar, wie bei unterschiedlichen Objekten. Auch bei zwei Objekten mit flacher Vererbungshierarchie kann die Optimierung hier umschlagen; dies ist wie viele Optimierungsmöglichkeiten ein zweischneidiges Schwert.

Plattformübergreifend native Implementierung

Auch wenn sich plattformübergreifend und nativ scheinbar widersprechen, so ist dies möglich. Auch die JVM basiert auf Betriebssystemroutinen und einige dieser Routinen werden durch das JDK direkt genutzt – die nativ implementierten Methoden. Während Ihre Eigenentwicklungen nicht plattformunabhängig, sobald Sie eigene native Methode hinzufügen, ist es jedoch möglich native Methoden, nämlich die des JDK, zu nutzen und trotzdem plattformunabhängig zu bleiben. Da jede JVM diese Methoden implementieren muss, egal für welches Betriebssystem Sie entwickelt wurde, können Sie diese schnelleren Methoden verwenden. Die Methoden `arraycopy()` und `drawPolyline()` sind zwei der bereits nativ implementierten Methoden.

Exceptions

`Exception` also Fehler oder Ausnahmen sind eine weitere Möglichkeit Geschwindigkeit zu gewinnen oder zu verlieren. Ein `Exception` erzeugt jeweils ein Standbild des Javastack zum Auslösungszeitpunkt und ist daher synchronisiert. Ein `try-catch-finally` Block hingegen kostet grds. lediglich Bytecode. Hinzu kommt, dass auch `Exceptions` in unserer Anwendung nur als Objekte vorkommen und deren Erzeugung einen gewissen Overhead mitbringt.

Die Wahrscheinlichkeit der Auslösung einer `Exception` in Ihrer Anwendung sollte daher deutlich unter 50% liegen. Sie können jedoch `Exceptions` auch nutzen. So kann das Abfangen einer `Exception` in Ihrer Anwendung deutlich performanter sein, als z.B. das Prüfen der Abbruchbedingung einer Schleife mit vielen Durchläufen. Sofern die Wahrscheinlichkeit zu hoch für das Auslösen einer `Exception` ist, sollten Sie besser eine Prüfung durchführen und ggf. einen Fehlerwert erzeugen. Eine weitere Möglichkeit die Performance zu erhöhen ist das zusammenfügen vieler kleiner `try-catch-finally` Blöcke zu einem größeren. Sie können somit die Anzahl des Erzeugens und Abfangens von `Exceptions` senken. Außerdem wird der Compiler an der Optimierung eines Quelltextes mit vielen `try-catch-finally` Blöcken gehindert. Dies ist jedoch abhängig von der Semantik Ihres Quelltextes und kann nicht generell vorgenommen werden.

Ein- und Ausgabe

Ströme

Ein Strom (engl. `Stream`) kann man als Schnittstelle der Anwendung nach außen definieren, sofern wir die Benutzeroberfläche [BNO] nicht berücksichtigen. Ströme sind eine Anwendungsschnittstelle, die weitreichend genutzt wird. Sowie das Einlesen von Daten und Dateien als auch das Übertragen dieser innerhalb von Netzwerken sowie die Sicherung und Wiederherstellung von Objekten – meist mit dem Begriff Serialisierung gleichgesetzt – erfolgt mittels dieser. Typische Beispiele sind Server, da ein Großteil derer Tätigkeit hier in der Übertragung von Daten lokal ins Dateisystem oder über Netzwerk besteht.

Allgemeines

Das Paket `java.io` enthält die wichtigsten Ein- und Ausgabeströme. Alle Ein- und Ausgabeströme sind von den beiden abstrakten Klassen `InputStream` und `OutputStream` abgeleitet, welche die grundsätzlichen Operationen definieren. Außerdem werden auch Reader und `Writer` zur Verfügung gestellt.

Eine grundsätzliche Eigenschaft fast aller Stromklassen ist die Möglichkeit Sie in andere Ströme zu schachteln. Dieses Prinzip wird z.B. bei den Filterströmen angewendet. Eine Möglichkeit die Performance -Geschwindigkeit- zu erhöhen, ist dieses Prinzip zu puffern der Ein- bzw. Ausgabeströme anzuwenden. Außerdem ist ein Vergleich der Reader bzw. `Writer` zu den Strömen aus Performancessicht nötig.

Reader contra Eingabeströme

Die Ströme selbst sind auf einem sehr niedrigen Level angesiedelt und basieren in den wichtigen Methoden `read()` und `write()` auf native Methoden. Das folgende kleine Listing zeigt ein Programm, welches mit dem `FileInputStream` und dem `FileReader` eine Datei öffnet und die Daten in einem `byte` Array sichert.

```
package
make_java.performance.buch.readerUndEingabestroeme;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;

public class Datei {
  public Datei(File file) {
    long time;

    time = System.currentTimeMillis();
      this.inputWithReader(file);
    System.out.println("Reader:\t"+
                        (System.currentTimeMillis()-
time));

    time = System.currentTimeMillis();
```

```
      this.inputWithInputStream(file);
    System.out.println("Stream:\t"+
                        (System.currentTimeMillis()-
time));
  }

  byte [] inhalt;

  public void inputWithReader (File file){
    inhalt = new byte [(int)file.length()];
    try{
      FileReader fr = new FileReader(file);
      for (int i =0; i< file.length(); i++){
        inhalt[i] = (byte) fr.read();
      }
    }
    catch (IOException e){
      System.out.println("Fehler");
    }
  }

  public void inputWithInputStream (File file){
    inhalt = new byte [(int)file.length()];
    try{
      FileInputStream fis = new FileInputStream(file);
      for (int i =0; i< file.length(); i++){
        inhalt[i] = (byte) fis.read();
      }
    }
    catch (IOException e){
      System.out.println("Fehler");
    }
  }
```

Das Programm wurde bewusst nicht weiter optimiert. Beim Laden einer Datei mit einer Größe von ca. 207 Kilobyte ergab sich ein Geschwindigkeitsvorteil von etwa acht bis neun Sekunden für den `FileInputStream`. Obwohl die Klassen die gleiche Aufgabe erledigen sollen ergibt sich ein größerer Performanceunterschied. Ein Blick in den Quellcode des `FileReader` (hier aus dem JDK 1.2.2) zeigt uns warum:

```
public class FileReader extends InputStreamReader {

  public FileReader(String fileName)
                    throws FileNotFoundException {
    super(new FileInputStream(fileName));
  }

  public FileReader(File file) throws
FileNotFoundException {
    super(new FileInputStream(file));
  }

  public FileReader(FileDescriptor fd) {
    super(new FileInputStream(fd));
    }

}
```

Es wird schnell deutlich das die Aufgabe der `FileReader` darin besteht, einen `FileInputStream` zu öffnen und diesen dann an den Konstruktor der Superklasse zu übergeben. Sobald wir nun noch einen Blick in den Quellcode der Klasse `InputStreamReader` werfen stellen wir fest, dass mittels der `FileReader` und auch der anderen Reader noch andere Aufgabe verbunden sind, als das bloße Einlesen der

Dateien. Tatsächlich werden die eingelesenen Byte mittels des `FileToCharConverter` in Zeichen umgewandelt. Bei dem oben dargestellten Beispiel öffnen wir also eine Datei lesen diese ein, wobei eine Umwandlung in `Character` erfolgt um diese dann wieder in Byte zu zerlegen. Dabei findet u.a. auch noch eine Synchronisation statt, die das Einlesen weiter verlangsamt.

Es bleibt festzustellen, dass aus Performancesicht die Verwendung von Readern und auch Writern nur dann sinnvoll ist, wenn diese genau die Aufgabe bereits wahrnehmen die gewünscht wird. Auch bei kleineren Abweichungen lohnt sich – wie meist – eher das Erstellen einer eigenen Klasse, welche mit Hilfe von Strömen die Aufgaben erledigt.

Gepufferte Ströme

Um die Vorteile von gepufferten Strömen nachzuvollziehen, ist es nötig zuerst das Vorgehen ohne Pufferung zu betrachten. Dazu betrachten wir einmal die Methode aus `InputStream` zum Einlesen von Daten aus der Klasse `InputStream`:

```
public abstract int read() throws IOException;
```

Die Methode wird nur von den Subklassen implementiert, so dass wir in eine dieser einen Blick werfen müssen.

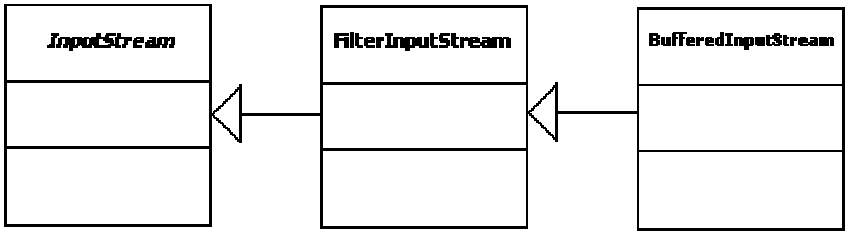
```
public int read(byte b[], int off, int len)
                                    throws IOException {
 if (b == null) {
     throw new NullPointerException();
 } else if ((off < 0) || (off > b.length) || (len < 0)
||
           ((off + len) > b.length) || ((off + len) <
0)) {
     throw new IndexOutOfBoundsException();
 } else if (len == 0) {
     return 0;
 }

 int c = read();
 if (c == -1) {
     return -1;
 }
 b[off] = (byte)c;

 int i = 1;
 try {
     for (; i < len ; i++) {
        c = read();
        if (c == -1) {
            break;
        }
        if (b != null) {
            b[off + i] = (byte)c;
        }
     }
 } catch (IOException ee) {
 }
 return i;
    }
```

Im Sinne guter Objektorientierung, wird das eigentliche Lesen der Daten an die abstrakte Methode `read()` übertragen. Wie wurde diese Methode jedoch tatsächlich implementiert? Wenn wir uns eine Implementation (`FileInputStream`) der Methode anschauen, sehen wir, dass diese Methode nativ realisiert wird. Wichtig zu wissen ist, dass Sie über die meisten Ströme jedes Byte einzeln bewegen. Dies ist etwa genauso performant, als ob Sie einen Stapel einzelner Blätter dadurch um 20 cm verschieben, in dem Sie jedes einzelne Blatt bewegen, anstatt den Stapel im Ganzen oder zumindest teilweise nehmen.

Genau für diese Vorgehensweise wurden die gepufferten Ströme (`BufferedInputStream` und `BufferedOutputStream`) entwickelt.



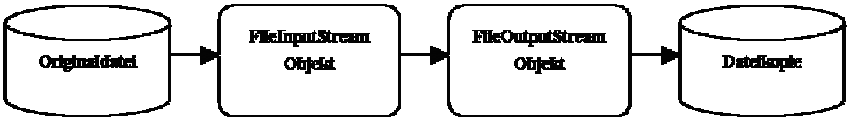
Vererbungshierarchie zur Klasse BufferedInputStream

Am Beispiel einer Kopieranwendung werden wir im Folgenden verschiedene Möglichkeiten vergleichen. Als Erstes betrachten wir eine einfache Anwendung:

```
private InputStream  in  = null;
private OutputStream out = null;

private void simpleCopy () throws IOException{
    System.out.println("Einfaches Kopieren");
    int data;
    while (true){
      data = in.read();
      if (data == -1){
        break;
      }
      out.write(data);
    }
    out.flush();
  }
```

Der Quellcode zeigt das einfache Kopieren einer beliebigen Datei. Dabei wird jedes Byte gelesen, überprüft und wieder geschrieben. Für das Übertragen einer Datei mit 1.269.333 Bytes wird etwa 13 Zeiteinheiten beim ersten Durchlauf und den folgenden Durchläufen (JIT).



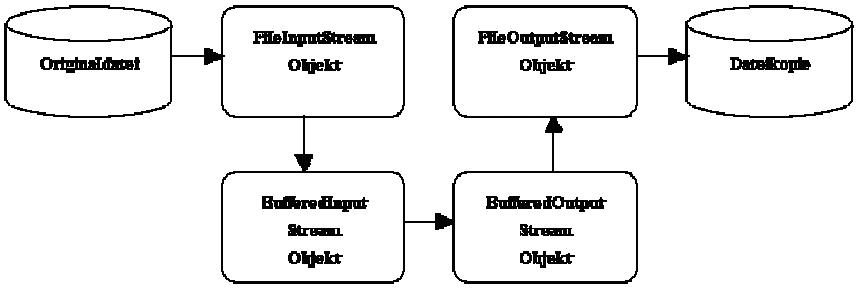
Kopiervorgang bei einfachem Kopieren

Durch das Aufsetzen der gepufferten Ströme auf Ströme kann bereits ein Geschwindigkeitsgewinn erreicht werden. Der gepufferte Strom sichert dabei die einzelnen Bytes zwischen bevor Sie die Bytes geliefert bekommen. Somit kann eine größere Geschwindigkeit erreicht werden. Da der gepufferte Strom als Parameter im Konstruktor lediglich ein Objekt vom Typ `InputStream` bzw. `OutputStream` benötigt, können Sie diesen mit jedem anderen Strom verschachteln.

```
private InputStream  in  = null;
private OutputStream out = null;

private void bufferCopy () throws IOException{
  System.out.println("Gepuffertes Kopieren");
  int data;
  BufferedInputStream  bIn  = new BufferedInputStream
(in);
  BufferedOutputStream bOut = new BufferedOutputStream
(out);
  while (true){
    data = bIn.read();
    if (data == -1){
      break;
    }
    bOut.write(data);
  }
  bOut.flush();
  out.flush();
}
```

Der gepufferte Kopiervorgang benötigt für eine Datei mit 1.269.333 Bytes nur noch etwa 3,3 Zeiteinheiten beim ersten Durchlauf. Mit einem JIT bzw. HotSpot –Compiler verringert sich die benötigte Zeit bei nochmaligem Kopieren auf etwa 1,6 Zeiteinheiten. Hier kommt die Umsetzung in nativen Maschinencode voll zum tragen.



Kopiervorgang bei gepuffertem Kopieren

Neben der reinen Geschwindigkeitsbetrachtung sollten wir jedoch einen Blick auf die Größe des Bytecodes werfen. Dieser wird entsprechend größer.

Eigene Puffer

Um die Geschwindigkeit weiter zu steigern bietet es sich an einen eigenen internen Puffer anzulegen. Dabei haben wir mehrere Geschwindigkeitsvorteile. Zuerst können wir über eine Methodenvariablen zugreifen, was uns den optimalen Zugriff sicher.

Außerdem können wir mit gepufferten Strömen arbeiten und können mit einem `byte` Array Geschwindigkeitsrekorde brechen.

```
byte [] dateiPuffer = new byte [pufferGroesse];
BufferedInputStream bis = new BufferedInputStream (in);
bis.read (dateiPuffer);
```

Bei Betrachtung dieses kleinen Codeausschnittes können wir bereits sehen, dass nunmehr unser Puffer der Methode übergeben wird. Anstatt die Daten also erst intern in einen anderen Puffer zu schreiben und uns auf Anforderung zu übersenden wird unser Puffer sofort mit den Dateiwerten gefüllt. Analog zum Einlesen des Puffers ist es auch möglich der `write`–Methode einen Puffer zu übergeben.

Synchronisierung bei Strömen

Nachdem wir bereits einen Blick in den Quellcode geworfen haben, konnten Sie feststellen, dass die `read()` und `write()` –Methoden nicht synchronisiert sind. Dies ist jedoch nur die Vorgabe der Klassen `InputStream` und `OutputStream`. Die gepufferten Ströme hingegen verwenden synchronisierte Operationen, um die Daten einzulesen und zu schreiben.

```
   public synchronized int read() throws IOException {
       ensureOpen();
if (pos >= count) {
    fill();
    if (pos >= count)
       return -1;
}
return buf[pos++] & 0xff;
   }
```

Dies, die Tatsache, dass Arrays mehrfach kopiert werden und dass für jede Kopie ein neues Array erzeugt wird, zeigt jedoch, dass weiteres Optimierungspotential vorhanden ist. Um die Vorteile eines Puffers für das Kopieren unserer Daten nutzen zu können, legen wir uns somit einen eigenen größeren Puffer an, in welchem wir die Daten zwischensichern. Eine andere Möglichkeit ist die Erhöhung des durch das `BufferedStream` Objekt genutzten Puffer, durch Übergabe eines entsprechenden Parameter an den Konstruktor. Diesen Puffer können wir dann den beiden Methoden `read()` und `write()` übergeben. Optimale Größe ist die Größe des Eingabestromes, welchen wir über die Methode `available()` des Eingabestromes bekommen.

Bei diesem Code ist jedoch der Speicherverbrauch kritisch zu betrachten. Bei langen Dateien wird der Puffer und somit unser Array entsprechend groß und beansprucht viel Speicher. Dies kann schnell zu einem Speicherüberlauf führen. Die Möglichkeit mittels Abfangen des Fehlers und Zugriff auf den Standartpuffer im Fehlerfall müssen wir Verwerfen, da die JVM erst versucht das Array zu füllen. Um die Laufzeitfehler zu vermeiden können wir einen festen Puffer auf Klassenebene nutzen. Dieser muss nun nur einmal initialisiert werden. Um jedoch Fehler zu vermeiden, muss der eigentliche Kopiervorgang synchronisiert werden.

Jedoch können wir bei großen Strömen in einigen Fällen die Laufzeitfehlerfehler nutzen um unsere Geschwindigkeit zu erhöhen.

Laufzeitfehler nutzen

Gerade für das Kopieren von Daten des Dateisystems können wir die `EOFException` und `ArrayIndexOutOfBoundsException` sinnvoll nutzen. Dabei ist jedoch zu beachten, dass Geschwindigkeitsvorteile nur mit JIT bzw. HotSpot Compiler erreicht werden können.

```
private InputStream  in  = null;
private OutputStream out = null;

private void ownBufferCopyWithExceptions (
        String fromFile,
        String toFile)
         throws IOException{
  BufferedInputStream  bIn  =
          new BufferedInputStream  (in,2048);
  BufferedOutputStream bOut =
          new BufferedOutputStream (out,2048);
  byte [] dateiInhalt = new byte[in.available()];

    // Lesen
    try{
        bIn.read(dateiInhalt);
    }
    catch (EOFException eofe){
      /* Datei eingelesen */
    }
    catch (ArrayIndexOutOfBoundsException aioobe){
      /* Datei eingelesen */
    }

    // Schreiben
    try {
      bOut.write (dateiInhalt);
    }
    catch (ArrayIndexOutOfBoundsException aioobe){
      /* Datei geschrieben */
    }

    bOut.flush();
    out.flush();
  }
```

Bei Nutzung von Laufzeitfehlern entsteht zwangsläufig ein Overhead durch das Erzeugen der Fehlerobjekte. Ob dieser geringer ist, als der Gewinn, welcher durch die Vermeidung der Prüfung auf das Stromende ist, hängt im wesentlichen von der Größe des Stromes ab.

Für unsere Datei benötigten wir beim ersten Durchlauf lediglich eine Zeit von 1,7 Zeiteinheiten. Bei den folgenden Durchläufen ergab sich jedoch kein Geschwindigkeitsgewinn mehr.

Es wird somit deutlich, das der Geschwindigkeitsgewinn beim Verwenden von Strömen von der jeweiligen Umgebung abhängt und dem Zweck der Anwendung abhängt. Eine Anwendung, deren Aufgabe lediglich das einmalige Kopieren ist – z.B. Installationsroutinen für eine geringe Anzahl von Dateien – haben Geschwindigkeitsvorteile eher durch Abfangen der Fehlermeldungen. Sofern viele Kopien oder auch

Kopien von kleinen Dateien Hauptaufgabe einer Anwendung ist, sollte man eine Prüfung auf das Ende des Stromes vorziehen.

Ergebnisübersicht

Das Vorgehen ist also einer der entscheidenden Punkte, wie wir die Performance einer Anwendung erhöhen können. Die folgenden Tabellen zeigen einige Ergebnisse aufgrund der vorgestellten Möglichkeiten:

Vorgehensart	Dateigröße 1,2 MB	Dateigröße 12 MB
Einfaches Kopieren	13,950 sek	155,053 sek
Gepuffertes Kopieren	3,364 sek	37,394 sek
Gepuffertes Kopieren (Wiederholung)	1,592 sek	18,086 sek
Kopieren mit eigenem Puffer	0,512 sek	5,638 sek
Kopieren mit Exception	1,692 sek	19,438 sek
Kopieren mit Exception (Wiederholung)	1,773 sek	19,148 sek

Tabelle 4 - Geschwindigkeitsvergleich des Kopieren von Dateien mit JIT Compiler

Wie aus der Tabelle 4 - Geschwindigkeitsvergleich des Kopieren von Dateien mit JIT Compiler ersichtlich wird, ist bei großen Dateien das Kopieren mit `Exception` auch mit JIT noch etwas performanter. Sie warten jedoch auf einen Anwendungsfehler und fangen diesen bewusst nicht auf. Sollte an dieser Stelle (warum auch immer) ein anderer Fehler die gleiche `Exception` werfen, können Sie diese nicht mehr sinnvoll abfangen. Da sich keine wesentlichen Geschwindigkeitsvorteile damit erreichen lassen, sollten Sie auf dieses Konstrukt besser verzichten.

Vorgehensart	Dateigröße 1,2 MB
Einfaches Kopieren	15,232 sek
Gepuffertes Kopieren	5,928 sek
Gepuffertes Kopieren (Wiederholung)	5,949 sek
Kopieren mit Exception	6,429 sek
Kopieren mit Exception (Wiederholung)	6,409 sek

Tabelle 5 - Geschwindigkeitsvergleich des Kopieren von Dateien ohne JIT Compiler

Sicherung und Wiederherstellung von Objekten

Die Sicherung und das Wiederherstellen von Objekten wird meist mit der Serialisierung gleichgesetzt. Obwohl die Serialisierung, aufgrund ihres Einsatzes bei Remote Method Invocation (RMI) und in Jini™, ein wesentliches Verfahren zur Sicherung von Objekten ist, gibt es weitere Möglichkeiten.

Allgemeines

Als Sicherung und Wiederherstellung von Objekten im hier verwendeten Sinne ist das Sichern aller wesentlichen Objektinformationen zu verstehen, so dass ein Objekt bzw. das Objekt wieder im Speicher angelegt werden kann. Das Sichern und Wiederherstellen kann somit auf verschiedenen Wegen geschehen. In der Java Standart API stehen uns für diese Aufgabe bereits zwei Schnittstellen im Pakte `java.io` zur Verfügung: `Serializable` – der Standart und `Externalizable` als eine besondere Form. Des weiteren ist die Möglichkeit des Sichern / Wiederherstellen des Objekts mittels XML und innerhalb von Datenbanken vorhanden.

Die Optimierung bzgl. Datenbanken und Zugriff auf diese wird in diesem Abschnitt nicht betrachtet, da dies ein weiteres komplexes Thema ist.

Serialisierung

Die Sicherung bzw. Wiederherstellung von Objekten mit Hilfe der Schnittstelle `Serializable` ist der übliche Weg, um Objekte zu sichern. Die Schnittstelle ist eine der kürzesten Typdefinitionen in der Java API:

```
package java.io;
public interface Serializable {
}
```

Ziel ist es lediglich den Objekten einer Klasse den Typ `Serializable` zuzuweisen. Dadurch wird die Sicherung der Objekte ermöglicht. Das Sichern der Objekte kann nunmehr über einen `ObjectOutputStream` erfolgen.

```
ObjectOutputStream oos = new
ObjectOutputStream(outputStream);
oos.writeObject(object);
```

Die Vorteile einer Serialisierung sind schnell aufgezählt. Serialisierung ist einfach umzusetzen, da die Implementation der Schnittstelle ausreichend ist. Serialisierung erfolgt zwingend über einen Strom, so dass das Versenden und Empfangen von Objekten in Netzwerken ebenso gut und schnell zu realisieren ist, wie das Sichern auf Datenträgern. Sämtliche Informationen eines Objektes werden automatisch berücksichtigt. Das Sichern eines Objektes auf einem Datenträgern kann dabei ähnlich dem Kopieren von Dateien vorgenommen werden:

```
public class MyObject implements Serializable{

  private int alter = 18;
  private int geburtsJahr = 1975;
  private int aktuellesJahr;

  public MyObject(){
  }
```

```
  public MyObject(int geburtsJahr) {
    this.setGeburtsJahr (geburtsJahr);
  }

  public void setAktuellesJahr (int aktuellesJahr){
    this.aktuellesJahr = aktuellesJahr;
  }

  public void setGeburtsJahr (int geburtsJahr){
    this.alter = aktuellesJahr - geburtsJahr;
  }

  public void saveObject(){
    FileOutputStream fos = null;
    try{
      File f = new File("Serial.ser");
      fos = new FileOutputStream(f);
      ObjectOutputStream oos = new
ObjectOutputStream(fos);
      oos.writeObject(this);
      oos.flush();
    }
    catch (IOException ioe){
      ioe.printStackTrace(System.out);
    }
    finally{
      try{
        fos.close();
      }
      catch (IOException ioe){
        ioe.printStackTrace();
      };
    }
  }
}
```

Die Serialisierung eines Objektes benötigt 142 Byte in unserem Ausgabestrom (JDK 1.3.0). Die Frage die sich somit zwangsläufig aufdrängt: welche Informationen werden in unserem Strom gesichert? Bei der Serialisierung werden der Klassenname, die Meta-Daten, Name und Typ jeder Eigenschaft auf Objektebene gesichert. Außerdem wird der gleiche Vorgang für jede Superklasse von `MyObject` vorgenommen. Da in Java Referenzen auf andere Objekte als Eigenschaften des Objektes behandelt werden, sichern wir mittels der Serialisierung auch diese Objekte grundsätzlich mit. Dies erklärt, warum für unser Objekt bereits 142 Byte benötigt wurden. Bei Beziehungen auf andere Objekte wird es durch deren Meta-Daten, wie Klassenname etc. schnell ein vielfaches. Die Sprachsyntax lässt uns jedoch nicht im Stich und bietet uns eine komfortable Möglichkeit die Größe und somit auch die Geschwindigkeit beim Sichern und Wiederherstellen eines Objektes zu beeinflussen.

Der ObjectOutputStream

Der `ObjectOutputStream`, mit welchem wir oben bereits gearbeitet haben, hat eine Eigenschaft, welche die Performance unserer Anwendung wesentlich beeinflusst. Sofern Sie nur einige wenige Objekte sichern, werden Sie kaum auf Probleme stoßen. An der Ausgabe des folgenden Beispieles werden Sie jedoch ein Problem erkennen, welches der `ObjectOutputStream` verursachen kann.

```
package bastie.performance.io.object;

import java.io.*;

public class Streams {

  private static final int MAX_COUNT = 10000000;

  public Streams() {
    FileOutputStream fos = null;
    BufferedOutputStream bos = null;
    ObjectOutputStream oos = null;
    try{
      fos = new FileOutputStream ("C:\\OOS.bst");
      bos = new BufferedOutputStream(fos,100000);
      oos = new ObjectOutputStream(bos);

      for (int i = 0; i < Streams.MAX_COUNT; i++){
        SaveObject save = new SaveObject();
        oos.writeObject(save);
        save = null;
      }
    }
    catch (IOException ioe){
      ioe.printStackTrace();
    }
    finally{
      try{
        oos.flush();
        bos.flush();
        fos.flush();
        oos.close();
        bos.close();
        fos.close();
      }
      catch (IOException ignored){}
    }
  }

  public static void main(String[] args) {
    Streams s = new Streams();
  }
}

class SaveObject implements Serializable{}
```

Damit Sie das Problem erkennen können, starten Sie die JVM unter Ausgabe der Arbeit der GarbageCollection[1]. Die Ausgabe, die Sie erhalten sollte dann ähnlich der unteren sein:

```
[GC 101K->91K(1984K), 0.0047880 secs]
[Full GC 1931K->939K(2144K), 0.1116996 secs]
[Full GC 2091K->1355K(2836K), 0.1551373 secs]
[Full GC 2443K->1611K(3264K), 0.1655439 secs]
[Full GC 3264K->2297K(4408K), 0.3183278 secs]
[Full GC 3973K->2885K(5388K), 0.2477401 secs]
[Full GC 5253K->4165K(7520K), 0.5004811 secs]
[Full GC 7301K->5701K(10272K), 0.4770002 secs]
[Full GC 10053K->8133K(14648K), 1.0023131 secs]
[Full GC 14341K->11205K(20152K), 0.9469679 secs]
[Full GC 19845K->16325K(29324K), 2.0997476 secs]
[Full GC 28677K->22469K(39756K), 1.9442968 secs]
[Full GC 39109K->31941K(55544K), 6.2937264 secs]
```

[1] Bei der Sun JVM 1.3.0 bedeutet dies, dass Sie den Parameter ‚-verbosegc' mit angeben müssen.

```
[Full GC 54725K->44229K(65280K), 4.4415177 secs]
[Full GC 65280K->54783K(65280K), 5.9746309 secs]
[Full GC 65280K->62976K(65280K), 9.3792020 secs]
[Full GC 65280K->98K(65280K), 44.2969396 secs]

java.lang.OutOfMemoryError

            <<no stack trace available>>

Exception in thread "main" Dumping Java heap ...
allocation sites ... done.
```

Es ergeben sich also mehrere Probleme. Das Augenscheinlichste ist, dass Sie einen `OutOfMemoryError` bekommen. Außerdem arbeitet die GarbageCollection sehr häufig, wobei Sie steigenden Speicher und Zeitverbrauch feststellen können. In unserem Beispiel[1] Steigt der Speicher von 1984 Kb auf das Maximum, wobei die GarbageCollection innerhalb unserer Anwendung bereits etwa 78 Sekunden unsere Anwendung benötigt.

Der `ObjectOutputStream` legt bei der Sicherung von Objekten eine Referenz auf diese an. Dies führt nun dazu, dass in unserem Beispiel der Speicherverbrauch stetig steigt und mit diesem auch die Zeit der GarbageCollection bis letztendlich kein Speicher mehr vorhanden ist. Um dieses Problem zu lösen gibt es die Methode `reset()`. Mit dieser wird die Referenzensammlung gelöscht. Beachten Sie jedoch, dass Sie ggf. zu übertragende Objekte neu in den Strom schreiben müssen.

Das Schlüsselwort transient

Mit Hilfe des Schlüsselwort `transient` können wir Informationen eines Objektes, welche wir nicht sichern wollen von der Serialisierung ausnehmen. In unserem Beispiel bietet sich das alter und das `aktuellesJahr` an. Das aktuelle Jahr dient nur als Rechengröße und ist für unser Objekt nicht relevant – aus Sicht der Objektorientierung wird es wohl kaum eine Eigenschaft unserer Klasse sein. Das Alter ist hingegen eine Größe die schnell errechnet werden kann und daher nicht benötigt wird. Bei Werten die erst durch umfangreiche und rechenintensive Vorgänge ermittelt werden, ist es hingegen meist sinnvoll diese zu mitzusichern.

```
transient private int alter = 18;
transient private int aktuellesJahr;
private int geburtsJahr = 1975;
```

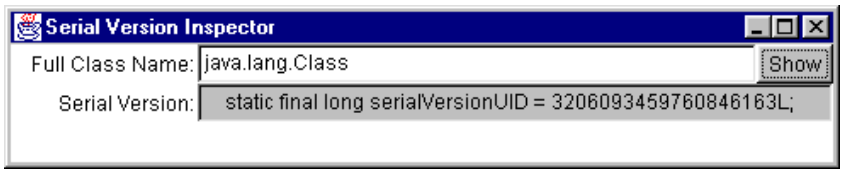
Mit Hilfe von `transient` ist es uns somit möglich die Größe des serialisierten Objektes klein zu halten. Außerdem können wir die Geschwindigkeit der Serialisierung zu erhöhen, da assoziierte Objekte oder auch Eigenschaften nicht gesichert werden. Mit Hilfe dieser Veränderungen können wir die Größe bereits auf 110 Byte senken. Das Schlüsselwort kann jedoch nicht verhindern, dass Daten der Superklassen unseres Objektes gesichert werden. Dies kann jedoch in

[1] Das Beispiel ist natürlich leicht übertrieben, da Sie wohl kaum 10.000.000 Objekte über einen Strom senden wollen. Bedenken Sie jedoch, dass die meisten Objekte wesentlich größer sind und spezielle Anwendungen eine genügend große Anzahl von Objekten erzeugen könnten.

einigen Bereichen durchaus nützlich sein. Das Sichern solcher Objekte, beispielsweise eine benutzerdefinierte Menüleiste sollte daher über einen anderen Mechanismus geschehen.

Das Tool serialver

Mit Hilfe des Tools „`serialver`“ (Serial Version Inspector), welches mit dem JDK ausgeliefert wird, können Sie die Serialversion einer Klasse berechnen. Das direkte Einfügen dieser in serialisierbare Klassen bringt einen weiteren, wenn auch geringen Geschwindigkeitsvorteil. Mit Hilfe des Startparameters ‚-show' können Sie die grafische Oberfläche des Tools starten. Durch Eingabe des qualifizierten Klassennamen erhalten Sie dann die `serialVersionUID`, welche Sie lediglich noch in Ihre Klasse kopieren müssen.



Statt mit der grafischen Oberfläche zu arbeiten können Sie auch direkt den qualifizierten Klassennamen als Startparameter übergeben.

Die Schnittstelle Externalizable

Mit Hilfe dieser Schnittstelle können wir weitere Performancegewinne erzielen. Im Gegensatz zur Serialisierung müssen wir uns selbst um die Sicherung der einzelnen Werte kümmern. Als Gegenleistung erhalten wir jedoch eine größere Flexibilität und kleinere gesicherte Objekte. Im Gegensatz zu `Serializable` werden lediglich der Name der Klasse, deren Meta-Daten und die Objektdaten ohne Meta-Daten gesichert. Durch die Einbindung der Schnittstelle müssen wir zwei Methoden implementieren:

```
public void writeExternal(ObjectOutput oo)
              throws java.io.IOException {
}
  public void readExternal(ObjectInput oi)
              throws java.io.IOException,
                     java.lang.ClassNotFoundException {
}
```

Mit Hilfe dieser Methoden ist es möglich, das Sichern und Wiederherstellen von Objekten gezielt zu steuern. Sie können diese Methoden auch bei der Serialisierung verwenden. Dies hat jedoch einen unschönen Beigeschmack, da Ziel der Implementation dieser Schnittstelle ist, dass Sie sich nicht um die Implementation der Methoden kümmern sollen. Wenn wir unser obiges Beispiel weiter abwandeln, kommen wir etwa zu folgendem Quellcode:

```
public class MyObject2 extends Object implements
Externalizable {
  private int alter = 18;
  private int geburtsJahr = 1975;
```

```
  private int aktuellesJahr;
  private static final String file = „External.ser“;

  public MyObject2(){
  }

  public MyObject2(int geburtsJahr) {
    this.setGeburtsJahr (geburtsJahr);
  }

  public void setAktuellesJahr (int aktuellesJahr){
    this.aktuellesJahr = aktuellesJahr;
  }

  public void setGeburtsJahr (int geburtsJahr){
    this.alter = aktuellesJahr - geburtsJahr;
  }

  public void saveObject(){
    FileOutputStream fos = null;
    try{
      File f = new File(file);
      fos = new FileOutputStream(f);
      ObjectOutputStream oos = new
ObjectOutputStream(fos);
      oos.writeObject(this);
      oos.flush();
    }
    catch (IOException ioe){
      ioe.printStackTrace(System.out);
    }
    finally{
      try{
        fos.close();
      }
      catch (IOException ioe){
        ioe.printStackTrace();
      };
    }
    FileInputStream  fis = null;
    ObjectInputStream ois = null;
    try{
      fis = new FileInputStream(file);
      ois = new ObjectInputStream (fis);
      myObject.readExternal(ois);
    }
    catch (Exception e){
    }
    finally {
      try {
        fis.close();
        ois.close();
      }
      catch (Exception e){
      }
    }
  }

  public void writeExternal(ObjectOutput out) throws
IOException {
    out.writeObject(this);
    out.writeInt(this.geburtsJahr);
  }

  public void readExternal(ObjectInput in)
                    throws IOException,
                           ClassNotFoundException {
    MyObject2 myObject2 = (MyObject2) in.readObject();
    myObject2.setGeburtsJahr(in.readInt());
```

```
    myObject2.setAktuellesJahr(2000);
  }
```

Das Ergebnis dieser Sicherung ist ein Strom von lediglich 84 Byte Länge. Obwohl dies bereits ein beachtliches Ergebnis ist, lässt sich die Länge weiter verringern. Dazu müssen wir etwas tiefer in den Quellcode einsteigen. Wir speichern in dem dargestellten Quellcode neben dem Objekt lediglich noch das `geburtsJahr`. Bei einer Analyse, würde man zu dem Schluss kommen, dass ein Vermerken innerhalb des primitiven Datentyps `short` vollkommend ausreichend ist. Aus Geschwindigkeitsgründen ist es jedoch durchaus sinnvoll den primitiven Typ `int` vorzuziehen. (Dies vergrößert natürlich den benötigten Speicherplatz zu Laufzeit.) Für die Sicherung von Objekten können wir diesen Teil der Analyse jedoch wieder einsetzen, in dem wir nur einen `short` Wert sichern. In großen verteilten Systemen lässt sich somit auch bei vielen Objekten sowohl eine hohe Ausführungsgeschwindigkeit als eine geringe Netzwerklast erreichen. Wir haben mit dieser Veränderung des Quellcode nochmals 2 Byte herausgeholt.

Sicherung als XML

Bei der Sicherung als XML gibt es wenige erweiterte Möglichkeiten, zu den bisherigen Ausführungen. Inwieweit Sie eine performante Umwandlung von und nach XML vornehmen können, hängt u.a. auch von der jeweiligen API ab. Einige Punkte sprechen für XML: XML ist unabhängig vom Betriebssystem und somit die ideale Ergänzung zu Java. XML kann als Unicode gesichert werden. Dadurch ist es möglich sprachunabhängig den Objektinhalt zu verteilen. Außerdem ist XML ein Standart, welcher auch außerhalb von Java eine breite Unterstützung genießt. Es gibt jedoch auch Punkte, welche hinsichtlich der Performance negativ zu bewerten sind. Wenn wir Objekte als XML-Datei sichern wollen, benötigen wir eine weitere API. Dies erhöht sowohl den Speicherbedarf zur Laufzeit als auch die benötigten Ressourcen auf unserem Speichermedium, da wir mehr Klassen mitliefern müssen. Eine Unicode XML Datei ist meist länger als die Umsetzung mit Hilfe der Schnittstellen Serializable und Externalizable. Ein gutes Beispiel ist das Sichern von primitiven Datentypen, da Sie immer mindestens 2 Byte für die Daten und einige weitere Bytes für die Meta-Daten benötigen.

Benutzeroberflächen

Die Benutzeroberfläche ist die verbleibene Schnittstelle nach außen und somit wollen wir diese jetzt näher betrachten. Die BNO ist für mich neben der Funktionalität der Punkt, in welchem eine Anwendung entweder aktzeptiert wird oder gnadenlos deinstalliert. Bei Netzwerkanwendungen kommt auch der rasche Aufbau einer solchen hinzu. Sie haben einen wesentlichen Vorteil bei der Entwicklung von Benutzeroberflächen. Sie können sich bei der Beschleunigung der Benutzeroberfläche, das subjekte Empfinden der Endanwender zu Nutze machen. Mit Hilfe von mehreren Threads können Sie so eine nicht vorhandene Geschwindigkeit vorgaukeln.

Bei dem Begriff BNO denken Sie jetzt bitte nicht nur an das AWT oder die Swing Bbliothek, sondern vergessen Sie nicht unsere Systemadministratoren, welche sich vielfach immer noch mit den guten alten ASCII bzw. ANSI Ausgaben unter DOS zufrieden geben können / müssen / wollen[1].

Ein wichtiger Punkt ist die Frage AWT oder Swing. Das AWT ist wesentlich schneller, da es auf die Ressourcen des Betriebssystem zurückgreifen kann, verbraucht jedoch mehr Ressourcen des Betriebssystems da für jedes AWT Objekt ein Peer Objekt[2] erzeugt wird. Swing benötigt nur in seltenen Fällen diese Peer Objekte. Lediglich die Klassen `JFrame`, `JDialog` und `JWindow` basiseren auf Objekten im Betriebsystem. Dies macht Swing nicht ganz so Speicherhungrig. Im Gegensatz zu AWT können jedoch auch nicht die schnellen Betriebsystemroutinen zum darstellen der grafischen Komponenten verwendet werden, so dass Swing langsamer ist als AWT. Das AWT ist jedoch nicht immer eine Alternative. Ein Baum oder eine Tabelle mit dem AWT zu generieren ist zwar möglich aber nicht mehr erstrebenswert. Da ein Mischen von AWT und Swing fast immer zu Problemen führt bleibt Ihnen dann nur noch der Schritt zu Swing.

Swing zu beschleunigen ist daher eine wirkliche Herausforderung. Eine Möglichkeit ist unter Windows xx der Einsatz einer guten DirektX- und OpenGL-Grafikkarte.

Die Klasse Component

Die Klasse `Component` stellt die Grundfunktionen für das Zeichnen zur Verfügung und ist daher für die Benutzeroberfläche eine der zentralen Klassen. Da auch die Grafikausgabe durch diese Klasse wesentlich optimiert werden kann, werden wir diese im Abschnitt Grafiken nochmals betrachten. Mittels Ihrer Methoden können schon die ersten Performancegewinne erreicht werden.

paint und repaint

Wenn Sie eigene Komponenten erstellen, welche innerhalb der `paint()` Methode umfangreiche Berechnungen ausführt, so können Sie durch Aufrufen der `paint()` Methode der Superklasse eine subjektiver Geschwindigkeitsverbesserung erreichen. Dabei muss der Aufruf möglichst früh erfolgen, so dass Ihre Komponente bereits sichtbar wird. Ihre `paint()` Methode überschreibt nun die bereits gezeichnete Komponente. Dies geht natürlich nur, sofern Ihre Komponente noch Gemeinsamkeit mit der Superklasse hat. Sofern dies jedoch gegeben ist, erfolgt auf langsameren Rechnern / JVM bereits eine Ausgabe, während auf schnelleren dieses Zeichnen optisch nicht oder kaum wahrzunehmen ist. Beachten Sie jedoch, dass die Performance in Bezug auf die Rechenleistung abnimmt, da Ihre Komponente zweimal gezeichnet wird.

---

[1] Es gibt insbesondere im Serverbereich auch JVMs, welche AWT und Swing gar nicht erst unterstützen!

[2] Sie haben somit eine Java Frame und ein Plattformspezifischen (Peer) Frame.

Die Methode `repaint()` kann die Geschwindigkeit Ihrer Anwendung deutlich erhöhen. Diese Methode wird u.a. aufgerufen, wenn Sie Komponenten neu zeichnen. Um eine beschleunigte Grafikausgabe zu ermöglichen, verwenden Sie die überladenen Methoden. Dadurch können Sie das Neuzeichnen auf den Bereich begrenzen, welcher geändert wurde und somit die Geschwindigkeit erhöhen.

Threads nutzen

Auch die Threads und somit von Nebenläufigkeit kann die subjektive Performance deutlich erhöhen. Lassen Sie in einem Low-Priority-Thread Ihre Komponenten bereits erzeugen und sichern sie diese in einem `SoftReference` Cache. Sie können somit den optisch wahrnehmbaren Overhead zum Erzeugen von grafischen Komponenten und darstellen dieser deutlich verringern.

AWT

Das AWT ist die Noch-Standard-Oberfläche von Java. Der wesentliche Vorteil des AWT liegt in der Geschwindigkeit. Ein bisschen will ich jetzt ausholen, damit der Grund für die hohe Geschwindigkeit von AWT und die dazu relativ geringe Geschwindigkeit von Swing deutlich wird.

Das AWT hat nur schwergewichtige Komponenten. Schwergewichtig werden diese bezeichnet, da jedes AWT Objekt im jeweiligen Betriebssystem einen Partner (Peer) hat. Dieses Peer-Objekt ist das eigentliche Objekt. Das Java-Objekt kann als ein Wrapper betrachtet werden. Die eigentlichen Aufgaben, wie das Zeichnen der Komponente wird tatsächlich durch das Peer-Objekt vorgenommen. Auch die Daten eines AWT Objektes (Farbe, Größe, Inhalt) befinden sich tatsächlich im jeweiligen Peer-Objekt. Da es sich um ein Objekt handelt, welches direkt im Betriebssystem liegt ist es wesentlich schneller als ein leichtgewichtiges Java-Objekt. Falls Sie schon einmal eine Tabelle oder eine Baumstruktur mit dem AWT fertigen wollten, haben Sie auch schon ein wesentliches Problem von AWT kennen gelernt. Da für jedes AWT Objekt ein entsprechendes Peer-Objekt vorliegen muss und Java auf möglichst vielen Plattformen laufen soll, fielen derartige Strukturen einfach weg. Ein weiteres Problem ist der Speicherverbrauch und die Erzeugungsgeschwindigkeit von AWT Objekten. Für jedes AWT Objekt muss ein entsprechendes Peer-Objekt angelegt werden und dies benötigt neben der Zeit natürlich auch Ressourcen. Das folgende Beispiel zeigt ein Fenster mit einem Knopf und einem Textfeld.

```
package makejava.performance.awtpeer;

import java.awt.*;

public class Fenster extends Frame{

  public static void main(String[] args) {
    Fenster fenster1 = new Fenster();
    TextField txt    = new TextField("Text");
    Button knopf     = new Button ("Knopf");
    fenster1.add(txt,BorderLayout.WEST);
    fenster1.add(knopf, BorderLayout.EAST);
    fenster1.setSize(100,100);
    fenster1.show();
  }
```

```
}
```

Für jedes Objekt in unserer Anwendung wurde ein Peer-Objekt auf

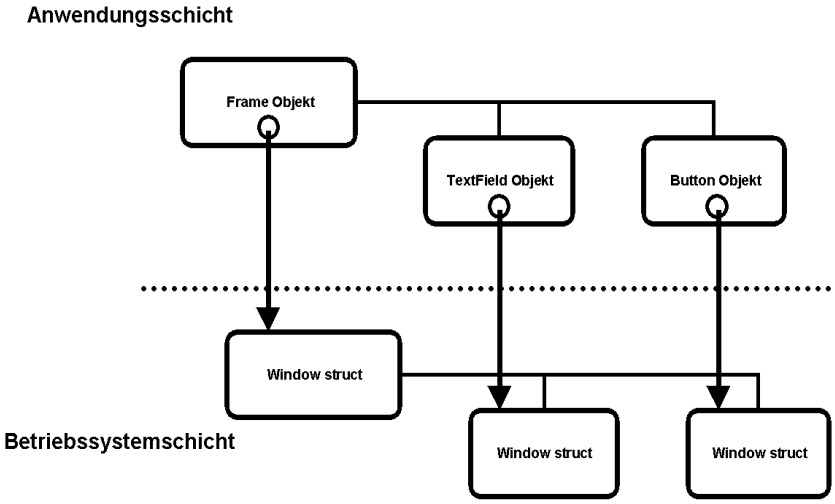


Abbildung 2 - Partner der Schwergewichtigen-BNO-Objekte im Betriebssystem

Betriebssystemebene angelegt. Wenn wir das Fenster minimieren und wieder maximieren so wird jedem dieser Peer-Objekt der Befehl gegeben sich zu zeichnen[1].

Falls Sie nun einem AWT-Objekt, beispielsweise dem `Button` den Befehl erteilen, er solle sich zeichnen, so sagt er dies seinem Partner auf der Betriebssystemebene. Dieser benutzt nun die Routine des Betriebssystems für das Zeichnen eines Button.

Der Geschwindigkeitsvorteil ergibt sich somit insbesondere aus der Tatsache, dass keine Javaroutinen verwendet werden, sondern indirekt auf das Betriebssystem zurückgegriffen wird. Da für jede Ihrer AWT Komponenten jedoch ein Partner im Betriebsystem vorliegen muss, ist der Speicherverbrauch zur Laufzeit jedoch entsprechend hoch.

Die Klasse java.awt.List

Die Klasse `List` des AWT ist zu großen Teilen synchronisiert, so dass der Zugriff auf diese Komponente relativ langsam ist. Insbesondere die Methode `clear()` ist daher nicht zu verwenden. Das Erstellen eines neuen `List` Objektes bringt Ihnen Geschwindigkeitsvorteile 100% bis 500%. Ein JIT Compiler kann dies zwar etwas optimieren, ist jedoch meist immer noch langsamer. Die Synchronisation der Klasse `List` macht diese gerade für das Objektpooling uninteressant.

[1] Da bei leichtgewichtigen Swingkomponenten der Befehl nicht von Betriebssystemobjekten durchgeführt wird, sollten leicht- und schwergewichtige Komponenten nicht gemischt werden

Swing

Wie beim AWT erfolgt hier erst einmal ein kleines Vorgeplänkel, um die Unterschiede in Bezug auf die Performance zu verdeutlichen.

Im Gegensatz zu Swing sind die meisten Swing-Komponenten partnerlos und beziehen sich nicht auf Peer-Objekte. Nur die eigentlichen Fensterklassen (`JFrame`, `JDialog`, `JWindow`) besitzen Partner im Betriebssystem. Die Partnerlosen Komponenten werden leichtgewichtig genannt.

```
package makeSwing.swingpeer;

import javax.swing.*;
import java.awt.*;

public class Fenster extends JFrame{

  public Fenster() {
  }

  public static void main(String[] args) {
    Fenster fenster1 = new Fenster();
    JTextField txt   = new JTextField("Text");
    JButton knopf    = new JButton ("Knopf");

fenster1.getContentPane().add(txt,BorderLayout.WEST);
    fenster1.getContentPane().add(knopf,
BorderLayout.EAST);
    fenster1.setSize(100,100);
    fenster1.show();
  }
}
```

Abgesehen von der Tatsache, dass die Optik sich etwas unterscheidet gleichen sich die Fenster. Die Abbildung auf das Betriebssystem ergibt jedoch ein anderes Bild.

Lediglich das `JFrame` Objekt hat immer noch ein Partnerobjekt im Betriebssystem. Die einzelnen Komponenten auf dem Frame nicht. Dies bedeutet u.a. dass der `JFrame` für das Zeichnen der einzelnen Komponenten zuständig ist. Die Folge ist, dass keinerlei Einschränkungen mehr bzgl der möglichen Komponenten vorliegen und Sie auch eigene Komponenten erstellen können. Außerdem wird nur noch ein Peer-Objekt im Betriebssystem erzeugt. Dies vermindert den Overhead der Objekterstellung. Der Nachteil ist jedoch, dass die Anwendung nun die Komponenten selbst zeichnen muss und nicht auf die schnellen Routinen (zum Zeichnen einer Komponente) des Betriebssystems zurückgreifen kann. Die Anwendung muss nun über Betriebssystemaufrufe einzelne Linien etc. zeichnen.

Auch wenn dies erst einmal keinen großen Vorteil zu bringen scheint, basieren jedoch Komponenten wie `JTable` und `JTree` auf diesem Prinzip und ermöglichen so die Plattformunabhängigkeit dieser. Auch das Look&Feel wäre ohne Swing nicht denkbar.

Sämtliche leichtgewichtigen Komponenten von Swing leiten sich von `JComponent` ab, welche eine direkte Unterklasse von `Component` ist.

Die Vorteile von Swing, u.a. einen geringeren Speicherverbrauch zur Laufzeit, wird mangels Peer Objekten somit mit dem Nachteil an Geschwindigkeit bezahlt. Abgesehen von dieser Tatsache bietet Swing

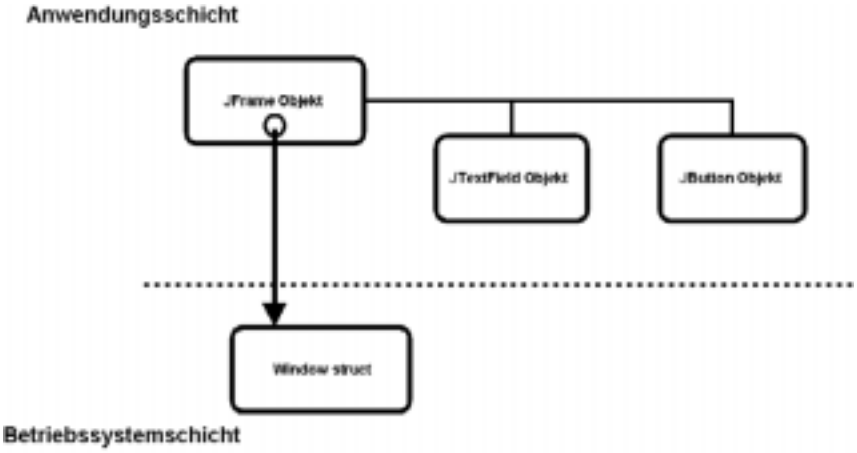


Abbildung 2 - Partner der Swing-Objekte im OS

jedoch die Möglichkeit plattformunabhängige ansprechende BNOen zu erstellen, welche Java bis dahin vermissen ließ.

JTextComponent löschen

Das Löschen einer `JTextComponent`, zu diesen gehören unter anderem das `JTextField` und die `JTextArea`, ist durch die Zuweisung von `null` als Text besonders performant. Im Gegensatz zu der Variante einer leeren Zeichenkette wird sowohl die Erzeugung des `String` Objektes vermieden, als auch die Methodendurchläufe vorzeitig beendet.

Eigene Wrapper nutzen

Besonders bei den Swingkomponenten machen sich die als `final` deklarierten Wrapperklassen der primitiven Datentypen negativ für die Performance bemerkbar. Das Anzeigen von Ganzzahlen wird beispielsweise häufig benötigt[1]. In Ihren Fachobjekten werden Sie diese Daten auch als primitive Datentypen abgelegt haben. Das bedeutet jedoch, sobald eine Komponente diese Daten anzeigen soll, müssen Sie ein `Object` haben. Sie müssen Ihre primitiven Datentypen somit in einen Wrapper verpacken. Hier liegt das Problem der normalen Wrapper. Für jeden Ihrer Datensätze müssen Sie somit ein neues Objekt erzeugen und somit den Overhead in Kauf nehmen. Besser ist es daher einen eigenen Wrapper zu entwickeln. Ein ganz einfacher Wrapper für Ganzzahlen des Typs `int` könnte daher etwa so aussehen.

```
class MyInteger {
  protected int value;
```

```
  public int getValue(){
    return this.value;
  }

  public void setValue(int newValue){
    this.value = newValue;
  }

  public String toString(){
    return Integer.toString(this.value);
  }
}
```

In Ihrer Model Implementation halten Sie dann eine Referenz auf ein Objekt dieses Typs. Die jeweilige Zugriffsmethode (z.B. `getElementAt()` beim `DefaultListModel`) müssen Sie dann lediglich so abändern, dass vor der Rückgabe des Objektes der Wert entsprechend geändert wird. Neben der erhöhten Geschwindigkeit nimmt somit der benötigte Speicherplatz zur Laufzeit erheblich ab.

Eigene Models nutzen

Die Swing-Klassenbibliothek arbeitet nach dem Model-View-Controller Prinzip. Das bedeutet für Sie, dass Ihre Daten nicht mehr in der grafischen Komponente, sondern in einem anderen Objekt, dem Model, gesichert werden. Wenn Sie eine Swing-Komponente erstellen, müssen Sie auch ein Model erstellen. Dies wird in einigen Fällen z.B. `JTextField` oder `JButton` bereits im Hintergrund vorgenommen; in anderen Fällen müssen Sie dies explizit vornehmen.

Die Nutzung eigener Models kann sich dabei positiv auf die Performance Ihrer Anwendung auswirken. Das Implementieren eines eigenen Models für eine `JList` etwa in der Form

```
class KundenJListModel implements ListModel {

  java.util.ArrayList kunden = new
java.util.ArrayList();
  java.util.ArrayList listDataListener
        = new java.util.ArrayList();

  public int getSize() {
    return kunden.size();
  }

  public void addElement(Kunde k){
    this.kunden.add(k);
  }

  public Object getElementAt(int index) {
    return kunden.get(index);
  }

  public void addListDataListener(ListDataListener l) {
    this.listDataListener.add(l);
  }

  public void removeListDataListener(ListDataListener l)
{
```

[1] Einige typische Beispiele sind Postleitzahlen, Bankleitzahl, Kontonummer, Alter, Artikelanzahl, Mitgliednummer oder auch nur das Durchnummerieren von Zeilen einer Komponente.

```
    this.listDataListener.remove(l);
  }
}
```

ist dabei etwa doppelt bis drei mal so schnell, wie die Verwendung des `DefaultListModel`.

Grafiken

Grafiken werden in den verschiedensten Varianten verwendet. U.a. als Teil der BNO oder auch in Spielen. Es gibt zwei wesentliche Arten von Grafiken. Bitmap Grafik ist eine Darstellungsform, wobei hier für jeden darzustellen Punkt die Informationen wie Farbe gesichert werden. Die andere Grafikart ist die Vektorgrafik. Diese kennt End- und Anfangspunkte bzw. Formeln, nach denen die Erstellung der Grafik geschieht sowie Farben oder auch Farbverläufe. Bitmapgrafiken werden hauptsächlich für die Verwaltung von Fotos verwendet. Vectorgrafiken haben den Vorteil, dass Sie wesentlich genauer und beliebig vergrößerbar sind. Schriftarten sind meist eine Art Vectorgrafik. Außerdem ist nochmals zu unterscheiden zwischen statischen Grafiken und nicht statischen Grafiken, den Animationen.

Java unterstützt standardmäßig die Bitmap Grafiktypen GIF und JPEG und seit Java 2 auch PNG. Die richtige Wahl des Grafiktyps ist dabei schon eine erste Möglichkeit den Speicherbedarf und somit ggf. auch die Netzwerklast zu minimieren. Ich bin kein Speziallist was Grafiken betrifft aber einige Eigenheiten sollten Ihnen bekannt sein. GIF Grafiken können maximal 256 Farben (8 Bit) darstellen. Für (hochauflösende) Fotos sind Sie daher ungeeignet. Allerdings können GIF Grafiken einen transparenten Hintergrund haben und auch Animationen enthalten. JPEG Grafiken können 24 Bit (True Color) Bilder verwalten. Die Wahl des richtigen Datenformats kann hier die Größe dieser Dateien verringern. Prüfen Sie auch, ob Sie wirklich animierte GIF Grafiken verwenden wollen. Sollten Sie hier Änderungen vornehmen, können Sie den Speicherverbrauch Ihrer Anwendung zu Laufzeit und auch die Netzwerklast stark vermindern. Besonders das PNG Format ist zu empfehlen. Es verbindet die Vorteile von GIF (z.B. Transparenz) und JPEG (photogeeignet) und unterliegt dabei wie JPEG keinen einschränkenden Rechten.

Graphics und Graphics2D

Wenn Sie zeichnen oder Bilder darstellen, so findet dies stets über eines `Graphics` Kontext statt. Die `Graphics2D` Klasse ist eine Subklasse von `Graphics`. Seit dem JDK 1.2 wird die Klasse `Graphics2D` statt `Graphics` verwendet, so dass Sie grds. diese Verwenden können, sofern Sie mit Java2 arbeiten. Diese beiden Klassen stellen bereits die grundlegenden Methoden zum Zeichnen bereit. Darunter ist auch die Methode `drawPolygon()`.

Die Methode `drawPolygon` ist eine nativ implementierte Methode und sollte daher, aus Geschwindigkeitsgründen den Vorzug vor eigenen Implementation oder mehrere `drawLine()` Aufrufen bekommen.

Die Klasse Component

Die Klasse `Component` stellt die Grundfunktionen für das Zeichnen zur Verfügung.

repaint

Die Methode `repaint()` kann die Geschwindigkeit Ihrer Anwendung deutlich erhöhen. Diese Methode wird u.a. aufgerufen, wenn Sie Komponenten neu zeichnen. Um eine beschleunigte Grafikausgabe zu ermöglichen, verwenden Sie die überladenen Methoden. Dadurch können Sie das Neuzeichnen auf den Bereich begrenzen, welcher geändert wurde und somit die Geschwindigkeit erhöhen. Bei dem Verwenden von Animationen sollten Sie ebenfalls prüfen, ob Sie nicht nur Teile Ihrer Animation erneut zeichnen müssen[1].

BufferedImage

Eine weitere Möglichkeit bietet das Zeichnen im Hintergrund, bevor die Ausgabe auf dem Bildschirm erfolgt, mit Hilfe der Klasse `BufferedImage`. Das Zeichnen von Bildern im Hintergrund können Sie in Verbindung mit Threads nutzen, um optischen einen schnelleren Bildaufbau vorzunehmen. Dieses Vorgehen führt dabei jedoch zu mehr Bytecode und benötigten Arbeitsspeicher zur Laufzeit. Auch das Caching von Bildern sollte man in Betracht ziehen – es ist bestimmt kein Zufall, dass Sun dies in seinem Tutorial zum Caching mittels Softreferenzen aufführt[2].

Icon optimieren

Fast jede grafische Oberfläche bietet u.a. eine Menüleite und Symbolleiste an. Unter Windows und OS/2 wird beispielsweise die Menüzeile am oberen Rand des Anwendungsfensters erwartet. Erster Menüpunkt ist dabei `Datei` mit den Menüpunkten `Neu` [Strg N], `Öffnen` bzw. `Laden` [Strg O], `Speichern` [Strg S], `Drucken` [Strg P] und `Beenden` [Alt F4]. Der zweite Menüpunkt `Bearbeiten` enthält meist die Einträge `Rückgängig`, `Wiederholen`, `Ausschneiden`, `Kopieren`, `Einfügen`, `Löschen` und `Alles Auswählen`. Ganz rechts kommt üblicherweise das Menü `Hilfe` mit `Hilfe` [F1], `Hilfeindex` und `Über das Programm`. Ein typisches Programm könnte demnach etwa so aussehen:

---

[1] Inzwischen werden durch das JDK auch animierte GIF Grafiken unterstützt. Prüfen Sie, inbesondere wegen der Wartung, ob Sie nicht besser mit GIF Grafiken arbeiten. Hier liegt die Wartung außerhalb des Quelltextes.

2 Und wenn Sie in einige Javabücher zu Softreferenzen schauen, finden Sie wahrscheinlich nur eine deutschsprachige Übersetzung – es gibt ja auch keine anderen Beispiele ☹!

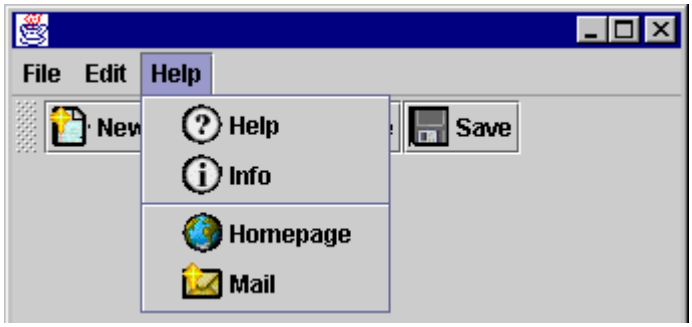


Abbildung 3 - Beispielscreenshot[1]

Um die Icons zu optimieren können Sie diese auch als „Tileset“ sichern. Das einmaligen Laden und Speichern eines solchen Image ermöglicht Ihnen bereits bei dieser kleinen Anwendung Geschwindigkeitsgewinne von etwa 0,2 Sekunden. Bei größeren grafischen Oberflächen konnte ich schon eine Steigerung von über 2 Sekunden erreichen. Der folgende Quellcode zeigt beispielhaft, wie Sie eine entsprechende Klasse umsetzen könnten.

```
class IconRessource extends ImageIcon {
  // Typen der Icons

  public static final int WORLD_1 = 0;
  public static final int WORLD_2 = 1;
  public static final int BLANK = 2;
  public static final int SAVE_ALL = 3;
  public static final int SAVE_AS = 4;
  public static final int SAVE = 5;
  public static final int UNDO = 6;
  public static final int REDO = 7;
  public static final int PRINT = 8;
  public static final int PASTE = 9;
  public static final int OPEN = 10;
  public static final int MAIL = 11;
  public static final int NEW = 12;
  public static final int INFO = 13;
  public static final int HP = 14;
  public static final int HELP = 15;
  public static final int DELETE = 16;
  public static final int CUT = 17;
  public static final int COPY = 18;
  public static final int BOMB = 19;

  private static Image icons;
  int paintFactor;

  //Index des Icons
  private final int index;

  public MyIcons(int index){
    super();
    if (this.icons == null){
        this.icons =
Toolkit.getDefaultToolkit().getImage(

IconRessource.class.getResource("icons.gif"));
```

[1] Die Icons entstammen der Sammlung, welche unter http://www.javalobby.org kostenlos für die Entwicklung von Java-Anwendungen zur Verfügung stehen. Eine weitere kostenlose Iconsammlung finden Sie auf der Sun – Java – Homepage.

```
    }
    new ImageIcon(this.icons).getIconHeight();
    this.index = index;
  }

  public synchronized void paintIcon(Component c,
                                     Graphics g, int x, int
y) {
    //Alle Icons 20x20
    paintFactor = index*20; //sonst: index*Icon_Breite
    g.drawImage(icons,      //Was
                x,
                y,
                x+this.getIconWidth(),
                y+this.getIconHeight(),//Wohin
                paintFactor,
                0,
                paintFactor+20,
                          //sonst:+this.getIconWidth(),
                20,
                          //sonst: this.getIconHeight()
                null);
  }

  public int getIconHeight() {
    //Icons sind immer 20 Pixel hoch
    return 20;
  }

  public int getIconWidth() {
    //Icons sind immer 20 Pixel breit
    return 20;
  }
}
```

Für den optimalen Gebrauch sollten Sie diese Klasse noch um einen Cache-Mechanismus erweitern.

Threads

Die Verwaltung von Threads ist sehr plattformabhängig (nativer Stack). Unabhängig davon bekommt jeder Thread von der JVM einen Java Stack um die Methodenaufrufe, Referenzen und Variablen innerhalb der JVM zu verwalten. Wichtig ist insbesondere jedoch die JVM und deren Verwaltung von Threads[1]. Der Speicherverbrauch ist zwar auf den meisten heutigen Systemen vernachlässigbar, jedoch sollte er bei der Entwicklung von Clients beachtet werden. Threads können sehr gut für subjektive Beschleunigung von Programmen verwandt werden. Prozess- und Benutzerorientierte Programme können zum Beispiel grafische Komponenten bereits in den Speicher laden ohne das dem Anwender dies auffällt. Dies kann mittels Low-Priority-Threads geschehen. Wichtig ist jedoch zu wissen, welche Ressource durch die verschiedenen Aufgaben belastet wird.

[1] Dazu schauen Sie am besten einmal unter „Thread-Umsetzung“. Ja eigentlich wollte ich Verweise vermeiden...

Hauptspeicherzugriff	Prozessor
Dateioperationen	Sekundärspeicher
Socket, Netzwerkverbindungen, RMI	Netzwerk (-last)

Um die richtige Semantik in Ihrer Anwendung zu erhalten werden Sie um Synchronisation von Threads nicht umherkommen können.

Synchronisation von Threads

Synchronisierte Methoden und Programmteile sind nicht gerade performant verschrieen, obwohl mit den HotSpot Compiler eine wesentliche Verbesserung zu spüren ist. Da eine synchronisierte Methode immer nur von einem Thread gleichzeitig aufgerufen werden kann, müssen andere Threads warten bis der synchronisierte Block verlassen wurde. Die Synchronisation wird innerhalb der JVM nach dem Monitor-Prinzip umgesetzt. D.h. derjenige Thread, welcher den Monitor auf ein synchronisiertes Objekt hat, kann synchronisiert Methoden ausführen. Andere Threads müssen warten bis der Monitor-Inhaber diesen freigibt. Wenn Sie eine Synchronisation erzeugen, erstellt die JVM zuerst einen Monitor für diesen Teil. Will ein Thread nun den Programmteil ausführen so muss er zuerst sich beim Monitors anmelden (und ggf. warten). Bei jedem Aufruf / Verlassen eines solchen Programmteils wird der Monitor benachrichtigt, was entsprechende Verzögerungen mit sich bringt. Bei Bytecode Interpretern kann der Geschwindigkeitsverlust den Faktor 100 erreichen. JIT Compiler beschleunigen dieses zwar, sind im Schnitt jedoch immer noch 3-4 mal langsamer als ohne die Synchronisation. Synchronisierte Methoden finden sich in vielen der Standardklassen von Java. Insbesondere die IO Klassen und die Sammlungen, welche bereits in Java1 vorhanden waren haben diesen Nachteil. Suchen Sie hier ggf. nach Alternativen oder entwickeln Sie selbst diese.

Was soll synchronisiert werden

Das Synchronisieren führt stets zum Speeren einzelner Objekte. Synchronisieren können Sie sowohl auf Methodenebene als auch auf Ebene einzelner Blöcke einer Methode. Das Synchronisieren von Methoden ist etwas performanter als bei Blöcken. Gerade bei Methoden mit häufigen Schleifendurchläufen bzw. langen Ausführungszeiten kann die Synchronisierung von einzelnen Blöcken zu einer besseren Performance führen, da Sie somit das Objekt schneller für andere Threads freigeben.

Aufruf synchronisierter Methoden synchronisieren

Der Aufruf synchronisierter Methoden führt meist zu einem erheblichen Geschwindigkeitsverlust. Sofern Sie in einer Multi-Threading-Umgebung mehrere synchronisierte Methoden auf einem Objekt hintereinander aufrufen müssen, können Sie diese durch eine äußere Synchronisation beschleunigen.

```
// ...
synchronized (stringBufferObjekt){
  stringBufferObjekt.append ("%PDF-");
```

```
    stringBufferObjekt.append (pdfVersion);
    stringBufferObjekt.append (pdfObjekt1.getPDFString());
    stringBufferObjekt.append (pdfObjekt2.getPDFString());
  // ...
  }
  // ...
```

Dies ist eine direkte Folge des Monitor-Prinzips. Durch den `synchronized` Block erhält Ihre Methode bereits vor dem Aufruf der ersten synchronisierten Methode auf dem `StringBuffer` Objekt den Monitor. Die folgenden Aufrufe können daher direkt hintereinander ohne Warten aufgerufen werden. Dies erhöht insbesondere die Geschwindigkeit dieses Threads, da dieser zwischen den einzelnen `append` Methodenaufrufen nicht durch einen anderen Thread unterbrochen werden kann. Es wird jedoch auch die gesamte Anwendung beschleunigt, weil die Anzahl der zeitaufwendige Monitorwechsel für Ihre Anwendung sich verringert.

Datenbankzugriff

Java ermöglicht es schnell und einfach Datenbankzugriffe zu implementieren. Neben der eingesetzten Datenbank kommt es für die Ausführungsgeschwindigkeit vor allem auf den Datenbanktreiber und Ihrer Implementierung an. Zu den Datenbanktreibern finden Sie etwas unter ‚Datenbanktreiber' im Teil System und Umgebung. Die Geschwindigkeit der Datenbank kann ebenfalls beeinflusst werden. Hier sollten Sie die Handbücher der Datenbank wälzen und ein gutes Design für Ihre Datenbank erstellen.

Statements

Statements Objekte anzulegen ist eine der häufigsten Aufgaben bei dem Zugriff auf Datenbanken. Leider ist dies eine sehr langsame Angelegenheit und sollte daher von Ihnen optimiert werden. Ein wirkungsvoller Ansatz ist dabei das cachen von Statements bzw. das Verwenden des Typs `PreparedStatement`.

Das Caching eines Statementobjektes verlangt dabei keine komplizierten Klassen oder Überlegungen. Bereits das Sichern der Referenz auf Objektebene mit Prüfung auf null vor dem Zugriff führt hier zu einem enormen Zeitgewinn. Die Größe des Bytecodes wird dabei nicht wesentlich erhöht, jedoch haben Sie einen Mehrbedarf an Arbeitsspeicher zur Laufzeit der nicht unberücksichtigt bleiben sollte. Der Zeitgewinn ist hierbei natürlich abhängig von der Datenbank kann jedoch schnell das dreifache und mehr betragen. Falls Sie der Speicherbedarf abschreckt, ist die Verwendung des `PreparedStatement` die andere Möglichkeit

Wenn viele Zugriffe selber Art nötig sind und somit das selbe Statement Verwendung finden soll, ist es sinnvoll das Sie die Klasse `PreparedStatement` verwenden. Dabei wird die SQL-Anweisung in einer (vor)kompilierten Weise in der Datenbank abgelegt. Dadurch muss die Datenbank nicht bei jedem Aufruf die SQL-Anweisung intern übersetzen. Der Geschwindigkeitsgewinn ist für derartige SQL-Anweisung ist wesentlich höher, als der eines normalen Statement. Sie haben außerdem den Vorteil des geringeren Speicherplatzbedarfs.

Netzwerk

Eine performante Netzwerkanwendung zu erstellen, bedeutet das System und die Umgebung, die Analyse, das Design und die Implementierung zu optimieren. Sie als Programmierer nur der Letzte in der Reihe.

Methodenaufrufe in verteilten Anwendungen

Während bei lokalen (nicht verteilten) Anwendungen der Methodenaufruf keine relevanten Performanceengpässe auslöst (und trotzdem optimiert werden kann), ist bei verteilten Anwendung ein Methodenaufruf anders zu betrachten. Bei der Kommunikation in lokalen Anwendungen besteht die Zeit, in welcher der Aufrufer blockiert wird, asynchronen Aufrufen lediglich aus der Zeit, in welcher die Methoden aufgerufen wird. Bei synchronisierten Aufrufen ist noch die Zeit der Ausführung der Methode hinzuzurechen. Bei verteilten Anwendungen kommen noch weiter Zeitfaktoren hinzu. Unter RMI müssen ggf. noch Objekte serialisiert werden. Der gleiche Effekt ist das Verpacken der Informationen, wie die Signatur und benötigten Daten bzw. Referenzen, sowie zusätzlich das Transformieren dieser in das IIOP-Format unter CORBA. Hinzu kommt ebenfalls die benötigte Zeit für den Transfer über das Netzwerk.

Für die Serialisierung und das Senden von Objekten über Ein-Ausgabeströme können Sie die erwähnten Optimierungsmöglichkeiten bei ‚Sicherung und Wiederherstellung von Objekten' verwenden.

Eine wichtige Optimierungsmöglichkeit bei allen Anwendungen, welche sich jedoch besonders bei verteilten Anwendungen bemerkbar machen, ist die Verwendung von asynchronischen Methodenaufrufen. Ihr Designer hat Ihnen (hoffentlich) dazu schon das entsprechende Framework mit Callback Objekten zur Verfügung gestellt, so dass Sie diese lediglich verwenden müssen. Hierdurch kann sich besonders die gefühlte Geschwindigkeit von Anwendungen erhöhen. Synchrone Methodenaufrufe werden insbesondere dann benötigt, wenn der Aufrufer das Ergebnis der aufgerufenen Methode benötigt. Dies lässt sich in Fällen, wo das Ergebnis für den Anwender sofort sichtbar sein soll und muss leider nicht ändern. Es gibt jedoch auch Nachteile. Bei Methoden deren Ausführungszeit gering ist, ist der Overhead durch Erzeugung von Callbackobjekten und der höheren Anzahl von Methodenaufrufen keine Geschwindigkeitsvorteile. Durch die Erzeugung von Objekten deren einziger Zweck es ist auf die Abarbeitung einer Methode und der Weiterleitung des Ergebnisses zu warten, erhöht sich der physische Speicherverbrauch des Systems, auf welchem das Callbackobjekt liegt. Dies kann durch Änderung des Callbackverfahrens in das Observermuster verringert werden. Dabei fungiert der Server als ein spezielles Callbackobjekt, welches dem Aufrufer nur mitteilt, das sich sein interner Zustand geändert hat. Der Callback Mechanismus führt jedoch fast zwingend zu einer höheren Netzwerklast, da das Clientobjekt sich die benötigten Daten nochmals anfordern muss.

Eine weitere weit verbreitete Möglichkeit die Performance zu erhöhen, ist das Teilen der Ergebnismengen. Dabei werden zuerst die wahrscheinlich gewünschten Ergebnisse geliefert. Benötigt der

Anwender / Aufrufer weitere Ergebnisse so werden diese auf Anfrage nachgeliefert. Dies können Sie insbesondere in Verbindung mit Swing wirksam nutzen. Im Gegensatz zu AWT fordert eine Swing Komponente nur die Elemente, welche Sie darstellen muss, bei ihrem Modell an. Genau diese Anzahl können Sie als Anhaltspunkt für die Splitgröße Ihrer Ergebnismenge berücksichtigen. Ein Vorteil ist, dass während der Zusammenstellung des Ergebnisses bereits der Anwender ein Ergebnis sieht und subjektiv die Anwendung schneller wirkt. Sofern der Anwender bereits durch die ersten übermittelten Daten zufrieden gestellt werden kann, wird die Netzwerklast wesentlich verringert. Dies ist einer der Gründe, weshalb Internetsuchmaschinen dieses Prinzip nutzen. Der Speicherverbrauch beim Aufrufer wird ebenfalls vermindert, da nur die gerade angeforderten Daten im System gehalten werden müssen. Bei der Umsetzung mittels Java sind u.a. noch andere Punkte ausschlaggebend.

Methodenaufrufe können noch weitere Probleme erzeugen. Eine große Anzahl von Methodenaufrufen in verteilten Objekten führt zu einer hohen Netzwerklast. Hierbei kann es z.B. in Verbindung mit CORBA sinnvoll sein diese in einem Methodenaufruf zu binden. Die Verwendung des Fassade Muster mit einer anderen Zielrichtung kann Ihre Anwendung hier beschleunigen.

Mathematik

Statt der Verwendung `Math.abs()`, `Math.min()` und `Math.max()` sollten Sie ggf. eigene Implementierungen vorziehen, welche etwa so aussehen könnten.

```
int abs = (i> 0) ? i : -i;
int min = (a> b) ? b : a;
int max = (a> b) ? a : b;
```

Hier ist wieder die JVM entscheidend. Mit JIT Compiler hat sich die Verwendung der Math.-Methoden als schneller ohne die eigenen Implementierungen erwiesen. (getestet unter Java 1.1.7)

TEIL 8
Implementierung mit Perl

Wer hat schon nur mit Java zu tun? Wie sieht es also bei anderen Programmiersprachen wie eben Perl aus? Gibt es auch hier Möglichkeiten zur Optimierung?

Die Antwort auf diese und weiter Fragen finden Sie nunmehr auch in diesem Dokument. Da sich dieser Teil noch im Aufbau befindet möchte ich Sie bitten Unzulänglichkeiten zu entschuldigen – aber vielleicht wollen Sie ja zur Vollständigkeit beitragen. Wie Sie dies bewerkstelligen können Sie unter „Unterstützung“ nachlesen.

Perlversionen

Die hier dargestellten Tipps wurden mit folgenden Perlversionen getestet:

Perl	OS
5.6.0 MSWin32-x86-multi-thread	Windows NT
5.6.0 MSWin32-x86-multi-thread	Windows 2000
5.005_53	OS2

Eigener Profiler

Auch bei Perl gilt, dass es nützlich ist die Performancelecks zu kennen bevor Sie zu Optimieren anfangen. Damit Sie die Ausführungsgeschwindigkeiten der einzelnen Operationen komfortabel messen können steht Ihnen das Modul `Benchmark.pm` zur Verfügung. Durch dieses Modul wird Ihnen ermöglicht in komfortabler Art und Weise die Ausführungsgeschwindigkeit zu messen.

Nutzung des Benchmark.pm Modul

Für die Nutzung des Moduls `Benchmark.pm` ist zuerst die Einbindung dieses notwendig.

```
use Benchmark;
```

Routine timethis ()

Die einfachste Methode die Ausführungszeit eines Quelltextabschnittes zu messen ist diesen mit Hilfe der Routine `timethis ()` aufzurufen. Der einfachste Aufruf erfolgt durch die Übergabe von zwei Parametern.

Der erste ist die Anzahl der Durchläufe während Sie innerhalb des zweiten den auszuführende Quelltext einbetten.

```
timethis (1000000,
          'my $home="http://www.Bastie.de";
         ');
```

Optional können Sie einen dritten Parameter übergeben, der eine Überschrift für Ihre Benchmark darstellt.

```
timethis (1000000,
          'my $home="http://www.Bastie.de";',
          "Meine Benchmark"
         );
```

Sofern Sie den ersten Parameter negativ wählen wird der zu testende Quelltext mindestens diese Anzahl von Sekunden ausgeführt, bevor der Test beendet wird. Bei der Angabe von 0 wird der Quelltext 3 Sekunden lang ausgeführt.

```
timethis (-3,
          'my $home="http://www.Bastie.de";',
          "Meine 3sek.-Benchmark"
         );
```

Als fünften Parameter können Sie noch die Art der Aufzeichnungen des Benchmarktests angeben. Zur Auswahl stehen dabei die Parameter `'all'`, `'none'`, `'noc'`, `'nop'` oder `'auto'`.

```
timethis (1000000,
          'my $home="http://www.Bastie.de";',
          "Option nop",
          'nop'
         );
```

Routine countit ()

Eine weitere Routine ist `countit ()` mit welcher Sie die Anzahl der Durchläufe in einer bestimmten Zeit. Dabei wird als erster Parameter die Zeit und als zweiter Parameter der auszuführende Quelltext übergeben. Zurückgegeben wird ein Benchmark Objekt. Im Gegensatz zu `timethis ()` wird das Ergebnis nicht sofort ausgegeben. Zu diesem Zweck kann die Routine `timestr ()` verwendet werden.

```
my $count=Benchmark::countit (
        2,
        'my $author="Bastie - Sebastian Ritter";');

print timestr($count);
```

`countit ()` wird durch `use Benchmark;` nicht importiert, so dass Sie dass Paket der Routine mit angeben müssen.

Das obige Beispiel ist somit identisch zu einem einfachen `timethis ()` Aufruf mit einer positiven Zahl als ersten Parameter.

Routine timethese ()

Die Routine `timethese ()` ist als Wrapperroutine zu `timethis ()` zu interpretieren. Wie diese wird als erster Parameter die Anzahl der Durchläufe angegeben. Als zweiten Parameter wird ein `Hash` erwartet.

```
my $hash = {'Addition mit +' =>
                        'my $num=2; $num=$num+$num;' ,
            'Addition mit +=' =>
                        'my $num=2; $num+=$num;'

           };
timethese (1000000, $hash);
```

Direkte Zeitmessung

Das Modul `Benchmark.pl` stellt noch weitere Möglichkeiten der Zeitmessung bereit. Durch den `new ()` Operator können Sie ein Benchmark Objekt erzeugen, welcher üblicherweise die vergangende Zeit in Sekunden seit dem 01.01.1970 enthält. Die Routine `timediff ()` ermöglicht das Berechnen des Zeitunterschiedes zweier Benchmark Objekte, während Sie mit `timesum ()`[1] die Summe ermitteln können. Eine einfache Zeitmessung kann daher wie folgt aussehen:

```
my $counter = 0;
my $timeLauf1 = new Benchmark;
while ($counter < 1000000){
  $counter ++;
}
$timeLauf1 = timediff (new Benchmark, $timeLauf1);
$counter = 0;
my $timeLauf2 = new Benchmark;
while ($counter < 1000000){
  $counter ++;
}
$timeLauf2 = timediff (new Benchmark, $timeLauf2);
print "\n\rZeit 1. Durchlauf\n\r".timestr($timeLauf1);
print "\n\rZeit 2. Durchlauf\n\r".timestr($timeLauf2);
print "\n\rSumme: ".
      timestr (Benchmark::timesum($timeLauf1,
                                  $timeLauf2));
```

Inlining

Das sogenannte Inlining von Quelltext ist eine wirksame Methode um die Ausführungsgeschwindigkeit Ihrer Perl Anwendung zu erhöhen. Dies erhöht zwar die Größe Ihres Skriptes in nicht unerheblicher Weise ermöglicht jedoch enorme Geschwindigkeitssteigerungen.

Die Benchmark der folgenden Anwendung zeigt eine Geschwindigkeitssteigerung von etwa 300%.

[1] Beachten Sie, dass timesum () im Gegensatz zu timediff () nicht in Ihren Namensraum importiert wird.

```
timethis (1000000, '&printIt()', "\n\rSubaufruf");

sub printIt {
  print "";
}

timethis (1000000, 'print "";', "\n\rInlining");
```

Der andere wesentliche Nachteil ist die Verschlechterung der Wartbarkeit Ihrer Anwendung, da Quelltext mit identischen Aufgaben an verschiedenen Stellen stehen.

Operatoren und logische Ausdrücke

Verbunden Zuweisungen

Perl (wie auch Java) versteht verbundene Zuweisungen. Diese werden durch den Interpreter schneller umgesetzt als getrennte, so dass `$i++;` schneller als `$i += 1;` und `$i += $i;` schneller als `$i = $i + $i;` ist. Dies gilt analog für die anderen Rechenoperationen.

Schreibweise der logischen Ausdrücke

Sie können für die logischen Ausdrücke Nicht, Und bzw. Oder zwei verschiedene Schreibweisen verwenden:

Logischer Ausdruck		
Nicht	Not	!
Und	And	&&
Oder	Or	||
Exklusives Oder	Xor	

Die ausgeschriebenen logischen Ausdrücke, die mehrere Ausdrück verbinden werden dabei anders interpretiert. So wird bei den ausgeschriebenen Ausdrücken nur dann der nächste Ausdruck ausgewertet, sofern dies nach Auswertung des aktuellen Ausdrucks noch nötig ist.

```
if (1 = 1 || 2 = 2){}
if (3 = 3 or 4 = 4){}
```

Der vorangegangene Quelltext wird in der || Schreibweise somit zu einer Prüfung beider Ausdrücke führen, bevor der Inhalt des if Blocks ausgeführt wird. Bei der or Schreibweise hingegen wird bereits nach der Prüfung des ersten Ausdrucks in den if Block verzweigt. Für die and bzw && Schreibweise gilt dies analog.

Übrigens waren die Auswirkungen auf dem OS/2 System mit dem Perlinterpreter 5.005_53 wesentlich gravierender als die auf dem Windowssystemen – hat hier jemand mitgedacht?☺

ANHANG I
Vorgehen

Die hier vorgestellten Vorgehensweisen sollten Sie sich entsprechend anpassen. Auch wenn ich mich persönlich mit Performance stark beschäftige so lege ich vielfach, insbesondere bei kleinen Projekten, mein Hauptaugenmerk auf die Funktionalität. Er wenn diese vorliegt beginne ich bei diesen Projekten mit Optimierungsüberlegungen.

vor / bei der Codierung

1. Performante Designentscheidungen
2. Performante Systementscheidungen - Das Externe!
3. Performante Algorithmen - Profiling
4. Low-Level Optimierung
5. Alternative „Native Programmierung“

Dies setzt natürlich Teilweise einen funktionierenden Prototyp voraus. Low-Level Optimierung deutet fast immer auf Fehler in den vorgenannten Punkten hin. Wenn nur noch Low-Level Optimierungen die Anwendung performant machen, wird auch die Native Programmierung nicht mehr viel helfen. Der Vollständigkeit halber ist Sie jedoch erwähnt.

nach der Codierung

1. Externes optimieren ohne Quellcode - Profiling
2. Algorithmusoptimierung - und wieder Profiling
3. Designoptimierung - und wieder Profiling
4. Alternative „Native Programmierung“

Externes optimieren kann u.a. durch Verkleinern der Bytecode-Dateien vorgenommen werden. Hierbei läuft die Anwendung allein dadurch schneller, dass die Klassen schneller geladen werden. Dies macht sich insbesondere bei Applets bemerkbar. Wo eine Low-Level Optimierung ansetzen sollte ist schwer zu entscheiden. Die Schritte sind nicht unbedingt in dieser Reihenfolge vorzunehmen. Die Alternative der Nativen Programmierung wird keine großen Vorteile mehr bringen, wenn die anderen Mittel ausgeschöpft sind.

bei Web-Anwendungen

1. Frühzeitig mit Tests beginnen

2. Messung und Analyse der Performance - Festlegung der Mindestperformance

3. Analyse des Web-Auftritts - Größe und Anzahl der Grafiken, Anzahl der Server und Serverbandbreite, DNS-Zeit

4. Optimierung des Auftritts - Implementation der Web-Anwendung, Aufbau des Web-Auftritts, Bandbreite...

5. Überwachung während des Einsatzes

Die Besonderheit bei einer Performanceoptimierung von Web-Anwendungen ist, dass Sie nicht die Benutzereinstellungen optimieren können, dass Sie nicht nur die Anwendung optimieren müssen und dass nicht nur Sie optimieren müssen. Hier ist insbesondere auch der Netzwerkadministrator gefragt.

ANHANG II
Quelltextmuster

Natürlich sind dies hier nur Kernsysteme, welche entsprechend angepasst werden sollten. Sie können jedoch schon unterschiedliche Ansätze erkennen.

Objektpool

```
// Beispiel 1 – Quelltext für einen Objektpool
// Quelle Java Spektrum 04/1998

public interface ReusableObject{
   public void delete();
}

// Quelle Java Spektrum 4/98

import java.util.Stack;

public class ObjectPool implements reusableObject{
   private static Stack objectPool = new Stack();
   public static ObjectPool newInstance (Class_1 cl_1,
..., Class_n cl_n){
      if (objectPool.empty()){

         return new ObjectPool(cl_1, ..., cl_n);
      }
      ObjectPool object = (ObjectPool) objectPool.pop();
      object.init (cl_1, ..., cl_n);
      return object;
   }

   private ObjectPool (Class_1 cl_1, ..., Class_n cl_n){
      init(cl_1, ..., cl_n);
   }

   private void init (Class_1 cl_1, ..., Class_n cl_n){
      // Initalisiere neues Objekt
   }

   public void delete(){
      objectPool.push(this);
   }

   // sonstige Methoden der Klasse
}
```

Das folgende Beispiel hat den Vorteil, dass mehrere Objektpools für unterschiedliche Datentypen angelegt werden können.

```
// Beispiel 1 – Quelltext für einen Objektpool
// Quelle Java Magazin 06/2000

public final class ObjectPool{
   private Stack pool = new Stack();
```

```
    public ObjectPool(){
       // Pooled Objekte anlegen
    }

    public Object getObject(){
       return pool.pop();
    }

    public void recycleObject(Object o){
       poop.push(o);
    }
}
```

ANHANG III
Small FAQ

Dieser Teil stellt lediglich einige wenige Punkte dar, welche m.E. für den sinnvollen, schnellen und anwenderfreundlichen Umgang mit Java nötig sind. Ein Großteil der Punkte zeigt sekundär andere Möglichkeiten Performanceeigenschaften zu nutzen.

Die Sun JVM scheint nicht mehr als 64 MB Speicher verwenden zu können!

Die JVM von Sun arbeitet standardmäßig mit einer maximalen zugewiesenen Speichergröße von 64 MB. Soll der Sun JVM mehr als 64 MB sofort zugewiesen werden, so müssen beide Werte verändert werden.

Beispiel: Es soll 80 MB der Anwendung „Start“ sofort zugewiesen werden. Ein Speicherverbrauch bis zu 128 MB sei erlaubt.

Lösung: „`java –Xms80m –Xmx128m Start`“

Wie kann ich die Dead Code Optimierung zu Debugzwecken nutzen?

Die Dead Code Optimierung des Compiler bedeutet, dass nicht erreichbarer Quelltext bei der Erzeugung von Java Bytecode nicht berücksichtigt wird. Um dies zu Debugzwecken zu nutzen, deklarieren Sie eine statische Variable vom Typ `boolean`. Diese können Sie mit Hilfe von `if` Verzweigungen nutzen. Setzen Sie den Wert auf `false` (für die Auslieferung) wird weder die `if` Anweisung noch der enthaltende Quelltext in Bytecode übersetzt.

```
public static final boolean debug = true;

//...
if (debug){
  System.out.println („Debugmodus aktiviert“);
}
//...
```

Ich habe nicht so viel Festplattenkapazität. Kann ich eine Version der JRE mit geringerem Platzbedarf betreiben?

Es gibt hier mehrere Möglichkeiten den Platzbedarf einer JRE zu minimieren. Sofern Sie Javaanwendungen betreiben, welche eine JRE mit einer niedrigeren Versionsnummer benötigen, sollten Sie diese installieren. Besonders der Umstieg auf Java 2 kann hier den Platzbedarf in die Höhe treiben.

Evtl. bietet sich auch ein Obsfucator für Ihre eigentliche Anwendung an. Mit diesem kann als Nebeneffekt die Größe des Bytecode deutlich verringert werden.

Ein weiterer Punkt ist die Datei „`classes.zip`“ neu zu packen. In dieser befinden sich die Dateien, welche die JRE zur Ausführung Ihrer Anwendung benötigt. Es handelt es sich jedoch nicht zwingend um ein nicht komprimiertes Archiv. Ein Komprimieren kann auch hier den Platzbedarf senken.

Letztendlich bleibt noch die Möglichkeit ein anderes SDK zu verwenden. Ähnlich wie die Enterprise Edition gibt es auch SDK's und daran angelehnt spezielle JREs für Geräte mit wenig Speicher.

ANHANG IV
Datenbank-Internaas

Oracle

1. Oracle Datenbanken werten die `where` Klausel von rechts nach links aus. Bei Verknüpfen von zwei Tabellen sollten größere Einschränkungen stets zuerst verarbeitet werden. Daher sollte bei Oracle Datenbanken die größerer Einschränkung rechts stehen. Zum Beispiel:

```
select * from Kunde K, Bestellung B
   where Datum '1.1.2000' and '31.1.2000' and Name=
'Anton'
   and K.kdkey = B.kdkey
```

Interbase

1. Interbase wertet die `where` Klausel von links nach rechts aus. Bei Verknüpfen von zwei Tabellen sollten größere Einschränkungen stets zuerst verarbeitet werden. Daher sollte bei Interbase die größerer Einschränkung links stehen. Zum Beispiel:

```
select * from Kunde K, join Bestellung B on
K.kdkey=B.kdkey
   where "Anton" and
   Datum between "1.1.2000" and "31.1.2000 23:59"
```

ANLAGEN
Quellen und Literatur

Die Quellen- und Literaturhinweise- befinden sich in keiner bestimmten Ordnung. Die Reihenfolge ist weder alphabetisch noch gibt Sie den Aufnahmezeitpunkt, Qualität o.ä. an.

[01] Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software, E. Gamma R. Helm, R. Johnson, J. Vlissides, Addison Wesley Verlag 1996, ISBN 3-89319-950-0, (Originalausgabe „Design Patterns“, 1995, ISBN 0-201-63361-2)

[02] Java optimieren – Die Jagd auf Taktzyklen (C. Ullenboom, Version 1.2, Stand 21.07.1999, http://www.java-tutor.com)

[03] Java ist auch eine Insel (C. Ullenboom, Stand 21.07.2000 ff., http://www.java-tutor.com)

[04] Writing Advantage Application – Chapter 8 (Sun Microsystem, Stand 13.10.1999, http://www.sun.com)

[05] Java Newsgroup (verschiedene Autoren, laufend, news://de.comp.lang.java)

[06] Java Performance / SoSe 1998 (unbekannt, 1998, Stand 05.07.2000, http://www.zuggi.de)

[07] JDK 1.x – Quelltext (Sun Microsystem, laufend, http://www.sun.com)

[08] Optimierung von Java-Programmen (Markus Kopp, Java Spektrum 04/1998, Seite 24 ff.)

[09] Performanceoptimierung in Java-Anwendungen (Andreas Holubek, Java Magazin 06/2000, S. 30 ff.)

[10] Tutorial ... – 15. Teil (Ralf Kühnel, Java Spektrum 04/1999, S. 50 ff.)

[11] Maschinenpark (Kay Glahn, Java Magazin 07/2000, S. 44 f)

[12] Java und JDBC – Ran an die Datenbank (Stefan Walter, Java Spektrum 1/1998, S. 50 ff.)

[13] Performance von Web-Applikationen... (Philippe Carlier, Java Spektrum 4/2000, S. 38 ff.)

[14] Performanz und Skalierbarkeit von Corba Applikationen (Oliver Ihns, Objekt-Spektrum 5/2000 S.70ff.)

[15] Mantel der SQL-Geschichte - ... (Andreas Tengicki, Java Magazin 11/2000, S. 28 ff.)

[16] SELFHTML (Stefan Münz, Version 8.0, http://www.teamone.de)

[17] Java – Programmierhandbuch und Referenz für die Java-2-Plattform (Middendorf und Singer, dpunkt Verlag 1999)

[18] Java Language Spezification (Sun Microsystems, http://www.java.sun.com)

[19] Thirty Way to Improve the Performance of Your Java™ Programs (Glen McCluskey, http://www.glenmccl.com/jperf/index.htm)

[20] Artikel auf verschiedenen Internetpräsenzen:

http://www.onjava.com
http://www.java.sun.com
http://java.oreilly.com
http://www.javaworld.com
http://www.javareport.com

[21] Perl Quelltexte

Da sich dieses Dokument nunmehr auf mehr als eine Programmiersprache bezieht beachten Sie bitter, den Aufbau des Dokumentes:
Die ersten Teile beschäftigen sich mit Java und der Objektorientierung, sowie allgemeinen Grundsätzen zur Optimierung von Anwendungen.
Ab Seite 106 finden Sie Informationen zur Optimierung von Perl Anwendungen.

8

80 / 20 Regel.....11

A

Abstract Window Toolkit.....93
Allgemeines..... 5
  Gründe nicht zu optimieren 9
  Gründe zu optimieren..... 9
  Javaversionen..... 5
Analyse
  asynchrone Methodenaufrufe.....32
  Nebenläufigkeit.....31
  Paralellität.....31
Analyse und Performance.....31
Applet.....73
Applets.....29
  Internet Explorer.....29
Array.....65
Ausnahmen.....*Siehe* Exception
AWT .....*Siehe* Abstract Window Toolkit

B

Benutzeroberfläche.....91
Buffer.....*Siehe* Ströme, gepufferte

C

Cache 39, 75, 93
  extern.....75
Callback 32
Cast 62, 64
Collection ..... *Siehe* Sammlung
Compiler
  alternative 23
  javac 22
  native.....20
  startische..... *Siehe* native Compiler
Component 92, 98

D

Datenbank.....26
  Treiber.....26
Datenbanktreiber
  Direkte JDBC Treiber 28
  JDBC ODBC Brücke 26
  Plattformeigene JDBC Treiber.....27
  Typen 26
  Universelle JDBC Treiber 27
Datentyp
  Array 36
  primitive 64
  Primitive und Referenzen 36
  Sichtbarkeit.....46
Datentypen 61
Debug.....23, 114
Design **35**
  Datentypen.....35
  Sammlung.....37
  Vor- und Nachteile.....35
DesignPattern .....*Siehe* Entwurfsmuster

E

Ein- und Ausgabe **77**
EMail 2
Entwurfsmuster.....42
  Fabrik Muster 45
  Fassade.....43
  Fliegengewicht.....42
  Null Pattern.....44
  Process-Entity.....45
  Singleton Muster.....45
Exception 76
  Ströme.....*Siehe* Ströme, Exception
Externalizable.....89

F

Fabrik Muster .....*Siehe* Entwurfsmuster
FAQ 114
Fassade.....*Siehe* Entwurfsmuster
Fehler .....2 *Siehe* Exception
Festplatte
  Kapazität erhöhen 30
Fliegengewicht .....*Siehe* Entwurfsmuster
Fragen 2
Fundstelle.....2

G

Garbage Collection 21
Grafiken
  Datenformat.....25
  Tiles 57

H

Hardware
  DirectX.....20
  Gleitpunktdarstellung.....20
  Grafikkarte.....20
  OpenGL.....20
HashMap.....67
Hashtable.....66
Hot Spot ... *Siehe* Java Virtual Machine, Hot Spot Compiler

I

Implementation
  Datentypen.....61
  Grundwissen Optimierung.....50
  Objektpool 75
  Operatoren.....61
  Variablen 62

Implementierung
Konstante Ausdrücke 55
Quelltextverschiebung 56
unereichbarer Quelltext 56
Implementierung
Algorithmus 54
Cache 75
Datenbank 103
Methoden 69
Netzwerk 104
Optimierung für Anwendungsbereiche 57
plattformübergreifend und nativ 76
Redundanz vermeiden 51
Schleifen 52
Sichtbarkeit von Variablen 62
Thread 101
Vor- und Nachteile 50
Inhalt 2
Inlining 46
instanceof 62
Internet Explorer 29
IO ..... *Siehe* Ein- und Ausgabe

J

jar 24, 29
Java Virtual Machine 16, 60
Gleitpunktdarstellung 19
Hot Spot Compiler 18
Just-In-Time Compiler 17
Profiler 25
stackorientiert 60
Thread 19
Javaprozessor 20
JIT ..... *Siehe* Java Virtual Machine, Just-In-Time Compiler
JTextComponent löschen 96
JVM ..... *Siehe* Java Virtual Machine

K

Klasse
BufferedImage 99
Component 92, 98
DefaultListModel 98
Graphics 98
Graphics2D 98
Icon 57, 99
Integer 64
JTextArea 96
JTextComponent 96
JTextField 96
List 94
ObjectOutputStream 86
PreparedStatement 103
Statement 103
Konstante Klassen
Wrapperklassen 73
Zeichenketten 73
Konstruktor 72
Kontakt 2

L

List
AWT Komponente 94
Literatur 117

M

Memory Access Pattern 62
Methoden
eigene 71
Inlining 69
synchronisierte 71
überladene 70
Model-View-Controller
Vermischung 48
Modul
Benchmark.pm 106
Motivation für dieses Dokument 2
Multiprozessorsystem 19, 31
MVC ..... *Siehe* Model-View-Controller

N

Netzwerk
Callback 40
Design 39
Methodenaufruf 41
NetzwerkNetzwerk
System und Umgebung 28
Null Pattern ..... *Siehe* Entwurfsmuster

O

Objekte
erzeugen 71
klonen 74
sichern*Siehe* Sichern und Wiederherstellen von Objekten
vergleichen 75
wiederverwenden 74
Objektorientiertes Design
Antidesign 48
Objektpool 39, 75, 112
Obsfucator 23, 24
Operatoren 61
Optimierung
Ende 12
Wann 10
Was 11
Wo 11

P

`paint` 92
Perl
Benchmark 106
countit() 107
Inlining 108
Perlversionen 106
timediff() 108
timestr() 107
timesum() 108

timethese() 108
timethis() 106, 107
Plattformunabhägigkeit
vs. Geschwindigkeit 60
PreparedStatement 103
Process-Entity ..... *Siehe* Entwurfsmuster
Profiler
eigener Java~ 59
eigener Perl~ 106
Tools 60

Q

Quellen 117

R

Reader 77
Redundanz 51
`repaint` 93, 99
RMI 84

S

Sammlung
Array 36, 65
ArrayList 67
Design 37
Gegenüberstellung 38
HashMap 67
Hashtable 66
IntHashtable 38, 67
LinkedList 67
Stack 67
synchronisierte 37, 68
Vector 67
Sammlungen 65
Schleifen 52
Zählschleifen 54
Schlüsselwort
strictfp 19, 60
transient 88
Serialisierung 85
Externalizable 89
transient 88
Sichern und Wiederherstellen von Objekten **84**
Serialisierung 85
XML 91
Sichtbarkeit 62
Methoden 69
Singleton Muster ..... *Siehe* Entwurfsmuster
Slotvariablen 61
SoftReference 93
Stack 67
Statement 103
Stream ..... *Siehe* Ströme
Ströme 77
Eigene Puffer 81
Ergebnisse 84
Exception 83
gepufferte 79
Laufzeitfehler 83
ObjectOutputStream 86
Reader 77
synchronisierte 82
Swing 95
eigener Wrapper 96
eigenes Model 97
Icon 99
ListModel 97
Synchronisation 102
Aufruf optimieren 102
System und Umgebung
Datenbanken 26
Hardware 20
Native Übersetzung 20
Netzwerk 28
Speicher 21
Tools 22
Vor- und Nachteile 16

T

Thread 93, 101
Java Virtual Machine 19
synchronisieren 102
Tiles 57
transient 88
Typen
Zugriffsgeschwindigkeit 60

V

Variablen 62
Variablenzugriff
direkt 48
Vector 67
verbundene Zuweisungen 61, 109
Vererbung 47
Vererbungshierarchie 47
Vergleich 53

W

Wrapper 73
eigene 96
Umwandlung der primitiven Datentypen 65

X

XML 91

Z

Zahlensysteme 64
Zeichenketten 64
Zugriffsgeschwindigkeit 60
Memory Access Pattern 62