Michael Pfeiffer

Java Micro Edition

Mobile Anwendungen mit der MIDP 2.0 entwickeln

Galileo Press

Auf einen Blick

Inhalt

Lots of men have become the tools of their tools.
(Viele Menschen sind zum Werkzeug ihrer Werkzeuge geworden.)
– Henry David Thoreau (1817–1862)

2 Die Entwicklungsumgebung

Im Folgenden geht es nun – nach der möglicherweise etwas langwierigen, aber sicher notwendigen Einleitung, die ein paar wichtige Grundlagen geschaffen hat – an das Eingemachte. Bevor Sie jedoch mit dem Programmieren loslegen können, benötigen Sie irgendeine Art von Entwicklungsumgebung, die natürlich auch das J2ME-SDK beinhalten muss. Es soll hier also die Installation beschrieben und in die Benutzung der Entwicklungsumgebung eingeführt werden.

2.1 Das Wireless Toolkit

Die Laufzeit- und Entwicklungsumgebung hört unter der Micro Edition – wie sollte es anders sein – auf einen anderen Namen als unter dem Desktop-Java J2SE, hier heißt sie Wireless Toolkit (kurz: WTK). In Anbetracht der Tatsache, dass sich die Virtual Machine in dieser Umgebung auch KVM nennt, ist dies sicher nur konsequent.

Anders als bei den anderen Java-Varianten existiert hier auch keine echte Trennung zwischen einer Laufzeitumgebung (die, wie Sie oben bereits erfahren haben, in dem Sinne auch gar nicht existieren kann) und einer Entwicklungsumgebung. Für beides existiert nur ein gemeinsames Paket, das man bei Sun frei herunterladen kann. Ob die installierte Umgebung zur Entwicklung taugt oder ob sie nur dazu geeignet ist, bereits fertige Applikationen in einer emulierten Umgebung auszuführen, unterscheidet sich daran, ob bei der Installation ein SDK (Software Development Kit) oder nur eine JRE der J2SE vorgefunden wird.

2.1.1 Download und Installation

Daran lässt sich eine zwingende Vorbedingung für die J2ME-Softwareentwicklung erkennen: Sie benötigen das J2SDK für das aktuelle WTK 2.3 in mindestens der Version 1.4.2. Dieses SDK findet sich an altbekannter Stelle zum freien

Download unter **http://java.sun.com** in Versionen für verschiedene Betriebssysteme. (Die Tatsache, dass mit dem aktuellen Java 1.5 – das jetzt ja eigentlich Java 5 heißt – das SDK auch wieder in ein JDK, ein Java Development Kit, umbenannt wurde, übergehe ich für den Rest dieses Buches einfach schweigend. Das erlaube ich mir für dieses hauptsächlich an Entwickler gerichtete Buch, da das eine rein marketingtechnische Entscheidung war und die Mehrheit der Java-Entwickler nach wie vor »1.5« zählt bzw. unverändert »SDK« statt »JDK« sagt.)

Ist das J2SDK vorhanden, kann das WTK installiert werden. Auch dieses ist direkt bei Sun zu finden, folgen Sie dort einfach den Links zu »J2ME« und »Download«. Auch hier werden Versionen für verschiedene Betriebssysteme angeboten, die dann entsprechend den dort gültigen Konventionen installiert werden müssen. Für Linux und Windows werden die Schritte im Folgenden noch kurz beschrieben.

2.1.2 Installation unter Linux

Kommt als Entwicklungsbetriebssystem Linux zum Einsatz, so erfolgt die Installation mit Hilfe der Konsole. Das J2ME-Package von Sun wird in einem selbstextrahierenden Archiv ausgeliefert, das als Erstes mittels des Kommandos »chmod« ausführbar gemacht werden muss:

```
chmod 755 j2me_wireless_toolkit-2_xx-bin-linuxi386.bin
```

Dieses Kommando setzt das Executable-Flag für Gruppen, Eigentümer und alle anderen User für die heruntergeladene Datei. Der String »xx« ist hierbei ein Platzhalter für das jeweils aktuelle Minor-Release, für das derzeit aktuelle WTK 2.3 würde der Dateiname also *j2me_wireless_toolkit-2_3-bin-linuxi386.bin* lauten.

Wichtig ist hier, dass das WTK im Home-Verzeichnis eines Users also auch mit dessen Rechten installiert wird und dass natürlich ein normales Java-SDK auf diesem System verfügbar sein muss. Der Installationsvorgang selbst wird durch die Eingabe von

```
./j2me_wireless_toolkit-2_xx-bin-linuxi386.bin
```

gestartet. Nach der Eingabe von `yes` als Zustimmung zu den umfangreichen Lizenzbestimmungen – die Sie sich gewiss aufmerksam durchlesen –, sucht das Installationsskript nach vorhandenen SDKs bzw. Java-Installationen. Es empfiehlt sich allerdings, keine der vorgeschlagenen gefundenen Installationen zu verwenden, sondern den Pfad ins *bin*-Verzeichnis des aktuellen SDK selbst anzugeben. Der Grund hierfür ist, dass das WTK-Setup-Skript Java-Executables in */bin* oder */usr/bin* akzeptiert, die eigentlich nur Symlinks zu den *bin*-Verzeichnissen innerhalb einer installierten JRE sind. Abhängig von der verwendeten Distribu-

tion können diese Symlinks Schwierigkeiten machen, weil sich in ihrer Nähe eben nicht der benötigte Rest der VM befindet (Runtime-JAR etc.). Es ist bei der entsprechenden Frage also die Option **Specify a path to a Java interpreter directory** zu wählen und anschließend der Pfad manuell anzugeben. Das kann beispielsweise */usr/java/jdk1.5.0_05/bin/* sein.

Als Installationsverzeichnis für das WTK selbst kann das vorgeschlagene Verzeichnis problemlos verwendet werden, abhängig vom Namen des aktuell angemeldeten Benutzers wäre das zum Beispiel */home/developer/WTK2.xx*.

Wurden all diese Informationen erfolgreich eingegeben, kann die eigentliche Installation mit **0) Begin copying files if you are satisfied with the settings** gestartet werden. Anschließend befindet sich das WTK im Verzeichnis *~/WTK2.xx/*, und die KToolbar kann gestartet werden, sobald in ihr *bin*-Verzeichnis gewechselt wurde:

```
cd WTK2.xx/bin/
./ktoolbar
```

Diese Beschreibung gilt wie oben bereits erwähnt für die 2er-Version des WTK (genauer: für Version 2.3). Zukünftig können sich hier Änderungen ergeben, vom Grundprinzip her sollte die Installationsprozedur allerdings gleich bleiben.

2.1.3 Installation unter Windows

Die Installation unter Windows gestaltet sich vergleichsweise einfach, aber das Prinzip findet sich im Vergleich zwischen Windows und Linux ja eigentlich immer wieder: Unter Windows sind Regelabläufe insgesamt recht einfach und unkompliziert zu handhaben, es wird jedoch immer dann kritisch, wenn Sie spezielle Funktionalitäten benötigen oder wenn (wieder) etwas nicht funktioniert und man gerne ein paar Log-Informationen hätte, die Auskunft über die Ursache der Probleme geben. In dieser Beschreibung des Installationsvorganges soll bzw. kann allerdings erst einmal nicht davon ausgegangen werden, dass etwas nicht klappt, so dass die (unter Windows leider meist vergebliche) Suche nach System-Debuginformationen nicht nötig ist.

Die Chancen für eine problemlose Installation stehen recht gut, da sie mit relativ wenigen Mausklicks erledigt ist. Wenn Sie das J2SDK noch nicht installiert haben, führen Sie zuerst dessen Setup aus. Hier ist die Vorgehensweise eigentlich die übliche: den Lizenzvereinbarungen zustimmen, bestätigen, dass installiert werden soll, eventuell den Installationsort auf der Festplatte ändern und zulassen, dass das ganze Paket installiert wird. Wichtig ist hier lediglich die Stelle bzw. der Pfad im System, an der/in dem das SDK installiert wird, das sollten Sie sich für den folgenden Schritt gut merken. Sofern das J2SDK an die vorgegebene Posi-

tion kommen soll, ist das im Allgemeinen *C:\j2sdkx.y.z* oder *C:\Programme\Java\j2sdkx.y.z*.

Anschließend wird mit dem Setup des WTK in gleicher Weise verfahren, sprich die üblichen Zustimmungs-/Bestätigungs- und Akzeptanzklicks sind zu tätigen. Interessant wird hier aber die Stelle während des Installationsvorganges, an dem ein SDK (oder aber eine JRE) vorgeschlagen wird, für welches das WTK konfiguriert werden soll. Befindet sich auf dem Zielrechner nämlich noch eine andere Java-Installation, so kann es sein, dass das Setup diese vorschlägt statt des eben explizit installierten SDK. In diesem Fall ist diese Angabe unbedingt zu ändern und der Pfad zu der eben installierten Software anzugeben, was mit dem entsprechenden Button problemlos möglich ist.

Hat das alles geklappt, findet sich eine neue Programmgruppe im Startmenü unter **Programme • J2ME Wireless Toolkit**.

2.1.4 Programme des WTK

Nach der erfolgreichen Installation finden Sie verschiedene neue Programme auf Ihrem System vor, die hier kurz vorgestellt werden sollen. Die detaillierte Benutzung wird im Rahmen dieses Buches an passender Stelle erklärt, sprich, sobald diese Tools zur Entwicklung und zum Testen der Beispielapplikationen benötigt werden.

Abbildung 2.1 Das Default Gray Phone – Mobiltelefon mit LCD-Graustufen-Display (Windows)

- **Default Device Selection**: Dieses kleine Utensil erlaubt es, später während der Entwicklung das voreingestellte Device, das durch das WTK emuliert werden soll, einzustellen bzw. zu verändern. Hier können Sie zwischen verschiedenen vorgegebenen Geräteprofilen wählen. Es stehen dabei Mobilgeräte mit Farb- und Monochromdisplay sowie mit und ohne vollständige QWERTY-Tastatur und mit ganz unterschiedlichen Displaygrößen zur Auswahl.
- **KToolbar**: Dieses Programm stellt gewissermaßen die Schaltzentrale für die Entwicklung dar, hier können existierende Projekte geladen und neue erzeugt werden, es ist möglich, ein Projekt zu kompilieren und es in der integrierten Device-Emulation laufen zu lassen, um dort erste Tests auszuführen.
- **OTA Provisioning** (Over the Air Provisioning – etwa: Bereitstellung auf dem Zielgerät): Hinter diesem recht wenig aussagekräftigen Namen verbirgt sich die eigentliche Emulation des Endgerätes, innerhalb derer später Ihre Programme laufen werden. Es sei hier schon so viel verraten, dass sich dieser Emulator direkt aus der KToolbar heraus starten lässt, Sie werden dieses Tool also nur in den seltensten Fällen direkt aufrufen müssen. An dieser Stelle können Sie bereits ein wenig experimentieren: Verändern Sie das Default Device mit dem **Default Device Selection** Utility, und starten Sie **OTA Provisioning** erneut: So lernen Sie die verschiedenen Gerätetypen kennen, die bereits fertig erstellt mit dem WTK mitgeliefert werden.
- **Preferences**: Wie es der Name bereits sagt, finden sich hier umfangreiche Einstellmöglichkeiten. So kann die Netzwerkverbindung konfiguriert, die Emulationsperformance dem gewünschten Zielgerät angepasst werden und einiges mehr. An dieser Stelle ist es auch möglich, neue Gerätetypen anzulegen und zu konfigurieren, die dann emuliert werden und es erlauben, die eigene Software möglichst realitätsnah zu testen. Diese Einstellmöglichkeiten werden wichtig, wenn eine Software tatsächlich einmal veröffentlicht werden soll. Nichts kann ein Projekt mehr in Schwierigkeiten bringen, als wenn man es erst ständig unter unrealistischen Bedingungen testet und dann kurz vor der geplanten Fertigstellung feststellt, dass es auf dem eigentlichen Endgerät viel zu langsam oder vielleicht sogar gar nicht läuft.
- **Run MIDP Application**: Auch hier verrät der Name, was das Programm tut. Es dient dazu, eine kompilierte J2ME-Applikation auszuführen. Auch dieses Utility werden Sie in der Regel eher nicht separat ausführen, da es ebenfalls in die **KToolbar** integriert ist und bequem von dort aus aufgerufen werden kann.
- **Utilities**: Dieses Hilfsprogramm schließlich bietet verschiedene Möglichkeiten, eigene Programme zu debuggen und zu optimieren, hier stehen unter anderem Speicher- und Netzwerkmonitore zur Verfügung, und es gibt die

Möglichkeit, Applikationen zu profilen, sprich, einzelne Programmteile auf Ausführungsgeschwindigkeit und Effizienz hin zu untersuchen.

Abbildung 2.2 Media Control Skin, ein CLDC-Gerät wie bspw. eine Fernbedienung (Linux/KDE)

Im Zusammenhang mit dem WTK war jetzt mehrfach vom Begriff »Emulation« die Rede. Sinngemäß – und ohne dass diese Erklärung jetzt den Anspruch auf eine exakte Definition erhebt – stellt eine Emulation eine vollständige, umfassende und exakte Nachbildung einer bestimmten Umgebung innerhalb einer anderen, zumeist völlig unterschiedlichen Umgebung dar. Somit steht eine Emulation im Gegensatz zu einer Simulation, die eine Umgebung immer nur ähnlich wiedergibt, also nur versucht, dem Original möglichst nahe zu kommen, aber niemals in der Lage sein wird, dieses exakt abzubilden.

Simulationen kommen zumeist bei extrem komplexen Abläufen zum Einsatz, bei denen es nicht möglich ist, alle Eingangsparameter vollständig zu verarbeiten und abzubilden; entweder weil es zu viele oder aber weil nicht alle bekannt sind. Ein gutes Beispiel für eine Simulation ist die Wettervorhersage: Da die Computermodelle, die hier verwendet werden, unmöglich in der Lage sind, jedes Luftmolekül einzeln zu berechnen, wird alles zwangsläufig etwas ungenauer simuliert. Und hier gilt beides: Die Anzahl der Luftmoleküle ist zu hoch, um sie alle berechnen zu können, und es ist nicht möglich, deren Zustand (Bewegungsrich-

tung, Energiegehalt etc.) zu ermitteln, um exakte Anfangsbedingungen für eine Emulation herzustellen.

Anders jedoch bei der Emulation eines Computersystems bzw. einer Hardwareplattform: Deren Parameter und Arbeitsweisen sind in der Regel bis ins letzte Detail bekannt und können deswegen vollständig und exakt nachgebildet werden.

2.1.5 Benutzung des WTK

Die Benutzung des WTK soll hier exemplarisch anhand eines der mitgelieferten Beispielprogramme demonstriert werden. Anschließend geht es dann endlich los, es wird ein erstes eigenes, wenn auch noch einfaches Programm geschrieben und auf einem innerhalb des WTK emulierten Gerät getestet.

Als Erstes starten Sie das **Default Device Selection** Utility und wählen dort nach Gutdünken ein beliebiges Geräteprofil aus, das Sie anschließend verwenden möchten, um diesen ersten Test ablaufen zu lassen.

Ist das erledigt, kommt der zentrale Anlaufpunkt für die weitere Entwicklungstätigkeit zum Zuge: die KToolbar. Wenn Sie sich hier zuerst einen kleinen Überblick über die verschiedenen Menüs verschaffen, finden Sie Menüpunkte wieder, die direkten Zugriff auf die bereits erwähnten Hilfsprogramme des WTK bieten: Unter **File** finden sich die **Utilities**, und unter dem Menüpunkt **Edit** tauchen die **Preferences** wieder auf.

Mit **Open Project...** ist es nun möglich, ein bereits existierendes Projekt auszuwählen und zu öffnen. Passenderweise werden mit dem WTK einige Beispielprojekte mitgeliefert. Diese sind dafür gedacht, verschiedene (fortgeschrittene) Programmiertechniken und Anwendungsfälle der J2ME API zu demonstrieren und zu dokumentieren. Das nötige Wissen, um auch mit den darin gezeigten Techniken klarzukommen, wird Ihnen dieses Buch vermitteln.

Wählen Sie in dem Fenster, das sich nach dem Klick auf **Open Project...** geöffnet hat, einfach eines der Projekte aus, und starten Sie dieses anschließend mit Hilfe des Buttons **Run**. Jetzt öffnet sich ein weiteres Fenster, in dem das zuvor ausgewählte und zu emulierende Endgerät dargestellt ist und in dem das gewählte Projekt zur Ausführung kommt.

Achtung: Die Darstellung des Displays des Mobilgerätes verleitet enorm dazu, einfach darin herumzuklicken. Da es sich hier um die Nachbildung z.B. eines Mobiltelefons handelt, kann das nicht funktionieren, schließlich sind Touchscreens dort (noch) nicht wirklich verbreitet. Die Emulation lässt sich also ausschließlich über die Tastatur des dargestellten Gerätes bedienen, sprich, es kom-

men die Cursortasten (Pfeil nach links/rechts/oben/unten), die Softkeys (die beiden nicht vorbelegten Tasten) sowie natürlich alle Nummern- bzw. Buchstabentasten zum Einsatz – und diese lassen sich entweder per Mausklick oder aber auch direkt mit der Tastatur Ihres Computers bedienen.

Abbildung 2.3 Die Darstellung des Default Color Phones

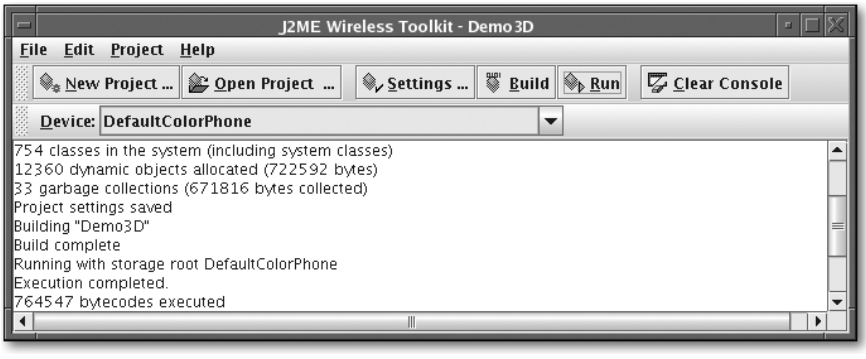

Abbildung 2.4 Die KToolbar nach dem Kompilieren und Ausführen eines MIDlets (Linux/KDE)

Ein Blick auf die Buttons der WTK KToolbar verrät bereits, wie wohl mit eigenen Projekten zu verfahren ist: Sie werden mit **New Project...** neu erzeugt bzw. spä-

ter, wenn sie einmal vorhanden sind, wie eben gehabt mittels **Open Projekt...** geladen. Diese Schritte sollen im Folgenden nicht nur demonstriert, sondern auch praktisch angewandt werden – mit einem ersten eigenen J2ME-Programm. Doch zuvor muss das Thema »SDK« noch mit einem Blick auf Spezielleres abgerundet werden:

2.2 SDKs der Gerätehersteller

Wie im Zusammenhang mit der J2ME/MIDP 1.x bereits erwähnt, gab es zu Zeiten dieser Version recht viele herstellerspezifische Erweiterungen und Schnittstellen, die Funktionalitäten bereitstellten, die der J2ME damals noch fehlten. Das betraf insbesondere die Soundunterstützung. Ähnliches gilt nach wie vor, nur dass die proprietären, herstellerabhängigen SDKs bzw. Programmierschnittstellen derzeit eher spezielle Funktionalitäten für die Endgeräte beinhalten, die meist nicht üblich und deswegen auch auf allen anderen Geräten zu erwarten sind. Das heißt im Klartext: Wenn Sie solche geräte- und herstellerspezifischen Spezialfunktionen verwenden, erhalten Sie nicht portable Applikationen. Sie müssten in diesem Fall also Spezialversionen für alle Geräte entwickeln, die Sie mit Ihrer Applikation unterstützen wollen – ein Aufwand, der sich sicher nur recht selten lohnt.

In den meisten Fällen dürfte es hingegen möglich sein, komplett ohne diese zusätzlichen Möglichkeiten auszukommen, da die Version 2.x der J2ME wirklich schon sehr vollständig ist. Sollten Sie aber doch solche Spezialfunktionen verwenden müssen, kommen Sie um die zusätzliche Software der Hersteller nicht herum. Im Rahmen dieses Buches ist es leider nicht möglich, all diese SDKs detailliert zu beschreiben, da hier – natürlich – jeder Hersteller das Rad neu erfindet.

Interessant werden diese SDKs aber auch in einem anderen Zusammenhang: Sie enthalten zumeist ebenfalls eine Mobilgeräte-Emulation wie das Tool OTA Provisioning des WTK. Der Unterschied dazu ist, dass diese Emulationen die Endgeräte spezifischer darstellen, während das WTK immer nur die Standardschnittstellen bereitstellen kann. Sollten Sie also vorhaben, Ihre Software zu veröffentlichen oder gar zu verkaufen, so empfiehlt es sich mindestens, diese ebenfalls auf allen zur Verfügung stehenden Entwicklungsumgebungen zu testen. Denn obwohl Sun ein wirklich strenges Zertifizierungsverfahren für J2ME-Konformität durchführt, sind kleine Unterschiede und Abweichungen – im Rahmen der Vorgaben – immer möglich. Das liegt nicht nur, aber in erster Linie an den KVMs, die für praktisch jedes Endgerät eine eigene Implementierung darstellen. Und schon kleine Abweichungen im Laufzeitverhalten können die lustigsten Bugs zutage fördern, die unter der WTK-Umgebung unter Umständen nie aufgefallen sind.

Diese Tests auf den Emulationen der Gerätehersteller können natürlich immer nur ein schwacher Ersatz sein für detaillierte Tests auf der realen Hardware selber, denn – Sie werden es sicher schon befürchten – selbst da kann sich die Software wieder ein klein wenig anders verhalten. Das gilt ganz besonders dann, wenn Sie mit Multithreading arbeiten, da die Verwaltung mehrerer Threads eine für das System nicht ganz einfache und stark timingabhängige Angelegenheit ist.

Um das vielleicht noch einmal festzuhalten: Die Entwicklung unter der J2ME kann so kompliziert sein, wie hier eben angedroht, aber in den allermeisten Fällen ist sie es sicher nicht. Man sollte allerdings auf die Möglichkeiten solcher Widrigkeiten vorbereitet sein, denn laut Murphys Gesetz wird einem so etwas – wenn überhaupt – genau dann passieren, wenn man es wirklich am allerwenigsten gebrauchen kann.

2.2.1 Motorola

Motorola bietet verschiedene SDKs für die verschiedenen Gerätetypen an – neben den unterschiedlichen Mobiltelefonen sind schließlich auch PDAs von diesem Hersteller verfügbar. Das jeweils passende SDK ist auf Motorolas Entwicklerplattform unter **http://developer.motorola.com** zu finden, nennt sich **Motorola iDEN SDK**, ist nur für Windows verfügbar und wird dementsprechend mit Hilfe eines Standard-Setups installiert. Anschließend existiert ein neuer Eintrag im Startmenü, über den das SDK gestartet werden kann.

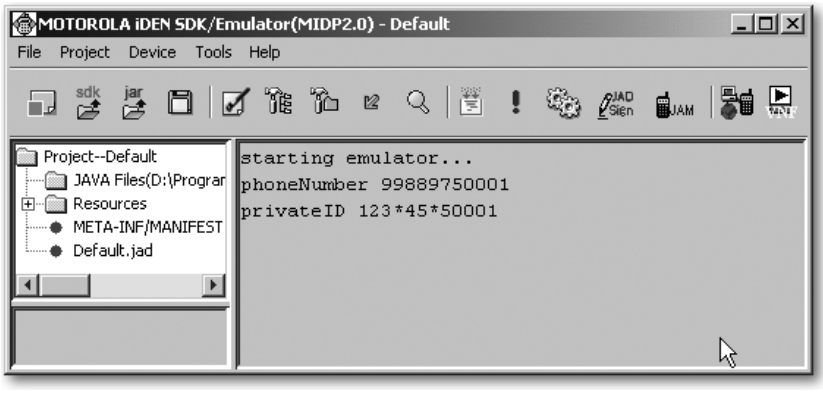

Abbildung 2.5 Motorola iDEN SDK (Windows)

Das iDEN SDK lässt sich sowohl als einfache IDE verwenden als auch zum Testen existierender Applikationen, die in Form von JAR-Archiven vorliegen.

Ähnlich wie in der KToolbar ist es hier möglich, Projekte anzulegen und zu verwalten, diese zu kompilieren, zu JAR-Archiven zusammenzupacken und auf

einer ebenfalls enthaltenen Emulation auszuführen und zu testen. Dabei wird hier immer ein ganz spezieller Endgerätetyp emuliert, weswegen es zuvor auch notwendig war, das richtige SDK für den gewünschten Zielgerätetyp herunterzuladen. Die Bedienung der gesamten Applikation ist ebenfalls genauso einfach und übersichtlich wie die der KToolbar – intuitiv und logisch aufgebaut.

Der Emulator stellt wieder das jeweilige mobile Endgerät dar, komplett mit allen Tasten und Displays – und da es auch Geräte mit mehreren Displays gibt, bei denen sich eines beispielsweise auf dem Deckel befindet, enthält diese Darstellung mehrere Perspektiven.

Das Gerät wird also nicht nur frontal, sondern auch von der Seite (für die dort angebrachten Zusatztasten) sowie auch in geschlossenem Zustand (für das Klappdeckeldisplay) dargestellt und lässt sich entsprechend in all diesen Ansichten bedienen.

2.2.2 Nokia

Nokia unterscheidet bei seinen SDKs nach Plattform-SDKs für bestimmte Geräteserien sowie nach Device-SDKs für ganz bestimmte Gerätetypen, die dann explizit für dieses eine Mobiltelefon geeignet sind. Diese SDKs lassen sich unter **http://www.forum.nokia.com** frei herunterladen. Hier sieht es in Sachen Plattformunabhängigkeit allerdings schlechter aus als bei Motorola, für diese SDKs wird zumeist Windows 2000 oder XP verlangt.

Die Installation geschieht auch hier über ein Setup. Eine Besonderheit ist allerdings, dass man während des Installationsvorgangs einen Forum-Nokia-Benutzernamen und das zugehörige Passwort eingeben muss, um das SDK überhaupt installieren zu können. Schön ist, dass im Setup gleich die Möglichkeit geboten wird, sich dort zu registrieren – nichtsdestotrotz ist das im Prinzip jedoch unnötig.

Allerdings funktioniert die Registrierung nicht immer einwandfrei. Klappt jedoch alles ordnungsgemäß, erhält man einen Bestätigungslink per E-Mail, und erst wenn man diesen angeklickt hat, ist es möglich, den eben erzeugten Benutzernamen sowie das zugehörige Passwort einzugeben. Im nächsten Schritt wird eine Seriennummer an die registrierte E-Mail-Adresse gesandt, die dann ebenfalls während des Setups eingegeben werden muss, um die Installation ausführen zu können.

Hat man schließlich alle Hürden hinter sich gebracht, fordert das Setup zu einem Reboot des Systems auf – diesen Neustart können Sie sich aber getrost sparen. Der anschließend neu zu findende Startmenüeintrag enthält auch einen Emulator

passend zu dem Gerätetyp, für den Sie das SDK heruntergeladen haben. Wenn Sie diesen starten, tut sich auf den ersten Blick recht Seltsames: Das System erkennt ein neues Gerät und installiert die Treiber dafür. Das erklärt sich allerdings später von selbst, dabei handelt es sich um Treiber für die Schnittstelle, über die es möglich ist, mit dem emulierten Endgerät zu kommunizieren.

Eine weitere Komponente des SDK soll hier keinesfalls unerwähnt bleiben, da sie unabhängig vom emulierten Gerät bei praktisch allen SDKs enthalten ist: das **Nokia Connectivity Framework**. Mit diesem Tool ist es möglich, SMS, MMS und Bluetooth-Messages zu transportieren.

Insgesamt ist Nokias SDK-Konzept im Vergleich zu anderen leider ziemlich verbesserungsbedürftig, wer damit arbeiten muss, ist nicht unbedingt zu beneiden.

2.2.3 Siemens/BenQ

Bei Siemens/BenQ hört das Developer Kit auf den Namen MTK (Mobile Toolkit) und verfolgt ein etwas anderes Konzept. Hier benötigen Sie ein Basissetup (das **MTK Core Pack**), das verschiedene allgemeine Komponenten enthält. Bei diesem Core wird nur nach Grundgerätetypen wie den Gerätegenerationen 45/55 und 65/75 unterschieden. Des Weiteren ist dann noch ein plattformspezifisches Plug-in (wie beispielsweise das **SX1 MTK Plug-in**) erforderlich, das die Funktionalitäten für einen bestimmten Gerätetyp hinzufügt.

Zu finden sind die entsprechenden Downloads unter **https://market.benqmobile.com/**, auch hier geht das leider nicht ohne eine Registrierung. Auch Siemens/BenQ scheinen die Existenz anderer Betriebssysteme als Windows zu ignorieren (selbst wenn man sich dank Java darum gar nicht kümmern müsste) und bieten nach erfolgreicher Registrierungs- und Login-Prozedur für den Core nur einen Download im Windows EXE-Format an.

Noch schwieriger kann es beim Download der Plug-ins werden. Ungeachtet der Tatsache, dass mit heutigen Breitbandanschlüssen große Download-Pakete kein Problem sind, wird dieses schon mal in vier ZIP-Archiven angeboten, die erst mit WinZIP wieder zusammengesetzt werden müssen. Zum Glück liegen die meisten der Emulator-Packs aber ebenfalls als komplettes Setup vor, so dass die Installation eines solchen Plug-ins oftmals doch recht einfach vonstatten geht.

Wirklich elegant an der Idee mit den Emulator-Plug-ins ist die Tatsache, dass dem Entwickler auf diese Art eine einheitliche Schaltzentrale zur Verfügung gestellt wird, aus der heraus eine Applikation auf den Emulatoren für die verschiedensten Gerätetypen getestet werden kann. So wird während der Installation eines Plug-ins auch kein neuer Startmenüeintrag erzeugt, vielmehr taucht der Emulator

dann innerhalb des **MTK Control Centers** auf. Dieses Control Center bietet dann die Möglichkeit, Applikationen auf dem Emulator laufen zu lassen und auch die Übertragung von Messages zu simulieren. Das Control Center erinnert dabei ein wenig an Eclipse, es gibt verschiedene so genannte Perspectives für die Emulatoren und die Simulation der Communication Services, zwischen denen hin und her geschaltet werden kann.

Abbildung 2.6 Das MTK Control Center mit CX65-Emulation (Windows)

2.2.4 Palm

Mit Palm soll auch ein Hardwarehersteller erwähnt werden, der keine Mobiltelefone herstellt, sondern in erster Line PDAs, einen weiteren Gerätetyp, für den die J2ME gedacht ist. Hier nennen sich die bereitgestellten Entwicklertools **WME Developer Toolkit** for Palm OS, sie sind ebenfalls erst nach der leider unvermeidlichen Registrierung unter **http://www.palmsource.com/developers/** herunterladbar.

Webadressen im Internet ändern sich oftmals schneller, als man es für möglich halten möchte, und speziell bei Palm dürfte das aufgrund der aktuellen Lage um die ausgelagerte Betriebssystemsparte nicht eben besser sein. Wer sich an die seltsame Politik bei Palm erinnert: Erst wurde die Betriebssystemsparte Palm OS ausgelagert, die dann von einer Firma übernommen wurde, die mit Palm nicht unbedingt freundschaftlich verbunden ist. In der Folge ist Palm nun gezwungen, wieder ein eigenes Betriebssystem aus dem Boden zu stampfen. In der Presseankündigung wurde betont, dass das kommende Palm OS, das wieder von Palm selbst entwickelt wird, jetzt auf Linux basiert.

Aufgrund dieser Verschiebungen kann es mehr als bei den anderen bereits erwähnten Herstellen nötig sein, eine Suchmaschine zu bemühen, um statt der hier angegebenen, möglicherweise nicht mehr aktuellen Download-Links die dann gültigen zu finden.

Zusätzlich zum WME Developer Toolkit, das die Komponenten für die J2ME-Entwicklung zur Verfügung stellt, ist für eine Test- und Emulationsumgebung der Palm OS Simulator erforderlich, der praktisch wieder die Emulation des Zielsystems darstellt. Beide Pakete werden als ZIP-Archiv bereitgestellt, enthalten aber auch wieder ausschließlich native Windows-Applikationen, so dass Entwickler auch hier keine Wahlfreiheit haben.

In diesen ZIP-Archiven ist leider noch nicht einmal ein Setup-Programm enthalten, sondern die reine Dateistruktur der Entwicklungsumgebung. Diese muss vor der Benutzung also an eine passende Stelle entpackt und nötigenfalls manuell mit eventuellen Zusatzpacks zusammengefügt werden.

Schnell arbeitender Verstand ist kein Vorteil, wenn er nicht auch gründlich ist. Die Vollkommenheit einer Uhr besteht nicht darin, schnell, sondern richtig zu gehen.
– Luc de Clapiers, Marquis de Vauvenargues (1715–1747), französischer Philosoph, Moralist und Schriftsteller

6 Effiziente Programmierung

Nachdem die Klassen und Bibliotheksfunktionen der J2ME jetzt größtenteils beschrieben sind – der Sinn der wenigen nicht explizit erwähnten und beschriebenen Klassen erschließt sich Ihnen mit Sicherheit von selbst –, bleibt noch ein wichtiges Thema übrig: Tipps und Tricks für eine effiziente und ressourcenschonende Programmierung.

Das wäre an und für sich nicht nötig, wenn es in der Softwareentwicklung heute allgemein üblich wäre, auf ein Codedesign zu achten, das mit den verfügbaren Ressourcen nicht umgeht, als wären davon immer und überall genügend vorhanden. Leider aber haben die Rechenleistungen, Speicher- und Festplattengrößen moderner Computer eben dieses sorglose Design praktisch zur Regel werden lassen. Von einigen wenigen Ausnahmen abgesehen, ist es sehr häufig so, dass – nicht zuletzt auch aus Kostengründen – lieber schnell etwas zusammengestellt wird, ohne dass dabei großartig optimiert wird. Passend dazu gibt es (in diesem Zusammenhang »leider«) auch Entwicklungsumgebungen, die so etwas noch fördern. Der Gipfel des Ganzen sind Programmiertools, von denen der Hersteller behauptet, damit könne man programmieren ohne überhaupt Programmierkenntnisse zu haben. Da die Codeelemente von Programmiertools, in denen Applikationen grafisch zusammengeklickt werden können, für sehr viele unterschiedliche Anwendungsfälle ausgelegt sein müssen, ist es zwangsläufig nicht möglich, dass dieser Code auf eine bestimmte Anwendung hin optimiert ist. Dass bei dieser Art von Entwicklungstool der Anwender, der ja angeblich kein Programmierer sein muss, allerspätestens dann scheitern wird, wenn Schwierigkeiten auftreten, die eben doch genauere Kenntnisse der Zusammenhänge erfordern, steht dabei auf einen anderen Blatt.

Doch zurück zum Thema Effizienz: Was zu den Anfängen der Heimcomputersysteme in der Softwareentwicklung zwingend notwendig war, weil die Computer

schlichtweg nicht genügend Kapazitäten hatten, um diese auch noch zu verschwenden, ist auch für Zielplattformen der J2ME mehr als nur sinnvoll. Und das aus ganz genau denselben Gründen: Rechenleistung, Arbeits- und nichtflüchtiger Speicher sind im Vergleich zu normalen PCs auf diesen Plattformen im Allgemeinen extrem knapp.

Im Folgenden möchte ich Ihnen deswegen verschiedene Hinweise und Tipps geben, die insgesamt zu kleinerem und schnellerem Code führen. Und ganz nebenbei lässt sich daraus auch etwas für die Programmierung im Allgemeinen lernen, denn auch auf großen Computersystemen kann ein kleines und schnelles Programm keinesfalls schaden: Was dank dieser Tricks auf Embedded-Geräten zügig läuft, dürfte auf schnellen Computersystemen geradezu rasen!

6.1 Bedingungen

Eine mögliche Stolperfalle findet sich an einer scheinbar recht einfachen Stelle: bei bedingten Verzweigungen. Nehmen Sie doch ein ganz einfaches Beispiel: Eine Variable `state` kann fünf verschiedene Zustände `STATE_1` bis `STATE_5` annehmen, die jeweils abgefragt werden und auf die dann anders reagiert wird. Ein möglicher Ansatz dafür könnte so aussehen:

```
if (state==STATE_1)
   {

   }
if (state==STATE_2)
   {

   }
if (state==STATE_3)
   {

   }
if (state==STATE_4)
   {

   }
if (state==STATE_5)
   {

   }
```

So weit, so gut, diese Lösung funktioniert. Je nachdem, welchen Wert `state` gerade hat, wird ein anderer Anweisungsblock ausgeführt, der hier nur durch drei Punkte symbolisiert wird. Aber abgesehen davon, dass hier keine Abfrage auf einen möglicherweise ungültigen Zustand vorgenommen wird, bei dem auf einen Wert `state` reagiert wird, der eben nicht einer der `STATE_x`-Konstanten entspricht, ist der Ablauf alles andere als optimal. Das erschließt sich schnell, wenn man den Code Schritt für Schritt durchgeht.

Für den Fall, dass `state` gleich `STATE_5` ist, passiert Folgendes: Es wird in der ersten `if`-Abfrage überprüft, ob `state==STATE_1` ist. Das ist nicht der Fall, also wird mit der zweiten `if`-Abfrage fortgesetzt. Auch hier ist das Ergebnis des Vergleichs `false`, bei der dritten und vierten ebenso. Erst der fünfte `if`-Block hat eine Bedingung, die erfüllt ist, also wird er ausgeführt. Was aber passiert, wenn `state` gleich `STATE_1` ist? Spannenderweise das Gleiche! Die erste Bedingung ist erfüllt, also wird der Block abgearbeitet. Anschließend trifft der Kontrollfluss des Programms auf die zweite `if`-Abfrage. Und obwohl bereits ein erfolgreicher Vergleich ausgeführt wurde, werden dieser und alle weiteren Vergleiche dennoch durchgeführt – und genau das kostet Rechenzeit. Das mag insgesamt nicht viel sein, handelt es sich aber um eine Codestelle, die sehr oft durchlaufen wird, summiert sich der Verlust stark.

Das Problem läst sich erst einmal recht einfach lösen, indem die `if`-Konstrukte durch `if-else`-Kombinationen ersetzt werden:

```
if (state==STATE_1)
   {

   }
else if (state==STATE_2)
   {

   }
else if (state==STATE_3)
   {

   }
else if (state==STATE_4)
   {

   }
else if (state==STATE_5)
   {

   }
```

Ist jetzt der erste Vergleich erfolgreich, so verhindert die anschließende `else`-Anweisung, dass weitere Vergleiche ausgeführt werden. Das heißt, für die ersten vier möglichen Zustände wurde einiges gewonnen, da mindestens eine unnötige `if`-Abfrage eingespart wurde. Lediglich für den letzten Wert wurde gar nichts gut gemacht, da für diesen nach wie vor alle Vergleiche ausgeführt werden müssen.

Was bei dieser Änderung praktisch nebenbei noch abfällt, ist eine Möglichkeit zur Fehlerbehandlung. Ein hinten angestelltes `else` ohne weitere Bedingung erledigt das recht einfach:

```
else
   {
   new IllegalArgumentException(»Ungültiger Zustand «+state)
   }
```

Hier stellt sich zu Recht die Frage, ob sich das Problem mit dem zuletzt abgefragten Zustand, für den immer noch alle `if`-Blöcke unter Aufwendung der dafür nötigen Rechenzeit durchlaufen werden, nicht auch noch lösen lässt. Der geneigte Leser mag hier folgendes Konstrukt vorschlagen:

```
switch (state)
   {
   case STATE_1
      ...
      break;
   case STATE_2
      ...
      break;
   case STATE_3
      ...
      break;
   case STATE_4
      ...
      break;
   case STATE_5
      ...
      break;
   default:
      new IllegalArgumentException()
      break;
   }
```

Dies scheint – zumindest vordergründig – die Lösung zu sein. Dieses `switch-case`-Konstrukt sorgt dafür, dass immer nur der `case`-Wert angesprungen wird, der dem Wert der `state`-Variablen entspricht oder aber dem `default`-Zweig, der

wieder den Fehlerfall behandelt. Wie Sie meinen vorsichtigen Formulierungen zu dieser Lösung jetzt sicher schon entnehmen können, ist dem wohl leider doch nicht so. Die Antwort darauf ist ein klares »Vielleicht«, wobei ich hier schon vorwegnehmen möchte, dass es tatsächlich die beste Methode ist, die Sie als Programmierer einsetzen können.

Im Prinzip würde diese Variante schon die gewünschten Einsparungen bewirken, leider hängt das aber sehr von der Implementierung innerhalb der jeweiligen KVM ab. Prinzipiell kann sie die `switch-case`-Statements leider genauso behandeln wie das obige Beispiel mit dem `if-else-if`-Konstrukt. Kann, muss aber nicht: Bei der Entwicklung von Embedded Software, die mehrheitlich in C geschrieben wird, ist es allgemein üblich, wenn möglich `switch-case` zu verwenden. Das aus dem einfachen Grund, dass bestimmte Prozessoren so genannte Jumptables kennen, die eben auf diesen Anwendungsfall abgestimmt sind und bei mindestens vier verschiedenen möglichen Zuständen hier einen Geschwindigkeitsvorteil bieten. Zusammen mit dem passenden Compiler ergibt dieser `switch-case`-Block dann also einen wesentlich schnelleren Code.

Das heißt für J2ME-Java-Code, dass es sich eigentlich immer lohnt, bei mindestens vier Zuständen statt `if-else-if` dieses `switch-case` zu verwenden. Läuft Ihre Applikation auf einer KVM, die den möglichen Vorteil nicht zu nutzen weiß, ist es egal, läuft sie jedoch auf einer Plattform, die hier tatsächlich so effektiv arbeitet, wie das eben prinzipiell möglich ist, so haben Sie schon etwas gewonnen. Hier handelt es sich also um einen Versuch, den man auf jeden Fall wagen sollte, da man nicht wirklich etwas verlieren kann, dafür aber Performance gewinnen. Oder um einen berühmten Satz eines nicht weniger berühmten Philosophen ein wenig zu vergewaltigen: Sie haben mit dieser Methode nichts zu verlieren als Performanceprobleme.

6.1.1 Die Klasse IllegalArgumentException

Die im Abschnitt oben benutzte und bisher auch schon mehrfach verwendete und erwähnte Klasse `IllegalArgumentException` möchte ich Ihnen hier unbedingt noch vorstellen. Da diese keine eigenen, sondern nur geerbte Methoden besitzt, lässt sich die gesamte Beschreibung dieser Klasse mit den beiden Konstruktoren abschließen:

- `public IllegalArgumentException()`
- `public IllegalArgumentException(String s)`

Beide Konstruktoren erzeugen ein neues Objekt vom Typ `IllegalArgument Exception`, wobei diesem über den Parameter `s` ein String mitgegeben werden

kann, der einen Fehlertext enthält. Sinnvollerweise sollten da Informationen über den Grund der Exception enthalten sein.

Diese Klasse leitet sich folgendermaßen ab:

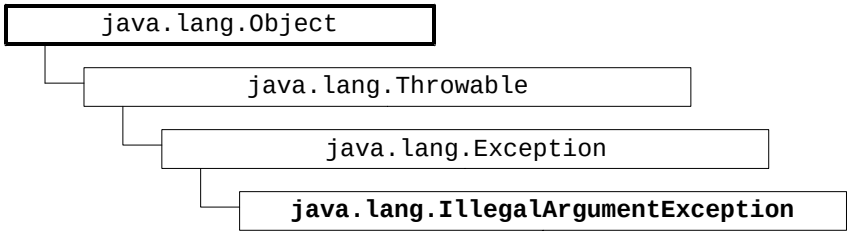

Abbildung 6.1 Vererbungshierarchie der Klasse IllegalArgumentException

6.1.2 Die Klasse Throwable

Die Methoden, die sich in sämtlichen Exception-Objekten wiederfinden, werden von der Klasse `Throwable` geerbt. Da Ihnen diese möglicherweise bereits aus der J2SE bekannt sind, möchte ich mich bei deren Vorstellung etwas kürzer fassen. `Throwable` kennt zwei Konstruktoren

- `public Throwable()`
- `public Throwable(String message)`

die ein neues Objekt dieses Typs erzeugen. Der zweite Konstruktor fügt diesem dabei wieder einen Fehlertext `message` hinzu, der den Grund für die Ausnahme möglichst verständlich beschreiben sollte. Folgende Methoden lassen sich verwenden, um Informationen über dieses Objekt zu erhalten:

- `public String getMessage()`
  Diese Methode liefert die mit diesem Objekt assoziierte Fehler-Message zurück. Hat dieses `Throwable`-Objekt keinen Fehlertext, da es mit dem ersten der oben beschriebenen Konstruktoren erzeugt wurde, so ist der Return-Wert dieser Methode `null`.
- `public String toString()`
  Der von dieser Methode zurückgelieferte String repräsentiert das `Throwable`-Objekt in Form eines Klartextes. Wurde das Objekt dabei mit einer Fehler-Message erzeugt, so setzt sich das Ergebnis aus dem Namen des Objektes, einem Doppelpunkt »:« sowie der Fehler-Message zusammen.
- `public void printStackTrace()`
  Diese Methode ist die für das Debugging wohl wichtigste. Wird sie aufgerufen, so wird der gesamte Stacktrace ausgegeben, anhand dessen sich nachvoll-

ziehen lässt, welche Codeteile und Objekte durchlaufen wurden, bevor die Ausnahme aufgetreten ist.

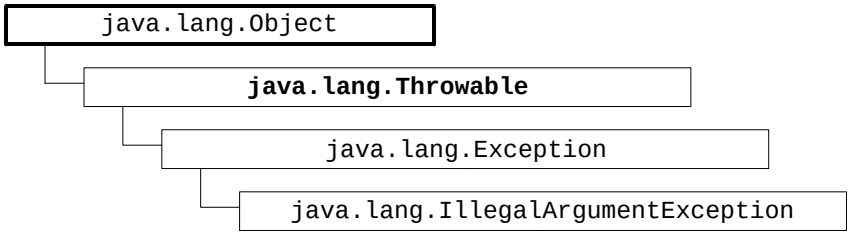

Abbildung 6.2 Vererbungshierarchie der Klasse Throwable

6.2 Exception oder Return-Wert?

Die Problematik, die in diesem Abschnitt beschrieben werden soll, steht bereits ein wenig in Zusammenhang mit dem noch folgenden Thema, weswegen Sie diese nicht gänzlich getrennt betrachten und das hier Gesagte für den kommenden Abschnitt ein wenig im Hinterkopf behalten sollten.

Hier geht es im Wesentlichen um das Thema Fehlerbehandlung. Da Java das Prinzip der Exceptions kennt, verleiten diese dazu, sie für die Behandlung auftretender Fehler zu verwenden. Diese Variante ist zwar durchaus elegant, nur ist sie auch wirklich schnell und effizient im Vergleich zur klassischen Variante mit einem einfachen Return-Wert, der mittels `true` oder `false` oder mit Hilfe eines Fehlercodes über den Erfolg oder Misserfolg einer Operation informieren?

Betrachten Sie das Problem also einfach logisch. Die KVM nutzt – als virtuelle Maschine – ebenfalls eine Einrichtung namens `Stack`. In der Informatik bezeichnet dieser Stack (oder Stapel- bzw. Kellerspeicher) eine Datenstruktur, die auch von den meisten Mikroprozessoren in der Hardware direkt unterstützt wird. Ein solcher Stack ist mit einem Stapel von Dominosteinen vergleichbar. Es kann immer ein neuer Stein oben auf den Stapel gepackt oder einer von oben heruntergenommen werden. Ein Zugriff ist also immer nur auf das oberste Element des Stapels möglich, ein Hinzufügen oder Entfernen eines Steines weiter unten im Stapel nicht. Compiler für moderne Programmiersprachen und virtuelle Maschinen wie die KVM nutzen diese Operationen zum Hinzufügen oder Entfernen eines Datums beispielsweise vor dem Aufruf einer Subroutine, um Parameter an sie zu übergeben. Ähnlich können so auch Ergebnisse der Unterroutine als Return-Wert zurückgegeben werden. Auch werden lokale Variablen auf diesem Stack gespeichert. Diese Operationen, bei denen Daten auf den Stack geschoben oder von dort geholt werden, kosten natürlich immer Rechenzeit – je mehr

Daten für Parameter, lokale Variablen und Return-Werte bewegt werden müssen, umso mehr Rechenzeit wird also für den Aufruf einer Subroutine (oder in Java für den Aufruf einer Methode) benötigt.

Basisdatentypen wie `boolean`, `int` oder `long` belegen in Java genau vier Bytes an Speicherplatz (lediglich doppelt genaue Fließpunktdatentypen wie `double` belegen mit acht Bytes ein wenig mehr Platz). Diese werden bei der Parameterübergabe oder bei der Rückgabe als Methoden-Return-Wert über den KVM-internen Stack geschoben, der Aufwand an Zeit, Speicher und Rechenleistung, um diese wenigen Bytes zu kopieren, ist also relativ gering.

Ein wenig anders sieht es jedoch mit echten Objekten aus. Diese benötigen zusätzlich zu den in ihnen enthaltenen Member-Variablen immer noch ein paar Bytes an Daten mehr. Diese zusätzlichen Datenbytes enthalten Informationen über die Art des Objektes, wovon es sich ableitet und anderes mehr. Es wird also schnell deutlich, dass Exceptions, die nichts anderes sind als Java-Objekte, in jedem Fall schon einmal mehr Ressourcen benötigen, da sie schlichtweg größer sind. Hinzu kommt noch ein anderes Problem: die Typüberprüfung. Bei vielen Operationen, bei denen ein Objekt im Spiel ist, muss von der KVM überprüft werden, ob sie wirklich vom erwarteten Typ ist. Ansonsten würde der Versuch, eine Methode aufzurufen, die in einem Objekt gar nicht existiert, weil eigentlich ein Objekt eines anderen Typs erwartet wurde, recht schnell zu ziemlich großen Problemen, sprich, zu einem Absturz führen.

Und ja, natürlich benötigt man auch die Überprüfung des Typs Rechenleistung.

Daraus jetzt aber zu schließen, dass Exceptions generell zu meiden wären und stattdessen immer mit primitiven Datentypen als Return-Wert gearbeitet werden sollte, ist allerdings auch nicht korrekt. Hier empfiehlt sich vielmehr das Prinzip »Jeder nach seinen Fähigkeiten«. Sprich überall da, wo in zeitkritischen oder sehr oft durchlaufenen Codeteilen eine Fehlerüberprüfung stattfinden muss, sollte der Lösungsansatz mit dem `boolean`-Return-Wert oder den Fehlercodes verwendet werden. An Stellen, an denen ein Fehler nur im Ausnahmefall zu erwarten ist – also beispielsweise bei einem echten Programmfehler oder bei einem kritischen Zustand, der im Regelfall nicht eintritt – sind Exceptions erste Wahl. In diesem Fall bietet eine Exception mit ihrem Stacktrace auch erweiterte Debugging-Möglichkeiten, die sonst eher nicht erforderlich sein sollten.

Da auch ein Beispiel manchmal mehr sagt als tausend Worte, möchte ich Ihnen das noch an einem solchen verdeutlichen. Sie erinnern sich noch an das kleine Brick-Game aus einem der vergangenen Kapitel? Dort gab es mit dem Spielball ein Element, das verschiedene Zustände annehmen konnte. In einer weiter ausgebauten Stufe dieses Spiels kann es passieren, dass auch mehrere Bälle vorkom-

men, deren Zustand abgefragt werden muss (der Ball wird im folgenden Beispiel von der fiktiven Klasse `GameBall` repräsentiert). Da diese Abfrage sehr häufig vorkommt – praktisch nach jeder Positionsänderung –, ist es hier sehr wichtig, auf Geschwindigkeit und Effizienz zu achten, hier ist also die Methode mit den Error-Codes bzw. Return-Codes die sicher zweckmäßigste:

```
static final int BALL_COLL_WALL  =1;
static final int BALL_COLL_BRICK =2;
static final int BALL_COLL_PADDLE=3;
static final int BALL_COLL_NONE  =4;
static final int BALL_COLL_LOST  =5;

...

private int getBallState(GameBall ball)
   {
   ...
   }
```

Es gibt hier fünf mögliche Zustände, die eine Kollision mit einer Wand (`BALL_COLL_WALL`), einem Spielstein (`BALL_COLL_BRICK`) oder mit dem Paddle (`BALL_COLL_PADDLE`) betreffen, den Fall abdecken, dass der Ball frei fliegt, ohne dass er irgendwo kollidiert (`BALL_COLL_NONE`) und dass der Spieler den Ball nicht mehr erwischt hat, so dass er nach unten fällt und verloren geht (`BALL_COLL_LOST`). Die Methode `getBallState()` erwartet dann eines der Ballobjekte als Parameter und liefert dessen Zustand hier mit Hilfe einer Konstante vom Typ `int` zurück.

So eindeutig wie dieser Fall zu sein scheint, so unklar wird es, wenn man den Fall abdecken möchte, dass der Parameter `ball` gleich `null` ist. Hier sollten Sie sich die Frage stellen, wann das passieren kann. Normalerweise sollte das kein Szenario im normalen Programmablauf darstellen, sprich, das würde eher im Fall eines Programmfehlers passieren.

Da bleibt trotzdem die Frage: was nun? Eigentlich handelt es sich ja um einen Zustand, der normalerweise nicht auftritt, hier könnte also eine `Exception` eingesetzt werden. Was aber, wenn dieser Programmfehler dann beim Endkunden auftritt? Ja ich weiß, Software hat eigentlich keine Fehler ... aber im Ernst, es sieht nicht wirklich gut aus, wenn das Spiel ab und zu wegen einer geworfenen `Exception` stehen bleibt oder zumindest spürbare Aussetzer hat. Wie hier zu verfahren ist, hängt vom Design der Applikation ab. So wäre es sinnvoll, diese `Exception` beispielsweise nur in einer Debug- oder Entwicklerversion zu werfen, so dass während der Entwicklung zu sehen ist, dass hier noch etwas schief läuft, und in der Release-Version den Fehler mehr oder weniger schweigend zu übergehen. Diese Entscheidung aber hängt wie bereits erwähnt sehr stark von Ihrem

bisherigen Codedesign und unter Umständen auch von möglicherweise vorgegebenen Programmierrichtlinien ab.

Ein Beispiel, in dem sich die Frage `Exception` oder nicht `Exception` eindeutiger beantworten lässt, ist die ebenfalls einige Abschnitte zuvor vorgestellte Streaming-Applikation. Ein- oder Ausgabedatenströme sind ein geradezu klassischer Anwendungsfall für Exceptions: In dieser Beispielapplikation wurde die Verbindung zu einem Streaming-Server hergestellt, von dem anschließend die abzuspielenden Audiodaten bezogen wurden. Tritt hier der Fall ein, dass diese Daten plötzlich enden, so kann dieser Fehler getrost mit einer `Exception` gemeldet werden, schließlich ist dann nichts mehr zeitkritisch: Es kommen ja sowieso keine Daten mehr, die noch decodiert und wiedergegeben werden könnten, die Applikation stellt an dieser Stelle ihre Arbeit sowieso ein.

6.3 Globale und lokale Variablen

Wie im vorangegangenen Abschnitt bereits angedeutet, existiert eine Einrichtung namens Stack, die bei der Parameterübergabe und bei der Rückgabe von Ergebniswerten einer Methode verwendet wird. Wird dieser Stack mit zu vielen Daten gefüllt – beispielsweise bei rekursiven Methoden, die zu oft aufgerufen werden –, so führt das zu einem Fehler (unter der J2SE gibt das einen `StackOverflowError`, unter der J2ME wird so etwas mit einem `VirtualMachineError` abgehandelt), einem so genannten Stack-Overflow, einem Überlauf des Stapelspeichers.

Sämtliche Operationen auf diesem Stack erfordern Zeit, woraus sich die Überlegung ergeben könnte, diese Zeit einzusparen, indem zum einen keine Parameter mehr übergeben, zum Zweiten methodenlokale Variablen vermieden und zum Dritten auch Return-Werte vermieden werden, indem man dafür ebenfalls eine globale Return-Variable verwendet.

Sollten Sie bereits Erfahrung in der Softwareentwicklung haben, so stellen sich Ihnen an dieser Stelle sicher gerade die Nackenhaare auf – und das zu Recht. Denn prinzipiell ist es zwar richtig, dass sich damit ein wenig Zeit sparen ließe, der große Nachteil, der meiner Meinung nach ein echtes K.-o.-Kriterium ist, besteht hier aber darin, dass der Code dadurch katastrophal unübersichtlich und schwer wartbar wird. Von Kollisionen und Problemen, die auftreten, weil beispielsweise eine globale Zählervariable versehentlich zweimal verwendet wird oder weil es Kollisionen zwischen zwei Threads gibt, die versehentlich die gleiche Variable verwenden, die statt eines Return-Wertes zum Einsatz kommt, ganz zu schweigen.

Meine Empfehlung in dieser Richtung ist also ganz klar: Finger weg davon! Das, was man durch solche Verrenkungen an Zeit gewinnt, verliert man ganz sicher wieder durch die Probleme, die man sich dadurch einhandelt, dass Fehler in einem solchen Code praktisch nicht aufzuspüren sind.

Oder noch schlimmer: Fehler lassen sich nicht eindeutig reproduzieren, da diese von seltsamen Race-Conditions abhängen, die nur schwer nachzuvollziehen sind. Das Ergebnis ist zumeist eine Applikation, die man kaum guten Gewissens veröffentlichen kann.

Aber etwas anderes lässt sich daraus lernen, bzw. an anderer Stelle bieten sich Möglichkeiten zu optimieren. Wie Sie eben gelernt haben, kosten Daten, die methodenlokal angelegt werden, Speicherplatz auf dem Stack und die entsprechende Zeit, die benötigt wird, um diese anzulegen. Das gilt auf alle Fälle für primitive Datentypen wie beispielsweise `boolean`, `byte`, `int` oder `long` (nicht zu verwechseln mit den Klassen `Boolean`, `Byte`, `Integer` oder `Long`!). Was aber ist mit Objekten, die so erzeugt werden?

Rekapitulieren Sie doch die grundlegende Arbeitsweise von Java: Ein solches lokal erzeugtes Objekt `A` könnte beispielsweise an ein anderes Objekt `B` übergeben werden, das eine Referenz auf Objekt `A` speichert. Wird die Methode, in der dieses Objekt `A` erzeugt wurde, wieder verlassen, so werden deren lokale Daten wieder vom Stack entfernt, da sie ja nicht mehr benötigt werden. Also kann bzw. darf ein lokal erzeugtes Objekt nicht auf dem Stack liegen, da es unter Java ja erst dann vom Garbage Collector entfernt wird, wenn keine Referenzen mehr darauf existieren. Und in diesem hier konstruierten Beispiel hat ja noch das Objekt `B` eine Referenz auf dieses so erzeugte Objekt `A`, das weiterhin existieren muss.

Aus Sicht des Stack-Wirkungsprinzips kann man mit methodenlokal erzeugten Objekten also nichts gewinnen oder verlieren. Allerdings kosten Objekte, die in häufig aufgerufenen Methoden massiv erzeugt und anschließend nicht mehr benötigt werden, an anderer Stelle verhältnismäßig viel Zeit: bei der Erzeugung, die – da so ein Objekt immer einigen Overhead zu dessen Verwaltung verursacht – wesentlich aufwändiger ist als bei primitiven Datentypen, und bei der Freigabe der Ressourcen dieses Objektes, wenn es nicht mehr benötigt wird. Das Überwachen der Referenzen, die auf ein Objekt möglicherweise noch existieren, ist für den Garbage Collector durchaus mit einigem an Arbeit verbunden. Das heißt, bei Objekten, die sonst massenhaft erzeugt werden müssten, nur um sie anschließend gleich wieder zu verwerfen, empfiehlt sich hin und wieder die Verwendung von globalen Objekten, die nur einmal bei der Instanziierung der Klasse, zu der sie gehören, erzeugt und erst wieder gelöscht werden, wenn auch die Klasse, deren Member sie sind, entfernt wird.

6.4 Rechenoperationen

Das Thema Rechenoperationen betrifft hier gleich zwei Bereiche. Zum einen gibt es hier Möglichkeiten und Kniffe, den Programmablauf zu optimieren, und zum anderen behandelt es die Problematik um die CLDC-Version 1.0 und 1.1. Wie Sie sich sicher erinnern, kennt die CLDC 1.0 keinerlei Fließpunktdatentypen wie `double` und `float` und dementsprechend auch nicht die Klassen `Double` und `Float`. Das heißt, wenn es nicht möglich ist, auf Version 1.1 zurückzugreifen, sind hier verschiedene Kunstgriffe notwendig, um die Fließpunktfunktionalität nachzubilden bzw. die Notwendigkeit zu umgehen, solche Datentypen zu verwenden.

Die im Folgenden beschriebenen Algorithmen und Methodiken beziehen sich also sowohl auf die eingeschränkten Möglichkeiten der CLDC 1.0 als auch auf Optimierungen im Interesse einer höheren Geschwindigkeit.

6.4.1 Die Klasse Math

Zuvor möchte ich Ihnen mit `Math` die Basisklasse für alle mathematischen Funktionen vorstellen, passend zur CLDC-Problematik getrennt nach den Funktionalitäten, die sie in Version 1.0 und 1.1 bietet.

Da `Math` als `static final` definiert ist, existiert kein Konstruktor, der notwendig wäre, um ein solches Objekt zu erzeugen. Auch sind alle Methoden `static`, so dass auf sie direkt zugegriffen werden kann. Unter der CLDC 1.0 kennt `Math` die folgenden Methoden:

- `public static int min(int a, int b)`
  Es wird derjenige von zwei als Parameter übergebenen `int`-Werten `a` und `b` alsReturn-Wert zurückgeliefert, der kleiner ist, d.h., der näher am Wert `Integer.MIN_VALUE` ist. Wenn beide Werte gleich groß sind, wird der gleiche Wert zurückgegeben.
- `public static long min(long a, long b)`
  Es wird derjenige von zwei als Parameter übergebenen `long`-Werten `a` und `b` als Return-Wert zurückgeliefert, der kleiner ist, d.h., der näher am Wert `Long.MIN_VALUE` ist. Wenn beide Werte gleich groß sind, wird der gleiche Wert zurückgegeben.
- `public static int max(int a, int b)`
  Es wird derjenige von zwei als Parameter übergebenen `int`-Werten `a` und `b` als Return-Wert zurückgeliefert, der größer ist, also der näher am Wert `Integer.MAX_VALUE` ist.

- `public static long max(long a, long b)`
  Es wird der kleinere von zwei als Parameter übergebenen `long`-Werten `a` und `b` als Return-Wert zurückgeliefert, d.h., also derjenige, der näher am Wert `Long.MAX_VALUE` ist.
- `public static int abs(int a)`
- `public static long abs(long a)`
  Diese Methode liefert den Absolutwert des als Parameter übergebenen Wertes `a` zurück. Das heißt, wenn `a` negativ ist, wird der positive Wert dieser Zahl zurückgegeben, ist `a` bereits positiv, so wird der gleiche Wert zurückgegeben.

Ab CLDC Version 1.1 kennt `Math` dann auch die folgenden Konstanten und Methoden:

- `public static final double E`
  Der `double`-Zahlenwert, der e, der Basis des natürlichen Logarithmus, am nächsten kommt.
- `public static final double PI`
  Der `double`-Zahlenwert, der Pi, dem Verhältniszahlenwert zwischen Umfang und Durchmesser eines Kreises, am nächsten kommt.
- `public static double sin(double a)`
  Es wird der Sinuswert des in der Einheit Radians übergebenen Winkels `a` errechnet und zurückgegeben. Ist der Wert dieses Winkels `NaN` (Not a Number – ein für `double` ungültiger Zahlenwert) oder unendlich, so liefert diese Methode auch `NaN` zurück. Ist das Argument positiv oder negativ null, so wird entsprechend auch eine positive oder negative Null zurückgegeben.
- `public static double cos(double a)`
  Es wird der Cosinuswert des in der Einheit Radians übergebenen Winkels `a` errechnet und zurückgegeben. Ist der Wert dieses Winkels `NaN` oder unendlich, so liefert diese Methode auch `NaN` zurück.
- `public static double tan(double a)`
  Es wird der Tangenswert des in der Einheit Radians übergebenen Winkels `a` errechnet und zurückgegeben. Ist der Wert dieses Winkels `NaN` oder unendlich, so liefert diese Methode auch `NaN` zurück. Ist das Argument positiv oder negativ null, so wird entsprechend auch eine positive oder negative Null zurückgegeben.
- `public static double toRadians(double angdeg)`
  Diese Methode konvertiert einen Winkel `angdeg` von der Einheit Grad nach Radians, wie er beispielsweise von den Methoden `sin()`, `cos()` oder `tan()` erwartet wird.

- `public static double toDegrees(double angrad)`
  Diese Methode konvertiert einen Winkel `angrad` von der Einheit Radians nach Grad.
- `public static double sqrt(double a)`
  Es wird die positive Quadratwurzel des als Parameter übergebenen Wertes `a` zurückgegeben. Ist der Wert `a` `NaN` oder kleiner null, so liefert diese Methode auch `NaN` zurück. Ist dieses Argument positiv unendlich, so ist das Ergebnis ebenfalls positiv unendlich. Wird ein Wert übergeben, der positiv oder negativ null ist, so wird auch der gleiche Wert zurückgegeben.
- `public static double ceil(double a)`
  Diese Methode rundet den übergebenen Zahlenwert `a` zum nächsten ganzzahligen Wert auf und liefert diesen gerundeten Wert zurück. Wenn der übergebene Wert `NaN`, unendlich, positiv oder negativ null ist, so wird der gleiche Wert zurückgegeben. Ist das Argument kleiner null, aber größer als –1.0, so ist das zurückgegebene Ergebnis negativ null.
- `public static double floor(double a)`
  Diese Methode rundet den übergebenen Zahlenwert `a` zum nächsten ganzzahligen Wert ab und liefert diesen zurück. Wenn der übergebene Wert `NaN`, unendlich, positiv oder negativ null ist, so wird der gleiche Wert zurückgegeben.
- `public static float abs(float a)`
- `public static double abs(double a)`
  Diese Methode liefert den Absolutwert des als Parameter übergebenen Wertes `a` zurück. Das heißt, wenn `a` negativ ist, wird der positive Wert dieser Zahl zurückgegeben, ist `a` bereits positiv, so wird der gleiche Wert zurückgegeben.
- `public static float max(float a, float b)`
  Es wird derjenige von zwei als Parameter übergebenen `float`-Werten `a` und `b` als Return-Wert zurückgeliefert, der größer ist. Wenn einer der Werte `NaN` ist (also ein für `float` ungültiger Zahlenwert), ist das Ergebnis auch `NaN`.
- `public static double max(double a, double b)`
  Es wird derjenige von zwei als Parameter übergebenen `double`-Werten `a` und `b` als Return-Wert zurückgeliefert, der kleiner ist. Wenn einer der Werte `NaN` ist (also ein für `double` ungültiger Zahlenwert), so ist das Ergebnis auch `NaN`.
- `public static float min(float a, float b)`
  Es wird der kleinere von zwei als Parameter übergebenen `float`-Werten `a` und `b` als Return-Wert zurückgeliefert. Wenn einer dieser beiden Werte `NaN` ist, soist das Ergebnis auch `NaN`.

- `public static double min(double a, double b)`
  Es wird der kleinere von zwei als Parameter übergebenen `double`-Werten `a` und `b` als Return-Wert zurückgegeben. Wenn mindestens einer dieser beiden Werte `NaN` ist, so ist das Ergebnis auch `NaN`.

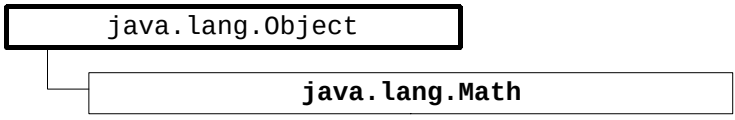

Abbildung 6.3 Vererbungshierarchie der Klasse Math

6.4.2 Grundrechenarten

Optimierungsmöglichkeiten gibt es bei den Grundrechenarten nur für die Multiplikation und die Division. Bei Addition und Subtraktion lässt sich praktisch nichts vereinfachen oder mittels programmtechnischer Tricks beschleunigen. Bis auf den Ratschlag, mögliche Berechnungen und Formeln soweit es geht zu vereinfachen, lässt sich in diesem Zusammenhang eigentlich nichts weiter empfehlen.

Anders jedoch bei Multiplikation und Division. Hier gibt es – abhängig vom verwendeten Faktor oder Divisor – durchaus ein wenig Spielraum. So ist es bei bestimmten Werten möglich, mittels der Operatoren << bzw. >> zu shiften. Ein Linksshift um ein Bit entspricht dabei einer Multiplikation um zwei, ein Rechtsshift ist entsprechend eine Division durch zwei, bei zwei Bits entspricht das dem Wert vier, drei Bits sind acht und so weiter.

Aber was macht diese Berechnung gegenüber einer echten Multiplikation oder Division effektiver? Nun, bei diesen Operationen rechnet der darunter liegende Prozessor auf jeden Fall – und Multiplikationen bzw. Divisionen sind für die Prozessorhardware wirklich aufwändig. Anders hingegen eine Schiebeoperation, wie sie durch einen Shiftoperator bewirkt wird. Diese ist für die Hardware extrem simpel, da dabei eine Schaltung namens Schieberegister zum Einsatz kommt, die diese Aufgabe ausgesprochen schnell erledigen kann.

Der Trick mit dem Shiften lässt sich auch verwenden, um die Rechengenauigkeit zu erhöhen, wenn mit relativ kleinen Zahlenwerten operiert wird, ohne dass Fließpunktvariablen zur Verfügung stehen. Ob diese nun nicht verwendet werden, weil die Software auf einer CLDC 1.0-Plattform läuft oder weil das eine Designentscheidung zugunsten der Geschwindigkeit ist, sei einmal dahingestellt. Auf den zweiten Grund möchte ich noch etwas näher eingehen. Die Fließpunkteinheiten von Mikroprozessoren sind ein recht aufwändiges Stück Hardware. Das heißt zum einen, dass solche Operationen einiges an Rechenzeit benötigen. Zum

anderen bedeutet es, dass diese aufwändigen Operationen den Prozessor auch mehr Strom aufnehmen lassen, was die Akkulaufzeit eines netzunabhängigen Gerätes unter Umständen nennenswert verringert. Und aus genau diesem Grund kommen oftmals Prozessoren zum Einsatz, die diese Fließpunkteinheit gar nicht erst besitzen. Auf solchen Systemen werden Fließpunktrechenoperationen dann per Software emuliert, was ebenfalls alles andere als schnell geht – an zeitkritischen Stellen ist so etwas also praktisch nicht zu verwenden.

Was aber lässt sich hier machen? Spielen Sie die Problematik doch an einer einfachen Rechenoperation durch, die exemplarisch zeigt, wo sich hier Ungenauigkeiten verstecken. Wenn Sie `10.0 * 5.0 / 6.0` mit doppelter Genauigkeit, also unter Verwendung von `double`-Variablen errechnen, so ist das Ergebnis `8.333333333333334`.

Anders ist es, wenn nur Ganzzahloperationen zur Verfügung stehen und Sie keine weitere Maßnahmen ergreifen: `10 * 5 / 6` ergibt hier `8`, da die Nachkommastelle abgeschnitten bzw. nicht erzeugt wird. Das Ergebnis ist also eine Abweichung um den Faktor `1.0416666666666667`. (`8.333333333333334` geteilt durch das jetzt erzielte Ergebnis `8`). Je nachdem, was Sie gerade berechnen, können die Auswirkungen ausgesprochen übel sein.

Anders sieht es jedoch aus, wenn Sie die Geschwindigkeit der Shiftoperation nutzen, um ein wenig Genauigkeit zu gewinnen. Ob dies nun geschieht, weil nur Ganzzahloperationen zur Verfügung stehen oder weil Sie aus Geschwindigkeitsgründen auf Fließpunktoperationen verzichten müssen, ist für dieses Beispiel egal.

Als Erstes werden alle beteiligten Werte – die bei einer realen Anwendung in entsprechenden Variablen stehen würden – um zehn Bit nach links geshiftet, was einem Faktor von `1024` entspricht. Anschließend sieht die Rechnung so aus: `10240 * 5120 / 6144`. Das Ergebnis der Ganzzahlberechnung dieses Terms ist `8533`. Würde dieser Wert durch `1024.0` geteilt werden, so ergäbe das `8,3330078125`, entsprechend einer Abweichung von nur noch `8,3330078125 / 8 = 1,0416259765625`. Das klingt nicht nach viel, aber für Operationen, bei denen eine höhere Genauigkeit erforderlich ist, kann natürlich auch mit größeren Faktoren bzw. Shiftgrößen gearbeitet werden.

Wie Sie sicher bemerkt haben, hat das Ganze einen Haken: Wird das Ergebnis `8533` durch einen Zehn-Bit-Rechtsshift zurückkonvertiert, so ist das Ergebnis wieder `8`, es ist also nichts gewonnen. Das ganze Verfahren funktioniert also tatsächlich nur dann wirklich gut, wenn das Ergebnis einer Operation für weitere Berechnungen verwendet werden kann, ohne dass zurückgeshiftet werden muss. Nötigenfalls wird ein Ergebnis nach der Rechtsshiftoperation ausgegeben, wäh-

rend der Ursprungswert für dieses Ergebnis ohne die Shiftoperation in die nächste Berechnung einfließt.

Ein weiteres Problem findet sich bei den Wertebereichen der verwendbaren Datentypen. Für obiges Beispiel mit dem Zehn-Bit-Shift ist das Ergebnis nach Berechnung des ersten Terms schon `52428800`, also viel zu viel für beispielsweise ein `short`. Wenn um größere Werte geshiftet wird, sind die Bereichsgrenzen eines Datentyps noch schneller erreicht. Das Ganze muss also immer ein Balanceakt zwischen Genauigkeit und verfügbarem Wertebereich sein.

6.4.3 Tangens, Sinus und Co

Ganz interessant wird es bei Funktionen wie Tangens, Sinus, Cosinus und anderen. Wie Sie der Beschreibung der Klasse `Math` oben entnehmen konnten, existieren die passenden Methoden zu deren Berechnung erst seit der CLDC 1.1. Aber auch wenn Ihnen diese zur Verfügung stehen, gibt es genügend Szenarien, bei denen sie aus Geschwindigkeitsgründen schlichtweg nicht benutzbar sind.

In solchen Fällen wird es richtig spannend, da es hier keine Möglichkeit gibt, diese Methoden durch einfachere und vor allem schnellere Rechenoperationen zu ersetzen. Also bleibt nur der Weg, diese vorneweg zu berechnen und die Ergebnisse in Form einer Tabelle zur Verfügung zu stellen. So eine Tabelle könnte für den Sinus beispielsweise so aussehen:

Winkel	Sin()
0	0.0
20	0.34202015
40	0.64278764
60	0.8660254
80	0.9848077
100	0.9848077
120	0.8660254
140	0.64278764
160	0.34202015
180	1.2246469E-16
200	-0.34202015
220	-0.64278764
240	-0.8660254
260	-0.9848077

Tabelle 6.1 Die Sinustabelle

Winkel	Sin()
280	-0.9848077
300	-0.8660254
320	-0.64278764
340	-0.34202015
360	0.0

Tabelle 6.1 Die Sinustabelle (Forts.)

Was hier der Übersichtlichkeit wegen tatsächlich in einer Tabelle dargestellt wurde, würde in einem realen Programm natürlich in beispielsweise zwei Arrays landen, in denen das eine die Winkel enthält und das andere an gleicher Indexposition den zugehörigen Sinuswert. Wie ist nun vorzugehen? An einer Stelle, an welcher der Sinus eines Winkels berechnet werden soll, muss nun derjenige Tabellenwert gesucht werden, der dem gegebenen Winkel am nächsten kommt. Der zugehörige Sinuswert ist dann das Ergebnis. Für dieses Beispiel hier würde das recht große Abweichungen bedeuten, da an dieser Stelle nur mit recht groben 20-Grad-Schritten gearbeitet wird.

Deswegen heißt es hier wieder abwägen zwischen Genauigkeit und Speicherverbrauch. Denn prinzipiell lässt sich so eine Tabelle zwar vergrößern und mit kleineren Abständen zwischen den Werten arbeiten, aber praktisch verbraucht das in kürzester Zeit sehr viel Platz. Eine alternative Variante wäre eine lineare Interpolation zwischen den Werten: Für einen gegebenen Ausgangswert wird errechnet, welchen Abstand dieser von zwei in der Tabelle enthaltenen Werten hat. Diese Abstände können dann – zueinander in Verhältnis gesetzt – verwendet werden, um die Position zu errechnen, die sich in gleichen Abständen zwischen den Ergebniswerten befindet. Das ergibt dann zwar auch einen Wert, der noch mit einer Ungenauigkeit behaftet ist, aber diese ist geringer als bei der ersten Methode.

Der Nachteil dieses Verfahrens ist ganz klar ersichtlich, wenn Sie sich die Methode ansehen, mit der diese lineare Interpolation erfolgen kann. Zum einen werden wieder Fließpunktvariablen benötigt, und zum anderen ist für diese Berechnung wieder einiges an Rechenzeit erforderlich, so dass im Einzelfall immer geprüft und nötigenfalls mit einem kleinen Benchmark ausprobiert werden muss, ob diese Klimmzüge überhaupt etwas bringen:

```
float[] angle=new float[19],sin=new float[19];

float fastSin(float inangle)
   {
```

```
int   i=0;
float f1,diff;

while (angle[i]<inangle) i++;
diff=angle[i]-angle[i-1];
f1=(inangle-angle[i-1])/diff;

diff=sin[i]-sin[i-1];
return sin[i-1]+diff*f1;
}
```

Die beiden `float`-Arrays `angle` und `sin`, die gleich zu Anfang definiert werden, müssen in einem hier nicht dargestellten Initialisierungsvorgang mit den Werten entsprechend obiger Tabelle gefüllt werden.

Die Methode `fastSin()` selbst tut dann exakt das, was ich oben kurz angedeutet habe: Zuerst wird ermittelt, an welcher Position bzw. zwischen welchen Positionen in der Tabelle sich der gesuchte Wert befindet, was mit der `while`-Schleife geschieht. Anschließend ist klar, das sich der Wert des Eingangswinkels `inangle` entweder direkt an der Position `i-1` oder aber zwischen `i-1` und `i` befindet. Anschließend wird die Differenz zwischen diesen beiden Positionen in der Tabelle ermittelt. Das ließe sich für dieses Beispiel zwar durch den Wert 20 ersetzen, allerdings ist es sinnvoll, diesen Wert dynamisch zu ermitteln. Dann sind nämlich Tabellen mit nichtlinearen Schritten zwischen den verschiedenen Indexwerten möglich, in denen in bestimmten Bereichen mehr Werte vorhanden sind, um dort die Genauigkeit noch einmal zu erhöhen. In anderen Bereichen, für die das weniger wichtig ist oder die seltener benötigt werden, können die Abstände dann wieder entsprechend größer werden.

Anschließend wird das Verhältnis errechnet, in dem `inangle` zwischen den beiden Tabellenwinkeln steht, und in `f1` gespeichert. Dieser Faktor `f1` wird dann gleich anschließend herangezogen, um einen Sinuswert zu ermitteln, der in gleichem Verhältnis zum Abstand der beiden Werte `sin[i-1]` und `sin[i]` steht – und mit der Rückgabe dieses Wertes ist die gesamte Interpolation bereits erledigt.

Zwei Dinge sind Ihnen sicher aufgefallen: Zum einen ist die hier vorgestellte Methode noch etwas begrenzt. Für Winkel kleiner `0` oder größer bzw. gleich `360` funktioniert sie nicht, sondern wirft eine `ArrayIndexOutOfBoundsException`. Das liegt daran, dass diese Größen noch nicht abgefangen werden – was beim Sinus aber einfach ist, indem man den Eingangswert einfach durch Addition bzw. Subtraktion von ganzzahligen Vielfachen von 360 innerhalb des gültigen Bereiches bringt.

Der andere Punkt ist eher ein Feature denn ein Problem: Die Tabelle enthält jeweils Wertepaare aus Winkel und zugehörigem Sinuswert. Die Umrechnung zwischen beiden muss natürlich keine Einbahnstraße sein, was hier exemplarisch für die Sinusberechnung gezeigt wurde, lässt sich genauso gut für deren Umkehrfunktion erreichen: Dazu müssen bei der Ermittlung des Ergebnisses lediglich die beiden Seiten der Tabelle getauscht werden.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z