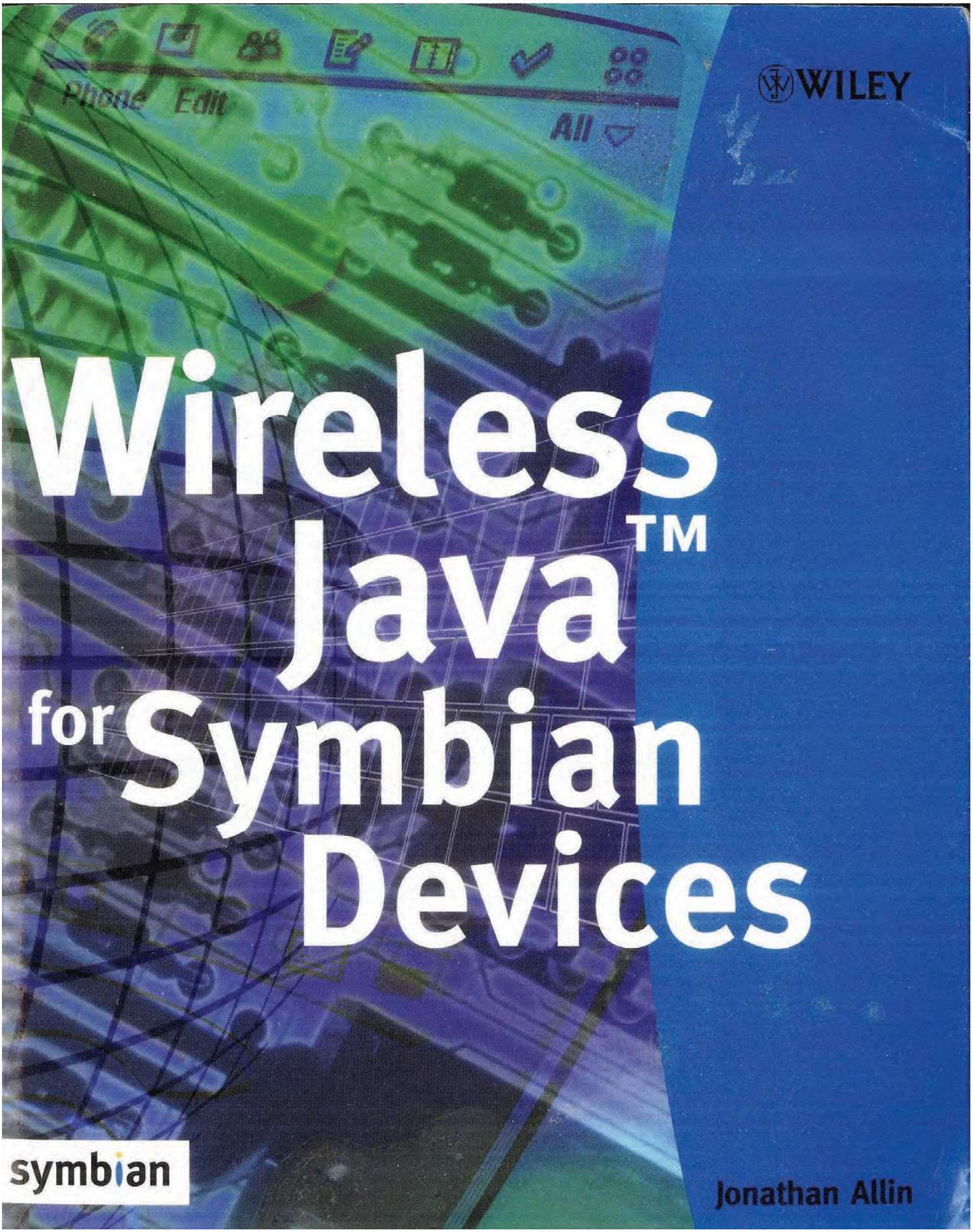


EX-1012
U.S. Patent 9,032,039

symbian

Wireless Java™ for Symbian Devices

Jonathan Allin



Wireless Java for Symbian Devices

Symbian Press

Symbian Press has been established to communicate authoritative, timely, and practical information on Symbian OS and related technologies.

Established as a partnership between Symbian Ltd (www.symbian.com) and John Wiley & Sons Publishers (www.wiley.com), Symbian Press aims to enable and encourage developers to build great products as well as to educate commentators and decision-makers within the wireless industry.

Symbian Press—meeting your development needs

Wireless Java for Symbian Devices

Jonathan Allin
Java Technology Manager Symbian Ltd

and

Colin Turfus, Alan Robinson, Lucy Sweet and John Bown
Symbian Ltd

JOHN WILEY & SONS, LTD
Chichester • New York • Weinheim • Brisbane • Singapore • Toronto

Copyright © 2001 John Wiley & Sons Ltd,
Baffins Lane, Chichester,
West Sussex PO 19 1UD, England

National 01243 779777
International (+44) 1243 779777
e-mail (for orders and customer service enquiries): cs-books@wiley.co.uk

Visit our Home Page on http://www.wiley.co.uk or http://www.wiley.com

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except under the terms of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency, 90 Tottenham Court Road, London, W1P 0LP, UK, without the permission in writing of the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the publication.

Neither the authors nor John Wiley & Sons Ltd accept any responsibility or liability for loss or damage occasioned to any person or property through using the material, instructions, methods or ideas contained herein, or acting or refraining from acting as a result of such use. The authors and Publisher expressly disclaim all implied warranties, including merchantability of fitness for any particular purpose. There will be no duty on authors or Publisher to correct any errors or defects in the software.

Symbian, Symbian OS, EPOC, the Symbian logo, and the Symbian Developer Network logo are trademarks of Symbian Ltd. This book refers to trademarks of third parties and Symbian recognizes their rights.

Copyright (c) 1997–2000, Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303-4900 USA. All rights reserved. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons is aware of a claim, the product names appear in initial capital or capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

Other Wiley Editorial Offices

John Wiley & Sons, Inc., 605 Third Avenue,
New York, NY 10158-0012, USA

Wiley-VCH Verlag GmbH
Pappelallee 3, D-69469 Weinheim, Germany

John Wiley & Sons Australia, Ltd, 33 Park Road, Milton,
Queensland 4064, Australia

John Wiley & Sons (Canada) Ltd, 22 Worcester Road,
Rexdale, Ontario, M9W 1L1, Canada

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01,
Jin Xing Distripark, Singapore 129809

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 0 471 48684 1

Typeset in 10/12pt Times by Laser Words Private Limited, Chennai, India
Printed and bound in Great Britain by Biddles Ltd, Guildford and King's Lynn
This book is printed on acid-free paper responsibly manufactured from sustainable forestry, in which at least two trees are planted for each one used for paper production.

CONTENTS

Preface ix

Introduction to Java on the Symbian OS 1

Section 1 Getting Started 21

1 Getting Started 23
- 1.1 The Development Tools 23
- 1.2 A Hello World Program 25
- 1.3 A JavaPhone Program 34
- 1.4 A Few Guidelines for Developing Applications 39

2 Java on Crystal 45
- 2.1 An Introduction to Crystal 45
- 2.2 The Crystal APIs 47
- 2.3 Crystal Utility Classes 53
- 2.4 XBombs: a Crystal Example 58

3 Java on Quartz 69
- 3.1 An Introduction to Quartz 69
- 3.2 Anatomy of a Quartz Application 71
- 3.3 Tuning Java Applications for Quartz 80
- 3.4 Limitations of Java Programming on Quartz 94
- 3.5 Conclusions 96

Section 2 Getting the most from Java on the Symbian OS 97

4 PersonalJava 99
- 4.1 Summary of PersonalJava 99
- 4.2 Symbian's PJAE Implementation 99
- 4.3 Double Buffering 102
- 4.4 Specifying Component Input Behavior 102
- 4.5 Handling unsupported optional features 104

	4.6	The Timer Classes	104
	4.7	The Timer Tester Example	106
	4.8	Compiling with PersonalJava	116
5	**A JavaPhone Primer**		**117**
	5.1	Introduction	117
	5.2	The JavaPhone Architecture	118
	5.3	Working with JavaPhone on the Symbian OS	121
	5.4	The JTAPI APIs	122
	5.5	The Database Package	141
	5.6	The Address Book API	143
	5.7	The Calendar API	156
	5.8	The User Profile API	169
	5.9	The Datagram API	170
	5.10	The Power Monitoring API	178
	5.11	The Communications API	183
	5.12	Further Reading	189
6	**Wireless Java Games**		**191**
	6.1	Developing a Strategy Game	191
	6.2	Developing Action Games	225
7	**Building User Interfaces for WIDs**		**229**
	7.1	Symbian Devices	229
	7.2	Running Java Application Written for Desktop Computers	231
	7.3	Creating a User Interface that Works	237
	7.4	Symbian Reference Designs	240
	7.5	WIDs are not PCs	243
8	**Writing Optimized Code for Constrained Devices**		**247**
	8.1	Introduction	248
	8.2	General Guidelines for Optimization	250
	8.3	Feedback and Responsiveness	251
	8.4	JAR Files and Class Path	251
	8.5	Object Creation	254
	8.6	Method Modifiers and In-Lining	255
	8.7	Strings	257
	8.8	Containers	259
	8.9	Optimization: How not to do it	261
	8.10	Going Native	262
	8.11	Avoid Polling	264
	8.12	Graphics	264
	8.13	Optimizing a Queue: A Semi-Fictitious Case Study	269
	8.14	Tools for Optimization	277

8.15 Design Patterns 283
8.16 Memory Management 286
8.17 JIT and Adaptive Compilers 288
8.18 Obfuscators 290
8.19 What's Next—Enhancing Java Performance 290
8.20 Conclusion 290
8.21 References 291

Section 3 Creating Wireless Java Services 293

9 Creating Wireless Java Services 295
9.1 Introduction to Services 295
9.2 Wireless Services 297
9.3 The Simple Conference and Handheld Travel Assistant Case Studies 301

10 Java Security on the Symbian OS 303
10.1 Developing a Secure Environment 303
10.2 The Components Needed to Create a Secure Environment 305
10.3 Java as a Secure Platform 308
10.4 Symbian OS v6.0 Security Capabilities 309
10.5 The Thumbprint Example 313
10.6 Security Management in the Future 320

11 Provisioning Java Services 323
11.1 Introduction to Provisioning 323
11.2 Applets 324
11.3 The MID Profile Recommended Practice Document 325
11.4 The Java Services Provisioning Manager 330
11.5 Mobilitec 333
11.6 Java Web Start 334
11.7 OSGi 335
11.8 MExE 336
11.9 iBus//Mobile Messaging Service 336

12 Case Studies 339
12.1 The Simple Conference Service 339
12.2 The Handheld Travel Assistant 351

Section 4 Looking deep and looking forward 367

13 JNI Programming for the Symbian OS 369
13.1 Introduction 369
13.2 JNI and the Symbian OS 374

13.3 Using Java Variables and Methods 381
13.4 Dealing with Java Strings 393
13.5 Exception Handling 397
13.6 Memory Management 409
13.7 Troubleshooting 428

14 Looking to the Future 433
14.1 The Next Release 433
14.2 The Java 2 Micro Edition 435
14.3 Symbian's Road Map and the J2ME 442
14.4 WAP 443
14.5 MExE 445
14.6 Last Word 447

Appendix A 449

Appendix B 455

Glossary 461

Index 481

Preface

The year 2000 marked a watershed. There are now more mobile phones than personal computers: 120 million PCs were sold in 1999, while mobile phone sales exceeded 450 million in 2000.

In 2001, Nokia, one of Symbian's founders, announced that 50% of its 3G terminals would be based on Symbian OS by 2004.

The Symbian OS has the necessary ingredients to set the standard for the wireless revolution. Java will be an important, perhaps essential, enabler for wireless services. This book therefore sets out to describe Java on Symbian devices.

Who the book is for

The book is aimed at developers who are creating, or are considering creating, Java applications for Symbian OS phones and other constrained devices.

It is also aimed at wireless service providers and operators who want to understand the potential of the wireless market.

Summary of the book

The book is organized in four sections.

Section 1

Chapter 1 introduces the tools and basic development processes used to create Java programs for the Symbian OS. The steps involved in compiling, running and deploying programs are presented with a traditional 'Hello World' program.

Chapter 2 introduces Crystal. 'Crystal' is the code-name for Symbian's keyboard-based reference design. The chapter describes the Crystal APIs and an example program, XBombs, to illustrate control button areas, UI issues and optimized graphics.

Chapter 3 introduces Quartz. 'Quartz' is the code name for Symbian's pen-based reference design. This chapter explains the unique way in which this task-based platform can be programmed.

Both chapters should be read in conjunction with the latter part of Chapter 7, which discusses Symbian's software reference designs.

Section 2

Chapter 4 describes Symbian's current implementation of PersonalJava. It looks at each of PersonalJava's unique features and illustrates the timer API with an example.

Chapter 5 looks at the JavaPhone APIs from the mobile phone perspective. It starts with an overview of the JavaPhone architecture and Symbian's JavaPhone implementation together with useful help on getting started with JavaPhone on the Symbian OS.

The remainder of the chapter covers the individual JavaPhone APIs. The explanations are intended to complement Sun's API documentation, not to replace it. A brief overview is given for each API, together with a discussion of those issues that might be harder to understand or that are Symbian specific. UML class diagrams show the relationship between the classes in a package and list those methods of particular interest. Each API is illustrated with an example.

Chapter 6 is devoted to game development. It describes how to develop a strategy game, using Whist as an example. It then considers approaches to developing action games on wireless devices, taking into account the constraints and advantages of WIDs for this purpose and describing the technology available for implementing action games.

Chapter 7 looks at guidelines and general rules for creating user interfaces for constrained devices, with examples focused mainly on EPOC Release 5 devices. The chapter's objective is twofold: to encourage the reader to think about opportunities for Java on constrained devices, and to emphasize that developing for such devices requires more care than developing for desktop machines.

The chapter starts with a round-up of the current Symbian devices and examples of applications running on these devices, looks at a range of large and small Java applications and considers their suitability for WIDs and PDAs, and then look at guidelines for creating UIs for constrained devices. The final section provides companion to the Quartz and Crystal chapters.

Chapter 8 looks at how the Java developer can get the most from constrained devices such as PDAs and WIDs. However, much of what is discussed is more widely applicable, being concerned not just with improving performance but also with minimizing memory requirements.

The chapter begins by examining a number of general issues, including benchmarking, required performance, and principles of optimization, and then

discusses individual areas of optimization. It then looks at several specific areas for optimization: object creation, method and variable modifiers, the use of Strings, using containers sensibly. These ideas are brought together in an example. The chapter then covers some more advanced techniques: using JNI, blocking techniques to avoid polling, issues with graphics.

A case study explores optimization issues in depth, and the chapter then looks at the use of profiling tools in the context of the case study.

Subsequent sections discuss memory issues on constrained devices and the need to cope with out of memory situations, JIT and adaptive compilation technologies, and a couple of design patterns relevant to optimization.

Section 3

The purpose of this section of the book is to provide operators and service providers in particular with an understanding of how Java can meet their needs and its importance to the wireless market.

Chapter 9 provides an introduction to creating wireless services. It discusses the creation of, and opportunities for, wireless services and applications.

Chapter 10 looks at the issues of security in relation to service and application delivery. It discusses the general requirements for a secure system and the components that are used to fulfill these requirements. It describes Java's security features, particularly as they relate to v6.0 of Symbian OS, and also summarizes the native security capabilities. The Java security APIs are illustrated with the Thumbprint example. The chapter ends with a short discussion on how platform security might evolve in the future.

Chapter 11 covers provisioning, which, along with security, represents perhaps one of the biggest challenges in enabling the widespread uptake of wireless services. Provisioning covers a number of concepts that relate to an application's life cycle: service discovery, service delivery, service installation, service update, service removal, push capability, device variability, and billing. The chapter looks at a number of technologies that address various parts of the over-the-air provisioning jigsaw.

Chapter 12 describes two case studies, Digia's *Simple Conference Service* and Telenor's *Handheld Travel Assistant*.

Section 4

Chapter 13 deals with Java Native Interface programming, for those who need to use it. JNI is a feature introduced by Sun Microsystems into JDK 1.1 in recognition of limitations of what could be accomplished with Java code alone in the Java Virtual Machine.

Chapter 14 concludes the book with a look at the road map beyond v6.0. It starts with a look at the next release of Symbian OS, and then sees how future Java releases will develop within the J2ME framework and integrate with technologies such as MExE and WAP.

Appendices

The appendices provide the reader with three resources

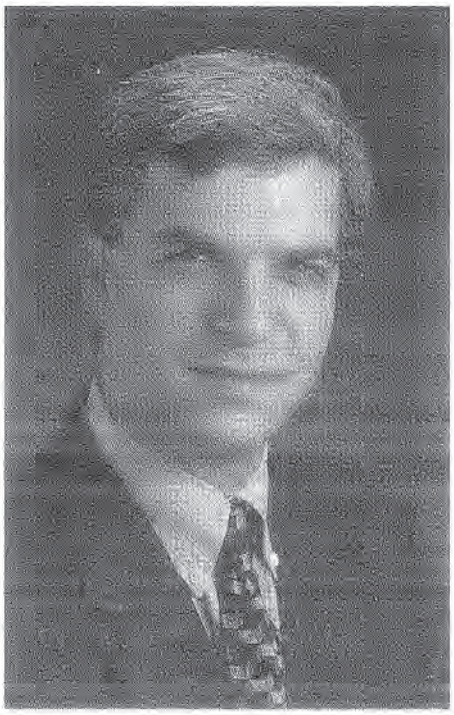
- an illustrated guide to those current Symbian devices available, with their specifications and details of which version of Java they support.
- an at-a-glance guide to resources that can be found in the book
- a comprehensive wireless glossary, including terms not only relevant to wireless Java programming, but also to the broader wireless industry

The code

Regularly updated sample code for this book can be downloaded when you register at the Symbian website at ***http://www.symbian.com/books/***. Further information on other Symbian Press titles can also be found there.

Foreword

Greg Papadopoulos, Chief Technology Officer of Sun Microsystems, Inc.



Worldwide sales of cell phones exceeded all expectations in 2000. More than 400 million units were shipped, eclipsing by far the number of personal computers shipped in the same 12-month period.

Despite such rapid growth, the days of the simple cell phone are numbered.

Over the next three to five years, cell phones will be replaced with a new class of device. Call them wireless information devices - WIDs, for short. They are personal communicators that combine the attributes of PDAs, pagers and cell phones in a simple, easy-to-use package.

Designed to take advantage of second- and third-generation wireless networks, WIDs will give us access to a wide range of personalized services: stock quotes, driving directions, integrated messaging, reservations, and services that would be impossible using today's technology.

However, as computing moves away from the desktop and into our hands, we demand ever higher reliability. We do not care what makes these devices work. We just want them to work and work all the time.

To appeal to a mass-market, these devices need to be not only compelling, but also compatible. Device manufacturers and service providers can no longer afford to think in terms of proprietary systems. They think instead of open standards and cross-platform Java technology.

In March 1999, Sun Microsystems and Symbian announced an alliance to incorporate Java technology into Symbian OS. This platform provides perhaps the most complete and mature Java implementation on constrained portable computing devices. It is the platform of choice for Ericsson, Kenwood, Motorola, Nokia, Panasonic, Psion, Sony, Sanyo and Siemens.

The combination of Java and Symbian OS enables you, the developer, to deploy services and applications in a multi-platform, mass-market, wireless environment. Using Java technology enables rapid development times, with easier coding and fewer bugs, in a secure and reliable environment.

Wireless Java for Symbian Devices provides a comprehensive introduction to all aspects of Java programming on Symbian OS. Much of the material here is also applicable to constrained devices in general, and the book is liberally sprinkled with sample code that will prove useful to all programmers targeting wireless information devices.

Service providers, network operators, and other organizations at the business end of B2C wireless commerce understand the opportunities provided by Java technology and the issues involved in delivering wireless services. This book covers the needs, constraints, and opportunities of WIDs.

Welcome to wireless Internet — make the most of it.

The authors

The Symbian writing team

Jonathan Allin

Jonathan is Symbian's Java Technology Manager and lead author of this book. He is one of a number of people in Symbian whose role is to ensure that Symbian provides a first-class Java platform for wireless information devices. He initiated Symbian's Wireless Java technology training program and is the author of the Java chapter in *Professional Symbian Programming* (published by Wrox Press in March 2000 — see appendices for details).

Jonathan has a BSc in Electronics from Southampton University in the UK, and a DSc in Biomedical Engineering from the Technion in Israel. His first job was as a residential social worker with severely disabled children, which migrated into designing and building electronic and computerized aids for such children. He was with Acorn Computers for eight years, helping to develop computers and software for schools and picking up an MBA on the way. Prior to joining Symbian he worked for Origin BV for three years, where he first became interested in Java technology, particularly in the role that it can play within the enterprise.

Jonathan is married to Lauren. They have three children: Benjamin, Daniel, and Victoria. All three play rugby and play music.

Java has reminded Jonathan that computing can be fun, interesting, and useful.

I would like to thank Phil Northam for his sterling project management and for keeping all the authors in line and on track, as well as for the material that he has contributed.

Colin Turfus

Colin graduated from the University of Dundee, Scotland with a BSc (Hons) in maths and physics, and from the University of Cambridge with a PhD in applied mathematics. He has researched and lectured at universities in the UK and in S. Korea, publishing papers in fluid dynamics and computational astrophysics. He has a strong personal interest in theories of chaos and self-organizing systems. He became interested in Java programming while

developing intranet-based math-teaching resources and lecturing about Internet technology in S. Korea.

He joined Symbian shortly after its inception in the summer of 1998, at first building up the C++ developer support service, then doing the same thing for Java, following the release of Symbian's Java Runtime in May 1999. He has since been involved with Jonathan in setting up Symbian's Wireless Java technology training program, focusing on the use of Java Native Interface programming to achieve Symbian look-and-feel for Java applications and on the use of the JavaPhone Datagram and PIM packages.

Colin's interests include jogging, hill-walking and classical guitar. He is married to Keum-ye from S. Korea. They have three girls, Selina, Sonya, and Emily, at elementary school.

Alan Robinson

Alan graduated from Cambridge University, UK, with a BA in literature and philosophy. Having become interested in applying logic to fuzzy domains, he pursued his technical leanings, and took a Computing MSc at Middlesex University.

His first job was working on developer kits for a fax server software company. When one of the company founders formed a start-up to produce an innovative platform for messaging middleware, he quickly joined the new company. He spent some exciting but stressful years learning that the stories people tell about start-ups are true, but also saw the opposite end of business when contracted to work with IBM on its MQSeries product. When RedBox's money ran out, he joined the recently created Symbian, and has worked mostly on SDK documentation and examples in the messaging and comms areas. Most recently, he managed Symbian's Java documentation for the Version 6.0 SDKs.

I'd like to thank Ben Morris for his support in this, as in all projects, and Jonathan and Colin for convincing me of the excitements of Java.

Lucy Sweet

Lucy is the youngest member of the Java Team at Symbian, having worked in the department since summer 2000. She is a graduate from Queen Mary College at the University of London, UK, with a BEng in Biomedical Materials Science and Engineering. She also has an MSc in IT, and wrote a thesis on Wireless Java Games at Symbian, before joining the company as a permanent employee. She is establishing herself as one of Symbian's wireless Java games experts, and aspires not only to be an international author, but an international speaker too. She will be presenting at JavaOne for the first time in 2001.

Lucy's other talents include grade 8s in clarinet and saxophone which she played in the National Youth Wind Orchestra of Great Britain "when she was young."

She would like to thank Colin for his mentoring, including his assistance during her student placement.

John Bown

From an early age, John was drawn to the world of electronics and embedded systems. After studying for his degree in Computer Science, he chose to develop software within the manufacturing industry.

Working on "in-product" software, John adopted the ethos that the software must reflect the user's requirements. With his belief that the most successful and usable software is that which works the way people want it to, John joined Symbian to work with the quickly evolving Symbian OS.

During his stint with the Symbian Developer Network, John embraced Java as the ideal environment in which to leverage world-class applications on the Symbian OS. Working with Symbian's Communicator reference design, John was involved in software development, user training and bringing Java to a wider audience.

Since writing the Crystal Communicator chapter, John has left Symbian to take up a "couldn't refuse" offer of employment closer to home. Now working as a software design engineer, he has not lost his belief that Java has a tremendous future in the field of Wireless Information Devices.

He thanks everyone at Symbian who helped with the chapter on Crystal, and wishes all Java developers good fortune as they embrace the Symbian Platform.

Digia Oy



Simple Conference, Chapter 12

Digia Oy is a wireless software company based in Finland, developing software and value-added services for the next generation Wireless Information Devices (WIDs), such as Smartphones and Communicators. Digia's mission is to enable Symbian licensees, mobile operators, and verticals to meet their time-to-market requirements by shortening product development cycles. The company's aim is also to bridge the fixed and wireless Internet, allowing real-time access to services, resources, and applications anytime, anywhere.

Digia is a Symbian Competence Center and a Training Partner, both in wireless software development and in training. Digia's Professional Services unit produces digital media and e-business solutions for its clients, which are pioneers in adopting wireless Internet technologies in their businesses. Some of the world's most successful handset manufacturers, wireless network operators, and portal developers are Digia's clients. The company also has several international partners, like IBM, Nokia, Sonera, and Symbian. Digia is working to build a wireless future with its investors, which include Cisco Systems, General Electric, Intel, Sonera, and Sony.

Digia Helsinki, Hämeentie 153 B, FIN-00560 HELSINKI
Tel: +358 424 77771, Fax: +358 9 777 2089
Web: http://www.digia.com
Email: scc_info@digia.com

The author: Jüergen Bocklage

Jüergen is a student in the final year of a degree of European Computer Science at the University of Applied Sciences, Osnabrueck, Germany. Currently he is an exchange student at the EVTEK Institute of Technologies, Espoo, Finland. Since November 1999 he has worked for Digia, where he has gained extensive knowledge about Symbian OS C++ and Java technologies. His final year thesis is about Java for Wireless Information Devices with a focus on Symbian OS technology.

His university experience included Java, C/C++, TCL/TK, Python, SQL, and the challenge of teaching object-oriented methods when programming with C to a group of C developers. When he moved to Finland, he realized that mobile phones were not just for yuppies and decided to become a professional Symbian developer. He is establishing himself as a speaker and his most recent presentation was on "Java for mobile devices—hype and reality" at

the Symbian Developer Expo in November 2000. He is now training Digia customers in Symbian OS C++ and Java.

I'd like to thank Jonathan and Colin for a very intensive course in London and for the trust they placed in me, Sami Leppanen and Kai Tulonen for allowing me to participate in this very exciting project, and Erno for his reviewing work.

Telenor R&D 

Handheld Travel Assistant, Chapter 12

Telenor is the leading telecommunications company in Norway. The country has some of the world's highest penetration rates for mobile phone, fixed line digital telephony, personal computer and Internet usage, and can boast one of the most advanced telecommunications markets in the world. Telenor also has substantial international operations and investments, particularly in the areas of mobile phone, Internet and Internet protocol-based communications services, satellite services, and pay-television services. Telenor R&D is Norway's largest research establishment within ICT (Information and Communication Technology). Its long-term research comprises future communication networks and services together with applications in various market arenas. Its activities help the Telenor Group stay at the leading edge of telecoms innovation.

Telenor AS, Universitetetsgt. 2, PO Box 6701, St. Olavs pl., 0130 Oslo, Norway.
Tel:+47 22 77 60 60, Fax:+47 22 77 87 20
Web — http://www.telenor.no
R&D Web — http://www.telenor.no/fou/english/about/index.shtml

The authors

Tore Urnes

Tore holds an MSc in computer engineering from the Norwegian University of Science and Technology and a PhD in computer science from York University in Toronto. Prior to joining Telenor R&D in 1998, he spent many years researching software architecture and programming language support for synchronous groupware applications, focusing on how higher-level specifications can be mapped to efficient distributed implementations. More recently, his research has focused on software infrastructure and applications within the domains of smart environments and mobile computing. Right now, he enjoys investigating how software infrastructure can simplify adapting to the dynamic service landscapes of ad hoc wireless networks.

Sigrid Steinholt Bygdås

Sigrid graduated from the University of Tromsø with an MSc in computer engineering. During her study she became interested in object-oriented languages and distributed systems. She joined Telenor R&D in 1992, and spent several years researching development efficiency related to software development tools. She took part in the development of a method for organization-dependent

systems planning before she returned to her main area of interest: software infrastructure for distributed object-oriented systems. Since 1999 she has been experimenting with Java applications on wireless devices, investigating software infrastructure within the domains of context awareness and mobile computing.

Rune Henning Johansen

Rune graduated from the University of Oslo in 1984 with an MSc in computer science. His first job was working with Simula on a machine specially designed for this language. He joined Telenor R&D in 1988. He has, among other things, participated in the development of the first Tandberg videophone, developed a gateway for automatic SMS services, and implemented Web, WAP, and SMS services for the kitchen of the future (***http://www.telenor.no/fou/english/projects/fk/kjokken.shtml***). Right now he experiments with WAP and J2ME (MIDP) on coming third-generation mobile phones

Øystein Myhre

Øystein studied mathematics at the University of Oslo. A computer science veteran, he was involved in the implementation and development of the first object-oriented language, Simula, at the Norwegian Computing Center and at the Norwegian Defense Research Establishment. In the seventies, he played a central role in development of the portable Simula system, a compiler development project with certain similarities to today's Java virtual machine and runtime environment. He was a co-founder of the company Simula a.s., which introduced Simula on a PC platform. He has been involved in developing Telecommunications Management Network (TMN) systems at Siemens and Telenor R&D using object-oriented techniques. When the Java language entered the scene a few years ago, he quickly became a great fan of the platform. He has always been fascinated by the challenge of putting seemingly large and complicated software onto hardware that appears to be too limited for that task.

Acknowledgements

This is the first book from Symbian Press. It's been a great achievement by all involved, particularly the authors, as they have generally found themselves scribbling away during "lulls" between software releases, seminar presentations, SDK work, during their weekends and other more critical events!

Many thanks also for the efforts made by the reviewers, by responding with comments often at short notice, to tight deadlines. Review was undertaken by Symbian in London and Ronneby and included: The Java Runtime

Team — Andrew Thoelke, Simon Lewis, Keith Playford, Jurgen Hoffges, Alex Rothwell, and Michael Aubert; The Java Integration Team — George Thucydides, Paul McKay, Jim Clarke, Barbara White, Jamie Hogg, Adam Stevenson, and Justina Oakes; Symbian Ronneby — Elisabet Gustafsson and Peter Wikstrom. Special thanks to Trevor Strudley, David Southeard, Emily Walton, Jonathan Hassell, and David Mery. Traditional thanks also to all the other people who made this book possible, but aren't included in the list above.

One of the underlying principles of Symbian is that it represents a "wireless community," with an emphasis on openness, industry-wide standards, and cooperation. With this in mind, special thanks from the whole Symbian team goes to Digia Oy and Telenor for their early commitment to the book and their excellent contributions with the case studies in Chapter 12.

We would also like to thank Leon Bovett for his input about action games.

Introduction to Java on the Symbian OS

Jonathan Allin

This chapter provides an introduction to Symbian's Java technology. It takes an inward look at the history behind Symbian's Java development and why Symbian regards Java as such an important technology.

The Symbian Story

The Symbian OS incorporates a comprehensive set of leading-edge applications and services that will set the standard for the wireless revolution. But where did it come from and why is it the best software platform for an emerging market of wireless information devices, or WIDs?

Symbian is jointly owned by Nokia, Ericsson, Motorola, Matsushita (Panasonic) and Psion. Its many other licensees include Philips, Sanyo, Siemens, Kenwood and Sony. The nucleus of the company was formed from Psion Software, and today's Symbian OS is an evolution of Psion's EPOC Release 5 (sometimes referred to as ER5), which appears in the Psion Series 5mx, Revo and netBook/Series 7.

Building a Software Platform for the Wireless Industry

History

In mid-1994, Psion began work on a new 32-bit operating system (OS) to replace SIBO, its sixteen-bit operating system. Products based on SIBO were a great success and included the Psion Series 3a, Series 3c, Siena, and Series 3mx, the last of which was launched in 1998. The platform supported internationalization with a full Unicode implementation on a 32-bit flexible architecture. The new OS was called EPOC, a name that reflected the intention to define a "new epoch" in wireless information devices.

Right from its inception, Psion intended that the EPOC operating system should support a commercial licensing model. The entire software team

working on EPOC, plus a sales team, were formed into a new company called Psion Software in June 1996. Its mission was to license EPOC to original equipment manufacturers (OEMs) outside the Psion Group. While EPOC was suitable for a wide range of applications, the sales group decided that its sweet spot was the mobile phone market. Its design supported the flexibility that would allow EPOC to be licensed to the many telecommunications OEMs who could then build innovative products based on it.

Psion's timing was good. In the mid-1990s, the mobile phone revolution was in full swing. Well-positioned in the harmonized GSM (global system for mobile communications) markets of Europe, Psion was also able to watch market development in the USA and Japan. It was clear that mobile communications and computing markets were converging, just as they had with the fixed Internet.

Parallels

Even as Psion began building EPOC, Nokia, market leader in the mobile handset market, introduced the Nokia 9000 Communicator in late 1996. The Communicator used the GEOS operating system from Geoworks Inc. Here was proof of concept: a combined GSM phone and personal data assistant (PDA), and an unexpected opportunity for Psion. Nokia had used GEOS to spare it the effort of writing all the PDA software, but GEOS, like SIBO, was limited by a 16-bit architecture. EPOC was the logical next step.

By early 1998, as Psion Software's sales group made steady progress licensing EPOC to the phone handset manufacturers, a new idea emerged. If Psion Software was an independent company, outside of the Psion Group, and owned by the mobile handset manufacturers, then the latter would own a strong technology asset that would enable them to boost the entire market for wireless information devices.

On June 24, 1998, the new company, Symbian Ltd, was announced to the world. Colly Myers, who had helped develop EPOC, was announced as Chief Executive Officer. Symbian's owners would be Psion, Nokia and Ericsson. Motorola announced its intention to join the alliance, and did so formally from October 1998. Japan's Matsushita, better known as Panasonic, joined in May 1999.

Symbian, owned by the world's leading telecommunications manufacturers, is now a global company with offices in London and Cambridge in the UK, Silicon Valley in the USA, Ronneby in Sweden, and Tokyo in Japan. There are now more than 700 employees worldwide.

EPOC Release 5 was formally launched in June 1999. It was a single reference design for devices based on a 640 × 240 pixel screen, pen and keyboard, although it was adapted for a VGA screen in Psion's netBook.

The Advanced Client

Psion's success licensing EPOC was because of the OS's inherent suitability for the wireless environment. The most effective wireless operating system will make the least use of wireless infrastructure. Wireless networks will not provide blanket next-generation (2.5G/3G) coverage in the near future nor will they offer the reliability of wireline ones. As a result, mobile devices must negotiate different amounts of bandwidth and may even lose connection altogether, which means switching between modes while using a service.

A WID will need to switch not only between 2.5G and 3G in packet-switched mode, but might occasionally need to negotiate legacy circuit-switched 2G networks, which require dial-up. WDP protocols such as SMS would allow continuous service, whatever the network or bandwidth. Figure 1 illustrates that bandwidth cannot be guaranteed in the air as it is terrestrially, so that wireless applications must cope with variations in bandwidth, as well as possible delays in data delivery. This may mean that an application must preserve data where appropriate and send or receive it when network coverage is restored.

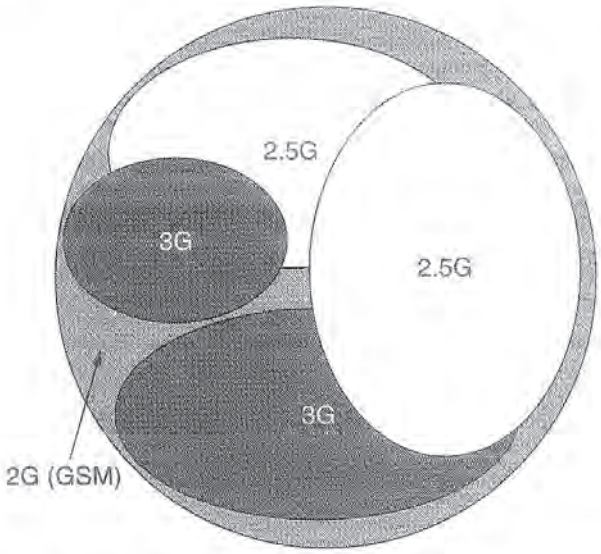


Figure 1 Symbian devices are advanced client portable computing systems, delivering useful applications in a sometimes unpredictable network environment, with variable bandwidth.

The advanced client will retain its rich functionality when there is no coverage by processing its information locally. This approach maximizes the value of airtime used by the network subscriber and allows efficient management of limited battery life. These key attributes were what mobile manufacturers were looking for: power-efficiency; the ability to operate when wireless connections are lost; and that benchmark of the telecoms industry, the "five nines," 99.999% efficiency. The last point is a significant advantage of the Symbian OS—WIDs must be "always-on", which means they must almost never crash, never lose their data, and be able to handle a variety

of service requirements as and when required. This is the kind of robustness embodied by Symbian's advanced client. Symbian has also adopted three key technologies that reflect industry-wide standards, further strengthening the Symbian OS proposition. These are WAP, Bluetooth, and, of course, Java.

The Symbian OS

Ericsson's R380 became available in September 2000. This is Symbian's first Smartphone.

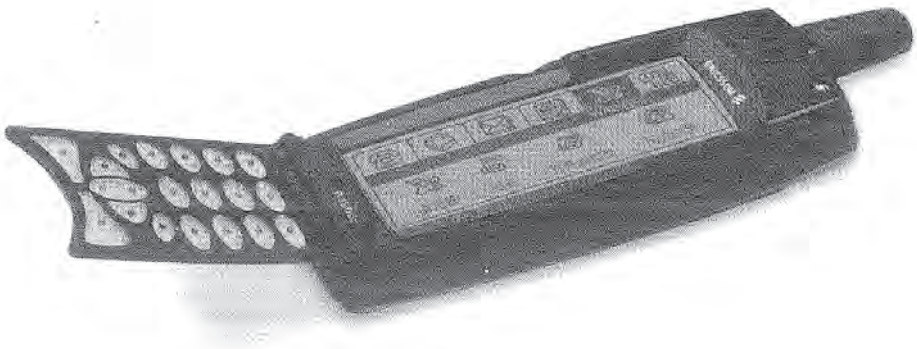


Figure 2 The Ericsson R380, Symbian's first Smartphone

Symbian has developed three reference designs for licensees, divided into:

- Smartphones, primarily for voice communication, with PDA-like sophistication that allows information to be rapidly accessed and reviewed
- Communicators with a PDA's form-factor and fully-integrated communications including voice. Entering information on these devices is straightforward using pen or keyboard

Communicators include two Symbian reference designs, code-named Crystal and Quartz. Crystal is the evolution of the Psion Series 5 into the product category defined by the Nokia 9000 Communicator. Its most recent incarnation is the Nokia 9210, based on Version 6.0, announced in November 2000.

Quartz is a pen-based tablet device with integrated telephony. With its half VGA sized screen it is designed to fit comfortably into the palm of your hand.

The Crystal and Quartz user interfaces (UIs) can be customized by Symbian licensees to give their products an individual "look and feel". The Smartphone reference design allows manufacturers still more product differentiation. It also comes with a standard UI, but gives licensees far greater freedom to customize applications and UI.

Version 6.0, as used in Nokia's 9210 and Ericsson's concept Communicator, provides some important enhancements to EPOC release 5.

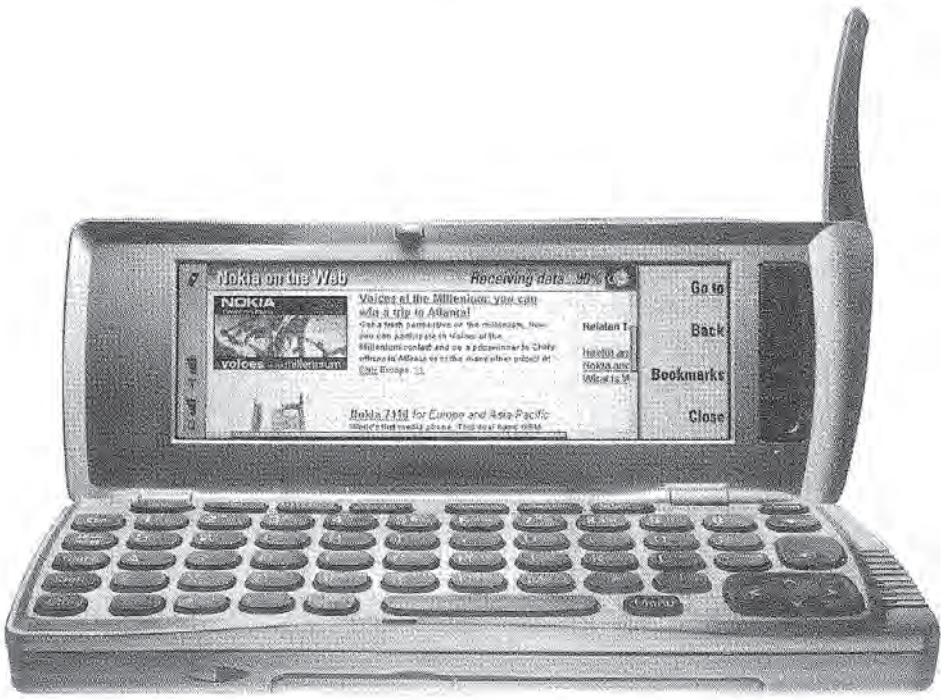


Figure 3 The Nokia 9210, based on the Crystal reference design.



Figure 4 The Ericsson Communicator concept, based on the Quartz reference design.

These include:

- The Quartz and Crystal reference designs for direct integration into real devices. Underneath the hood the GUI system has been restructured into several cooperating components, making it simpler for a Symbian licensee to adapt the look-and-feel to meet their particular requirements
- Integrated wireless telephony for single box Communicators and Smartphones, which seamlessly combine voice and advanced data functionality



Figure 5 Symbian's Smartphone reference design.

- Bluetooth technology
- WAP browsing
- Security, including full-strength encryption and certificate management, integrated into communications (SSL, HTTPS, WTLS) and application installation
- Full international locale support: EPOC C++ now uses 16-bit Unicode characters rather than 8-bit narrow characters. Text formatting, text input, and font management systems have been amended to support a wider range of locales
- New connectivity package, Symbian Connect, providing increased usability
- A media server with support for several audio and image formats
- New toolchain supporting ARM4 and Thumb instruction sets, an ARMI interworking binary standard, improved GCC compiler, and more convenient program building
- Java enhancements: implementations of PersonalJava PJAE 1.1 and JavaPhone 1.0, and performance improvements.

The Crystal and Quartz reference designs are discussed in Chapters 2 and 3.

Why Java?

There are several choices of language for developers programming for the Symbian OS, which include C++, Java and, on the content side, WML and HTML. Programming with C++ is expected to appeal to OEMs, including Symbian licensees, for the design of application engines. Java is expected to become the preferred language for the programmer community when developing applications across the whole range of Symbian licensee devices (SLDs). Java offers a powerful programming environment and key Symbian technologies—telephony, contacts-management, and Datagram messaging (all relevant to games and value added services)—are exposed through the JavaPhone APIs, which are detailed in Chapter 5. Furthermore, Java applications can also be adapted to have the look-and-feel of Symbian's Smartphone and Communicator reference designs, as we mentioned earlier.

The Symbian OS provides a full Java implementation within the Java 2 Platform, Micro Edition (J2ME) framework. This includes PersonalJava and the first JavaPhone implementation for wireless devices. The PersonalJava implementation is extensive, including all mandatory features and most optional features defined in the PersonalJava Application Environment (PJAE) specification.

The revenue from WIDs will be generated by wireless services enabled by these devices, rather than through the actual sale of the device. Java will be an important, perhaps essential, enabler for these services. Throughout this book, we will be discussing the relationship between the Java environment and wireless services.

The most significant benefits of the Java language, probably in order of importance, are these:

- Security
- Standardization
- Robustness
- Device independence
- Fast development

Security

Security is crucial to the success of wireless devices.

Wireless services depend on the secure delivery of trusted applications and the secure exchange of information. Security was, and is, one of Java's critical design goals, and it is built into Java from the ground up. Features such as the byte code verifier, prevention of random memory accesses, and disallowing inappropriate casting, ensure that Java can be used to develop secure systems. Achieving the same watertight security with other languages is difficult, if not impossible. We'll look at security in more detail in Chapter 10.

Standardization

Numbering over 2 million, skilled Java developers have overtaken skilled C++ programmers. As a consequence, a wide range of development tools, documentation, books, technical support, and training is available. Java is the preferred teaching language, and certainly the preferred object-oriented (OO) language, in an increasing number of computer science courses.

Robustness

Java code can be developed more quickly and is easier to maintain than C++ code and at the same time is likely to be more robust and contain fewer faults than the equivalent C++ code.

Fewer Faults

Here's an example. This snippet of C++ code is intended to reverse the contents of an array, `ar`:

```
include <stdlib.h>
void reverseArray(int ar[])
{
  if(ar = NULL) return;
  unsigned int len = sizeof(ar);
  for(unsigned int index = len-1; index >= 0; index--)
  {
    int temp = ar[index];
    ar[index] = ar[len-index-1];
    ar[len-index-1] = temp;
  }
}
```

The example is based on a genuine piece of code. It actually contains four mistakes: three are coding errors and the fourth is an algorithmic error. These are mistakes that won't generate compiler errors but will cause runtime errors. Let's go through them.

The code `if(ar = NULL) return;` is intended to check that `ar` is a valid array. However, because the assignment operator has been used rather than the equality (==) operator, `ar` will be set to null. This is interpreted as boolean `false` so that the program will carry on with `ar` equal to null.

`sizeof(ar)` was expected to return the number of elements in the array. In fact it returns the size of an object in bytes, so at best it will return the size of the array in bytes, not the number of elements. However, even though `ar[]` looks like an array, it is actually a pointer. Therefore `sizeof(ar)` will return the size of a pointer, which is typically four for a 32-bit machine.

The third mistake arises because an unsigned integer, `index`, was used for the loop counter. This means that `index` will never go negative and therefore the loop won't terminate.

The last mistake is a design error. The number of times the code was intended to go through the loop is equal to the number of elements in the array. But think about an array of five elements: I want to swap the first with the last, the second with the second to last, and the third, or middle element, can stay where it is. In other words, I want to make two swaps, not five!

This is the equivalent Java code:

```
void reverseArray(int[] ar)
{
  if(ar == null) return;
  int len =  ar.length;
  for(int index = len-1; index >= 0; index--)
  {
    int temp = ar[index];
    ar[index] = ar[len-index-1];
    ar[len-index-1] = temp;
  }
}
```

The Java compiler expects the argument of an `if` statement to be a `boolean`: therefore I have to use the equality operator. Java arrays are first class objects and so `ar.length` will return the number of elements in the array. Primitive types in Java (bytes, ints, longs) are always signed, so my loop counter will go negative.

I've removed three out of four faults by using Java, leaving me to concentrate on the algorithmic error. Arguably a fourfold improvement in productivity!

Less Code to Write

Java applications also require fewer lines of source code. The following Symbian OS C++ code sends the message "Hello Imperial" to the server 193.63.255.1 (which belongs to Imperial College in London) on port 7, which is the echo port. The code then reads the echoed reply:

```
RSocketServ ss;
err = ss.Connect();
RSocket sock;
err = sock.Open(ss, KAfInet, KSockStream, KUndefinedProtocol);
const TInt KEchoPort = 7;
TInetAddr imperial(INET_ADDR(193,63,255,1), KEchoPort);
TRequestStatus stat;
sock.Connect(imperial, stat);
User::WaitForRequest(stat);
```

```
TBuf8<14> text = _L("Hello Imperial");
sock.Write(text, stat);
User::WaitForRequest(stat);
sock.Read(text, stat);
User::WaitForRequest(stat);
sock.Close();
```

Although this is Symbian OS C++ code, WIN32 code is very similar.

Here's the equivalent Java code:

```
Socket socket = new Socket("193.63.255.1", 7);
DataOutputStream out = new DataOutputStream
     (socket.getOutputStream());
DataInputStream in = new DataInputStream
     (socket.getInputStream());
out.writeUTF("Hello Imperial");
String echoedText = in.readUTF();
socket.close();
```

Not only is the code shorter (6 lines versus 14), but it's a lot clearer as well.

This isn't quite the end of the story. Java byte code itself is compact so that in general a line of Java source will generate less code than a line of C++. This is because instructions like `add`, `new`, etc. are expressed as single byte codes whereas the C equivalent will expand into word instructions or multiple instructions. On the other hand, we need to be aware of the higher overhead associated with each Java class.

The Benefits of Smart Arrays and Automatic Garbage Collection

Robustness is also achieved through two other important features: smart arrays and automatic garbage collection.

The absence of pointers and the use of Array objects means that Java code cannot point to either a non-existent object or to the wrong sort of object. It is too easy in C++ code to write or read from an incorrect area of memory (e.g. off the end of an array). The real problem is that the mistake won't be identified immediately but can generate apparently unrelated misbehavior, frequently a crash, hours or even months later. If a Java application attempts to access an inappropriate object, or access a non-existent element in an array, an exception will be thrown immediately that identifies both the cause of the problem and where in the source code the problem occurred.

Java's automatic garbage collection means fewer, or no, memory leaks. In C++ we must be careful to match object construction with object destruction: failing to destroy an object results in a memory leak, attempting to destroy an object twice will cause the program to crash. (As in C++, we must ensure that our Java code doesn't keep hold of references to unwanted objects: such objects cannot be garbage collected and will effectively cause a memory leak.)

Java is not a complete answer to porting and device independence, but is almost certainly the least painful. This is because Java is inherently device agnostic: the same binary will run on a MIPs processor or an ARM processor; it is relatively easy to port: APIs in general are simpler and UI APIs are standard; and the code is generally simpler and there's less of it.

In Summary

By combining a large developer community, a fast development environment, and a secure and robust execution environment, new services can be deployed rapidly and safely onto a wide range of devices. The fruit of this union is added value and differentiation for the service provider.

Nonetheless, Java should not be seen as a silver bullet. Java applications, especially poorly written ones, can be slow; and in general they won't have full access to a platform's resources.

Speed is not a problem on desktop machines. Lots of MIPS (millions of instructions per second) and lots of memory for JIT (just-in-time) compilers mean that nowadays a Java application feels just as crisp as an application written in C++. Speed is, however, an issue on small devices, where memory is tight and CPU performance limited. Personal Java and J2ME technologies provide certain performance and memory improvements (see Chapter 14), and we will look in depth at how you can write leaner and meaner Java applications in Chapter 8.

Java has restricted access to a device's resources. For instance, a pure Java application running on a Symbian device won't have access to GUI (graphical user interface) controls such as command buttons or date editors, or to EPOC data types such as Word, Sheet, DBMS or Agenda documents. JavaPhone (which we'll discuss in Chapter 5) opens up considerable Symbian OS functionality to Java applications. Other areas can be addressed by design guidelines or classes provided by Symbian's developer support or by third parties. Some features, in particular GUI controls that aren't also used by the AWT, are difficult to access.

Where does the Symbian OS Fit in?

If developers, licensees, and network operators are convinced that they should be developing in Java, why should they choose the Symbian OS rather than any other platform running Java?

The answer is in the added value provided by the Symbian OS:

- Symbian devices do not rely on being connected to a network to offer users applications and services. Devices will continue to provide intelligent capabilities even when disconnected.

- integration of Java with the Symbian OS will continue. Application engines, GUI controls, communications protocols, and other Symbian OS facilities are becoming increasingly accessible to the Java developer.
- Symbian's Java implementation is robust. It builds on the Symbian OS, which is itself designed to operate 24 hours a day, 7 days a week.
- Symbian's Java implementation is fast. Symbian has optimized, and continues to optimize, both the Java virtual machine port and the Java libraries. The result is probably the fastest Java implementation in its class.
- the implementation has a tight footprint, taking advantage of the lean and mean philosophy of the Symbian OS.

A bit of History

Porting Java to the Symbian OS began in February 1997. Symbian obtained a source license from Sun for JDK 1.1.2, which arrived in the form of 32 megabytes (MB) of C and Java source code. Most of the code was a generic core, but there were platform-specific areas that required tailoring in the application layer, the Standard C library, and the virtual machine.

The Mismatch between EPOC and POSIX

The first hurdle to overcome was the mismatch between what was available on EPOC (the early name for Symbian OS) and what Sun assumed was available. The source code was organized for ANSI C and POSIX and assumed process-wide file handles usable by any thread, while EPOC file handles are thread-relative. There is no standard thread applications programming interface (API), so inevitably work had to be done to match the EPOC thread APIs and its active-scheduler model to a multi-threaded model.

Overcoming these problems required a partial implementation of the Standard C library (STDLIB), which drives the relevant EPOC C++ APIs, to handle the ANSI C and POSIX areas. It was also necessary to develop a CPosixServer thread to own the thread-relative file sessions (RFs) and RFile handles, a sockets API on top of ESOCK, a limited C++ API for asynchronous IOCTL, and a variation of the Unix `select()` function to work on a single file descriptor. 64-bit integer arithmetic was handled by the GNU C++ compiler on EPOC hardware and by Microsoft's Visual Studio for the Emulator. Java monitors are implemented using a variation on the `RCriticalSection` class, which avoids the full cost of an `RSemaphore::Wait()` and `RSemaphore::Signal()` if the object is not currently locked. Further, the Java virtual machine (VM) widely uses writeable static data, which is not generally available to an EPOC DLL. This meant the build tools had to be modified so that the static data could be relocated at link time.

"Hello world" ran on July 24, 1997.

Implementing the AWT

Implementing the AWT was the second hurdle. Being pen based, the Symbian user interface is quite different to Windows and Motif. EPOC peer classes were written in Java and an EPOC C++ layer developed to package up the arguments and pass the calls through to an AWT server. It is this server that does the real work: it has access to the EPOC's GUI environment (known as CONE) and all the necessary handles, and has responsibility for converting input events into Java events.

The GraphLayout Applet from Sun's JDK ran on October 15, 1997.

But there were still problems. The AWT server was originally known to Java; however, this could cause deadlocks during garbage collection and so was redesigned as a secret thread. Problems with memory leaks and further problems with garbage collection were also overcome.

Optimization

Hurdle number three was also opportunity number one: optimization to ensure that Symbian's Java implementation performed well. The core of the byte code interpreter, a big switch statement in C code, was rewritten in ARM assembler. The more commonly used classes are pre-loaded: this reduces the RAM (random access memory) required and improves start-up time, though can increase the ROM (read-only memory) needed.

Finally, C++ classes and functions were written to generate the common assembler elements used by the byte code interpreter instructions. This makes it straightforward to try out new code or algorithms for these elements and to take advantage of the optimizations that ARM assembler can provide.

Development moved from JDK 1.1.2 to JDK 1.1.4, and on June 19, 1998, the JDK 1.1.4 version of the GraphLayout Applet ran successfully.

Certification

Obtaining certification was the fourth hurdle. The Java certification kit contains some 31,000 files and defines 6,000 individual tests. The validity of some of these tests had to be negotiated, for instance those that made too generous assumptions about the amount of memory available. The test harness, written in Java, drove multiple slave machines. Typically, four machines were used, which allowed a test run to complete in about a day.

Java certification was obtained on August 24, 1998.

Integration and EPOC Release 5

The final hurdle was integrating Java into EPOC. Java had to be integrated into the Web application to allow Applets to be run and, because Java applications are generally started from a command line, an application launcher was

required. Java applications on the Symbian OS need a document UID (unique ID) and class names or titles should appear in the task list. Tools to package Java applications for EPOC deployment were also developed.

The first Symbian runtime for Java was included in EPOC release 5, which became available in May 1999. The first Symbian SDK for Java was released at the EPOC World conference in June 1999.

JavaPhone

The Wireless Java working group was formed in October 1998 to develop the JavaPhone 1.0 specification (this was pre-JSR days). Symbian began implementation in October 1999, though the specification wasn't finalized and published until March 2000. The implementation was completed in May, though getting a full test pass was difficult as the many EPOC services that it relied upon were being actively developed at the time. The first full pass was achieved on September 6, with final release on October 16.

The specification was evolving as development progressed, and in particular a number of problems with the specification were only discovered once serious testing started. There was no official test suite for JavaPhone and so the developers had to write their own.

JTAPI was implemented without access to a true telephony platform. Up to that time, EPOC was "two-box" based, with the telephony implementation using AT commands to drive a GSM mobile phone. None of the interesting JTAPI behavior (voice calls, networks etc.) had been implemented and so the team had to rewrite and extend the generic "GSM" TSY to support JTAPI as best as possible.

In Fall 1999, EPOC server behavior was changed to allow multiple threads to use the same client-side resource/handle. This allowed a simpler and more efficient implementation of the JavaPhone APIs that used EPOC services such as addressbook, calendar, JTAPI, and datagram.

PersonalJava (Symbian OS v6.0)

The PersonalJava code arrived on March 19, 1999. The basic port to EPOC Release 5 went quickly: HelloWorldApp and EmbeddedCaffieneMark ran on May 7, 1999, and GraphLayout on May 12. The first Unicode build ran shortly after, though the internal VM interface was still narrow. The internal interfaces such as STDLIB were ported to Unicode in June and the AWT ported to UIKON (from EIKON) in July. The Thumb build of the JVM was made in October, with a full pJCK pass on the Emulator and Thumb achieved on January 16, 2000. A round of bug fixing followed, with final release in July 2000.

And Products...

The Version 6.0 Quartz SDK for Java was released at Symbian's DevExpo in November 2000. The first Version 6.0 hardware platform for Crystal was Nokia's 9210 Communicator, launched at the Nokia Mobile Internet Conference just three weeks later. By March 2001, Symbian had unveiled its platform for GPRS-enabled Smartphones and Communicators, incorporating support for WAP 1.2 and advanced Bluetooth technology. This provides all the building blocks necessary to create services and applications for wireless devices running on GPRS networks. Version 6.1 products are scheduled to ship from H2, 2001 with the next Quartz SDK shipping in the third quarter of 2001.

It's interesting to note that the front-ends for tools such as AIF builder and the CS Help Compiler are written in Java. So even C++ developers are now exposed to Java!

Introduction to the Wireless Marketplace

Wireless Services

Wireless information devices are characterized by timeliness and mobility, features that open up opportunities for many new and exciting services. Two key areas for innovative development are location-based services and games. In Chapter 12 we see an example of a location-based application demonstrated by the Telenor's "Travel Assistant," case-study, while a case-study from Symbian Competence Center, Digia Oy, gives an example of a location-independent service. Other development areas for new services are described below and we include an in-depth look at wireless games, demonstrated in the Whist case-study in Chapter 6. These new markets will allow networks to deliver value added services to customers, assisting them in their quest to reduce customer churn rates while increasing ARPU (average revenue per user)

Wireless Commerce

Wireless Commerce provides the opportunity to purchase goods or services securely and easily, using a mobile phone. Examples include hard goods such as the weekly shopping, soft goods that can be instantly downloaded such as stock information or a piece of music, or services such as banking: the ability to transfer money between accounts, control standing orders, or transfer money to someone else's account.

Cashless Payment

This enables you to make small purchases without having to count the change, for example newspapers, car parking, road tolls.

Mobile and Paperless Ticketing

The airlines are moving toward paperless ticketing. A ticket confirms that the holder has paid for a service, such as a train journey or an airline flight, or a concert performance or a film, but it doesn't have to be a piece of paper. It will be possible to buy tickets directly using your wireless phone. The ticket will be held in a secure format on the phone and then checked and cancelled as you enter the cinema or theatre, or board the plane or train.

Travel and Traffic Congestion Information

How often have you been on a car journey and suddenly hit the back of a queue? Yet a few hundred meters down the road was a turn-off that might have provided an alternative route. We need traffic and congestion information to be pushed onto our phones in a timely manner so that we can plan our journeys better and avoid delays.

Rail and Flight Information

We need information on flight and train times, changes to these times, and additional information such as delays, gate and platform numbers.

Location-Based Services

Location-based services can be divided into two types: "coordinate" services that provide positional information (using GPRS or signal strength), and push services that vary depending on your location. (See Chapter 12).

"Coordinate services" answer questions like "where am I?", "where is. . .?", "how do I get there?". If you are walking through Florence, the answer to "where am I?" might be Giotto's bell tower and your destination could be the Palazzo Pitti. And of course we can also answer "where are my friends?".

Push-type location services will enable you to receive information about local items that are of interest to you, such as restaurants, museums, and shops.

Networks will also be seeking ways to continue engaging their customers when they've enabled them to get to (or on their way to) their destinations — and one way to do that will be by offering games. Wireless games contrast with location-based services, drawing players together on the network and making their location irrelevant. Of course, wireless games could — and do — also use location as a feature and we will see games being developed that rely on location as part of their intrinsic offering. This book includes an in-depth case-study of a wireless card game, Whist, developed by Symbian for the Quartz reference design. We look in more detail at the wireless games market below and why it is so important. Many of the paradigms that apply to games also apply to other applications and services on wireless platforms.

Understanding Wireless Games

To understand the importance of Java for wireless games, we need to look at the wider context of developing applications for the wireless market. Let's begin by looking at some numbers. In a report published in September 2000, Datamonitor published forecasts of the wireless games market across Western Europe and the USA. By 2005 they predict this market will comprise 200 million people, all playing Internet games on their mobile phones. Western Europe and Japan will be at the center of this revolution. Take-up in the USA will be delayed, mostly because of the speed of network rollout and the lack of common standards in the US. The market is expected to be turning over a staggering US$6 billion by 2005, although revenues are not expected to pick up until 2003. For developers, Europe's primary markets will be in the UK and Germany. (Datamonitor: *The future of wireless gaming*, Sept 2000, ***http://www.datamonitor.com***)

The New World of Wireless Games

The popularity of simple games like Snake and Tetris on mobile phones shows that most of us, whether we care to admit it or not, like to play games. Games, however, can become much more sophisticated. Much of the gaming AI (artificial intelligence) can be provided by the server, or the server can be used to find us a partner of the appropriate level for Dungeons and Dragons, or chess. Such server-mediated games provide opportunities for chargeable services.

The games industry is full of heavy hitters. While Sega's Megadrive drew a (large) cult following, it was that consummate mass-marketer, Sony, that established the games console as a fixture in many living rooms. In the home there are now some very powerful games platforms. But now games are extending their reach out of our homes and into our hands. The wireless market space differs from the traditional console-driven market, defined by some analysts as the domain of "casual gamers," as opposed to the "hard core" of console-users. There's little precedent, though, and Nintendo has the only successful platform so far with the Gameboy. But the "casual vs. hard core" gamer argument may help to explain the way games should and will be targeted in a new, networked, market.

With the wireless games market still in its infancy, there is a buzz of activity from telecoms OEMs, and network operators, as well as those companies that have traditionally built games consoles and published games, to build market share in the wireless space. The ways in which they are doing this are manifold. Nokia is developing its own games and announced some time ago that it intended to provide content, but has also signed agreements with software houses Rage and Eidos for third-party development. Others, such as Motorola, are encouraging and supporting independent developers to build applications

for their next generation platforms (see table over). Content-providers like AOL are also looking for ways to expand their offerings in this area. For companies like Sony, wireless is an opportunity to extend games from the console to the hand. Sony's alliance with Vodafone (which has UK and Germany as prime territories), and development work with leading handset manufacturer Nokia, underlines its intentions. But the race is far from won; in fact, it has only just begun.

Write or Port?

Should you write or port applications? In the short run, many of the games appearing on WIDs will be ported from other platforms—in fact this is expected to be an early enabler of the WID market, according to analysts like Datamonitor: "Gradually, popular PC and console games will be ported onto mobile phones. With their design altered to accommodate a smaller screen, they will appeal to the public." Classic games such as cards are expected to be very popular, too. However, porting games and any other application poses a set of problems in such areas as user interaction and user interface, so that the better approach and long-term route to a successful games market will be through designing original content and applications. Network operators are now looking for innovative ways to differentiate their brand in a competitive market by taking full advantage of a network's functionality.

Device or Platform?

With games moving onto wireless platforms, we can introduce the concept of pervasive games. With pervasive games, an application that is running on a tablet Communicator could also be required to run on a Smartphone, or even a desktop. This will not only require developing several versions of the game at the UI level to take into account any differences in the form factor, but might have implications for the kind of interactions that take place between each form factor and the network. Java offers games developers the best tools to achieve this with the minimum of difficulty (Chapter 11).

Business Case

Who should you target your game at—operator, handset manufacturer or user? Where does it sit in the mobile space? Should you sell your program over the air or over the counter? It is possible that games could be sold over the counter direct to users, with no network functionality required, but a key driver for selling applications to network operators is that the application uses airtime. This is a crucial consideration. It is a goal of infrastructure vendors to build up the market by helping operators maximize profit from their hardware.

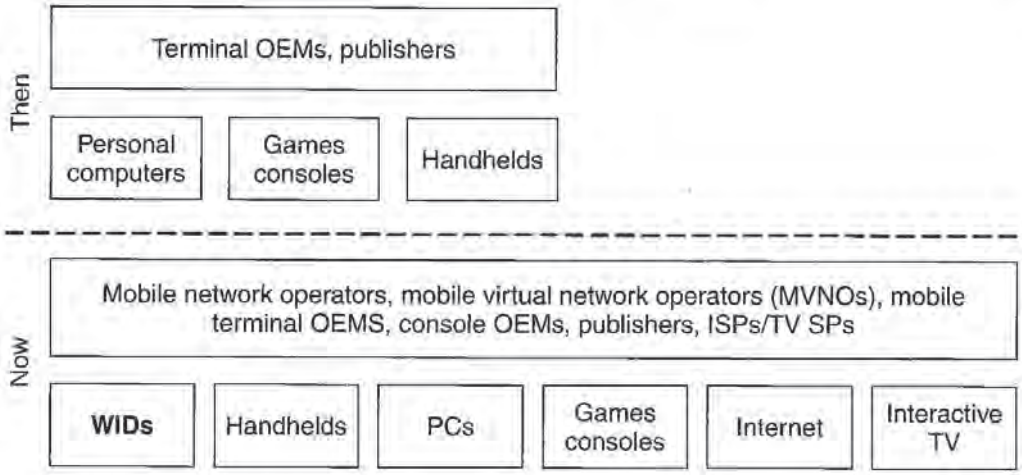


Figure 6 The games market is altogether more complex now than it used to be and has grown to include WIDs and many other new form factors as well as new developers and publishers.

Most of Symbian's licensees now run developer forums designed to support independent developers and bring their applications to market. Below is a list of some of the developer links Symbian partners and licensees provide.

Symbian licensee and partner developer links online	
Nokia	www.forum.nokia.com/
Motorola	www.motorola.com/developers/
Ericsson	www.ericsson.com/developerszone/
NTT DoCoMo*	www.nttdocomo.com/imode/

*(NTT DoCoMo announced a Java implementation on its iMode service in January 2001)

WIDs offer a far richer array of enabling technologies than today's simple feature phones. This allows users to make the most of those little bits of "niche time," perhaps while they are waiting for a train or in a queue. We can catch up on our messages, continue working on our holiday plans, pay the newspaper bill, or we can have a bit of fun.

The Future of Java on Symbian OS

This chapter demonstrates the importance of Java to Symbian and the wireless market in general. Symbian will continue to establish Java as the preferred developer language for its devices. Java is already a permanent part of Symbian OS and the intention is that every Symbian WID will include a JVM.

Symbian's goal is to provide the best possible Java platform on WIDs and Symbian actively encourages the general software houses to adopt Java. However, C++ will continue to play a significant role. It will still be used by specialist Symbian OS developers for areas such as application engines and device drivers. Java and C++ developers will be catered for on an equal footing with the same level of support, quality of tools, and visibility.

Section 1: Getting Started

How to get started on Symbian OS, the Symbian OS emulator, and developing for Symbian's Quartz and Crystal reference designs

1. Getting Started page 23–44
2. Crystal page 45–68
3. Quartz page 69–96

1

Getting Started

Alan Robinson

In this chapter, I'll give an introduction to the tools and basic development processes that you'll use to create Java programs for Symbian OS Version 6.

The steps of compiling, running, and deploying programs are presented through a traditional "Hello World" program. We'll then go on to have a first glimpse of one of the most significant parts of Symbian's Java implementation, the JavaPhone APIs.

There isn't anything particularly specific in this chapter to either of Symbian's software reference designs, code-named Crystal and Quartz. For simplicity's sake, we'll use Crystal, and mention any significant differences for Quartz.

1.1 The Development Tools

Symbian enables Java development for its platform by providing software development kits (SDKs). You may be asking why, given Java's cross-platform capabilities, you need more than the standard tool set provided by a Java Development Kit (JDK) or Java IDE. There are at least a couple of answers.

For one thing, the APIs in Symbian's Java implementation, PersonalJava and JavaPhone, are specific to WIDs and other smaller devices, rather than being part of a standard JDK (although the Personal Java implementation provided is a rich one, little different from JDK 1.1.6). The SDKs provide an effective device emulator that enables efficient development for these APIs, and for targeting the particular look and feel of Symbian devices.

Regarding tools, you'll still use the familiar JDK **javac** to compile your programs. Later in the development cycle though, you'll need to use the SDK's tools to allow a user to install and run Java applications in the standard Symbian ways.

We'll start with a brief survey of what's in an SDK.

1.1.1 The SDKs

For Version 6 of the Symbian OS, there are separate SDKs for C++ and Java for each reference design.

The central component of the SDK is its emulator, a Windows NT 4.0/98 program that emulates a Symbian device. For the Java SDKs, this includes an emulation of the Java virtual machine and class libraries.

Next is the SDK's online documentation, which is composed of the *Developer Library* and the *Technical Papers*. Readers of this book will probably find the SDK documentation most useful for the detailed reference information for the tools and Symbian-specific APIs. For Quartz developers, the *Quartz Style Guide* is a key guide to how applications should work.

The SDK's example code addresses many of the important issues in developing for the Symbian OS, including designing user interfaces, using the JavaPhone APIs, and writing JNI programs.

Symbian also supplies tools that you will use to deploy and install programs on Symbian devices, and that are required to support native method development.

A significant change for users of previous Symbian SDKs is that you no longer need to install each SDK in the root of a different drive. Instead, you can now have multiple SDKs on the same drive, and install them anywhere in the directory structure. An environment variable, `epocroot`, gives the directory (such as **\Symbian\6.0\CrystalJava**) for the SDK currently being used for development. This is set initially by the SDK's installation program. If you later need to develop with a different SDK, you will need to reset `epocroot` manually to the relevant directory.

There are a few key directories that are useful to know. The **epoc32** directory contains release (**release\wins\urel**) and debug (**release\wins\udeb**) versions of the emulator, including the Java Virtual Machines. The **erj** directory is provided exclusively for Java use, and contains the class libraries for Personal Java (**lib\classes.zip**) and JavaPhone (**ext\JavaPhone.jar**), as well as example code, and files for JNI support. The emulator is configured by default to see the **erj** as part of the device's file system, mapping it to drive **J:**. Finally, **Shared\tools** has the Symbian tools.

For JNI development, you will need to install an SDK for C++ *first*, and then install an SDK for Java in the *same* directory. The SDKs for C++ supply the Symbian C++ headers, libraries, and tools that you need to build native libraries. You will also need Microsoft Visual C++ 6.0 to compile the C++ code.

1.1.2 The Emulator

The emulator allows most of the development cycle to be done on a PC using familiar Windows-based tools (Symbian does not supply a Java compiler that runs on target devices). It can run both Java applications and Applets. Applets

are run as on other operating systems, by starting the **Web** browser application and opening the relevant HTML file. There are several options for running applications, which we'll explore with the Hello World program.

The emulation is comprehensive and realistic, but, nevertheless, there are some situations where a degree of early testing on real devices is required:

- performance is likely to be an important attribute for many programs, and to assess this you will need real hardware
- basic telephony functionality can be emulated by connecting a PC to a suitable mobile phone, but some more advanced features, such as DTMF tone control and signal strength monitoring, are not available. Other communications functionality, such as serial comms and TCP/IP, and the higher-level protocols built on these, *can* be used on the emulator
- for JNI programs, any native libraries that you supply must address C++ issues. There are some minor differences between the language features supported by the GCC compiler used to build code for target devices, and the Microsoft Visual C++ compiler used to build code for the emulator. Differences in thread and process handling—the emulator is a single Windows process that emulates Symbian OS processes using Windows threads—may also have an impact. These differences are documented in the C++ SDKs.

1.1.3 Java Tools

You will need a Java Development Kit 1.1.8 from Sun Microsystems to compile your programs. Note that JDK 1.2 is *not* suitable. The JDK 1.1.8 can be downloaded free from ***http://java.sun.com/products/jdk/1.1/***.

1.2 A Hello World Program

That's about as much background as you need. The rest of the chapter will be using the emulator and tools in a real, step-by-step, way.

Before going further, you will need to have installed the Crystal SDK for Java, and to have become familiar with the emulator's basic operation, such as how to use the shell-like **Extras** application, and a standard application such as **Contacts**.

1.2.1 The Hello Program

Hello is an unremarkable program that uses the AWT interfaces to create a Frame with a Label and a Button to close the program.

```
package com.symbian.devnet.crystal.hello;

import java.awt.*;
import java.awt.event.*;
```

```
public class Hello extends Frame implements ActionListener
{
    public static void main(String[] args)
    {
        Hello hi = new Hello();
    }

    Hello()
    {
        Label lab = new Label("Hello world");
        Button but = new Button("Dismiss");
        but.addActionListener(this);
        addWindowListener(new Goodbye());
        setLayout(new GridLayout(2, 1, 10, 10));
        lab.setAlignment(Label.CENTER);
        add(lab);
        add(but);
        setSize(200,100);
        show();
    }

    public void actionPerformed(ActionEvent e)
    {
        dispatchEvent(new WindowEvent(this,
                      WindowEvent.WINDOW_CLOSING));
    }

    private class Goodbye extends WindowAdapter
    {
        public void windowClosing(WindowEvent e)
        {
            System.out.println("Goodbye");
            System.exit(0);
        }
    }
}
```

1.2.2 Building Hello

Note: It is a known problem that if the CLASSPATH environment variable is set, javac can crash when you attempt to build Java projects for the SDKs. If this happens, type:

```
set CLASSPATH=
```

at the command prompt before building any Java projects for the SDK.

Hello is built using **javac** in the standard way. We've packaged up the commands into a batch file, **Build.bat**, which also creates a **code** directory

in which to put the compiled classes, and makes a **.jar** file for the program. The batch file assumes that **Hello** is in an **erj\wjsp** directory under the SDK installation: just change the paths in the fourth and fifth lines if you've put it somewhere else.

```
@echo off
set appname=Hello
set localclasspath=%epocroot%erj\lib\classes.zip;
cd %epocroot%erj\wjsp\%appname%\
if not exist %epocroot%erj\wjsp\%appname%\code\ mkdir
%epocroot%erj\wjsp\%appname%\code
javac -classpath %localclasspath% -d code source\*.java
if errorlevel 1 goto failed

echo Successfully compiled %appname%

cd code
jar -cfv ..\Hello.jar.
cd ..
goto exit

:failed
echo Failed to compile %appname%

:exit
```

Its only interesting feature is that the class path is set, in variable `localclasspath`, to point to the emulator's implementation of the PersonalJava classes in the **erj** directory. Note also the use of the `epocroot` environment variable, mentioned earlier, to get the SDK path.

To build **Hello**, start up a command prompt, move to the **erj\wjsp\Hello** directory in your Crystal SDK for Java directory, and enter `Build`. You should see a message back: `Successfully compiled Hello`.

1.2.3 Running Hello from the Command Line

Users will expect to run Java applications in the same way as other (C++) applications. For Crystal, this means from the **Extras** application, or for Quartz, from the **Launcher** screen — generically, we'll call these the *shell* programs. It takes a bit of extra developer effort (though not much) to add an application to the shell, so you may prefer to use the Windows command line while you're in the development cycle on the emulator.

Continuing in the command prompt window, you can now run **Hello**. Again, a batch file has been provided that contains the necessary command, so just enter its name, **AppRun**. You'll see a new window called **EPOC Emulator** start, with the appearance of a Crystal-based device.

When the emulator has started up, you may find that you are looking at the **Extras** or **File Manager** application, and that it appears that **Hello** has not

started at all. In fact, it *has*. There is a slight timing problem: the VM has started **Hello** and drawn its frame, over which the emulator has then drawn the other applications. So, keep closing the topmost application until you see the program as in the following figure:

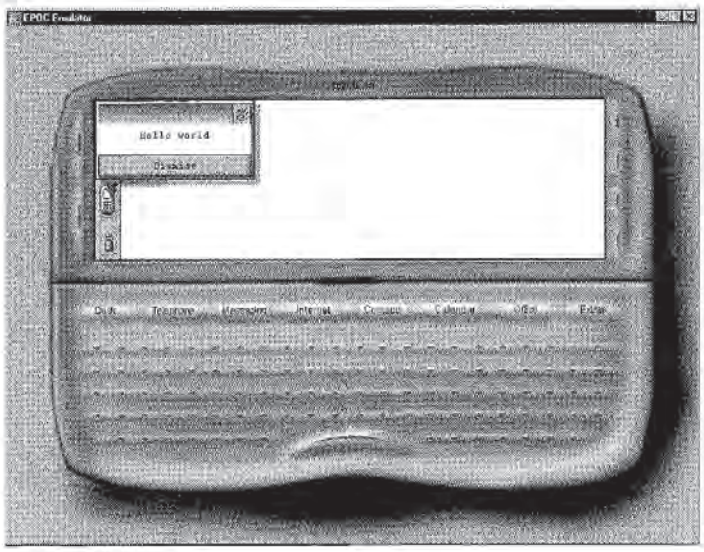


Figure 1.1 The Hello program running.

The **AppRun** batch file is as follows:

```
%epocroot%epoc32\release\wins\udeb\pjava_g -cd
j:\wjsp\Hello\code
com.symbian.devnet.crystal.hello.Hello
```

The directory **epoc32\release\wins\udeb** holds the debug versions of the emulator's executable files, one of which is the debug PersonalJava VM, **pjava_g.exe**. The possible options are mostly the same as for the standard **java/java_g** VMs. You should note though the Symbian-specific option `-cd`, which sets the current directory. This option is necessary as Symbian OS does not have the concept of a current working directory in the way that other operating systems do, so it must be set explicitly to allow the classes to be found.

To run **Hello** in the release version of the emulator, which is quicker to start, you would use **pjava.exe** in the **urel** directory, i.e.

```
%epocroot%epoc32\release\wins\urel\pjava -cd
j:\wjsp\Hello\code
com.symbian.devnet.crystal.hello.Hello
```

1.2.4 Closing a Program

In a real Crystal program, you would provide a **File Exit** command, or similar, for the user to quit. While you are developing, you have also got a couple of other options. One is the **Close** command on the so-called Java menu, which is automatically added to the title bar of Java applications.

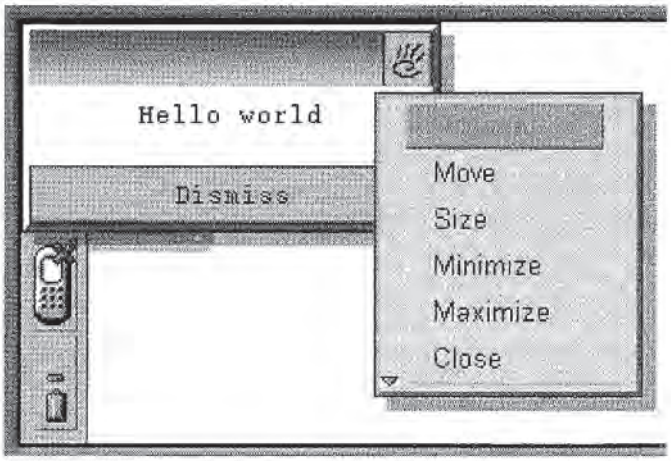


Figure 1.2 The Java menu.

This menu is provided to allow users to manipulate windows, usually of Java programs that have not been specifically written for the Symbian OS, and so may need resizing to fit the screen well. Its use should not be required, though, on well-written programs designed for Symbian devices.

Returning to closing down, the other method is simply to shutdown the emulator window in the normal Windows ways. Notice that when you shutdown a Java program, however you do it, the emulator also shuts down. This results from the way the emulator models the multi-process environment of a real machine with a single Windows process—an aspect of the emulation that otherwise has little impact on Java programming.

1.2.5 Running Hello from a Shell Program

To make an application available, a shell program needs to know a bit more information than can be supplied with class or Jar files. Specifically, it needs ways to get:

- a unique identifier number (UID) for the application
- the parameters that should be passed to the Java runtime to launch the application
- the caption to use (if not the default)
- the icons to use (if not the default).

These pieces of information are supplied by various additional files that you must supply. The files are put in a directory dedicated to the application in **\system\apps**. In general, you should also put your application's Jar files, classes, and DLLs into this directory. This reduces the risk of name clashes, makes application removal easier, and perhaps most importantly, unambiguously identifies the class or Jar file with the application. If you need to share code between applications, you can put it in the **ext** or **classes** directories in the root of any drive. The **ext** directory is for Jar files that are used by several applications (e.g. **javaphone.jar**), while the **classes** directory is for shared classes. In general your application should not use these directories. The **lib** directory is reserved for implementation specific classes such as **classes.zip** and properties files.

If you've used the EPOC Release 5 SDK, you may have postponed creating these files, as it was undeniably laborious, with hand-coding of files and command lines tools. Fortunately, Version 6 has introduced a Windows-based GUI tool, called **AIF Builder**, that makes the process much simpler.

Start **AIF Builder** from the **Symbian 6.0 SDKs\Development tools** program group, and do **File, New** to begin a new project. We'll now briefly go through the options, as you would use them to add **Hello** to **Extras**. The files that this would generate are already in the **Hello** directory, but you can do it yourself and overwrite these if you would like to follow step by step.

The initial page is **Application**:

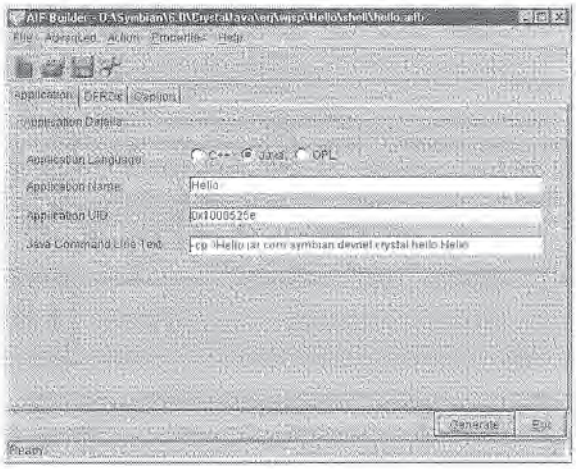


Figure 1.3 AIF Builder application page.

Set **Application Language** to **Java**, so that the tool shows only the options relevant to Java.

Set **Application Name** to **Hello**. You can use any name you like—it doesn't have to be the same as the **.class** or **.jar** file, but, by default, the name is used as the caption by which to identify the application to the user.

Set **Application UID** to a globally unique 32-bit number. To ensure that each UID is genuinely unique, the Symbian Developer Network allocates UIDs from

a centrally administrated database. For details, see ***http://www.symbiandevnet.com/techlib/techcomms/techpapers/papers/tn_uid/uidinfo.html***. For our program the UID is 0x100052E.

Set **Java Command Line Text** to the arguments that you want to pass to the Java runtime. In the case of **Hello**, we want to get the `Hello` class from the **Hello.jar** file, so we need to put that on the class path with the `-cp` option. The shell sets the current directory by default to the application's directory, so if that's where the **.jar** file is, the command line is:

```
-cp .\Hello.jar
com.symbian.devnet.crystal.hello.Hello
```

Alternatively, if the class files themselves were put in the application directory, the command line would just be:

```
com.symbian.devnet.crystal.hello.Hello
```

Note that if the emulator shuts down when you attempt to start your application from the shell, it's likely that the command-line text is wrong.

The **DFRDs** page is mostly concerned with application icons:

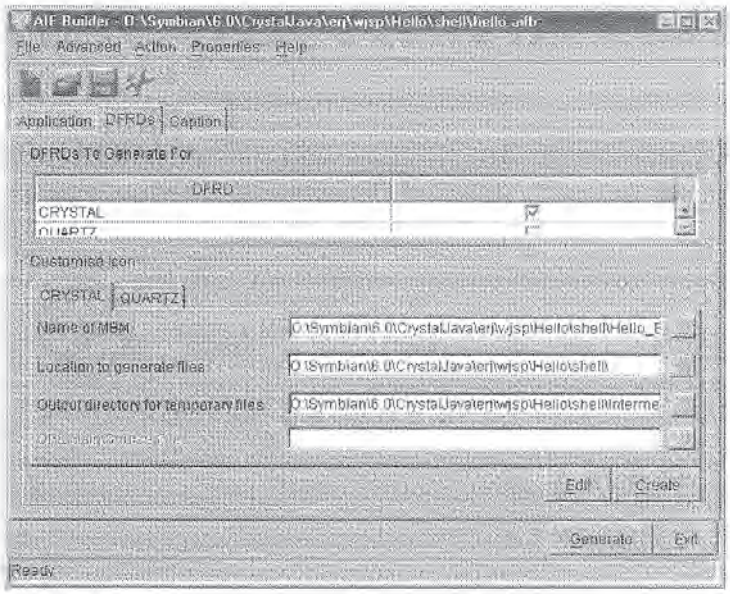


Figure 1.4 AIF Builder DFRDs page.

Again, we'll go through the options used for **Hello**:

In **DFRDs To Generate For**, deselect **Quartz**, as we're only going to be creating files for Crystal. Then, set the options on the **Crystal** tab. The only thing that this choice really affects is the size of the application icons that are acceptable, as these are different in the two device families.

Set **Name of MBM** to the file that holds the application icons. **MBM** refers to the Symbian-specific multi-bitmap (**.mbm**) file format. The easiest way to

create such a file is to draw the icons in the **AIF Icon Designer** program, which you can launch through the **Edit** or **Create** commands. A suitable **.mbm** file is **Hello\shell\Hello_BMPs\Hello.mbm**.

Set **Location to generate files** to the directory where **AIF Builder** should write the files it creates. This doesn't really matter, as we're going to move them afterwards. For **Hello**, we'll use a sub-directory named **shell**.

Likewise, **Output directory for temporary files** isn't important, so we'll leave it blank to accept the default value.

The final page is **Captions**.

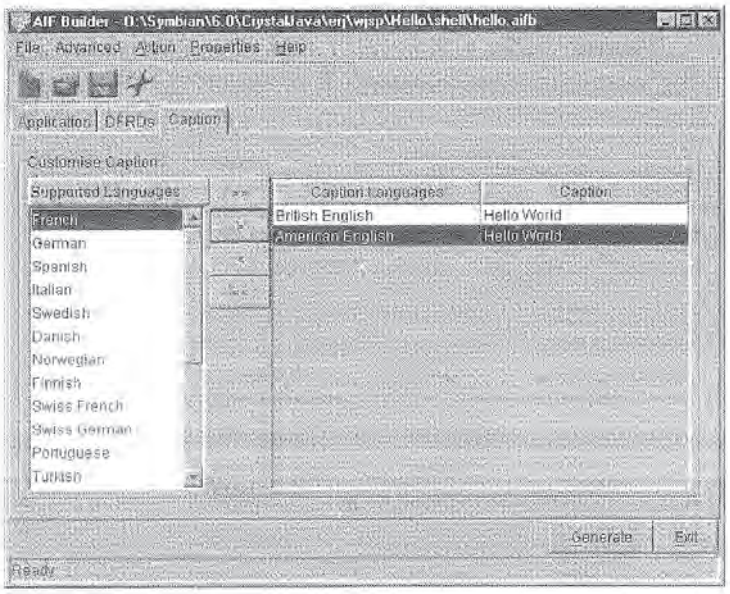


Figure 1.5 AIF Builder captions page.

If you're developing just for a single language, and you're happy that your application name is suitable to show to end-users, then you don't have to do anything. If you want the shell to show different captions, depending on the locale of the device, then you enter them here. For **Hello**, we've set the caption for American and British English locales to be "Hello World".

That's just about it. Save the project, and press the **Generate** button and after a short time, **AIF Builder** will tell you that it has created the following files:

Hello.app
Hello.txt
Hello.aif

and the temporary file **Hello.rss**. Now go back to a command-prompt at the **Hello** directory. There's a final batch file to run, **WinsInstall.bat**, that copies all these files, together with **Hello.jar**, into a directory for the Hello application on the emulator's C: drive, i.e. **epoc32\release\wins\c\system\apps\Hello**.

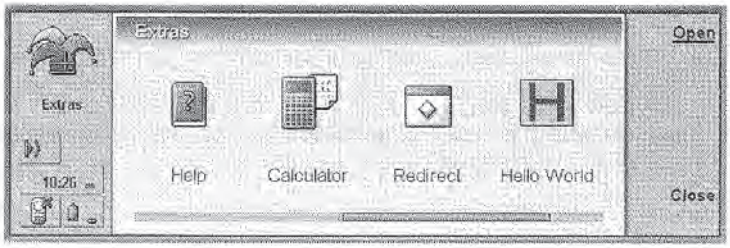


Figure 1.6 Hello on extras application.

To check that it all works, run the emulator and go to the **Extras** application. You should see something like Figure 1.6.

1.2.6 Writing to the Standard Output Stream

A problem that might strike you when launching applications from a shell program is that, without a command prompt, where do the messages printed to the standard output steam go? The short answer is that for output to end-users, *don't* use standard output. For development, such as for debugging, you can redirect the stream to a number of destinations. If you open **Extras**, you will see an application called **Redirect**, which allows you to view standard out on a console window, or redirect it to a file or serial port.

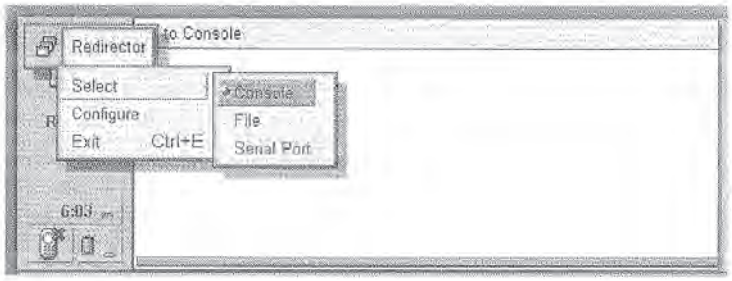


Figure 1.7 The redirect application.

1.2.7 Installing Programs on Target Devices

We've seen that, for a fully developed application, you are likely to produce quite a number of files, some of which require being put in particular directories. While you are developing, and you wish to test on a real target device, just copying them from the development machine through a connectivity product is probably adequate. To deliver to end-users though, you'll want the files properly packaged in a simple-to-use installation process. The Symbian tool for this is the Installation File Generator (**makesis**), which takes a text package file (**.pkg**) that lists the files to be transferred, and produces a single installation file (**.sis**). The end-user runs this file on their device, or on a connected Windows PC, to install the application. Full details of the Installation File Generator are

in the SDK, so we'll limit ourselves to a simple example here. EPOC Release 5 users might particularly like to note the introduction of signed installation files for secure installs.

A package file for **Hello** is given in **Hello\install\Hello.pkg**:

```
#{"Hello"},(0x1000525e),1,0,0
"..\Hello.jar"-"!:\system\apps\Hello\Hello.jar"
"..\shell\Hello.txt"-"!:\system\apps\Hello\Hello.txt"
"..\shell\Hello.app"-"!:\system\apps\Hello\Hello.app"
"..\shell\Hello.aif"-"!:\system\apps\Hello\Hello.aif"
```

The first line specifies the package header. In a simple installation, this consists of the application's name, its UID, and its major, minor, and build numbers. The other lines specify the files to put in the installation file, and their target locations. The target drive will be (probably) selected by the user. This is indicated by specifying ! as the destination drive.

You can now create the installation file by going to the command line, moving to **Hello\install** and running:

```
makesis Hello.pkg
```

Hello.sis file is created in the current directory.

That concludes the basics of how to build, run, and install a basic Java application on Version 6. More interesting applications will probably need to use the JavaPhone APIs specially designed for WID capabilities. We'll take a brief look at these next.

1.3 A JavaPhone Program

JavaPhone is a number of related APIs, defined by Sun Microsystems, that offer facilities particularly suited to mobile telephony and Internet devices. There's considerable flexibility for implementers in which parts of the APIs to support. Details of the Symbian implementation and each JavaPhone API are discussed later in Chapter 5.

Basic telephony functions to create, answer, and disconnect calls are provided by the JTAPI Core API (`javax.telephony`). It's accompanied by JTAPI mobile (`javax.telephony.mobile`) to manage network, transmission, and signal features that are unique to mobile telephony devices.

On the personal information management (PIM) side of things, there's an Address Book API (`javax.pim.addressbook`) to manage contacts information, which works in a compatible way with the Internet Mail Consortium's vCard standard. The User Profile API manages information about the current user's own contact information. There's a Calendar API (`javax.pim.calendar`) for schedule and calendar information, compatible

with the IETF's iCalendar standard. The PIM APIs use the same simple database model, defined in `javax.pim.database`.

For comms, the Communications API (`javax.comm`) provides access to a device's serial and parallel ports. The Datagram API (`javax.net.datagram`) supports Datagram transports such as UDP, SMS, and WAP's WDP.

Finally, the Power monitoring API (`javax.power.monitor`) is used to monitor the power level of the system.

The next example program, **JPhoneIntro**, shows four of these APIs — JTAPI Core, Datagram (SMS), Address Book, and Calendar — in action.

1.3.1 Hardware Setup on the Emulator

To test the JTAPI and Datagram API examples, you will need access to an Ericsson i888 phone (for the US or Europe) or an Ericsson SH888 phone (for Europe) and an Extended Systems JetEye IR podule for your PC. Other phones which appear to work, though haven't been fully tested, are the Ericsson T28 and the Ericsson R320.

If you want to use the serial port, either for direct serial communications using `javax.comm`, or to drive the IR link to a mobile phone, you must ensure nothing else is using the port (such as PsiWin and the PsiWin connection server).

Using the IR port emulation may require modifications to a setup file called **irda_wins.esk**, found in **\epoc32\wins\c\system\data**. It contains a section like this:

```
filename= irda.prt
index= 2
irPod= jeteye7401
irBaud= 115200
irPhysicalComPort= 1
irSupportedBaudRates= 319
```

`irPhysicalComPort` refers to the COM port as seen by the Emulator, which counts ports from 0. So, set this value to 0 if you have plugged your IR podule into COM1 on your PC, and to 1 if you have plugged it into COM2.

If the IR podule link seems unreliable, try changing `irBaud` to, say, 57600.

The Ericsson 888 requires some setting up:

1. Ensure that its phone number is set. Most phones do not know their own phone number, so menu through to **Settings, Yes**, menu through to **Phone number, Yes, Yes**, enter phone number. From then on, the phone number can be read off the SIM card.
2. You need to enable the IR port. Go to **Activate IR port** and select **Yes**. Note that the IR port is disabled when you switch the phone off.
3. Instead of using the Extended Systems Jeteye pod, you can use the serial cable that comes with the Ericsson phone. Everything else remains the same; in particular, the IR port must still be enabled on the phone.

1.3.2 The JPhoneIntro Program

We'll now take a brief look at the application's main classes. If you want to study the code more, it's documented in **JPhoneIntro\docs**.

`JPhoneIntro` is the main class. It creates instances of the user interface class (`JPhoneIntroFrame`) and the engine (or model) classes that directly use the JavaPhone APIs. Its only significant point is that it solves the slight annoyance of being covered up by later applications by introducing an optional delay at program startup, to ensure that it is drawn last:

```
static public void main(String[] args) throws
InterruptedException
{
   if (args.length > 0)
   {
            String s = args[args.length-1];
            if (s.equals("wait"))
            {
            Thread.sleep(20000);
            }
   }
   new JPhoneIntro();
}
```

`JPhoneIntroFrame` is the user interface class. It creates four panels, one for each of the JTAPI Core, Datagram, Address book, and Calendar APIs. For each panel, there are a couple of labels, a simple text input field, and a button that makes a call to the relevant API.

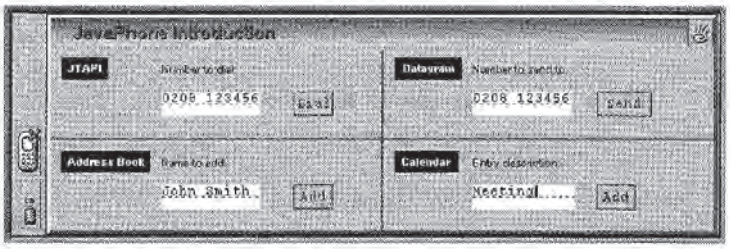


Figure 1.8 The JavaPhone intro user interface.

Finally, there are four model classes, `JTAPIIntroModel`, `DatagramIntroModel`, `AddressBookIntroModel`, and `CalendarIntroModel`, each of which uses a single JavaPhone API, as indicated by their names. Each class has a similar form: let's take `AddressBookIntroModel` as an example.

```
package com.symbian.devnet.crystal.jphoneintro;

import javax.pim.addressbook.*;
import javax.pim.database.*;
```

```
/**
 * Provides a simple example of using the JavaPhone Address Book
 * API: creates a new contact with the specified name
 */

public class AddressBookIntroModel
{
    /** The address book database */
    private ContactDatabase addressBook;

    /** Open the default address book database */
    public AddressBookIntroModel()
    {
        try
        {
            addressBook=ContactDatabase.openDatabase();
        }
        catch(DatabaseException dbe)
        {
            System.out.println("Can't open the address book
                    database: " + dbe.getMessage());
        }
    }

    /**
     * Add a new entry to the address book *
     * @param firstName First name
     * @param secondName Surname
     */
    public void addPerson(String firstName, String secondName)
    {
        AggregateField name = new
        AggregateField(ContactDatabase.N);
        name.addField(new ItemField(ContactDatabase.N,
                secondName));
        name.addField(new ItemField(ContactDatabase.N,
                firstName));

       try
       {
            ContactCard contact = new ContactCard();
            contact.addField(name);
            addressBook.addItem(contact);
       }
       catch(DatabaseException dbe)
       {
            System.out.println("Unable to create a new
                    contact: " + dbe.getMessage());
       }
    }

    /** Close database */
```

```
    public void close()
    {
        try
        {
            if(addressBook !=null)
            {
                addressBook.close();
            }
        }
        catch(DatabaseException dbe)
        {
            System.out.println("Unable to close the Contact
                database: " + dbe.getMessage());
        }
    }
}
```

The constructor performs whatever initialization is required to use the API. In the case of the Address Book API, that means opening the address book database (`ContactDatabase`). There's one action method that actually uses the API to carry out a task. For `AddressBookIntroModel`, that's `addPerson`, which takes two strings (first and second name of a person), puts these in a contact's entry name field, and then creates a new contact entry with that name. The `close` method carries out any clean-up required when use of the API is complete. Here, the database is closed.

1.3.3 Building and Running JPhoneIntro

You should have run the emulator's Calendar application at least once after installing the SDK before running JPhoneIntro. This ensures that the required database is created. If you don't, you will get an error "Can't open the calendar database: null."
If you get an error, "Can't get a provider: Javax.telephony.ProviderUnavailable-Exception: Provider not available," the emulator can't connect to your mobile phone, and you should check the setup.

The process for building and running **JPhoneIntro** is very similar to that for **Hello**. Again, batch files have been provided to do most of the work.

To build, start a command prompt window, go to the **JPhoneIntro** directory, and just type `Build`. The only difference from the **Build.bat** used for **Hello** is that `%epocroot%erj\ext\javaphone.jar`, the location of the JavaPhone classes, is added to the class path.

To run the application in the debug emulator, type **Apprun**. After a short time, the **JavaPhone Introduction** window will appear. The various panels work as follows:

JTAPI: enter a phone number (as you would dial it from your mobile phone) and press **Dial**. The emulator connects to the phone and rings the number.

Datagram: enter a phone number that can receive SMS messages and press **Send**. The emulator connects to the phone and sends an SMS message with the text "Test message from Java".

Address book: enter a name (first name then last name) and press **Add**. The emulator makes a new entry with this name in the emulator's Contacts database. You can check the entry has been made by opening the **Contacts** application.

Calendar: enter a description for a calendar entry and press **Add**. The emulator makes a new appointment with this description at the current time, of one hour in length. You can check the entry has been made by opening the **Calendar** application.

You can close **JPhoneIntro** as before, either from the Java menu or by closing the emulator window.

1.4 A Few Guidelines for Developing Applications

Having got your first application up and running, you now need to ensure that you continue to write maintainable code, both for yourself and others who will use your code. Our guidelines will help you with this.

1.4.1 Use Packages

Our very first example was created in a package: `code.com.symbian.devnet.crystal.hello`. This was done to emphasize how important it is to work with named packages.

Here are a few of the benefits:

- it makes it easier to partition even a medium sized project
- it supports the object-oriented principal of encapsulation
- it prevents name clashes
- it allows you to expose only what you want and so helps with security: with the default package all variables and methods are effectively public, unless they are specifically marked as private.

1.4.2 Keep Source and Code Files in Separate Directory Structures

The batch file to build our **Hello** demonstration contained the line

```
javac -classpath %localclasspath% -d code
source\*.java
```

This command takes all the Java sources in the **source**\ directory and, thanks to the `-d code` switch, puts the compiled class into the **code**\ directory. The **code**\ directory must already exist, but the subdirectories are created automatically: **Hello.class** ends up in **code\com\symbian\devnet\crystal\hello**\.

However, the command would fail if there were files in different packages that reference each other. For example, if we have a class in `mainpackage` that references a class in `secondpackage`,

```
package mainpackage;
import secondpackage.*;

public class MainPackageClass
{
  public static void main(String[] args)
  {
     SecondPackageClass spc
       = new SecondPackageClass(new MainPackageClass());
  }
}
```

and the class in `secondpackage` references our class in `mainpackage`,

```
package secondpackage;
import mainpackage.*;

public class SecondPackageClass
{
   MainPackageClass mpc;

   public SecondPackageClass(MainPackageClass mpc)
   {
       this.mpc = mpc;
   }
}
```

then the source directory tree must reflect the package structure, as shown in Figure 1.9.

To compile both classes, you have to change to the **Source**\ directory and compile with:

```
javac -d .. \Code mainpackage\*.java
```

Alternatively, try to avoid mutual references! In general, a main class should depend on support and library classes. Dependencies the other way around should be designed out if possible.

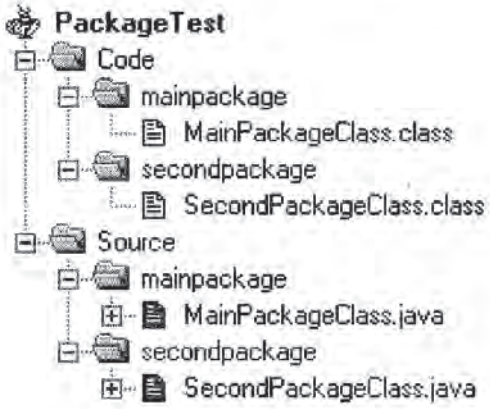


Figure 1.9 File structure required where `MainPackageClass` and `SecondPackageClass` reference each other.

1.4.3 Coding Conventions

The Java conventions we recommend are taken from broad industry standards. The motivation is that as Java is a widely used language — Sun claims over two million developers — where possible, we employ standards that Java developers are used to. This is perhaps in contrast to Symbian's C++ environment, which, because it is more specialized, uses Symbian-specific constructs and coding conventions to minimize the risk of memory leaks and coding errors.

Here's a summary of Symbian's Java conventions:

- class names start with an upper case letter
- variables and methods start with a lower case letter
- final variables use upper case, with underscores used to separate words
- package leaf names are generally all lower case
- one brace per line.

This code fragment illustrates some of these practices:

```
package com.symbian.devnet.awt;

import java.awt.*;
import java.awt.event.*;

/**
* The EFrame class is intended to provide some of the
* functionality needed to make a Java application look and behave
* like a C++ EPOC application
*
* @author Jonathan Allin. (C) Symbian Ltd 1999, 2000
* @version 0.1, 25/04/99
*/
```

```
public class EFrame extends Frame implements ActionListener
{
    public MenuBar menuBar = new MenuBar();
    public AboutDialog aboutDialog;
    static final int BORDER_WIDTH = 5;

/**
* A sub class of EFrame that implements actionPerformed should
* call super.actionPeformed(ActionEvent)
*/
    public void actionPerformed(ActionEvent ae)
    {
        if(ae.getSource() == closeItem)
        {
;
;
```

A few more points to note from this sample:

- for object names, it is more usual to use the class name with a lower case initial letter, rather than prefixed by "my", e.g. use `menuBar` rather than `myMenuBar` or `aMenuBar`
- indentation is with tabs, not spaces. Users can then set the number of spaces per tab in their editor to suit their own preferences
- C++ guidelines recommend a space between an `if` statement and its clause in order to avoid potential problems with macro expansions. This is not a problem in Java so we can write `if(ae.getSource() == closeItem)`.

Finally, take advantage of JavaDoc to document variables, methods, classes, packages, and the application itself. The JDK documentation describes the syntax at ***docs\tooldocs\win32\javadoc.html***.

To create the JavaDoc for our Hello application, change to the **source** directory and enter:

```
javadoc -d ..\docs
com.symbian.devnet.crystal.hello
```

This will generate HTML documentation in the **docs** directory at the same level as the **source** directory.

1.4.4 Internal Symbian Coding Standards

In addition to the above ideas, Symbian's developers prefix names with particular letters, in order to help with code development. We don't use prefixes in this book because if you're not used to them, they can make the code a little harder to read.

These are the prefixes:

- instance variables are prefixed with an "I".
- method arguments are prefixed with an "a".
- static variables are prefixed with an "s".

The following example doesn't use prefixes:

```
class Demo
{
   String name;
   int value;
   static float interestRate;

   public Demo(String name)
   {
      this.name = name;
   }

   public boolean checkValue(int check)
   {
      int value = 0;
      :
      :
      if(check > value) return true;
      else return false;
   }
}
```

The `if` statement in `checkVal()` compares `check` with `value`. Which `value` was intended - the local variable or the instance variable?

This version does use prefixes:

```
class Demo
{
   String iName;
   int iValue;
   static float sInterestRate;

   public Demo(String aName)
   {
     iName = aName;
   }

   public boolean checkVal(int aCheck)
   {
   //  int value = 0;
     :
     :
     if(aCheck > iValue) return true;
     else return false;
   }
}
```

And the ambiguity is removed. It also replaces `this.name = name` with `iName = aName`, which some people prefer.

For your own projects, think about adopting these, or similar conventions.

1.4.5 Java Coding Conventions on the Web

Sun's conventions are widely adopted and can be found at ***http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html***.

Netscape's Software Coding Standards Guide for Java is located at ***http://developer.netscape.com/docs/technote/java/codestyle.html***.

Further examples are Java Programming Style Guidelines: ***http://www.geosoft.no/javastyle.html*** and AmbySoft Inc. Coding Standards for Java v17.01d: ***http://www.ambysoft.com/javaCodingStandards.pdf***.

2

Java on Crystal

Jonathan Allin and John Bown

The Crystal user interface has many features that distinguish it from the UIs of other devices for which you may have developed. Outstanding amongst these are:

- a small screen (640 × 200), though large by WID or PDA standards
- no mouse or other pointer input
- additional input mechanism through soft keys (programmable hardware buttons).

These considerations mean it is unlikely that a user interface designed for a PC will give an effective user experience. Instead, it is recommended that you design your user interface taking careful account of the device's UI features. A more detailed examination of Symbian UIs and reference designs can be found in Chapter 7.

The best way to do this is to follow the Crystal User Interface style guide in the SDK. Those guidelines show how a native Crystal C++ application should appear. Java applications can, and should usually, aim for a similar look. AWT helps you achieve this by using the native user interface controls. Additionally, Symbian APIs are provided for handling soft key input, the virtual cursor, and the status pane.

2.1 An Introduction to Crystal

2.1.1 Screen Capabilities and the Java Menu

Crystal has 12-bit color (4096 tones), so applications can display high-quality images.

To maximize the use of the limited screen space, Crystal applications typically have a single, full screen window, rather than multiple document interfaces. Moving and resizing windows is not something users typically need to do. Instead, developers should try to ensure that windows are the correct size for the screen.

It may, however, be useful, during development and porting of applications, to have multiple windows. When executing a Java application on a Crystal device, or emulator, pressing the CTRL and SPACE keys together will bring up a menu, which allows the user to minimize, resize, move or close a Java program, and to enable the virtual cursor (see below).

2.1.2 Application Control Without a Pointing Device

Crystal devices do not support a mouse or pen, or any other kind of hardware pointing device. The absence of a hardware pointer means that users cannot click on controls to select or activate them. The advanced pointer idioms of right-clicking, double-clicking, dragging, and pointer-over events are of course equally unavailable.

It is therefore desirable to design Crystal applications so that the user interface can be easily operated without a pointing device. In general this will entail the use of soft keys and the menus, and avoiding controls that require "clicking". Examples are buttons, checkboxes, radio-buttons (checkbox groups) and scroll-bars.

Where it is not possible, or not desirable, to create a pointer free design, a virtual (software) cursor can be utilized to provide a 'point and click' mechanism for interacting with controls which control the application.

Controls can be traversed using the standard Java focus traversal keys of Tab and Shift-Tab. This works well if only two or three controls need to be traversed, but can be cumbersome and confusing if there are many components in the UI.

I frequently use a TextArea to display status and debug information for my applications. Calling `setEditable(false)` on this TextArea means that it cannot be edited by the user. If instead I was to call `setEnabled(false)` this would, in addition, mean that the TextArea could not accept the focus. It then becomes impossible to scroll the TextArea (even with the virtual cursor described below).

2.1.3 Soft Keys and the Command Button Area

A Crystal device has soft keys which the user can use for input, typically for a rapid choice of common commands. Crystal programs use part of the screen, referred to as the Command Button Area (CBA), to display the commands that the soft keys carry out. When a user presses a soft key, the program receives this as an event. Java programs can create a CBA and receive events from the keys using the Command Button Area API.

2.1.4 Status Pane

The status pane shows system status. This includes information such as connection, battery, field strength indicators and infra-red connection status. The status pane API is used to control the presence, or absence, of the status pane, and its size.

2.1.5 Virtual Cursor

Crystal provides a virtual (software) cursor that the user can control with the keyboard. This can be used to provide "point and click" capabilities for Java applications, but is not recommended for use if a pure Crystal "look and feel" is required; most non-Java applications do not have access to a virtual cursor.

No special programming is needed to handle the virtual cursor. The user can always enable the virtual cursor, and an application receives its actions as normal pointer events. Programs can also enable the virtual cursor with the virtual cursor API.

2.2 The Crystal APIs

2.2.1 Packages and Class Files

There are a number of classes that are specific to Crystal which are part of the com.symbian.epoc.awt package:

- com.symbian.epoc.awt.CKONToolkit
- com.symbian.epoc.awt.EikCommandButtonGroup
- com.symbian.epoc.awt.CBAListener
- com.symbian.epoc.awt.CBAEvent
- com.symbian.epoc.awt.EikStatusPane
- com.symbian.epoc.awt.EikVirtualCursor

These classes are included in the **classes.zip** file delivered with the Crystal Java runtime.

2.2.2 The Crystal Toolkit APIs

The Crystal Toolkit API is used to determine information that is specific to the Crystal device and not available elsewhere. The Toolkit consists of a single class that is used to determine the maximum screen area that a Java application has to use.

com.symbian.epoc.awt.CKONToolkit

The `CKONToolkit` class provides the means for discovering the usable screen area.

getAvailableScreenRect() Use `Rectangle getAvailableScreenRect()` to determine the dimensions of the maximum screen area available for use by an application.

```
Rectangle r = CKONToolkit.getAvailableScreenRect();
int width = r.width;
int height = r.height;
```

The maximum usable screen area is the screen area rectangle between the status pane and the CBA.

An application can be made to fill the maximum available screen area as follows:

```
show();
setBounds(CKONToolkit.getAvailableScreenRect());
```

Note that it is necessary to make the application visible (by using `show()`) before invoking `getAvailableScreenRect()`.

2.2.3 The Command Button Area APIs

The Command Button Area is the recommended method for allowing the user to choose commands simply and rapidly. Crystal provides a class that allows Java programs to create Command Button Areas and receive events from them. The API can be used by both Applets and applications.

com.symbian.epoc.awt.EikCommandButtonGroup

The main CBA class is `EikCommandButtonGroup`, which extends `java.awt.Component`. An application creates an instance of this class, and adds it to its main Container (Applet, Window, or Frame). You do not have to create individual buttons within the CBA. An application or Applet only ever has a single `EikCommandButtonGroup`.

The following code fragment shows how to create a CBA:

```
class SimpleApplet extends Applet
{
    EikCommandButtonGroup cba = null;
    public void init()
    {
        cba = new EikCommandButtonGroup();
        add(cba);
```

getButtonCount() Use `int getButtonCount()` to determine the number of soft keys provided by the CBA.

```
int buttonCount = cba.getButtonCount();
```

The value returned is the actual number of soft keys available; zero indicates there are no soft keys. The generic Crystal reference design has four soft keys.

Button Definitions The generic Crystal reference design has four soft keys in the CBA, for which constants have been defined:

```
final static int EikCommandButtonGroup.BUTTON1 = 1;
final static int EikCommandButtonGroup.BUTTON2 = 2;
```

```
final static int EikCommandButtonGroup.BUTTON3 = 3;
final static int EikCommandButtonGroup.BUTTON4 = 4;
```

setText() Use `void setText(int, String)` to assign a label to a soft key, which will be shown in the CBA on the screen.

```
cba.setText(EikCommandButtonGroup.BUTTON1, "Open");
cba.setText(EikCommandButtonGroup.BUTTON2, "Edit");
cba.setText(EikCommandButtonGroup.BUTTON3, "Options");
cba.setText(EikCommandButtonGroup.BUTTON4, "Close");
```

When a soft key has previously been set as the default button (using `setDefaultButton()`), invoking `setText()` will cause that soft key to no longer be the default button. If you wish to make that soft key the default button again, another `setDefaultButton()` invocation must be made.

getText() Use `String getText(int)` to determine the label assigned to a soft key.

```
String label = cba.getText(EikCommandButtonGroup.BUTTON1);
```

setEnabled() Use `void setEnabled(int, boolean)` to enable and disable the soft keys.

```
cba.setEnabled(EikCommandButtonGroup.BUTTON1, true);
```

enables the first soft key.

```
cba.setEnabled(EikCommandButtonGroup.BUTTON1, false);
```

disables the first soft key. When a soft key is disabled, the text label assigned to it, using `setText()`, will appear grayed-out (dimmed) in the CBA.

isEnabled() Use `boolean isEnabled(int)` to determine whether a soft key is enabled or disabled.

```
boolean enabled = cba.isEnabled(EikCommandButtonGroup.BUTTON1);
```

returns a boolean result which is true if the specified soft key is enabled.

setDefaultButton() `void setDefaultButton(int)` specifies a default soft key. When the user presses the return key, it will be as though the user had pressed the default soft key.

```
cba.setDefaultButton(EikCommandButtonGroup.BUTTON1);
```

sets the first soft key as the default button. When a soft key is set as the default, the text label assigned to it, using `setText()`, will appear underlined, in the CBA.

To return to the state where there is no default soft key, after a default has been set, it is necessary to use `setText()` on the current default button.

Note that there is no `isDefaultButton()` method. The application must keep track of which soft key has been set as the default button if its text label is to be changed using `setText()`, which will stop that soft key being the default button.

addCBAListener()
`void addCBAListener(CBAListener listener)` registers a listener for CBA events.

removeCBAListener()
`void removeCBAListener(CBAListener listener)` removes an existing listener for CBA events.

com.symbian.epoc.awt.CBAListener and com.symbian.epoc.awt.CBAEvent

To detect when the user presses a CBA button, a program creates a class that implements the `CBAListener` interface, i.e. it must implement the method:

```
public void cbaActionPerformed(CBAEvent cbae)
```

The following code fragment creates an `EIKCommandButtonGroup`, adds it to the application's Frame, adds the application as a CBAListener to the EIKCommandButtonGroup, and implements the CBAListener interface:

```
import com.symbian.epoc.awt.*;
import java.awt.*;

class MyApp extends Frame implements CBAListener
{
    EikCommandButtonGroup cba = new EikCommandButtonGroup();

    public static void main(String[] args)
    {
        new MyApp();
    }

    public MyApp()
    {
        add(cba);
        cba.addCBAListener(this);
        show();
    }

    public void cbaActionPerformed(CBAEvent cbae)
    {
        System.out.println("cbaActionPerformed: " + cbae);
    }
}
```

Event listening is implemented using the `java.awt.AWTEventMultiCaster`, so normal AWT behavior for handling multiple listeners is available.

When a soft key is pressed, `cbaActionPerformed()` is called with a `CBAEvent` argument. `CBAEvent` extends `java.awt.AWTEvent`. You can use `AWTEvent.getID()` to find which soft key has been pressed:

```
public void cbaActionPerformed(CBAEvent cbae)
{
    switch (cbae.getID())
    {
        case EikCommandButtonGroup.BUTTON1:
            doOpen();
            break;
        case EikCommandButtonGroup.BUTTON2:
            doEdit();
            break;
        case EikCommandButtonGroup.BUTTON3:
            doOptions();
            break;
        case EikCommandButtonGroup.BUTTON4:
            doClose();
            break;
    }
}
```

2.2.4 The Status Pane APIs

Crystal provides a class that allows Java programs to control the status pane. The API can be used by both Applets and applications.

The status pane API allows access to Crystal's status pane. The status pane provides information such as connection, battery, field strength indicators and infra-red connection status. Control is limited to varying the presence, or absence, of the status pane, and its size. There are two styles for the status pane, when present: a wide area of 92 × 200 pixels, and a narrow one of 32 × 200 pixels. The wide area has a place for time, the application's icon, and the application's name.

com.symbian.epoc.awt.EikStatusPane

The class for controlling the status pane is `EikStatusPane`. An application cannot create an instance of this class, but may use the static method provided by this class.

setStatusPaneStyle() Use `static void setStatusPaneStyle(int)` to set the style of the status pane. The native method takes an integer as an argument specifying the style of the status pane. The following code shows how to set the style of the status pane:

```
EikStatusPane.setStatusPaneStyle(EikStatusPane.NONE);
```

sets the style of the status pane to NONE. No status pane will be displayed.

```
EikStatusPane.setStatusPaneStyle(EikStatusPane.NARROW);
```

sets the style of the status pane to NARROW, 32 pixels wide.

```
EikStatusPane.setStatusPaneStyle(EikStatusPane.WIDE);
```

sets the style of the status pane to WIDE, 92 pixels wide.

2.2.5 The Virtual Cursor APIs

The virtual cursor is a useful UI extension providing users with a software pointer in the absence of a hardware pointer or for applications that explicitly require pointer events. For example, image maps on a web browser requiring pointer events can be handled by the virtual cursor: the virtual cursor is enabled and its actions are received as normal pointer events by the browser application.

com.symbian.epoc.awt.EikVirtualCursor

Crystal provides a class in the package `com.symbian.epoc.awt` that allows Java programs to control the virtual cursor and receive pointer events from it. The API can be used by both Applets and applications.

The class for controlling the status pane is `EikVirtualCursor`. An application cannot create an instance of this class, but may use the static methods provided by this class.

setState() Use `static void setState(int)` to turn the virtual cursor on and off.

```
EikVirtualCursor.setState(EikVirtualCursor.ON);
```

activates, or turns on, the virtual cursor.

```
EikVirtualCursor.setState(EikVirtualCursor.OFF);
```

deactivates, or turns off, the virtual cursor.

getState() Use `static int getState()` to find the state of the virtual cursor. This returns an integer that represents the state of the virtual cursor.

```
int state = EikVirtualCursor.getState();
if(state == EikVirtualCursor.ON)
{
    // Cursor is on
}
```

```
if(state == EikVirtualCursor.OFF)
{
    // Cursor is off
}
```

2.3 Crystal Utility Classes

The Symbian Developer Network has developed three utility classes to help with writing Crystal applications. They are provided in the `com.symbian.devnet.crystal.awt` package. The classes are:

CBAHandler	Provides a simple wrapper around the EikCommandButtonGroup class. Simplifies the job of providing multiple CBAs for different Dialogs or Frames
CFrame	A useful starting point for Crystal applications
GridPanel	A useful general-purpose utility that simplifies the use of GridBagLayouts

2.3.1 The CBAHandler Class

There are a few "gotchas" with using the EikCommandButtonGroup class to control the CBA:

- each application can only have one EikCommandButtonGroup, which makes it hard to have different CBAs for things like dialogs or multiple views
- the EikCommandButtonGroup must be added to the application's Frame prior to adding other Components
- given that an application can have only one EikCommandButtonGroup, a number of access methods should be static
- if the default CBA button's text is changed, it ceases to be the default. Conversely, if you want to unset a button's default status, without making another button the default, you have to change the button's text.

The CBAHandler class provides a wrapper class around EikCommandButton-Group to make it easier to use.

The CBAHandler class has both instance functionality and class functionality:

1. A CBAHandler instance stores information about the buttons: their text labels, whether the buttons are enabled or disabled, and the default button. In addition, it keeps a reference to the CBAListener class used by this instance.

2. The CBAHandler also contains static information, in particular a reference to a single instance of the underlying EikCommandButtonGroup class, a list of CBAHandlers, and a reference to the current, active, CBAHandler.

The XBombs game below shows the CBAHandler in practice. For now we'll go through the methods roughly in the order we'd encounter them in an application (the details can be found in the documentation for the com.symbian.devnet.crystal.awt package).

The first thing we need to do is add an EikCommandButtonGroup to the application's Frame. This should only be done once per application, using the static method `CBAHandler.add(Frame)`. This adds the private EikCommandButtonGroup to the Frame.

For each Frame or Dialog in our application, or more specifically for each unique set of CBA functionality we require, we need to create a CBAHandler:

```
CBAHandler(com.symbian.devnet.crystal.awt.CBAListener parent)
```

This takes a CBAListener as its parameter.

Using our CBAHandler instance we set up the CBA buttons using `setText(int buttonNumber, String text)`, `setDefaultButton(int buttonNumber)`, and `setEnabled(int buttonNumber, boolean enable)` as appropriate.

When our Frame or Dialog is opened we call `activate()` to make this the current CBAHandler, together with the referenced CBAListener.

`Deactivate()` is called to remove the current CBAHandler, whereupon the previous CBAHandler is activated. `Deactivate()` would typically be used when we close a Dialog (CFrame, below, uses this).

2.3.2 The GridPanel Class

The Crystal version of the GridPanel class provides a simple wrapper around the GridBagLayout class. It extends Panel, setting the layout manager to a GridBagLayout for maximum flexibility when laying out components. GridPanel uses an instance of GridBagConstraints, `constraints`, which is made public so that the layout's flexibility is not lost:

```
package com.symbian.devnet.crystal.awt;

import java.awt.*;

public class GridPanel extends Panel
{
    private GridBagLayout layout;
    public GridBagConstraints constraints;

    public GridPanel()
```

```
    {
            layout = new GridBagLayout();
            setLayout(layout);
            constraints = new GridBagConstraints();
            constraints.insets = new Insets(2, 2, 2, 2);
            constraints.anchor = GridBagConstraints.NORTH;
    }
```

GridBagLayout is the most complicated LayoutManager. GridPanel's `add()` method makes its use relatively straightforward:

```
    public void add(Component item, int gridx, int gridy)
    {
            constraints.gridx = gridx;
            constraints.gridy = gridy;
            layout.setConstraints(item, constraints);
            add(item);
    }
}
```

2.3.3 The CFrame Class

CFrame is a utility class that can be used as the parent class for a Crystal application. CFrame extends java.awt.Frame, providing the following additional functionality:

- sizes the application to the screen. This takes into account the status area and the CBA
- exit handling. Entering Ctrl E will close the application
- an About dialog
- menu entries to configure the Status pane, close the application, present the About dialog, and enable and disable the virtual cursor.

You don't have to use CFrame as it is, but feel free to adapt it to your own needs. However, if you do modify it, move it to your own package and keep it in your application directory, not in a shared directory.

Sizing the Application to the Screen

Both the Crystal and Quartz style guides state that applications should fill the screen. CFrame uses `getAvailableScreenRect()` to get the available screen size. The last two lines of CFrame's constructor are:

```
show();
setBounds(CKONToolkit.getAvailableScreenRect());
```

CFrame and the CBA

CFrame's constructor includes the code

```
CBAHandler.add(this);
```

to create a single instance of an EikCommandButtonGroup and add it to the application's Cframe.

CFrame itself does nothing else with the CBA; however, Cframe's About dialog shows how CBAs can be attached to additional Dialogs (XBombs below shows how an application's main Frame uses the CBA).

AboutDialog is an inner class to CFrame. It implements CBAListener and declares an instance of a CBAHandler:

```
public class AboutDialog extends Dialog
    implements KeyListener, CBAListener
{
    CBAHandler cba;
    :
```

The constructor creates a new CBAHandler, passing in this Dialog as the CBAListener. `setCloseText()` is a utility to set the label for the last CBA button to "Close" and make it the default button:

```
AboutDialog(Frame parent)
{
     :
     cba = new CBAHandler(this);
     cba.setCloseText();
}
```

`setVisible(boolean visible)` overrides `Dialog.setVisible()`. If `visible` is true, we activate the CBAHandler (we want to show the "About" dialog). If `visible` is false we deactivate it (we now want to hide the "About" dialog). Deactivating the About dialog's CBAHandler activates the application's CBAHandler. `setVisible()` also uses `CKONToolkit.getAvailableScreenRect()` to position the Dialog in the center of the screen, taking into account the width of the status area and the CBA area:

```
public void setVisible(boolean visible)
{
     if(visible)
     {
          cba.activate();
          Rectangle r =
              CKONToolkit.getAvailableScreenRect();
          Dimension dim = getPreferredSize();
```

```
            int y = r.y + ( ( r.height-dim.height ) / 2 );
            int x = r.x + ( ( r.width-dim.width ) / 2 );
            setBounds( x, y, dim.width, dim.height );
            validate();
        }
        else
        {
             cba.deactivate();
        }
        super.setVisible(visible);
    }
```

The About dialog implements KeyListener so that we can close it when the escape key is pressed. Calling `setVisible(false)` achieves this, and also deactivates the CBAHandler:

```
public void keyPressed( KeyEvent ke ) {}
public void keyTyped( KeyEvent ke ) {}

public void keyReleased( KeyEvent ke )
{
    if( ke.getKeyCode() == KeyEvent.VK_ESCAPE )
    {
        setVisible(false);
    }
}
```

We can also close the About dialog by pressing the ENTER key. This is picked up by the CBAHandler because the Close CBA button is the default. This also calls `setVisible(false)`:

```
public void cbaActionPerformed(CBAEvent e)
{
    :
    if (e.getID() == EikCommandButtonGroup.BUTTON4)
    {
        setVisible(false);
    }
}
```

Finally, back to CFrame itself. The event handler for the About... menu entry uses

```
aboutDialog.setVisible(true);
```

to make the About dialog visible and make its CBAHandler the current CBA.

Setting the Status Area's Width

The menu entries to set the width of the status area are handled by the ActionListener implementation:

```
public void actionPerformed(ActionEvent ae)
{
    :
    else if (ae.getSource() == noStatusPaneItem)
    {
        EikStatusPane.setStatusPaneStyle(EikStatusPane.NONE);
        setBounds(CKONToolkit.getAvailableScreenRect());
        validate();
    }
    else if (ae.getSource() == narrowStatusPaneItem)
    {
        EikStatusPane.setStatusPaneStyle(EikStatusPane.NARROW);
        setBounds(CKONToolkit.getAvailableScreenRect());
        validate();
    }

    else if (ae.getSource() == wideStatusPaneItem)
    {
        EikStatusPane.setStatusPaneStyle(EikStatusPane.WIDE);
        setBounds(CKONToolkit.getAvailableScreenRect());
        validate();
    }
}
```

2.4 XBombs: A Crystal Example

XBombs is a version of the well-known Bombs or Minesweeper game, written for the Crystal DFRD. The following discussion assumes that you're familiar with the game!

The XBombs game has been written to illustrate a number of Crystal related features:

- using the CBA Handler class
- creating a user interface for a graphical application that can be controlled by the keyboard. However, the game can also be controlled with a pointer, and so can easily be ported to the Quartz reference design
- updating graphics elements without flicker.

Different sized boards can be selected using the menu, while the arrow keys are used to move the Cursor (the black square in the screen shot shown in Figure 2.1) around the board. The Cursor uses an ImageFilter to highlight the active square: the ImageFilter is chosen to ensure that the contents of the active square are always visible.

Pressing the return key activates the default CBA button, which exposes the highlighted square. Pressing the space bar cycles through "mark as bomb," "mark as unknown," and "blank." Alternatively the CBA buttons can be used to mark the active square appropriately.

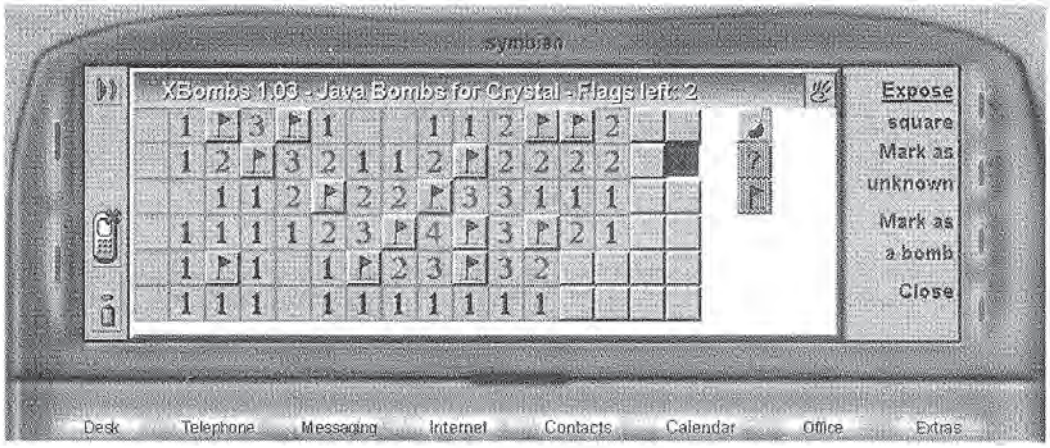


Figure 2.1 Nearly there!

The three buttons in a column to the right of the board are used for mouse control: clicking on, say, the "mark as bomb" button (the button with a picture of a flag) means that subsequent taps on unexposed squares will mark those squares with a flag. Of course, this isn't useful for Crystal devices, but as we said earlier, it does simplify porting to Quartz devices.

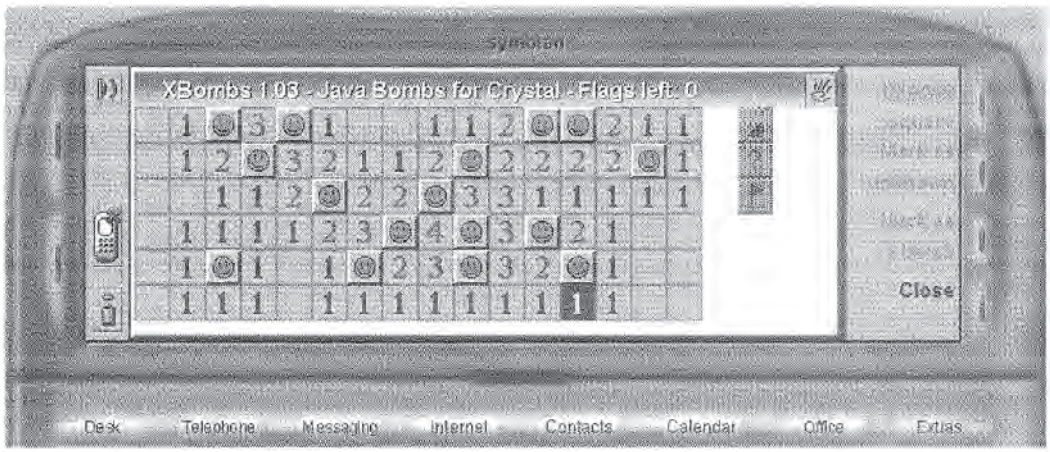


Figure 2.2 Success!

2.4.1 Game Architecture

We will look at features of XBombs that are specific to the Crystal reference design or relevant to small devices in general. The game is well documented so we don't need to go through all the code.

The principal classes that make up the game are:

XBombs

This is the main class. It does little more than set up the menus, create the Board, and receive CBA events (which it passes on to the Board).

Board

Board represents the playing board and the Squares that make up the board. It picks up keyboard and mouse input and contains the playing algorithms.

Board calls `repaint()` with an appropriate clipping region to ensure that only the required Squares are repainted.

Square

Square represents a square on the Board. Each Square has an image and a location. A Square also knows whether it is hiding a bomb, and the number of bombs adjacent to it.

Cursor

A Cursor stores information about the active, or highlighted, Square. The Cursor class is responsible for keeping track of the selected Square and creating a suitable Image to highlight the Square.

CursorImage

CursorImage is an ImageFilter used to create a highlighted image for the active Square.

Both the CursorImage and Cursor classes are defined in **Cursor.java**.

Option

An Option is one of the three mouse control buttons.

2.4.2 Implementing the ActionListener Interface

CFrame implements the ActionListener interface to handle menu events. Because the XBombs class is also an ActionListener, its `actionPerformed()` method must call `super.actionPerformed()` so that any unhandled events are passed on to the CFrame parent:

```
public void actionPerformed(ActionEvent ae)
{
    if(ae.getSource() == restartMenuItem)
    {
        :
    }
    else if(ae.getSource() instanceof MenuItem)
    {
        :
    }
    else super.actionPerformed(ae);
}
```

2.4.3 Using the CBA

All the code to handle the CBA is found in **XBombs.java**.

The game follows our usual packaging conventions, and so it's created in com.symbian.devnet.crystal.xbombs. We then import the packages that contain the CBAHandler classes and the Crystal specific classes:

```
package com.symbian.devnet.crystal.xbombs;
import com.symbian.devnet.crystal.awt.*;
import com.symbian.epoc.awt.*;
import java.awt.event.*;
import java.awt.*;
```

XBombs extends the CFrame utility class, implements ActionListener to listen for menu events, and CBAListener to listen for CBA events:

```
public class XBombs extends CFrame implements ActionListener,
CBAListener
{
```

We start by redefining the button definitions from EikCommandButtonGroup to simpler and shorter mnemonics, and then declare an instance of the CBAHandler utility class:

```
final static int EXPOSE = EikCommandButtonGroup.BUTTON1;
final static int UNKNOWN = EikCommandButtonGroup.BUTTON2;
final static int BOMB = EikCommandButtonGroup.BUTTON3;
final static int CLOSE = EikCommandButtonGroup.BUTTON4;
static CBAHandler cba;
:
```

In the constructor we create an instance of the CBAHandler, passing in the class that will listen for CBA events, i.e. that implements CBAListener (which happens to be this instance of XBombs). We then set the text for the CBA buttons, setting the EXPOSE button as the default.

```
:
cba = new CBAHandler(this);
cba.setText(EXPOSE,"Expose\nsquare");
cba.setText(UNKNOWN, "Mark as\nunknown");
cba.setText(BOMB, "Mark as\na bomb");
cba.setText(CLOSE, "Close");
cba.setDefaultButton(EXPOSE);
cba.activate();
:
```

`cbaActionPerformed()` is called when the user activates one of the CBA buttons. This uses the CBAEvent Id to identify the CBA button, and then takes the appropriate action:

```
public void cbaActionPerformed(CBAEvent cbae)
{
     switch (cbae.getID())
     {
          case EXPOSE:
               playingBoard.imgShowClicked();
               playingBoard.cbaExposeSquareClicked();
          break;

          case UNKNOWN:
               playingBoard.imgPossClicked();
               playingBoard.cbaMarkPossibleClicked();
          break;

          case BOMB:
               playingBoard.imgFlagClicked();
               playingBoard.cbaPlantFlagClicked();
          break;

          case CLOSE:
               super.shutDown();
          break;
     }
}
```

Note that when the Close CBA button is pressed we call `super.shutDown()`, rather than, say, `System.exit(0)`. In general, when an application that extends CFrame exits it should first clear up its own resources as necessary, then call `super.shutDown()`.

2.2.4 Graphics Drawing

A large part of XBomb's playability comes from the way the graphics provides an effective user interface, giving fast updating with no visible flicker. This section shows how we've achieved this.

Each Square is displayed as one of 16 possible icons in order to show its state. The first few states are valid during the game:

- an unexposed square
- a square with no adjacent bombs up to 8 adjacent bombs (9 different states)
- a flagged square
- a possible square (a question mark).

The remaining states are valid only at the end of the game:

- an exposed bomb
- an exploded exposed bomb (the one you trod on!)

- a correctly flagged bomb (a smiley face)
- an incorrectly flagged bomb (a flag with a cross through it).

There are also six other icons used for the three mouse control buttons.

The icons are loaded from the **Resources** subdirectory. They are loaded once by `loadImages()`, which is called from the Board's constructor:

```
private void loadImages()
{
    iImageNum[0] = loadImage("bombnum0.gif");
    iImageNum[1] = loadImage("bombnum1.gif");
    iImageNum[2] = loadImage("bombnum2.gif");
    iImageNum[3] = loadImage("bombnum3.gif");
    iImageNum[4] = loadImage("bombnum4.gif");
    iImageNum[5] = loadImage("bombnum5.gif");
    iImageNum[6] = loadImage("bombnum6.gif");
    iImageNum[7] = loadImage("bombnum7.gif");
    iImageNum[8] = loadImage("bombnum8.gif");
    iImageDuno = loadImage("bombduno.gif");
    iImageHere = loadImage("bombhere.gif");
    iImageBlow = loadImage("bombblow.gif");
    iImageFlag = loadImage("bombflag.gif");
    iImageOkay = loadImage("bombokay.gif");
    iImageNope = loadImage("bombnope.gif");
    iImagePoss = loadImage("bombposs.gif");

    iImageSsel = loadImage("bombshow.gif");
    iImageSoff = loadImage("bombsoff.gif");
    iImageFsel = loadImage("bombflag.gif");
    iImageFoff = loadImage("bombfoff.gif");
    iImagePsel = loadImage("bombposs.gif");
    iImagePoff = loadImage("bombpoff.gif");
}
```

`loadImage()` loads an individual icon from a JAR file, hence the use of a URL, and returns an Image:

```
private Image loadImage(String imageFilename)
{
    URL urlForImageFileName =
    getClass().getResource("Resources/" + imageFilename);
    return getToolkit().getImage(urlForImageFileName);
}
```

The game's algorithms are responsible for which Image should be displayed in a Square in response to the player's input, including exposing adjacent Squares as required (the algorithms are quite straightforward, take a look at **Board.java**).

The algorithms call `setImage()` when a Square's Image changes. It's called with the row and column coordinates of the Square, and the new Image (the Squares are stored in `squares[][]`, a two-dimensional array of Squares).

`setImage()` assigns the new Image to the Square: it checks if the cursor is at this Square and if it is calls `Square.setCursor()` to convert the new Image to the cursor image, and then calls `repaint()` with a clipping region equal to the coordinates of the Square. This is so that only the part of the Board covered by the Square is updated in subsequent calls to `paint()`:

```
private void setImage(int row, int column, Image requiredImage)
{
     if ( squares[row][column].currentImage != requiredImage )
     {
          squares[row][column].currentImage = requiredImage;
          if ( squares[row][column].hasCursor )
          {
              squares[row][column].setCursor();
          };
          repaint( squares[row][column].offsetX,
                   squares[row][column].offsetY,
                   Square.SIZE,
                   Square.SIZE
                  );
     }
}
```

`paint()` paints the whole board by calling `paintImage()` for every Square (it also paints the three mouse control buttons). However, only the region specified by the `repaint()` method is actually updated (as an aside, we've also taken the trouble to optimize the loops by counting down rather than up: see Section 8.10):

```
public void paint(Graphics g)
{
     :
     for( int row = numberOfRows; --row >= 0; )
     {
          for( int column = numberOfColumns; --column >= 0; )
          {
               squares[row][column].paintImage(g, this);
          }
     }
}
```

Finally `Square.paintImage()` paints its image onto the Board (which is a Panel). It will paint either the cursor Image or the normal Image:

```
void paintImage(Graphics g, Panel parent)
{
     if (hasCursor)
     {
          g.drawImage(cursorImage, offsetX, offsetY, parent);
     }
```

```
        else
        {
            g.drawImage(currentImage, offsetX, offsetY, parent);
        }
    }
```

We talked earlier about the cursor image. A Square obtains its cursor image by calling the static method `Cursor.getCursorImage()` with the Square's current image:

```
void setCursor()
{
    hasCursor = true;
    cursorImage = Cursor.getCursorImage(currentImage);
}
```

`getCursorImage()` uses an ImageFilter, `cursorFilter`, to modify the pixels of the original image, `rawImage`:

```
static Image getCursorImage(Image rawImage)
{
    return Toolkit.getDefaultToolkit().createImage(
        new FilteredImageSource(rawImage.getSource(),
            cursorFilter));
}
```

`cursorFilter` is an instance of our CursorFilter class. This extends RGBImageFilter, which provides a convenient way to modify the pixels in an Image:

```
class CursorFilter extends RGBImageFilter
{
    CursorFilter()
    {
        canFilterIndexColorModel = true;
    }

    public int filterRGB( int x, int y, int rgb )
    {
        return (rgb ^ 0x00FFFFFF);
    }
}
```

The filtering does not depend on the coordinates of the pixel being filtered; if we chose a filter that was coordinate sensitive then in the constructor we would set `canFilterIndexColorModel` to `false`. In this case we're simply exclusive ORing the RGB pixel values with 0xFF, which has the effect of flipping the state of each bit (this is not the same as taking the "negative" of an image).

2.4.5 Handling Keyboard Input

Keyboard input is handled by the Board class. This implements KeyListener and therefore has to implement `keyPressed()`, `keyReleased()`, and `keyTyped()`. The last two are empty methods. Only `keyPressed()` has to do anything:

```
public void keyPressed(KeyEvent e)
{
     if(gameOver) return;

     switch (e.getKeyCode())
     {
          case KeyEvent.VK_LEFT:
               if ( cursor.moveLeft()) redrawCursor();
          break;

          case KeyEvent.VK_RIGHT:
               if ( cursor.moveRight() ) redrawCursor();
          break;

          case KeyEvent.VK_UP:
               if ( cursor.moveUp() ) redrawCursor();
          break;

          case KeyEvent.VK_DOWN:
               if ( cursor.moveDown() ) redrawCursor();
          break;

          case KeyEvent.VK_SPACE:
               imgSquareClicked(cursor.row, cursor.column, true);
          break;
          }
}
```

If the game is over we ignore any key inputs. The only inputs we respond to are cursor up, down, left, and right, and the space key. We don't need to handle the enter key because that's taken care of by our CBA Handler (remember pressing ENTER activates the default CBA button).

The only trick is to make sure that the Board has the focus. The line

```
playingBoard.requestFocus();
```

in `XBombs.selectNewBoard()` achieves this.

2.4.6 Handling Mouse Input

The Board class implements our MouseListener. The only mouse method we need is `mousePressed()`.

We convert the mouse coordinates to row and column coordinates. If these are valid coordinates (which means within the area of the board) we set the cursor at the relevant Square. Using `isMetaDown` allows us to see if the right mouse button or an equivalent meta-key was pressed.

If the mouse event wasn't on the board we check to see if the user pressed one of the three mouse control buttons:

```
public void mousePressed( MouseEvent e )
{
     int column = e.getX()/Square.SIZE;
     int row = e.getY()/Square.SIZE;

     if ( validSquare( row, column) )
     {
          setCursorAt( row, column );
          imgSquareClicked( row, column, e.isMetaDown() );
     }

     else if ( theShowOption.inside( e.getX(), e.getY() ) )
     {
          imgShowClicked();
     }

     else if ( thePossOption.inside( e.getX(), e.getY() ) )
     {
          imgPossClicked();
     }

     else if ( theFlagOption.inside( e.getX(), e.getY() ) )
     {
          imgFlagClicked();
     }
}
```

3

Java on Quartz

Colin Turfus
Quartz is the code-name for Symbian's tablet communicator reference design. It is designed for hardware with the following capabilities:

- a quarter-VGA portrait screen — 240 × 320 pixels
- pen operation
- no keyboard
- hardware keys for up and down, and a single, large, CONFIRM button which works like ENTER on a PC.
- hardware keys for right and left (optional)
- built-in phone, with speaker, microphone, and probably two additional keys (yes/no, dial/hangup)

You can see all these features on the screenshot of Symbian's PC-based Quartz emulator shown in Figure 3.1.

3.1 An Introduction to Quartz

3.1.1 Quartz User Paradigm

Quartz is aimed at a considerably less technical end-user than EPOC Release 5 or the Crystal reference design (Chapter 2): the type of customer who uses a mobile phone, rather than the type who is comfortable with a Windows-based PC. You are probably so used to the PC that you have forgotten how intimidating PCs are for the average end-user — unless you've recently done some technical support for your parents or grandparents!

Navigation Between Tasks

Quartz is task-oriented, not application-oriented. The most common task on a mobile phone is to dial a number. In an application-oriented system, you



Figure 3.1 Symbian's PC-based quartz emulator.

would start with the Phone application, look through the contacts database using some shared UI to select a contact, and then dial that person. In a task-oriented system, you start with the Contacts application, select the person you want to talk to, and then tap their phone number. Control then transfers to the Phone application so you can dial them. Likewise, to send an email, you start in the Contacts application, select your contact, and then tap their email address. Control transfers to the Messaging application, with a blank message to the contact's email address.

Quartz does have applications, but it is structured to make it easy to transfer between them when needed. Quartz presents a single application at a time to the user. There are no overlapping application windows; each application takes up the full screen. The user does not quit or launch applications but switches between them, presenting the illusion that all applications are always active. You typically navigate to a different application or task by selecting its icon in the app launcher screen shown in Figure 3.1. This is equally true for Java applications: the Quartz system architecture ensures that no more than one instance of your Java application is ever running at any one time.

Technical necessities are hidden from users—there is no file system, and no task list. There *is* a file system, and Quartz implements sophisticated multi-tasking. But these technicalities are hidden. To Windows users—or even to EPOC Release 5 users—this may seem revolutionary. But today's mobile phones are already sophisticated multi-tasking systems, and they maintain significant databases, such as the phone book or text message log. Phones do all this without exposing their file system or task list to the end-user.

Quartz is much more sophisticated than today's mobile phones, but it delivers this sophistication in a deceptively simple-looking package. The removal of technicalities from the user interface, single-tap navigation, and task-oriented application switching are all a part of this simplification that delivers a much more attractive end-user experience.

Accessing Data

The paradigm of Quartz is paper and pencil: what you see is what you have. Any data you enter on a Quartz screen is saved when you move away from the view on which you're entering the data—however you move away. There is no PC-like "Save data? yes/no/cancel" dialog when you close a program: data is always saved. Data is saved not only when you close a program, but when you switch to another task. So what you last saw is what you still have—always. This simplifies the user interface, increases the level of user confidence, and provides better insurance against data loss.

Quartz is optimized for browsing and single-tap pen operation. Without a keyboard, it's very easy to use the pen to browse information on a Quartz device. Just as web browsers use only single-click navigation, Quartz uses single tap, with the pen, to open up links, programs, and items. Most applications are structured as list views, in which all data in an application can be browsed by navigating around a list. Items from the list can be opened by tapping on them with the pen.

Quartz is not browse-only. Data can be entered using convenient pointer-friendly controls for dates, numbers, and so on. Text can be entered using handwriting recognition or a virtual keyboard. Symbian supplies CIC's Jot handwriting recognition engine with the emulator in the SDK, although it is not formally a part of the reference design: the Quartz device manufacturer may license CIC Jot or some other handwriting recognition engine for their own device. Data may also be entered by synchronization with desktop-based data, or beamed from another device.

3.2 Anatomy of a Quartz Application

3.2.1 Screen Layout

Figure 3.2 shows a representative view of the Quartz Agenda application—or rather task—displaying a day's appointments. Different areas of the screen

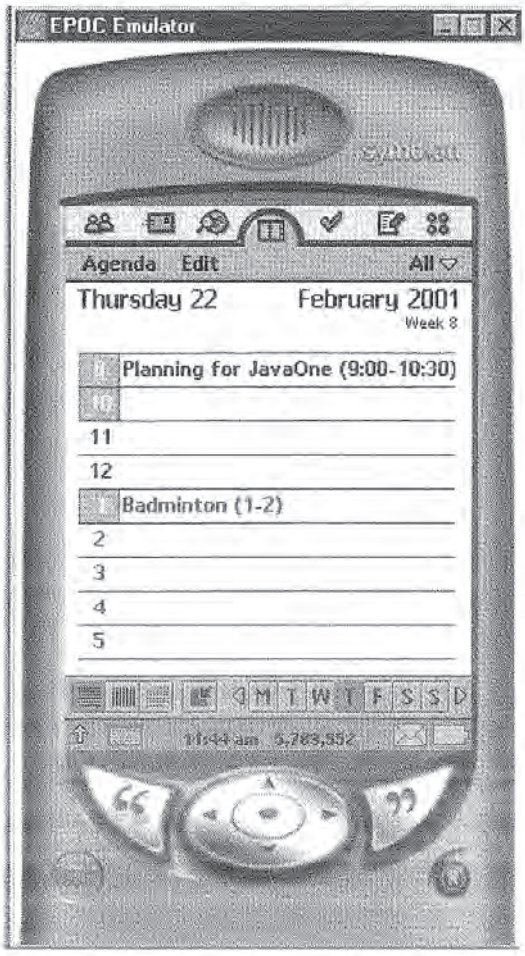


Figure 3.2 The Quartz Agenda task displaying a day's appointments.

are devoted to different purposes, some being accessible to the third-party developer and some being controlled by the system. A more detailed examination of Symbian's UIs and reference designs can be found in Chapter 7.

System Areas

App launcher
The area at the top of the screen controls switching between applications. The rightmost icon always invokes the shell (in Quartz, the shell is called the App launcher), which allows any application/task to be launched or foregrounded. The first six icons are customizable by the end-user: the manufacturer may also set the defaults.

Status bar
The area at the bottom of the screen displays device status information such as battery charge, time, text entry mode, and inbox status. Its content may be customized by the manufacturer. It is not accessible to, nor may it be overwritten by, the third-party developer.

Application Areas

Menu bar
The area between the app picker and the status bar is the application area. At the top, just below the app picker lies the menu bar. Although it can be omitted from either C++ or Java applications, its inclusion is mandated by the style guide and, unlike in EPOC Release 5, it is always visible. It should contain two menu items on the left. You should try to write your application so that all the menu items associated with each screen are available from one of these two menus. The two left-hand menus always have the same form:

- the application menu, whose name is always the same as the application or task, and which contains the system services used by most applications, such as Find, Send as, and Beam;
- the Edit menu, with editing items such as Cut, Copy, Paste, Zoom, Preferences and Help.

These are described further below. Note that because an application or task can always be identified from the first entry on the menu bar, there is no need for a title bar. Some sleight of hand is possible as a result, with a single process masquerading as two applications or tasks. But it is all part of the Quartz paradigm that processes should not be distinguishable to users—only tasks. According to the Quartz Style Guide, the menu bar may also have a folder selector on the right (see **Folders** section below).

Button bar
Optionally, an application view displays a button bar at the bottom of the application area. Not all application views require a button bar.

Main area
The remaining area is available for application developers to use in any way they please.

Views

Because of the limitations of what can be presented in a single screen, Quartz applications make use of views, each with a different screen layout and serving a different purpose. Standard applications in Quartz have an initial view called a base view. Base views display a list of entries for the application. In Figure 3.2, the Agenda base view shows a list of today's appointments. The base view is the topmost level of the application, providing the user with an overview of all entries.

The interaction style is to tap an item from the base view, which immediately opens the detail view for the item. When the JavaOne appointment is selected in the above screen, the details shown in Figure 3.3 are displayed.

The detail view is where the user can view an entry and make changes. The user can then close this window by tapping the "go back" icon in the lower



Figure 3.3 The details for the JavaOne appointment

right corner. This interaction style can be thought of as opening a more detailed view, which you then close to return to the overview. This implies that you view a single item at a time. Although in the Agenda task, the user is allowed to make changes directly in the detail view, in other cases this is not appropriate. For example, in the Contacts applications, you hit a "Change" button in the detail view of an individual contact card and are taken to a separate edit view of the data. Only there can you make changes. When you "go back," the navigation paradigm is that you will be taken to the *base view* again, not to the *detail view* you just left.

There are no Save or Cancel buttons in the detail view: changes are automatically saved.

3.2.2 User Interaction Mechanisms

User interaction mechanisms take various forms, from physically pushing hardware buttons, through entering text via the *handwriting recognition* software or *virtual keyboard*, to key events and mouse events.

Hardware Buttons

Quartz requires UP and DOWN arrows and a CONFIRM button, and may optionally have right and left arrows. The CONFIRM button opens the "current" item. UP and DOWN usually scroll items in a list, while the left and right buttons, if available, typically navigate between tabbed panes. Note that Quartz does not allow you to switch focus between UI components or to navigate text using cursor keys.

The hardware arrow keys and CONFIRM button are there to promote one-handed use in simple browsing tasks. A typical scenario would use the arrows to move the highlight from item to item in the base view, while the CONFIRM button would open the highlighted item in the detail view. Once in the detail view, CONFIRM acts as a short cut for the GO BACK button on the button bar: it closes the detail view and returns to the list view. So the CONFIRM button acts as a toggle, opening and closing items displayed in the list view.

Text Input Mechanisms

There are two ways of inputting text to Quartz, either by using handwriting recognition or via the virtual keyboard. Handwriting recognition is only active when needed; writing your own custom controls allows you to turn it on and off explicitly. Any data entered in handwriting recognition is interpreted as letters, numbers, or other characters, and is sent to the application as text. Special characters are usually interpreted as text characters, not as navigation or control characters. For example, an ENTER key press does not activate the Done button. If handwriting recognition is active, you can toggle between that and the virtual keyboard, by tapping a control on the status bar.

Event Handling in Java Applications

This interaction model translates conveniently into a Java context, albeit that there are a few issues you need to be aware of. The two text input mechanisms work in exactly the same way with Java text components as they do with native ones; as soon as the user clicks in a `TextField` or `TextArea`, they are able either to invoke the virtual keyboard or to take advantage of handwriting recognition. You can scroll up and down the elements of a `List` component with the UP and DOWN arrows and select with the CONFIRM button.[1] With a `Choice` list, however, you are confined to making selections with the pointer.

Java applications can detect the various cursor key and CONFIRM button presses and implement their own event handling by implementing a

[1] Although in Quartz v6.1 `List` navigation works as stated, on the v6.0 emulator you have to use the UP and DOWN arrows and the ENTER key on your PC keyboard rather than the hardware buttons on the fascia.

`KeyListener` interface. These events are associated with Java key codes in accordance with the following table:

Button	Key event
Confirm	VK_ENTER
Up	VK_UP
Down	VK_DOWN
Left	VK_LEFT
Right	VK_RIGHT

They can be distinguished by calling `getKeyCode()` on the `KeyEvent` detected.[2] An important point to remember is that, although the virtual keyboard has a SHIFT key, Quartz has no way of associating CTRL, ALT or META modifiers with a key press. So make sure your application doesn't depend on handling them. Other events you should avoid dependence on are:

- double-clicks
- right-clicks
- mouse motion

None of these are supported on Quartz. Detection of mouse drag is possible but should be used sparingly as it tends not to work well with a pointer, except when used in the context of a single UI component, e.g. dragging a scrollbar or selecting a region of text.

Menus

As we said above, the Menu bar is always visible and the two menu items on the left always have the same form:

- the application menu's name is always the same as the application or task, and contains the system services used by most applications, such as Find, Send As, and Beam.
- the edit menu contains editing items such as Cut, Copy, Paste, Zoom, Preferences, and Help, if these are supported.

Menus are context-dependent, i.e. vary in their content depending on the view being displayed, as can be seen in any standard Quartz application.

It is important to observe that there are no "Save" or "Close" options on these menus. This because it is the Quartz model that *both* of these tasks are

[2] A defect in Quartz 6.0 means that you have to use `getKeyChar()` instead of the more natural `getKeyCode()`. However, this has been fixed for version 6.1.

controlled by the system. As already stated, data modified in an edit view is saved immediately to the underlying persistent data store. Quartz applications are closed not by users but only by the system, and only when the system is running low on memory resources and needs to recover some. Applications should respond to a request to close down by saving their state to persistent storage and exiting. When the application is next activated, it should restore itself to its previous state, the idea being that the user is unable to distinguish whether or not it has been closed down in the meantime.

Dialogs

Quartz dialogs are always full screen width and sit immediately above the status bar. They are application modal, i.e. must be disposed before the user can continue using the application. They have a title bar which should display a meaningful title (*not* including the word "dialog"). Buttons in dialogs are always at the bottom of the dialog, where the rightmost is the action button, typically displaying the message "Done," with a Cancel button immediately to its left. A dialog generally has a default button, which is highlighted. The default button is activated by pressing CONFIRM. Query dialogs ask the user a simple question, and have No and Yes buttons, in that order. There are two types of progress dialog: if the action can be canceled, but leave a net result that nothing happened, the progress has a Cancel button; and if the action can be stopped, but leave the action partly completed, there is a Stop button. Dialogs notifying the user of received data have Reject or Accept buttons. Information dialogs give information to the user, and have a Continue button. Labels for controls in the dialog layout are right-aligned, and the controls themselves are left-aligned. The effect is that controls are centered neatly on a vertical line. This latter effect is tricky to achieve using Java layout conventions. The easiest way is to ensure where possible that all controls are of the same width.

Messages

There are different messages for the user for different events. The messages themselves should be in clear, unambiguous, jargon-free English, so say "create new contact" rather than "new," "information" rather than "error." Message types include notifications, which are messages from an application or server, that interrupt the current application with a system-modal dialog. The user should be able to read the notification and press Done, and continue with whatever they were doing before. Examples include "SMS messages can be read and deleted" and "Phone calls can be notified and answered." Infoprints are messages which appear for about three seconds in the top right-hand corner of the Quartz screen. The user has plenty of time to read them, but does not have to tap any buttons to dismiss them. Infoprints are used to confirm that something has happened, or not happened (e.g. an item was copied to the clipboard), to give an instant error response to things the user just did (such as invalid dialog

control state when the user pressed done). Busy messages flash at the bottom right-hand corner of the screen, and inform the user that the application cannot process user input at the moment, for example, "loading...," "searching...," etc. Error messages should be notified to the user by a dialog, with a continue button and an information title. We will see below (see **Infoprints** section) how you can, if you want, conveniently make use of Infoprints from Java code.

3.2.3 Quartz Utilities

Folders

The Folders menu is on the right side of the Menu bar. Folders are used to divide user data — contacts, jotter entries, to-do lists, etc. — into user-definable sets. There are recommended folders for most applications: unfiled, personal, and business; two folders are recommended for each application: Business, and Personal. The default folder in list views is All, which causes the list to show all items.

The user can add folders, or edit the names of existing ones, through the Edit Folders menu item. In List view the Folder menu acts as a filter. From the detail view, the Folders menu does not work as a filter, but as a control, displaying the folder an item is currently assigned to. You can use the menu to change the assignment. The Folders menu is optional in Quartz applications.



Figure 3.4 The folders menu.

Find

Make a Find item available from the application menu, in either list or detailed view. Find normally searches items in all folders, not just the currently selected folder.

Starting a Find The standard Find dialog contains a single text edit box with two buttons, Cancel and Find (Figure 3.5)

When the dialog first opens, the text entry field gets input focus, which automatically turns on handwriting recognition. You can add a Within choice to the standard find dialog. In this case, the Within contexts might include "this message," "this folder," "all folders."

Displaying the Results When the user selects the Find button on the Find dialog, your application should create a found dialog to display the results.



Figure 3.5 The standard find dialog.

The title should be in the form Found "Fred" (where "Fred" was the search string). The Found dialog should have a single button, Cancel, which closes the dialog. Set the dialog title to "Finding..." during the search, to emphasize that the find is still being processed and change the title to "Found..." when the search has been completed. If nothing is found, the results dialog should display the title "Nothing found..." after the "Finding..." phase has been completed.

Undo

Undo is there to protect the user's data. It is easy to "do" actions in Quartz, so it is important to be able to "undo" them also. Applications support only a single level of Undo. Most user tasks are expected to be short in Quartz, so the granularity of the Undo is usually all the data contained in an item—i.e. an entry in a List view, or the entire item being worked on, in a Detail view. Undo should normally work only in the current view where the action takes place. If Undo needs to change you to a different view in order to effect the change, you are keeping the Undo state around too long. Undo should not navigate the user to another view.

Zoom

Zoom increases font size so that, for example, users with poor eyesight, or those in difficult visual conditions, can see more easily.

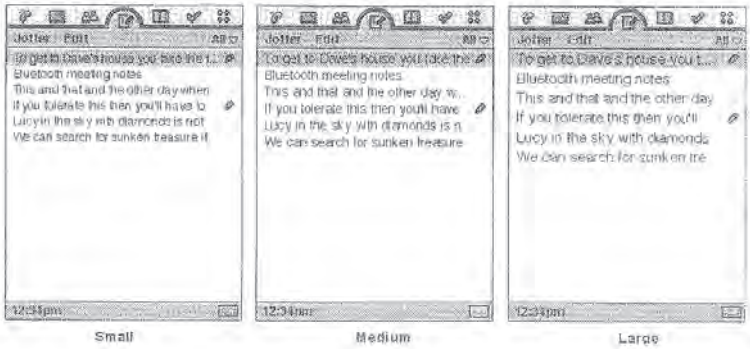


Figure 3.6 Zoom increases font size.

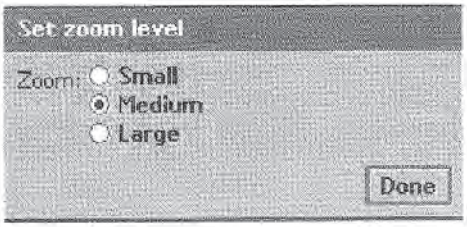


Figure 3.7 The standard zoom dialog.

Zoom is a function that should be placed on the edit menu. Figure 3.7 shows the standard Zoom dialog.

Most views with text, including List views, should be able to zoom, though tabs, menus, buttons, status bars, and dialogs cannot.

Send As

Initiating the Send As command calls a standard send as dialog in which the user can select the message transport (SMS, email, fax, etc.). Send As is used to send exactly one item:

- you cannot send less than an item (e.g. some highlighted text).
- you cannot send more than one item (e.g. highlighted items in a list).

Beam

It is very convenient to be able to exchange data using infrared between two devices that support the IR object exchange (IrOBEX) standards. One user specifies Beam, and both users put their machines into IR contact. The receiving user does not have to initiate anything. As a precaution, however, the receiving user does have the opportunity to accept or reject the item sent. All types of data that a Quartz device can accept by beaming open up the corresponding viewer (vCal viewer, vCard viewer, etc.). In the viewer, the user can see the data that was beamed, before deciding if they want to discard it or save it in the corresponding application (such as Agenda for vCals and Contacts for vCards).

3.3 Tuning Java Applications for Quartz

So now you've got an idea how the Quartz paradigm works, let's take a look at how a Java application can be designed to conform to this paradigm. The good news here is that by default a Java application running on the Quartz platform will conform to many of the guidelines described above. Other features, as will be seen below, can be implemented by following some simple design guidelines. Others again are unfortunately not possible at this time in Quartz

Java applications. We'll return to the subject of the limitations of Quartz Java applications in the final section below (**Limitations of Java programming on Quartz**).

3.3.1 What you get for Free

Because `java.awt` components are implemented in Quartz in terms of native (so-called "QIkon") components, any Java application will automatically pick up a certain amount of Quartz look-and-feel. For example, the default background color for a Frame or Dialog is the characteristic Quartz blue in v6.0, although this has been changed to white, the more common choice for windows, in v6.1.[3] If you want to obtain explicitly the standard Quartz blue, you can get this with `new Color(165, 203, 247)`.[4]

In addition, the sizing and positioning of Frames and Dialogs is done for you by the Quartz Runtime with virtually no effort on your part; indeed, one might say, *in spite of* efforts on your part. For, irrespective of the specifications you make:

- Frames will rescale themselves exactly to fill the application area. If you call `pack()`, after constructing a Frame, it will even size itself appropriately to allow for the presence or absence of a Menu bar. No Title bar will be used on your Frame; any title you specify will be ignored.
- Dialogs will position themselves appropriately just above the Status bar. If you call `pack()` on the Dialog, it will also size itself appropriately, taking up the full width of the screen with its height just as large as it needs to be to allow all the contained components to be displayed at their preferred sizes. Quartz Java dialogs do, like all Quartz dialogs, have Title bars, with titles set in the usual way.

3.3.2 The QFrame Class

Beyond the basic assistance described above, the Quartz SDK for Java provides a Java package `com.symbian.devnet.quartz.awt` containing a class `QFrame` which you can subclass in preference to `java.awt.Frame`. This furnishes easy access to a number of the other characteristic Quartz features.

---

[3] It is not a necessity that licensees of the Quartz platform use the standard Quartz colors. Instead they can implement their own system color scheme. If they do, Quartz Java Frames, Dialogs and other UI components will, unless hard-coded to the standard Quartz blue color scheme, pick up the new system colors automatically.

[4] Using the `SystemColor` class in a Quartz Java class in v6.0 gives you access not to the Quartz system colors, but to Sun's hard-coded VM defaults. The situation is different from v6.1 with `SystemColor` providing a Java application with information about whatever system colors have been specified for the particular Quartz-based device on which the application is running. For example, the standard Quartz blue will then be obtainable with `SystemColor.control`.

The package ships with full sources allowing you, if you prefer, to cut and paste parts of the code rather than subclassing, and with a simple test class called `QFrametest` which shows how `QFrame` might be used in practice. Here is how its constructor starts. The first thing it does is to call the parent class `QFrame`'s constructor, passing as an argument an integer representing the number of views supported by that application.

```
public QFrameTest(int _numCards)
{
      super( numCards);
```

If no argument is passed a single view is assumed. As a result, the application is endowed with a `CardLayout` with the specified number of cards called "`card1`," "`card2`," etc. Each of these cards is by default a `Panel` object whose layout you can specify any way you choose. `QFrame` provides a number of utility methods to allow you to recover and/or display the cards. The cards are most conveniently selected with `getCardAt(int n)` method which returns a reference to the (n – 1)th card.

```
Panel pt = getCardAt(0);
pt.setLayout(new BorderLayout());
```

`QFrame` also provides a convenient mechanism for equipping your application with an "About" dialog. This has a number of constructors, the most useful taking an array of strings, each representing a line of text in the dialog, as an argument. A `getVersion()` method provides by default a string representing the version number of `Qframe`, currently 1.21. You would normally override this method to return the version number of your own application.

```
String[] desc = {"QFrame Test v. " + getVersion(),
   "(c) Copyright Symbian Ltd",
      "http://www.symbiandevnet.com"};
createAboutDialog(desc);
```

This "About" dialog is launched from a menu item included automatically in an application menu called `appMenu` which you will find has been provided on a Menu bar. The menu's caption will by default be the class name (with the package qualification stripped off). You may want to set this instead to be the same as the caption that sits below the application's icon in the app launcher by ensuring that the flag

```
-Dcom.symbian.appName="<caption>"
```

where `<caption>` is the required name, is passed in as an argument to the Runtime when the class is launched. There will also be an Edit menu called `editMenu`. Both of these menus are declared `protected` in `QFrame`

so as to allow you to add items to, or remove items from, them in your subclasses. So:

```
editMenu.add(mnuStart);
editMenu.add(mnuStop);
editMenu.add(mnuDialog);
```

and so on. The inclusion of a `main()` method in `QFrameTest` allows it to be run as an application.

```
public static void main(String[] args)
{
```

We create `QFrametest` with three views.

```
    QFrameTest qTest=new QFrameTest(3);
```

We take advantage of the `setCardAt(Panel p, int n)` method which allows you to replace the default `Panel` at position `n` with the `Panel p` or, more interestingly, a `Panel` *subclass*. In this instance we set `Panel` at position 0 to be a `com.symbian.devnet.awt.quartz.GridPanel`, then display it.

```
    GridPanel cardToSet=new GridPanel();
    cardToSet.add(new TextArea("", 10, 20, TextArea.
        SCROLLBARS_VERTICAL_ONLY), 0, 0);
    cardToSet.add(new Button("Testing GridP"), 0 , 1);
    qTest.setCardAt(cardToSet, 0);
    qTest.displayCardAt(0);
```

Observe that retrieving the `GridPanel` with `getCardAt(0)` now requires an explicit cast:

```
    GridPanel cardToGet = (GridPanel)qTest.getCardAt(0);
    TextArea textArea = (TextArea)cardToGet.getComponent(0);
    textArea.setText("Testing QFrame!");
}
```

3.3.3 Access to Native Utilities

In order to have your application comply with the Quartz style guidelines, or further to display the full range of capabilities native applications can demonstrate, it is necessary to make use of some Symbian OS functionality not accessible through standard Java classes. In other words, you need to provide, or at least to invoke, Java native methods through the Java Native Interface. Instructions on how you would go about this can be found in Chapter 13. However, the core methods you are likely to need for the purposes of a

compliance with Quartz style are provided on the Quartz SDK for Java in a package called `com.sumbian.devnet.util`. Most of the methods are static methods of a holder class called `TaskSwitch`.

The `util` package should be imported into your class if you want to make use of it. When you install your v6.0 Quartz SDK for Java, this will be available as **\erj\ext\util.jar** so it will be on the default class path of the Symbian Runtime. However, you will need to include it explicitly in your Windows class path when you compile your own Java classes importing or using the package. The Symbian Runtime will also need access to a DLL called **Taskswitch1** which will be put in place for you automatically on installation of your v6.0 Quartz SDK for Java. Symbian installation files (**.sis**) allowing you to install this package, with appropriately recompiled versions of the DLL, onto real Symbian devices will be provided with the v6.1 SDK.

Infoprint

One requirement which should be satisfied by a Quartz application is that, if an attempt is made by the user to access an unsupported feature or if confirmation needs to be given in an unobtrusive way that an operation has completed in the background, this should be communicated to the user in an Infoprint. Just send the (Unicode) string you want displayed as an argument to the Infoprint(String) method, e.g.

```
try
{
      TaskSwitch.Infoprint("Connected to server");
}
catch (RuntimeException e)
{
      System.err.println(e.getMessage());
}
```

More serious errors should be communicated through an Information dialog with a "Continue" button. This can conveniently be achieved with a standard Java Dialog.

System-Modal Dialogs

Suppose you design your application so that it is able to process incoming data, e.g. an FTP download, even while the application is in the background. If something goes wrong you will likely want to notify the user of that fact. For example, the system might try to shut the application down to conserve memory and you might want to offer the user a choice as to whether that was their wish. The ideal way to do this is with an EPOC system-modal Notify dialog. This is a native facility allowing display of a small Dialog with a title, a question, and two buttons presenting a choice to accept or reject. 0 is returned

for a "no" and 1 for a "yes." This functionality is available directly to Java programs through the `notify(...)` method of TaskSwitch, used as follows:

```
try
{
      if(1 == TaskSwitch.notify("Exit dialog", "Do you want to
          exit?", "No", "Yes"))
      {
            shutDown();
      }
}
catch (RuntimeException ex)
{
      shutDown(); // if notify fails
}
```

This code causes the Dialog shown in Figure 3.8 to be displayed in front of whichever application currently has focus. The user cannot continue with any other work until they answer either yes or no.

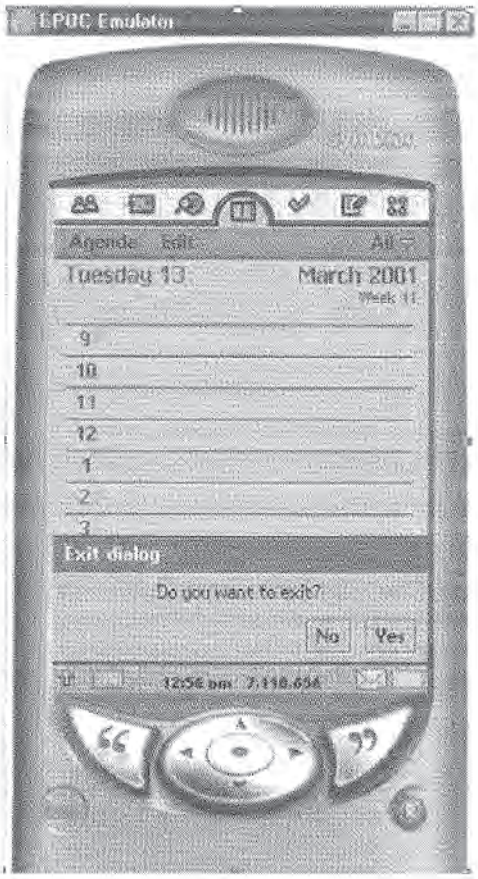


Figure 3.8 Exit dialog.

Interacting with the Symbian OS Window Server

There are a number of tasks which you might want to accomplish in Quartz Java applications which require interaction with the Symbian OS Window server. One of these is finding out whether your application is in the foreground or the background. As we shall see (in the section on **background\foreground event handling**, below), Quartz applications have a requirement to be able to handle "app backgrounded" and "app foregrounded" events, a capability which the Java Runtime just doesn't provide. To this end the TaskSwitch class provides a convenient static utility method `TaskSwitch.isForeground()` which returns a boolean indicating whether the current Java application (the one identified by the Java system property `com.symbian.appName`) is the owner of the window group currently in the foreground. Notice that this method can throw exceptions, in particular a custom `com.symbian.devnet.util.NotFoundException` if no window group is found associated with your application's name.

A second useful method is `TaskSwitch.taskToForeground(String)`. This allows you to change context to another, currently running Quartz application (Java or otherwise). It receives the name of an application as an argument and checks whether an application with that name is currently running. The "name" in this instance is the caption which appears under the application's icon in the app launcher, which name is formally independent of `com.symbian.appName` property used by `isForeground()`. If the named application is running, its window group is brought to the foreground. If it is not running or if no such application exists, a `NotFoundException` is thrown.

In line with the Quartz navigation paradigm, you might want to be able to start up an instance of the named application if this is not already running. Indeed you might want to be able to select a particular task or view within that application, e.g. to distinguish between the Agenda and ToDo tasks, or to open up the web browser seeded with a given URL. Quartz in fact does provide such functionality natively—and indeed the Quartz system applications make extensive use of this—but through the so-called View server (a feature introduced in Version 6.0 of the Symbian OS) rather than through the Window server. The details of this are beyond the scope of the present overview. `TaskSwitch` does not provide access to the View server, although it is possible to do so with JNI code and it is expected that the `com.symbian.devnet.util` package will be extended at a later date to encompass such functionality.

Finally, there is also `TaskSwitch.displayTaskList()`. This displays the task list which was a feature of EPOC Release 5, and was incorporated into the Crystal reference design. It allows you to switch context to, or to close, any application currently running on a device. However, as this breaks the Quartz user paradigm, it is not intended to be made use of in any application shipped to end users, but is provided for debugging purposes only. Note that on the debug emulator SHIFT-CTRL-ALT-T has the same effect of launching the task list at any

time, while CTRL-SPACE launches a menu providing you with the option to close the current application. The latter can be useful if you are testing the shutdown behavior of your Java application on the emulator. The TaskSwitch-based utility is obviously of more use if you are doing such testing on real devices.

3.3.4 Completing the Picture

We saw above (**Section 3.3.2**) how you can get many of the features of a Quartz application into your application simply by having it subclass `QFrame`. To implement other aspects of the Quartz paradigm, particularly relating to event handling, you have to do a little extra work.

Shutting Down Applications

One of the distinctive features of a Quartz application is that the user never explicitly closes it. Rather the application will simply be backgrounded when they move on to another task. You as the programmer are responsible for ensuring that, if and when the system decides that your application needs to be shut down, it responds to that request and shuts down in an orderly fashion. The way that the Quartz system requests a Java application to shut down is effectively by sending it a `WINDOW_CLOSING` event. You capture this event in client Java code by adding to each window in your application a `WindowAdapter` with a `windowActivated(WindowEvent event)` method containing your code to handle shutdown.

If you use `QFrame` rather than `Frame`, a `WindowAdapter` will already have been added, which calls a method `shutDown()` which shuts the application down with a call to `System.exit(0)`.To save your application state before shutting down, simply override `shutDown()` in your subclass and end your method with a call to `super.shutDown()`.

Saving Application State

If you follow the Quartz paradigm in your application design, you should be saving user data in persistent storage as it is modified. However, as it is the goal of Quartz application design that, when a user reactivates an application, they should be unaware whether it has been shut down in the interim, it is necessary that you store sufficient application state data on shutdown to facilitate replication of the prior state.

In this regard, there are some additional rules about what should happen when a Quartz application is backgrounded that you should be aware of:

- first, any open dialogs should normally be disposed.
- secondly, if an edit view is active it should be closed, the edited data being saved to persistent storage.
- finally, the application should be restored if possible to its base view.



Figure 3.9 The DiceMachine.

This coarse-graining of a Quartz application's "state" helps reduce the amount of information you need to save on shutdown. If you consider the DiceMachine example on the Quartz SDK for Java, you will see an example of this (Figure 3.9). Here the data needed to restore state are

- the number of "dice" currently displayed;
- the number each die is currently displaying.

This information is saved in `shutDown()` to a file called **DiceMachine.ini** which is read on start-up to configure the initial state.

```
public void shutDown()
{
      FileWriter iniFile = null;
      try
      {
            iniFile = new FileWriter("DiceMachine.ini", false);
            iniFile.write(numberDice);
            for (int i=0; i < numberDice; i++)
            {
                  iniFile.write(dice[i].offset);
            }
      }
```

```
        catch (Exception ex)
        {
              // do nothing
              System.out.println("Exception occurred writing
                  ini file.");
              ex.printStackTrace();
        }
```

Now close the file, if it was successfully opened, and exit.

```
        try
        {
              if (iniFile != null) iniFile.close();
        }
        catch (IOException ex) {}
        super.shutDown();
}
```

If a dialog is open, this should and does get closed when the application is backgrounded. In other words, the state of the dialog is not considered to be part of the application's state.

It may be that for your particular application, you decide such auto-closing of dialogs is inappropriate. If so, you should bear in mind that *a Java Dialog is a Window*, i.e. a subclass of `java.awt.Window`. The important corollary to this is that, just like a Frame, it intercepts (and consumes) `WINDOW_CLOSING` events. So, if you do not ensure the Dialog is disposed when your application is backgrounded, you must make sure it handles `WINDOW_CLOSING` events appropriately, e.g. by calling the `shutDown()` method of the parent `QFrame`. Otherwise the system will be unable to recover the resources used by your application, until the user manually closes the offending dialog. Of course, the only time a Quartz Java application is closed in practice is when it is in the background, so dialogs which dispose themselves when the application is backgrounded do not need to handle `WINDOW_CLOSING` events.

Quartz dialogs normally close themselves when the application is backgrounded. A Java Dialog not following this formula should handle `WINDOW_CLOSING` events by initiating application shutdown.

Background/Foreground Event Handling

Unfortunately, to satisfy the requirements in the preceding subsection, it is necessary to know when an application has been backgrounded, so that, for example, open dialogs can be closed. But Java has no such thing as an "app backgrounded" event for us to handle. In this regard, Quartz's user paradigm could be said to be incompatible with Java's. Nonetheless, a reasonable work-around is possible, taking advantage of the `TaskSwitch.isForeground()` method described above (**Interacting with the Symbian OS Window server**).

What you can do is intercept WINDOW_DEACTIVATED events. These are generated when a window loses its foreground status, something that can happen in a number of ways:

1. A Dialog or Menu may be displayed in front of a Frame.
2. A PopupMenu may be displayed in front of a Dialog.
3. A Choice list may be activated on a Dialog or Frame.
4. The application may be backgrounded by another application.

Clearly only the last of these is of interest to us. So how do we distinguish it from all the others? Answer: call `TaskSwitch.isForeground()`!

In other words, the strategy is that your Dialog should in all cases implement `WindowAdapter`, just like `QFrame` does. If the Dialog is to be closed when the application is backgrounded, WINDOW_DEACTIVATED events should be handled thus:

```
WindowAdapter wl = new WindowAdapter(){
      public void windowDeactivated(WindowEvent e)
      {
            handleBackground();
      }
};
addWindowListener(wl);
```

If the WINDOW_DEACTIVATED event has resulted from the application being backgrounded, `handleBackground()` should close the Dialog. Otherwise it does nothing.

```
public void handleBackground()
{
      if (!TaskSwitch.isForeground())
      {
            dispose();
      }
}
```

On the other hand, if the Dialog is *not* to be closed when the application is backgrounded, WINDOW_CLOSING events should instead be handled as follows:

```
WindowAdapter wl = new WindowAdapter(){
      public void windowClosing(WindowEvent e)
      {
            parent.shutDown();
      }
};
addWindowListener(wl);
```

where `parent` is a reference to the parent `QFrame`, which should be passed into the Dialog's constructor and cached.

As we discussed, when an application is backgrounded, disposing active Dialogs is only one of the things you might want to do. You would normally also close any active edit view and return to the base view. To ensure that this worked properly, you would have the code to do it called from a `windowDeactivated(Event)` method associated with the Frame, conditional upon `TaskSwitch.isForeground()` returning `false`, ensuring in addition that the same code was run after any Dialog was disposed in a `handleBackground()` method. To facilitate this, QFrame provides an empty `handleBackground()` method for you to override, which is called in response to a `WINDOW_DEACTIVATED` event. Unfortunately, the method is not called conditional upon the application being found to be in the background, so you will have to test for this in your overriding version. This should be corrected in a future release of QFrame in v6.1.

There are other things again you might want to do when an application is backgrounded. One of these is that you might want to suspend certain threads if, say, they were:

- performing intensive background computation which might impact the performance of the foregrounded application
- polling a remote server for information to update, say, a ticker-tape-style visual display
- controlling the action of a real-time user-interactive game, like Tetris or PacMan.

In such cases you would want to have the threads restarted automatically when the application was brought back to the foreground. To this end QFrame supplies a `handleForeground()` method called in response to a `WINDOW_ACTIVATED` event. The code you put in an overriding version of this method should be called conditional upon `TaskSwitch.isForeground()` returning `true`. Such a condition is not, however, a sufficient condition to guarantee that threads would have been stopped by a prior call to a `handleBackground()` method. For, just as we saw that `WINDOW_DEACTIVATED` events can be generated other than by an application being backgrounded, likewise for `WINDOW_ACTIVATED` events. You can get round this problem by setting a flag when the application is backgrounded (and the threads stopped) and unsetting this only when threads are restarted in a `handleForeground()` method. You will need, however, to take care that all the `handleForeground()` and `handleBackground()` methods getting called from various Dialogs and QFrames do work in sync with each other.

More on Dialogs

There are a number of other issues relating to the layout and disposal of Quartz Java Dialogs. Let's look at disposal first. As we saw above (**Event handling in Java applications**), a Quartz Dialog will typically have a default button, its

status being indicated by a darker border. The default button behavior event handling code gets run when

- the default button is pressed
- the application is backgrounded while the Dialog is open
- the ENTER key or CONFIRM button is pressed while the Dialog is open.

Clearly this is quite different from the way Java Dialogs work. First, Java has no concept of a default button. Nor is there a convenient way of providing a Java Button with a darker border. It is planned that future releases of Quartz will provide a custom button with the required behavior. But for the moment the only option available is to replicate Quartz default button behavior using Java code.

The default button's status might conveniently be indicated by displaying the text in bold font. Suppose we put the default button's event-handling code in a method called `handleDefault()`.

```
public void handleDefault()
{
        // do something useful here
        dispose();
}
```

You can arrange to have this code run when the default button is pressed by implementing an ActionListener in the familiar way. You can have it run in response to the application being backgrounded by using the `handleBackground()` method described above, replacing the call to `dispose()` with a call to `handleDefault()`. The third of the above three situations cannot, however, be conveniently handled by Java code. This is because the ENTER key event generated by the CONFIRM button is delivered to the component which currently has focus, and you have no way of controlling which this will be. This remains for the moment a limitation of Quartz Java applications *vis-à-vis* native applications.

Turning to Dialog layout issues, the Quartz style is for buttons to be positioned flush right at the bottom of the Dialog. In the QFrameTest class (pictured over) accompanying the `com.symbian.devnet.quartz.awt` package , this is accomplished as follows. The Dialog contains a TextArea sitting above two buttons, "Cancel" and "Done":

```
Button butCancel = new Button("Cancel");
Button butDone = new Button("Done");
String description = "\t\tTesting Dialog.\nNote the positioning
      of the Cancel & Done buttons which in line with the Quartz
      style guide are right aligned with Cancel as the first
      placed button, Done as default.";
TextArea txtDesc = new TextArea(description, 5, 10,
    TextArea.SCROLLBARS_NONE);
Dialog dialog;
```

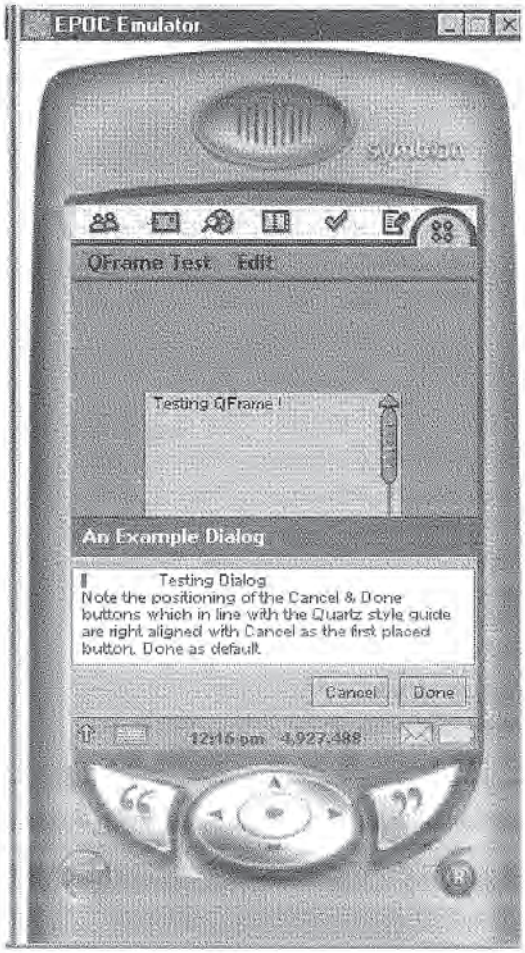


Figure 3.10 The cancel and done buttons.

In the `QframeTest` constructor, the Dialog is assigned:

```
dialog = new Dialog(this, "An Example Dialog");
dialog.setLayout(new BorderLayout());
```

The buttons are not added directly to the Dialog but to a GridPanel which controls their layout. This comes with a `GridBagLayout`, a reference to the constraints object which is recovered with the `getConstraints()` method. Each button is assigned its own horizontal grid cell. The button is forced to sit flush right within this cell by setting the constraints anchor to be EAST. The (leftmost) Cancel button is added with horizontal weight (`GridBagConstraints.weightx`) maximized at 1.0; any other buttons (here just the Done button) are then added with weight 0.0. This means that the leftmost cell expands to be as wide as it can at the expense of all the others, with the result that the buttons position themselves flush right as shown. The TextArea is added in position "Center," while the buttons are forced to the bottom of the Dialog by adding the button panel in position "South":

```
GridPanel buttonPanel = new GridPanel();
buttonPanel.getConstraints().weightx = 1.0;
buttonPanel.getConstraints().anchor = GridBagConstraints.EAST;
buttonPanel.add(butCancel, 0, 0);
buttonPanel.getConstraints().weightx=0.0;
buttonPanel.add(butDone, 1, 0);
txtDesc.setEditable(false);
txtDesc.setFont(new Font("Times", Font.PLAIN, 7));
dialog.add(txtDesc, "Center");
dialog.add(buttonPanel, "South");
```

Alternatively, you could get a similar effect by choosing `buttonPanel` to be a `Panel` with `FlowLayout`:

```
Panel buttonPanel = new Panel();
buttonPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
buttonPanel.add(butCancel);
buttonPanel.add(butDone);
...
```

This is simpler, albeit giving you less control over the layout of the buttons. Finally, action listeners are added to the buttons and `pack()` is called on the Dialog so that it can take up its preferred (minimum) height:

```
butCancel.addActionListener(this);
butDone.addActionListener(this);
dialog.pack();
```

The other layout issue with Quartz Dialogs is labeled controls: the labels should be right-aligned but the controls left-aligned. This effect can be achieved in a manner similar to the above, using a `GridPanel`, but with the anchor set to EAST for the labels and WEST for the controls. It is planned that a `QDialog` class conveniently incorporating the features of Quartz dialogs discussed above will be introduced into a new release of the `com.symbian.devnet.quartz.awt` package in the v6.1 Quartz SDK for Java.

3.4 Limitations of Java Programming on Quartz

We have already mentioned above one limitation of Quartz Java programs in that it is not possible to invoke a default button on a Dialog using the CONFIRM button. We mention below a few other areas where you may find your aspirations to replicate Quartz native functionality in Java applications thwarted.

3.4.1 Closing Activated Menus and Choice Lists

If a Quartz application is backgrounded, it should on reactivation be displayed in its base view with the user unable to distinguish whether it has been closed in the interim. Java applications have a certain difficulty complying with this insofar as activated Choice lists and menus will not by default get closed when an application is backgrounded; nor is there a convenient way to force them to deactivate. So, they will still be active when the application is reactivated, except if the latter has been closed in the interim. However, as the startup time for the Java Runtime will in that case be a more obvious indicator, this is not a significant shortcoming.

Potentially more troublesome is the fact that, although activated menus and Choice lists do not interfere with the reception of `WINDOW_CLOSING` events, they do raise problems for the strategy we proposed for handling application background and foreground events. You will recall that strategy depended on these events generating `WINDOW_DEACTIVATED` and `WINDOW_ACTIVATED` events, respectively. The problem is that, if a Menu or Choice list is active, the underlying Frame or Dialog considers itself to be deactivated so is temporarily unable to receive `WINDOW_DEACTIVATED` or `WINDOW_ACTIVATED` events. There is not much you can do about this other than hoping your users don't task away from your application while in the midst of making a selection from a Menu or Choice list!

3.4.2 Using the System Clipboard

It is possible to access the EPOC system clipboard using the standard Java clipboard APIs. However, although the EPOC system is capable of retaining formatting information for rich text, this is not available when the clipboard is accessed from Java code. Rather the text will be converted to plain text.

3.4.3 Quartz Utilities

None of the native utilities described above (**Section 3.2.3**) are immediately accessible to Quartz Java applications. A number of them (Find, Undo, Zoom) could be fairly straightforwardly replicated with Java code. Others (Beam, Send As) would need some JNI coding to expose the native functionality to Quartz Java applications. Folders, the other main utility, cannot, even with recourse to JNI coding, be displayed in Java applications as they are in native applications on the Menu bar. Nonetheless, the folder information stored by the native Contacts application is accessible through standard JavaPhone APIs as ContactGroups. And there is nothing to stop you enacting your own folder system in a Quartz Java application.

There is one further utility we have not heretofore mentioned: printing, not being supported in native Quartz applications, is not supported in Quartz Java applications either.

3.5 Conclusions

We have looked at the new user paradigm introduced by the Quartz reference design. We have seen how most of the requirements it makes on applications can be satisfied equally by Java applications as by native applications, albeit with recourse in the former case to some Java native methods provided in Quartz-specific utility classes on the v6.0 Quartz SDK for Java. Quartz native utilities can in some cases be replicated with Java code. In other cases they would require JNI development. The few outstanding areas of discrepancy between the capabilities of Java and native applications will be addressed in Version 6.1 and subsequent releases.

Section 2: Getting the most from Java on the Symbian OS

Explores the Java APIs available to Java developers, illustrated with an example game. Looks at creating user interfaces that work on constrained devices, and provides a detailed discussion on optimization and creating first class code for constrained devices

4.	PersonalJava	page 99–116
5.	JavaPhone	page 117–190
6.	Java Games	page 191–228
7.	UI Guidelines	page 229–246
8.	Optimization	page 247–290

4

PersonalJava

Jonathan Allin

This chapter describes Symbian's implementation of PersonalJava. It looks at each of PersonalJava's unique features and illustrates the timer API with an example.

4.1 Summary of PersonalJava

The PersonalJava Application Environment (PJAE) addresses the software requirements of networked applications running on personal consumer devices such as WIDs and PDAs, rather than desktop computers.

Symbian has implemented PJAE 1.1. This is based on JDK1.1 but makes a number of packages, classes, and methods optional. A vendor can therefore save memory by not implementing those optional features they don't require.

PJAE also defines a number of additions to the JDK1.1 specification. In summary, these allow an application to:

- identify if a component is double buffered
- specify component input behavior
- handle unsupported optional features
- implement timers using a new timer API.

Each of these additions is discussed in detail later in the chapter.

The detailed PJAE specification can be found at ***http://www.javasoft.com/products/personaljava/***.

4.2 Symbian's PJAE Implementation

4.2.1 Optionality

Symbian has ported Sun's PersonalJava 3.0.1 implementation of the PersonalJava Application Environment 1.1.1a specification. All of the PJAE's optional

features are implemented by Symbian. A licensee is not required to take all of these but is required to take more than the minimum subset. The following table summarizes the PJAE's optional features and which of these must be taken by a licensee.

Class, method, or package	Comment	Must be taken by licensee
Partial Frame	A minimal implementation must allow the Frame constructor to be called once to create a root for its component hierarchy	Yes
Partial Dialog	A minimal implementation must allow a single modal dialog to be visible at a time	Yes
Window	A minimal implementation can prohibit direct creation of Window objects	Yes
FileDialog	Optional. Not relevant for e.g. the Quartz reference design which has no user-visible filing system	No
Toolkit.getPrintJob()	Should throw UnsupportedOperation exception if printing is not supported. Note that Symbian's current Java implementation does not support printing	n/a
Graphics.setXORMode()	Should throw UnsupportedOperation exception if the display is not capable of drawing in exclusive-or mode. This is true of anti-aliased displays	n/a
java.io	The following IO classes must be completely implemented or not implemented at all: • java.io.File • java.io.FileInputStream	Yes

Class, method, or package	Comment	Must be taken by licensee
	• java.io.FileNotFound Exception • java.io.FileOutputStream • java.io.FileReader • java.io.FileWriter • java.io.FilenameFilter • java.io.RandomAccessFile The rest of the classes in java.io are required, e.g. support for buffered stream, filter streams, object streams	
java.math	Either implemented fully or not implemented at all	Yes
java.rmi (and sub packages)	Either implemented fully or not implemented at all	No
java.sql	Either implemented fully or not implemented at all	No
java.security (and sub packages)	Either implemented fully or not implemented at all	Yes
java.text.resources	The following three classes are required: • java.text.resources. DateFormat ZoneData • java.text.resources. LocaleData • java.text.resources. LocaleElements The rest of the classes in java.text.resources are optional	Yes

The PJAE does not support the java.security.acl package. This package defines interfaces for creating and managing access control lists (i.e. who can do what to whom!), so isn't relevant given that PDAs and WIDs are single user devices.

4.2.2 Performance Enhancements

Memory Footprint

PersonalJava uses smaller structures to represent loaded classes, methods, etc. Preloading the same classes in PersonalJava and Java 1.1 results in about a 25–30% reduction in memory for the classes. This saving is also transferred to classes loaded dynamically which will therefore require less RAM. However, the size of the Java objects themselves has changed little.

Speed

Java is significantly faster on those Symbian devices that run Version 6 on ARM9 or StrongARM cores. This is because the memory model has been changed to avoid the need to flush the code cache on a context switch. SpecJVM98 reports a 37% improvement in performance compared to EPOC Release 5, almost entirely due to this change.

The AWT implementation is a little quicker: better event handling and repainting/updating algorithms cut down on the amount of flicker by removing unnecessary repaints. This means that lightweight UI toolkits will run slightly faster.

4.3 Double Buffering

`isDoubleBuffered()` is a new method that has been added to the java.awt.Component class. The method returns `true` if all the drawing done during `update()` and `paint()` is automatically double buffered. The return value is not valid until after the peer has been created.

The default value is platform specific. Version 6.0 returns `false`; however, as part of the performance improvements in the next major release a number of components will be double buffered, in which case the method will return `true` for these components.

4.4 Specifying Component Input Behavior

PersonalJava provides a number of interfaces in the com.sun.awt package which can be used by a UI component to specify its preferred input behavior.

4.4.1 ActionInputPreferred

ActionInputPreferred indicates that a user can click on or otherwise "activate" a component by using a pointer device.

4.4.2 KeyboardInputPreferred

KeyboardInputPreferred indicates that the component will be used primarily via keyboard input. Some platforms will respond by popping up an on-screen keyboard when the user navigates to or selects this component. The developer should ensure that `Component.isFocusTraversable()` returns `true` for this component.

For instance in the following

```
Class MyTextComponent extends Component implements
    KeyboardInputPreferred
{
    :
}
```

`MyTextComponent` could be a lightweight component that is part of a game which requires keyboard input. When it gets the input focus the underlying implementation should, if the platform does not have a keyboard, display an on-screen keyboard or enable handwriting recognition. (This is the behavior you would expect on a Quartz device.)

4.4.3 NoInputPreferred

NoInputPreferred indicates that the user may not navigate to the component. This interface may be appropriate for components such as labels.

4.4.4 PositionalInputPreferred

PositionalInputPreferred indicates that the component will be used primarily by the user selecting x,y coordinates within the component.

The system should provide some mechanism for selecting specific x,y coordinates within the component's bounds and provide mouse movement events. Typically this will be done by overlaying a virtual cursor or pointer on the component that is driven by the keyboard's cursor keys.

In the following:

```
Class MyGraphicsMap extends Component implements
    PositionalInputPreferred
{
    :
}
```

`MyGraphicsMap` could be an image map (perhaps from a web page or part of a game) that requires mouse input. When it gets the input focus the underlying implementation should, if the platform does not have a pointer device, display a virtual cursor or mouse pointer. (This is the behavior you would expect on a Crystal device.)

4.5 Handling Unsupported Optional Features

There will be situations when a Java application attempts to access an unimplemented optional PJAE feature. In this situation one of a number of exceptions will be thrown:

- java.lang.NoClassDefFoundError will be thrown if an attempt is made to access a class in an optional package, e.g. attempting to access java.sql.DriverManager if the sql package is not implemented
- com.sun.lang.UnsupportedOperationException will be thrown if an attempt is made to access an unsupported class in the java.awt or java.util.zip package, e.g. attempting to access java.awt.FileDialog on a Quartz device
- an UnsupportedOperationException will also be thrown if an application attempts to access an unsupported optional method. For example, `Toolkit.getPrintJob()` will throw an UnsupportedOperationException if printing is not supported, as will calling Frame's constructor if an attempt is made to create a second Frame when only a single Frame is supported
- java.lang.NoSuchMethodError is only thrown if the java.util.zip.ZipFile class is not supported and an application attempts to access the constructor `ZipFile(File file)`.

4.6 The Timer Classes

PJAE includes a set of PJAE-specific classes that simplify creating timers and generating timer events. Using the timer API means that it is not necessary to create separate threads for timers: an important consideration on constrained devices where threads are a relatively expensive resource.

The timer classes are in the com.sun.util package and are documented as part of the PJAE specification (***http://www.javasoft.com/products/personaljava/***).

4.6.1 PTimerSpec

The PTimerSpec class declares when a PTimerWentOffEvent should be sent. These events are sent to the listeners registered with a PTimer instance.

A PTimerSpec may be absolute or delayed. An absolute PTimerSpec sends a PTimerWentOffEvent at the specified time; a delayed PTimerSpec sends the event after waiting the specified amount of time.

A delayed PTimerSpec may be repeating or non-repeating. A repeating PTimerSpec automatically reschedules itself after sending a timer event.

A repeating PTimerSpec may be regular or non-regular. A regular PTimerSpec attempts to send timer events at fixed time intervals, irrespective of system load or how long it takes to notify the listeners.

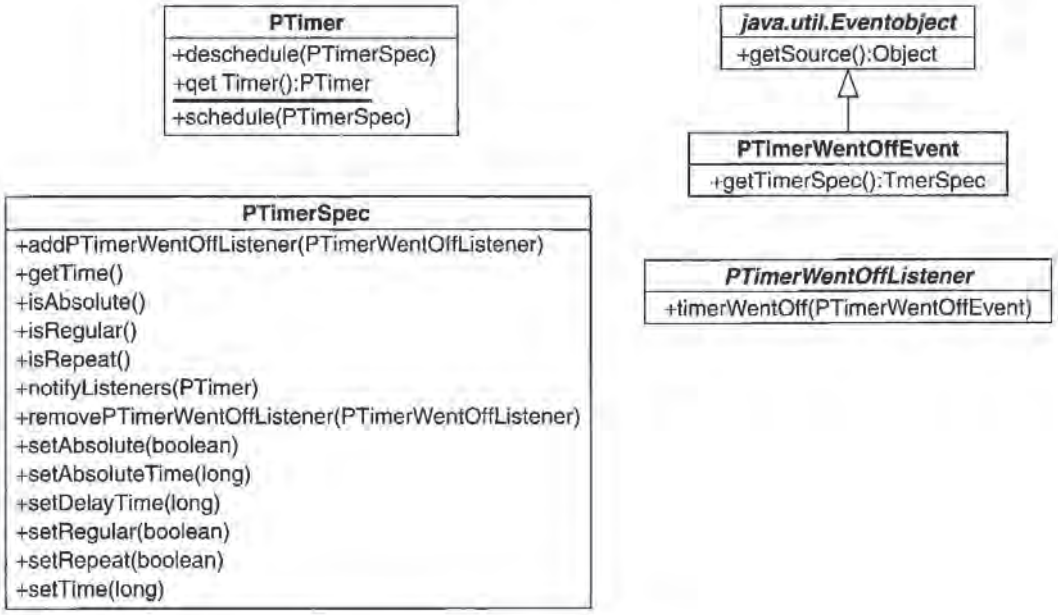


Figure 4.1 The PersonalJava timer classes.

A non-regular PTimerSpec will first call its registered listeners then wait the specified amount of time before calling its listeners again. For example, create a repeating PTimerSpec that goes off every 100 ms. Imagine that it takes 5 ms to notify the listeners. If the PTimerSpec is regular the listeners will be notified after 100 ms, 200 ms, 300 ms, and so on. If the PTimerSpec is non-regular the listeners will be notified after 100 ms, 205 ms, 310 ms, and so on.

`setDelayTime(long delay)` is a convenience function equivalent to:

```
setAbsolute(false);
setTime(delay);
setRepeat(false);
```

4.6.2 PTimer

The PTimer class is responsible for managing a set of timer events specified by PTimerSpec instances. PTimer calls the PTimerSpecs' `notifyListeners()` methods whenever timer events should be sent.

`getTimer()` is a static method used to obtain the default timer for the system. There may be one PTimer instance per virtual machine, one per Applet, one per call to `getTimer()`, or some other platform-dependent implementation.

`schedule(PTimerSpec)` registers PTimerSpecs with the PTimer. If you schedule an absolute PTimerSpec that should have gone off already, it will go off immediately. You may schedule a PTimer with multiple PTimerSpecs. You may schedule a PTimer with the same PTimerSpec multiple times, in which case it will go off multiple times. If you modify a PTimerSpec after it has been scheduled with a PTimer the results are unspecified.

4.6.3 PTimerWentOffListener

The PTimerWentOffListener interface defines a listener for PTimerWentOffEvents. The only method that needs to be implemented is `timerWentOff(PTimerWentOffEvent)`.

4.6.4 PTimerWentOffEvent

PTimerWentOffEvent has one method, `getTimerSpec()`, which returns the TimerSpec responsible for the timer event.

4.7 The Timer Tester Example

Timer tester's primary purpose is to illustrate the PersonalJava timer APIs, though it can also be used to investigate the performance of the underlying timer.

Timer tester has been written for the Crystal reference design and takes into account Crystal guidelines. In particular, it uses the Command Button Area and illustrates how components can be navigated in the absence of a pointer.

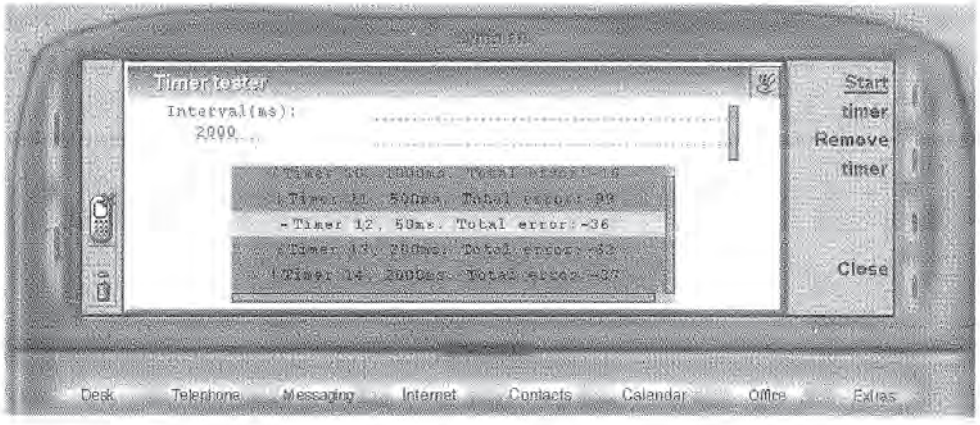


Figure 4.2 Timer tester running in the crystal emulator.

Pressing the Start new timer CBA button creates and starts a new timer with the specified interval (2000 ms in this case).

The ScrollPane lists the currently running timers. The leftmost character is one of |, /, -, \. This character provides a ticker, changing on each timer event generated by that timer. The timer number differentiates between timers. The next number shows the timer's interval (50 ms for the highlighted timer). "Total error" is the accumulated error for that timer.

Pressing the "Remove timer" CBA button removes the highlighted timer.

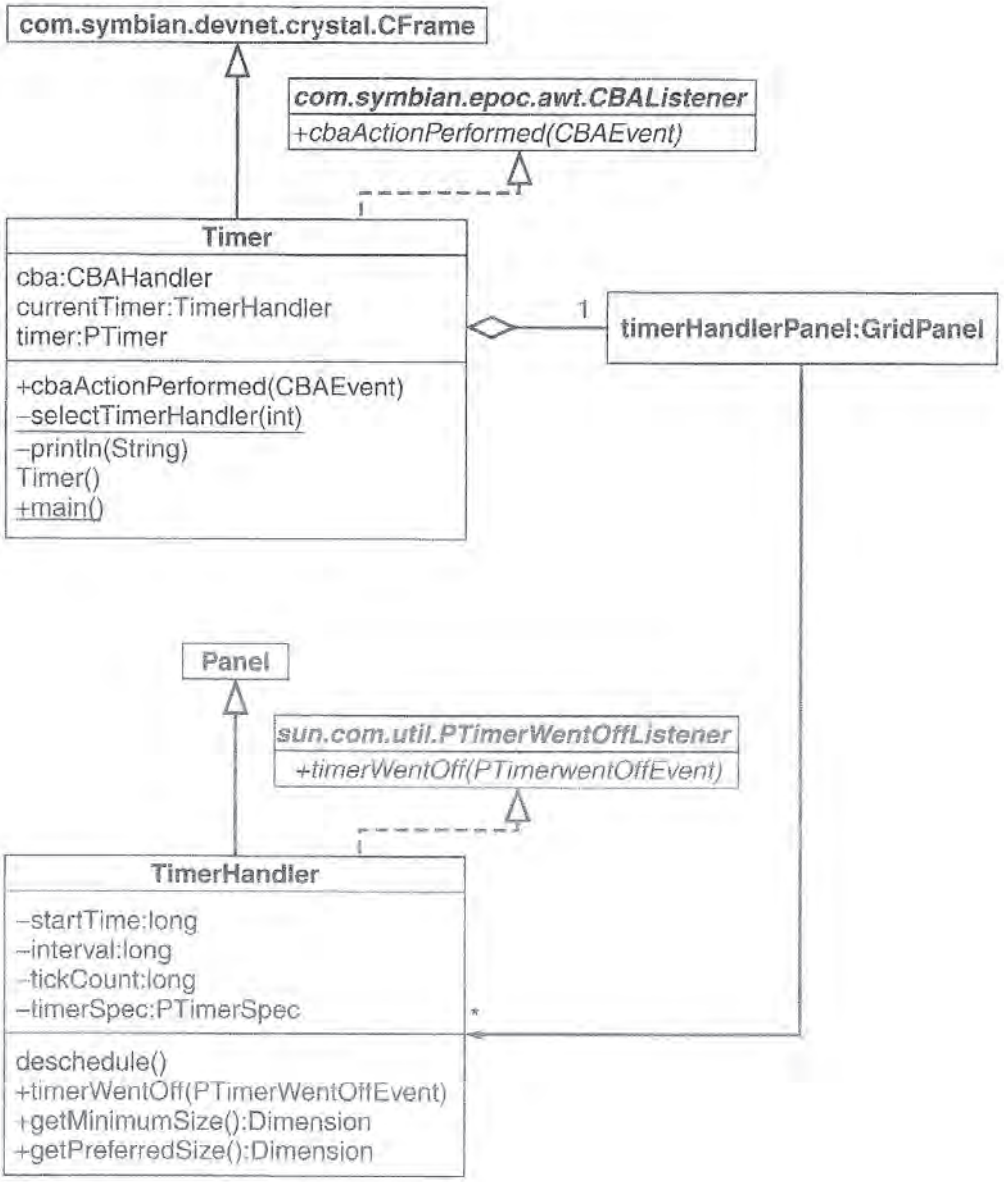


Figure 4.3 Timer tester's class diagram. TimerHandler is an inner class of Timer.

4.7.1 Architecture

The main class is Timer. Because it is written for the Crystal reference design it extends CFrame and implements CBAListener. The `timer` variable is the default PTimer timer (as returned by `PTimer.getTimer()`).

Each timer we create is an instance of a TimerHandler. TimerHandler is a Panel and so can be added to `timerHandlerPanel`. This is a GridPanel, which is basically a Panel with a GridBagLayout. TimerHandler implements PTimerWentOffListener, i.e. it is a timer event listener that can be added to `timerSpec`, a PTimerSpec. `timerSpec` is then registered with Timer's PTimer instance, `timer`, using `PTimer.schedule(PTimerSpec)`.

4.7.2 Setup

Timer is created in com.symbian.devnet.crystal.timer. The com.symbian.devnet.crystal.awt package contains CFrame, the application's super class, utility classes such as GridPanel (a Panel with a GridBagLayout, plus various methods to make it simpler to use), and CBAHandler, a wrapper class for the lower-level CBA classes supplied in com.symbian.epoc.awt (see Chapter 2). com.sun.util contains the PersonalJava timer classes.

```
package com.symbian.devnet.crystal.timer;
import com.symbian.devnet.crystal.awt.*;
import com.symbian.epoc.awt.*;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.sun.util.*;
```

cba is a reference to the CBAHandler used to control the application's CBA buttons. The next few lines define aliases for three of the four CBA buttons.

TimerHandlers are added to `timerHandlerPanel`. This GridPanel is added in turn to a ScrollPane, `timerScrollPane`. This allows an unrestricted number of TimerHandlers to be displayed.

`timer` is a reference to the system's default PTimer implementation. Each TimerHandler has an identifying number: `lastTimerNumber` is the number of the most recently created TimerHandler. `currentTimer` keeps track of the currently selected TimerHandler. `currentTimer` is a Component which has been added to `timeHandlerPanel`, a Container: timerPosition is this TimerHandler's position in the `timerHandlerPanel` Container.

```
public class Timer extends CFrame implements CBAListener
{
    private static CBAHandler cba; // Handles the CBA buttons
    static final int NEW = EikCommandButtonGroup.BUTTON1;
    static final int REMOVE = EikCommandButtonGroup.BUTTON2;
    static final int CLOSE = EikCommandButtonGroup.BUTTON4;

    private static TextField intervalT = new TextField("1000", 6);
    private static GridPanel timerHandlerPanel = new GridPanel();
    private static ScrollPane timerScrollPane = new ScrollPane();
    private static TextArea statusT = new TextArea
       ("", 3, 40, TextArea.SCROLLBARS_VERTICAL_ONLY);
    private static final Color NILE_BLUE = new Color(221, 221, 255);

    private static PTimer timer;
    private static int lastTimerNumber;
    private static TimerHandler currentTimer;
    private static int timerPosition = -1;
```

4.7.3 Constructor

The constructor creates the user interface. This allows a timer to be created with a specified interval, displays the created timers, and allows a timer to be removed.

The constructor adds a KeyListener and FocusListener to the timerScrollPane. The former is used to pick up Up arrow and Down arrow key presses and so move highlighting to a previous or later TimerHandler. The latter is used to make the `timerScrollPane's` background color lighter or darker when it has lost or gained focus:

```
Timer()
{
     setTitle(Version.TITLE);
     aboutDialog.setTitle("About " + Version.TITLE);
     aboutDialog.text = Version.DIALOG_TEXT;

     GridPanel panel = new GridPanel();
     panel.constraints.weightx = 1.0;
     panel.add(new Label("Interval(ms):"), 0, 0);
     panel.add(intervalT, 0, 1);
     panel.constraints.gridheight = 2;
     panel.add(statusT, 1, 0);
     statusT.setEnabled(false);

     timerHandlerPanel.constraints.insets = new Insets(0, 0, 0, 0);
     timerScrollPane.setSize(350, 108);
     timerScrollPane.setBackground(Color.lightGray);
     timerScrollPane.add(timerHandlerPanel);
```

Add a KeyListener to `timerScrollPane`:

```
     timerScrollPane.addKeyListener(
          new KeyAdapter()
          {
               public void keyPressed(KeyEvent ke)
               {
                    if(ke.getKeyCode()==KeyEvent.VK_DOWN)
                    {
                         if(timerPosition <
                         timerHandlerPanel.getComponentCount() - 1)
                         {
                              timerPosition++;
                              selectTimerHandler(timerPosition);
                         }
                    }
```

```
                if(ke.getKeyCode() == KeyEvent.VK_UP)
                {
                    if(timerPosition > 0)
                    {
                        timerPosition--;
                        selectTimerHandler(timerPosition);
                    }
                }
            }
        }
    );
```

Add a FocusListener to the `timerScrollPane`:

```
timerScrollPane.addFocusListener(
    new FocusListener()
    {
        public void focusGained(FocusEvent fe)
        {

            timerScrollPane.setBackground(timerScrollPane.
                getBackground().darker());
            if(currentTimer !=null)
                currentTimer.setBackground(NILE_BLUE);
        }

        public void focusLost(FocusEvent fe)
        {

            timerScrollPane.setBackground(timerScrollPane.
                getBackground().brighter());
            if(currentTimer !=null)
                currentTimer.setBackground(NILE_BLUE);
        }
    }
);
```

`timerScrollPane`'s vertical adjustment is chosen so that it scrolls about the height of a TimerHandler when the Up or Down cursor keys are pressed:

```
Adjustable adjustable = timerScrollPane.getVAdjustable();
adjustable.setUnitIncrement(15);

panel.constraints.gridheight = 1;
panel.constraints.gridwidth = 2;
panel.add(timerScrollPane, 0, 2);
```

Set up a CBAHandler. Making the `NEW` CBA button the default means that pressing the return key will create and display a new TimerHandler. This will happen irrespective of which UI component has the focus:

```
cba = new CBAHandler(this);
cba.setText(NEW, "Start new\ntimer");
```

```
cba.setText(REMOVE,"Remove\ntimer");
cba.setEnabled(REMOVE, false);
cba.setText(CLOSE,"Close");
cba.setDefaultButton(NEW);
cba.activate();

add(panel);
show();

intervalT.requestFocus();
```

Get the system timer implementation:

```
        timer=PTimer.getTimer();
}
```

4.7.4 Miscellaneous Methods

`main()` creates an instance of Timer:

```
public static void main(String[] args)
{
    new Timer();
}
```

`println()` displays a line of text in the status area:

```
public final void println(String text)
{
    statusT.append(text + "\n");
}
```

`selectTimerHandler(timerPosition)` selects the specified TimerHandler in the `timerHandlerPanel` as the current timer. The selected TimerHandler is highlighted. The method takes care to ensure that the required position is greater than or equal to zero and less than the number of TimerHandlers in the ScrollPane. If all the TimerHandlers have been deleted (i.e. timerPos is −1) then the REMOVE CBA button is disabled:

```
private static void selectTimerHandler(int timerPos)
    {
        if(currentTimer ! = null)
currentTimer.setBackground(timerScrollPane.getBackground());
        if(timerPos == timerHandlerPanel.getComponentCount())
            timerPos--;

        if(timerPos < 0)
        {
             currentTimer = null;
             cba.setEnabled(REMOVE, false);
        }
```

```
            else
            {
                cba.setEnabled(REMOVE, true);
                currentTimer =
(TimerHandler)timerHandlerPanel.getComponent(timerPos);
                currentTimer.setBackground(NILE_BLUE);
            }
            timerPosition = timerPos;
        }
```

4.7.5 The CBA Handler

`cbaActionPerformed()` handles the three CBA buttons we've defined.

Pressing the NEW button creates a new TimerHandler with the specified interval. If an incorrect interval is entered we'll catch and handle the NumberFormatException, though because this is a RuntimeException we are not forced to do so.

Pressing the REMOVE button deschedules the current timer. This removes it from `timerHandlerPanel` and cleans up various resources.

Pressing the CLOSE button shuts down the application.

```
    public void cbaActionPerformed(CBAEvent cbae)
    {
        switch(cbae.getID())
        {
            case NEW:
                try
                {
                    int interval =
Integer.parseInt(intervalT.getText()); // Get the interval
                    TimerHandler timerHandler = new
TimerHandler(lastTimerNumber, interval); // Create our timer
                    lastTimerNumber++;
                    if(timerPosition == -1) selectTimerHandler(0);
                }
                catch(NumberFormatException nfe)
                {
                    println("You have entered an invalid number");
                }
            break;

            case REMOVE:
                currentTimer.deschedule();
            break;

            case CLOSE: // Close the program
                super.shutDown();
            break;

            default:
        }
    }
```

4.7.6 The TimerHandler Class

An instance of a TimerHandler represents a timer. It encapsulates a PTimerSpec, referenced by `timerSpec`. TimerHandler implements the PTimerListener interface which means we can add it to the `timerSpec` PTimerSpec and listen for timer events generated by it. This code can be modified to make the timer a one-shot absolute or delayed timer, or a repeating regular or irregular timer.

TimerHandler extends Panel, so we can treat it as a UI Component.

`tickCount` keeps track of the number of times the timer has ticked, and together with `startTime`, is used to calculate the accumulated error.

```
final class TimerHandler extends Panel implements
    PtimerWentOffListener
{
    private int interval;
    private int tickCount;
    private long startTime;
        private PTimerSpec timerSpec;
    private Label tickerLabel;
    private Label nameLabel = new Label();
    private  Label errorMessageLabel = new Label(" Total error:");
    private Label errorLabel = new Label(".....");
    private String[] tickerSymbols = {"\\","| ","/","-" };
    private int tickerSymbolIndex;
```

The constructor takes the identifying number for the TimerHandler and the timer interval as arguments. We then create four labels and add them to this TimerHandler (which we can do because it is a Panel):

- `tickerLabel`: displays a sequence of four characters so that we can see that the timer is ticking
- `nameLabel`: the name of this timer (eg "Timer 3")
- `errorMessageLabel`: just the text "Total error: "
- `errorLabel`: displays the timer's accumulated error in ms

```
TimerHandler(int timerNumber, int interval)
{
    this.interval = interval;
    tickerLabel = new Label(tickerSymbols[tickerSymbolIndex++]);
    add(tickerLabel);
    nameLabel.setText("Timer " + timerNumber + ", " + interval
        + "ms. ");
    add(nameLabel);
    add(errorMessageLabel);
    add(errorLabel);
    timerHandlerPanel.add(this, 0, timerNumber);
    timerScrollPane.validate();
```

We create a PTimerSpec, make it relative and set the interval. We then schedule the PTimer with the PTimerSpec, which starts our timer, and get the current time. By default, timers are regular timers; however, in this case we've explicitly set the timer to be regular (`setRegular(true)`):

```
    timerSpec = new PTimerSpec();
    timerSpec.setAbsolute(false);
    timerSpec.setRepeat(true);
    timerSpec.setRegular(true);
    timerSpec.setTime(interval);
    timerSpec.addPTimerWentOffListener(this);
    timer.schedule(timerSpec);
    startTime = System.currentTimeMillis();
}
```

`timerWentOff()` implements the PTimerWentOffListener interface. It calculates how much time should have elapsed and how much has elapsed according to the system clock. It uses the `errorLabel` Label to display the difference. A positive error means the timer is too slow, a negative error means it is too fast.

The method also displays the next ticker symbol to show that the timer is still ticking:

```
public void timerWentOff(PTimerWentOffEvent woe)
{
    tickCount++;
    long desiredElapsedTime = tickCount*interval;
    long elapsedTime = System.currentTimeMillis() - startTime;
    long error = desiredElapsedTime - elapsedTime;
    tickerLabel.setText(tickerSymbols[(tickerSymbolIndex++)% 4]);
    errorLabel.setText("" + error);
}
```

Deschedule our timer. We need to remove the listener so that the instance can be garbage collected. `selectTimerHandler()` is used to select the next TimerHandler in the `timerHandlerPanel`:

```
private void deschedule()
{
    timer.deschedule(timerSpec);
    timerSpec.removePTimerWentOffListener(this);
    timerHandlerPanel.remove(this);
    currentTimer = null;
    timerScrollPane.validate();
    selectTimerHandler(timerPosition);
}
```

The following two methods ensure that our TimerHandler isn't resized by a LayoutManager:

```
        public Dimension getMinimumSize()
        {
            return getPreferredSize();
        }

        public Dimension getPreferredSize()
        {
            return new Dimension(350,20);
        }
    }
}
```

4.7.7 Experimenting with Timer Tester

If you start a timer you will probably see the error steadily increase and then drop back to zero or near zero as the timer corrects itself. It is worth noting that timer resolution under Windows is typically 100 ms whereas the Symbian OS is 1/64th of second, or 15.625 ms.

The shortest timer under the emulator appears to be around 30 ms, though the reported error is frequently greater than this.

A regular timer will constantly try to correct itself. Start a timer in the emulator and then put it into sleep mode (you should be able to do this by pressing the F9 key, though you may have to bring another application to the foreground first). When you wake up the emulator you'll see the timers catch up with themselves. This effect becomes most apparent with a fast timer, say 20 ms.

Unfortunately, PersonalJava timers are effectively deactivated when in sleep mode, which means that they can't be used for alarms.

We can make the timers irregular rather than regular by modifying the TimerHandler's constructor. Change the line

```
timerSpec.setRegular(true);
```

to

```
timerSpec.setRegular(false);
```

It is instructive to then watch the timer's accumulated error field. You should see the error steadily increase because the timer isn't reset until after the timer's listeners have been called. If calling the listeners takes, say, 5 ms then each tick will take an extra 5 ms.

You can make the timers single shot by removing the lines:

```
timerSpec.setRepeat(true);
timerSpec.setRegular(true);
```

4.8 Compiling with PersonalJava

There is a small potential "gotcha"when compiling PersonalJava applications. For the most part you can get away with compiling against the JDK1.1 classes; however, this won't work if you want to use PersonalJava-specific classes or methods. In this case you must compile against classes.zip as supplied in the relevant Symbian Emulator, and in general it's good practice to compile against these classes.

For instance, here's the command used to compile the Crystal version of the Timer application discussed in the previous section:

```
javac -d code -classpath
\erj\lib\classes.zip;\erj\utilities\awt\cawt.jar source\*.java
```

5

A JavaPhone Primer

Jonathan Allin

This chapter looks at the JavaPhone APIs from the mobile phone perspective. It provides an overview of the JavaPhone architecture and Symbian's JavaPhone implementation in Section 5.2 while Section 5.3 provides useful help on getting started with JavaPhone on the Symbian platform.

The remainder of the chapter covers the individual JavaPhone APIs. The explanations are intended to complement Sun's API documentation, not to replace it. A brief overview is given for each API, together with a discussion of those issues which might be harder to understand or which are Symbian specific. UML class diagrams show the relationship between the classes in a package and list those methods of particular interest. Each API is illustrated with an example.

5.1 Introduction

Scenario 1:	**Scenario 2:**
You're on the train to work. You need to schedule an urgent meeting with several colleagues. You take out your personal organizer, read off some phone numbers from the address book and key them into your mobile phone. You talk to each colleague in turn to try to establish a mutually convenient time. When you have finally succeeded you write the appointment into your calendar and embark on another round of calls asking your colleagues to do likewise. The train pulls in at the station and in your haste you leave your newspaper unread on the seat...	Same place. Same problem. You take out your JavaPhone-enabled Smartphone and start up your "Meeting Minder" Java application. This assembles a list of colleagues from your address book and sends agents to their Smartphones to check their daytime calendars. You pick up and read your newspaper. You're notified when the agents have reported back. You turn away from your newspaper for a moment to hit "Go." This pages your colleagues advising them of the mutually acceptable meeting time.

The JavaPhone APIs are important in enabling Java technology to deliver the wireless opportunities we discussed in the Introduction by making access to the telephony and PIM features of a Wireless Information Device straightforward and device independent. Through the APIs an application can:

- initiate and handle telephone calls via the JTAPI interfaces
- access the address books: read, write, and search. Both the phone's SIM card and main address book can be accessed
- access the calendar functionality
- send SMS messages and send and receive other message formats such as UDP or WDPSMS using the Datagram APIs
- make direct use of the device's serial or infrared ports
- respond appropriately to low power situations, perhaps by saving data or posting a warning.

And Symbian is providing the JavaPhone reference implementation.

5.2 The JavaPhone Architecture

5.2.1 Overview

JavaPhone is a vertical extension to the PersonalJava platform developed through an open process between Sun Microsystems and key leaders in the telecommunications industry, including Symbian.

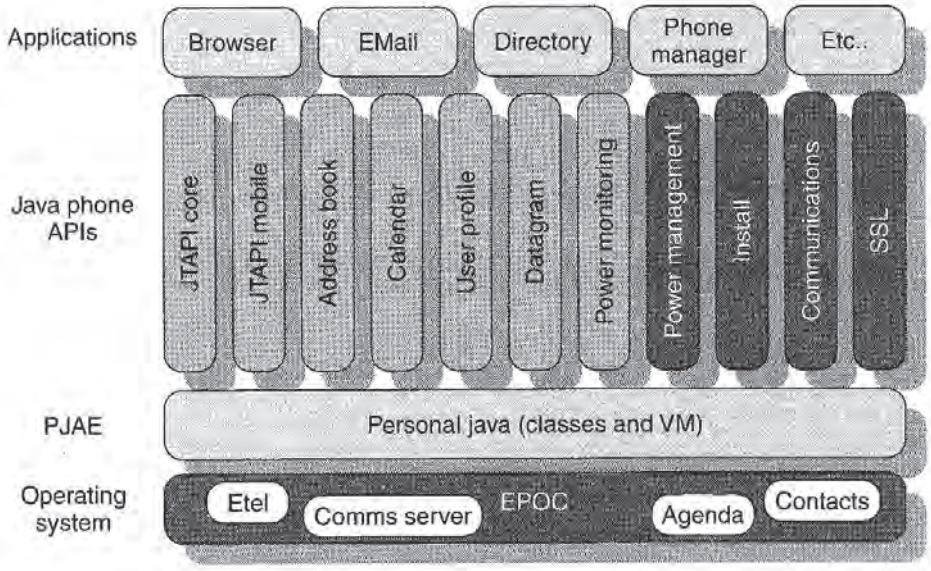


Figure 5.1 The JavaPhone architecture.

JavaPhone defines two profiles: an Internet screen phone profile and a mobile phone profile. The differences between the two are essentially to do with the required and optional APIs. The diagram represents the mobile phone profile, where the required APIs are in light gray and the optional APIs are in dark gray. The only required APIs for the Internet screen phone profile are JTAPI core, Address book, and User profile.

5.2.2 The JavaPhone APIs

The JavaPhone APIs give Java applications access to the telephony-related features and functionality of the underlying operating system, which in Symbian's case is the Symbian OS.

JavaPhone comprises a number of extension packages and, as with other extension packages from Sun, the intention is that the required functionality is provided by the host operating system. Factory methods are used to obtain an API implementation which is then accessed using classes that implement the relevant JavaPhone interface.

Table 5.1 summarizes the JavaPhone packages.

Table 5.1 The JavaPhone packages

API (required/optional) Implementing package	Description	Symbian's implementation
JTAPI Core (required) javax.telephony	Provides the basic framework to model telephone calls and to create, answer, and disconnect a call	
— (optional) javax.telephony.callcontrol	Provides advanced telephony features such as conferencing and transferring calls, call forwarding	Not currently supported
— (optional) javax.telephony.phone	Provides details of, and control over, a handset's advanced features, such as speakerphone, microphone, display, buttons, ringer	Not currently supported
JTAPI mobile (required) javax.telephony.mobile	Manages network selection and identification, network and radio status, switching the transmitter/receiver on, monitoring the signal level, plus other features which are unique to mobile telephony devices	
Address book (required) javax.pim.addressbook	An application using the Address book APIs can locate and update contact information such as phone numbers, postal addresses, email addresses, and other information about individuals, groups, and organizations. Schema strings used in querying and storing contact information are those specified in vCard, a personal data interchange format that has been accepted as a standard method to transfer contact information	

(continued overleaf)

Table 5.1 *(continued)*

API (required/optional) Implementing package	Description	Symbian's implementation
Calendar (required) javax.pim.calendar	Provides applications with access to schedule and calendar information, such as dates and task entries, and the ability to access, add, delete, and update the event information in a Calendar database. Schema strings used in querying and storing are those specified in iCalendar, an interchange format that has been accepted as a standard way to transfer scheduling information	
— (required) javax.pim.database	Provides Addressbook and Calendar with a common database model	
User profile (required) javax.pim.userprofile	Defines objects to get and set the current user contact information, which is returned in the same format as the AddressBook package	
Datagram (required) javax.net.datagram	Allows bearer agnostic messaging. Used for exchanging messages with devices that don't have an IP address, such as SMS over GSM. Addresses consist of a service name and location. The physical network or bearer used to deliver the message is selected by the device when the message is sent	Supports UDP over IP, SMS over GSM, and WDP over SMS
Power monitoring (required) javax.power.monitor	Monitor and respond to changes in the power level of the system. An application is able to retrieve the current battery level, an estimate of remaining battery life, and whether an external power source is being used. This package also enables an application to be notified when the battery power is running low and a service is about to be terminated	
Power management (optional) javax.power.management	Monitor and respond to changes in the power states of the system. For example, the "on" power state can be identified, and an application can respond to various conditions such as full power, power managed for efficiency, suspend, and sleep states. Privileged applications can also set the power state, e.g. switch a device off or put it into sleep mode	Not currently supported
Install (optional) javax.ce.install	Supports the following functions: • packaging an application for distribution • installation mechanisms • versioning to identify what version of an application is present • removing an application	Not currently supported. Symbian provides its own native installer

Table 5.1 *(continued)*

API (required/optional) Implementing package	Description	Symbian's implementation
Communications (optional) javax.comm	Provides access to a device's serial and parallel ports	Supported and typically provides access to an EPOC device's IR and RS232 ports
SSL (optional) javax.net.ssl	Provides secure communication over TCP/IP sockets	Not currently supported. Support for the HTTPS protocol via URL Connections is supported but the SSL APIs are not

In summary, the following packages are currently unsupported by the Symbian implementation:

- javax.ce.install
- javax.power.management
- javax.telephony.callcontrol
- javax.telephony.phone
- javax.net.ssl

PersonalJava does not support a fine-grained security model, therefore Applets cannot be given access to any of the JavaPhone APIs.

5.3 Working with JavaPhone on Symbian OS

The next few sections explore in turn each of the JavaPhone APIs supported by Symbian's implementation. Each section is accompanied by example code.

The examples are targeted at either the Quartz or Crystal reference designs. They are intended to run under the relevant Quartz or Crystal emulator or on actual hardware. You will need to download either the Quartz SDK for Java or the Crystal SDK for Java from the Symbian Developer Network website, ***http://www.symbiandevnet.com***.

To test the telephone API examples you will also need access to an Ericsson i888 phone (for the US or Europe) or an Ericsson SH888 phone (for Europe) and an Extended Systems JetEye IR podule for your PC. Other phones that appear to work, but have not been fully tested include Ericsson T28 and R320.

If you want to use the serial port, either for direct serial communications using javax.comm, or to drive the IR link to a mobile phone, you must ensure nothing else is using the port (such as PsiWin and the PsiWin connection server).

Using the IR port emulation may require modifications to a file called **irda_wins.esk**. This is found in **\epoc32\wins\c\system\data**. This a text file that contains a section like this:

```
filename= irda.prt
index= 2
irPod= jeteye7401
irBaud= 115200
irPhysicalComPort= 1
irSupportedBaudRates= 319
```

`irPhysicalComPort` refers to the com port as seen by the Emulator. So set this value to 0 if you have plugged your IR podule into COM1 on your PC, and to 1 if you have plugged it into COM2.

If the IR podule link seems unreliable, try changing `irBaud` to, say, 57600.

The Ericsson 888 requires some setting up:

- ensure that its phone number is set. Most phones do not know their own phone number, so menu through to Settings, Yes, menu through to Phone number, Yes, Yes, enter phone number. From then on the phone number can be read off the SIM card.
- you need to enable the IR port. Go to Activate IR port and select Yes. Note that the IR port is disabled when you switch the phone off.
- instead of using the Extended Systems Jeteye pod you can use the serial cable that comes with the Ericsson phone. Everything else remains the same, in particular the IR port must still be enabled on the phone.

The examples make use of simple frameworks that provide some of the behavior important to Symbian applications: Quartz versions of the examples extend QFrame, while Crystal versions extend CFrame. These frameworks are provided by the com.symbian.devnet.quartz.awt and com.symbian.devnet.crystal.awt packages respectively and are described in detail in the Quartz and Crystal Chapters 2 and 3.

In order to avoid unnecessary and potentially confusing code, the UIs are kept as simple as possible and there is the minimum of boilerplating. However, where appropriate we have kept to PDA coding guidelines in general and Symbian style guidelines in particular. For instance, to improve performance and reduce code size, methods and variables are made private, static, and/or final unless there is a good reason not to do so.

5.4 The JTAPI APIs

We take the simplicity of telephones for granted and the basic concepts are straightforward. I have a phone (mobile or fixed) which has its own telephone

number. I make a phone call to another number and (hopefully) someone at the other end answers.

This is the most common model; however, telephones can become much more complicated. Many handsets have a private number and a work number, a conference call will involve more than two parties, and so on. Further, JTAPI must also cover call centres, routers, PC and desktop applications, fixed line phones, and mobile phones.

In order to cope with the required flexibility JTAPI carefully defines a number of concepts and relationships. We'll now go through the core concepts, roughly in the order in which you would encounter them when creating a simple JTAPI application (a note on notation: Class names are capitalized, e.g. Address, Connection. A general concept, such as a handset, is not):

JtapiPeer: JTAPI accesses the underlying system's telephony implementation through a set of defined interfaces. However, it needs a starting point: the JtapiPeer object is a Java reference to this implementation.

Provider: A Provider provides access to a particular telephone network service, such as GSM, voice over IP, or fixed line. The Provider is the "window" through which JTAPI applications see the underlying telephony system. It allows us to create a Call and to query the capabilities of our implementation. A JtapiPeer can support multiple Providers.

Address: An Address represents what we commonly think of as a "telephone number." However, in implementations where the underlying network is not a telephone network the address may represent something else; for example, if we are using IP, the address might represent an IP address (e.g. "172.16.204.15").

Terminal: A Terminal represents the physical endpoints of a telephone call (the nearest thing to a handset). One Address may have several Terminals, and one Terminal may have several Addresses. For instance, we have two secretaries, Sally and Jo. If I phone the secretariat both their phones ring (i.e. one Address, two Terminals). However, in addition, Jo and Sally each have their own personal phone numbers, so their Terminals also have two Addresses.

Call: A Call is used to represent the state of a telephone call and the parties connected to that call. When a Call is first created, it has no Connections. When a simple two-party call is set up, two Connections are created, representing the local Address from which the Call is being made and the Address being called.

Connection: A Connection describes the relationship between a single Call and a single Address and the state of that relationship, e.g. idle, alerting, connected, disconnected.

5.4.1 The Basic Model

The basic framework to model telephone calls and to create, answer, and disconnect a call is provided by the javax.telephony package. Figure 5.2 helps

explain the relationships between the principal JTAPI objects that make up the package.

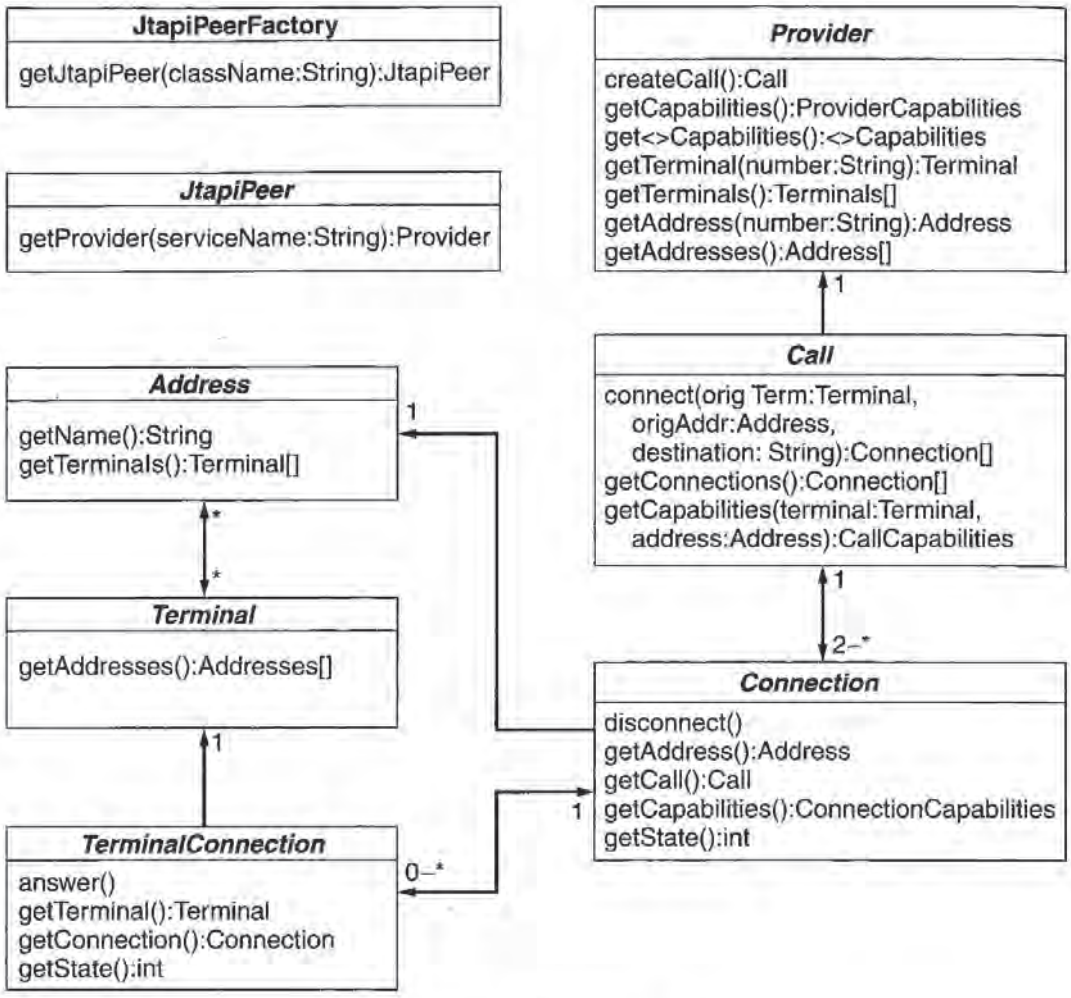


Figure 5.2 The JTAPI model.

JtapiPeerFactory Class

The only concrete class is JtapiPeerFactory (for simplicity all the other objects are named after the JTAPI interface they implement, rather than showing the implemented interface as a separate object).

`JtapiPeerFactory.getJtapiPeer(String jtapiPeerName)` is used to obtain a reference to the system's telephony implementation. It is a static method that returns an instance of a JtapiPeer object. `jtapiPeerName` is the fully qualified name of a class which implements the JtapiPeer object, e.g. "com.symbian.javax.telephony.EpocJtapiPeer." If a default implementation is defined (perhaps by an environmental variable) it will be returned if `jtapiPeerName` is null.

Provider Class

Given a JtapiPeer we can obtain a Provider using `JtapiPeer.getProvider(String serviceName)`. To obtain the default service we can use `getProvider(null)` or we can provide a service name, e.g. `getProvider("EpocGsmMobileProvider")`. A list of available service names can be obtained using `JtapiPeer.getServices()`.

5.4.2 Capabilities

Capabilities define what our JavaPhone implementation can and cannot do. Static capabilities define the limits of our implementation, and these limits will not change over the lifetime of the Provider (e.g. will this Provider allow me to connect a call?). Dynamic capabilities, on the other hand, define what a JTAPI object (e.g. a Provider, Address, Call) can do at the time the query is made (e.g. can I connect this call?).

Static capabilities are established using methods of the form `Provider.get<>Capabilities()`, where `<>` can be one of Address, Connection, Provider, Terminal, or TerminalConnection. These methods return the appropriate capabilities object, e.g. AddressCapabilities.

The Provider's dynamic capabilities are obtained using `Provider.getCapabilities()`, which returns a ProviderCapabilities object. Other JTAPI objects provide their own methods for determining their dynamic capabilities, e.g. `Call.getCapabilities(Terminal terminal, Address address)` returns the dynamic capabilities for the Call instance in the form of a CallCapabilities object.

The telephony, telephony.callcontrol, and telephony.phone packages each has associated subpackages for capabilities and events. The telephony.mobile package excludes these associated packages to minimize the implementation's memory footprint, an important issue in mobile devices.

5.4.3 Addresses

An Address is not explicitly created: local Addresses, which are Addresses known to the Provider (and include the handset's phone number(s)), are obtained from the Provider using `Provider.getAddress(String name)`. The Address of a remote Terminal can be obtained from a Connection using `Connection.getAddress()`.

An Address object has a String name which corresponds to this telephone address, and this can be obtained using `Address.getName()`.

5.4.4 Making (and Disconnecting) a Call

Perhaps the Provider's most important method is `createCall()`. This returns a Call object which is used to make a connection.

To actually make a connection to another handset we use `Call.connect(Terminal origTerm, Address origAddr, String destination)`. Looking at each parameter in turn:

`origAddr` is an originating Address, i.e. the Address, or one of the Addresses, of our handset. This is obtained using `Provider.getAddress(String name)`. `name` is the String representation of one of the Addresses known to the Provider (the list of known Addresses is returned by `Provider.getAddresses()`) and will typically be a phone number, e.g. "+447746456321," but could be an IP address.

`origTerm` represents the originating terminal. `Provider.getTerminals()` returns an array containing the Terminals associated with our handset so we simply pick one of these.

`destination` specifies our destination. This is a String which is used by the Provider to create a destination Address. Typically this String will be a phone number, e.g. for GSM-based providers, but as discussed earlier it could also be an IP address.

If successful, `Call.connect()` returns an array containing exactly two Connections: the first is the originating Connection and the second is the destination Connection (a normal Call has two Connections, while a conference call will have three or more Connections).

A Connection describes the relationship between a Call and an Address. The state of the relationship is obtained using `Connection.getState()` and will be one of alerting, connected, disconnected, failed, idle, in progress, or unknown. `getCall()` and `getAddress()` return the Call and Address: in a typical Call the latter will return either the originating Address or the destination Address (depending on whether we are accessing the originating or destination Connection). `disconnect()` is used to drop the connection between the Call and the Address.

Connections are containers for zero or more TerminalConnections: an Address may have more than one Terminal, and so a Connection will require a TerminalConnection for each of these Terminals. A TerminalConnection describes the relationship between a Connection and a Terminal. It provides a physical view of the Call. The relationship's state can be obtained using `TerminalConnection.getState()`.

`TerminalConnection.answer()` answers an incoming telephone call on this TerminalConnection. It blocks until the telephone call has been answered at the endpoint. When it returns successfully the state of this TerminalConnection object is `TerminalConnection.ACTIVE`.

5.4.5 Of Observers, Listeners and MetaEvents

A single change to a call as represented by the telephony platform, such as a dropped line, may generate several JTAPI events at once.

A JTAPI **Observer** will obtain an array of events. For example, a CallObserver implementation will provide a method with the signature:

```
void callChangedEvent(CallEv[] eventList)
```

The Observer should then iterate through this array, responding in the relevant way to each event.

JTAPI 1.3 introduced the Java 1.2 event model, whereby single events are delivered to a Listener object. In order to accommodate smaller platforms the memory footprint has been lessened by reducing the number of interfaces. A JTAPI **Listener** implements the relevant Listener interface and provides methods for each event to which the Listener can respond. Because events are delivered one at a time we need a way of grouping events that are generated by single state change in the underlying implementation. JTAPI 1.3 provides new events, called **MetaEvents**, which show when such a group begins and ends. A Listener should therefore wait until its `<>Meta<>Started(MetaEvent)` method has been called, store all subsequent events until its `<>Meta<>Ended(MetaEvent)` method has been called, then process the stored events.

With the old Observer event model, large-scale activities such as third-party call transfer and conferencing are awkward to detect and process. With the new Listener model and meta events an application has a Listener interface with methods and events which convey, directly and with necessary details, what applications need to know to respond to higher-level events.

Not all JTAPI packages provide both Observers and Listeners. The breakdown is like this:

javax.telephony	Observer, Listener
javax.telephony.callcontrol	Observer
javax.telephony.mobile	Listener
javax.telephony.phone	Observer

Figure 5.3 lists all the JTAPI event listeners. The signature for the listener methods are identical: they take a single argument of type `<className>Event` and are all void.

Events carry information about the JTAPI object whose state has changed. The state change is conveyed to the application in two ways. First, the provider reports the event to one of the relevant Listener's methods (generally the method corresponds to a particular state change). Second, the event that is presented to the method has an identification integer which indicates the specific state change: `Event.getID()` returns this identification number for each event.

Each event is associated with a cause or reason for the event. The `Event.getCause()` method returns this cause value.

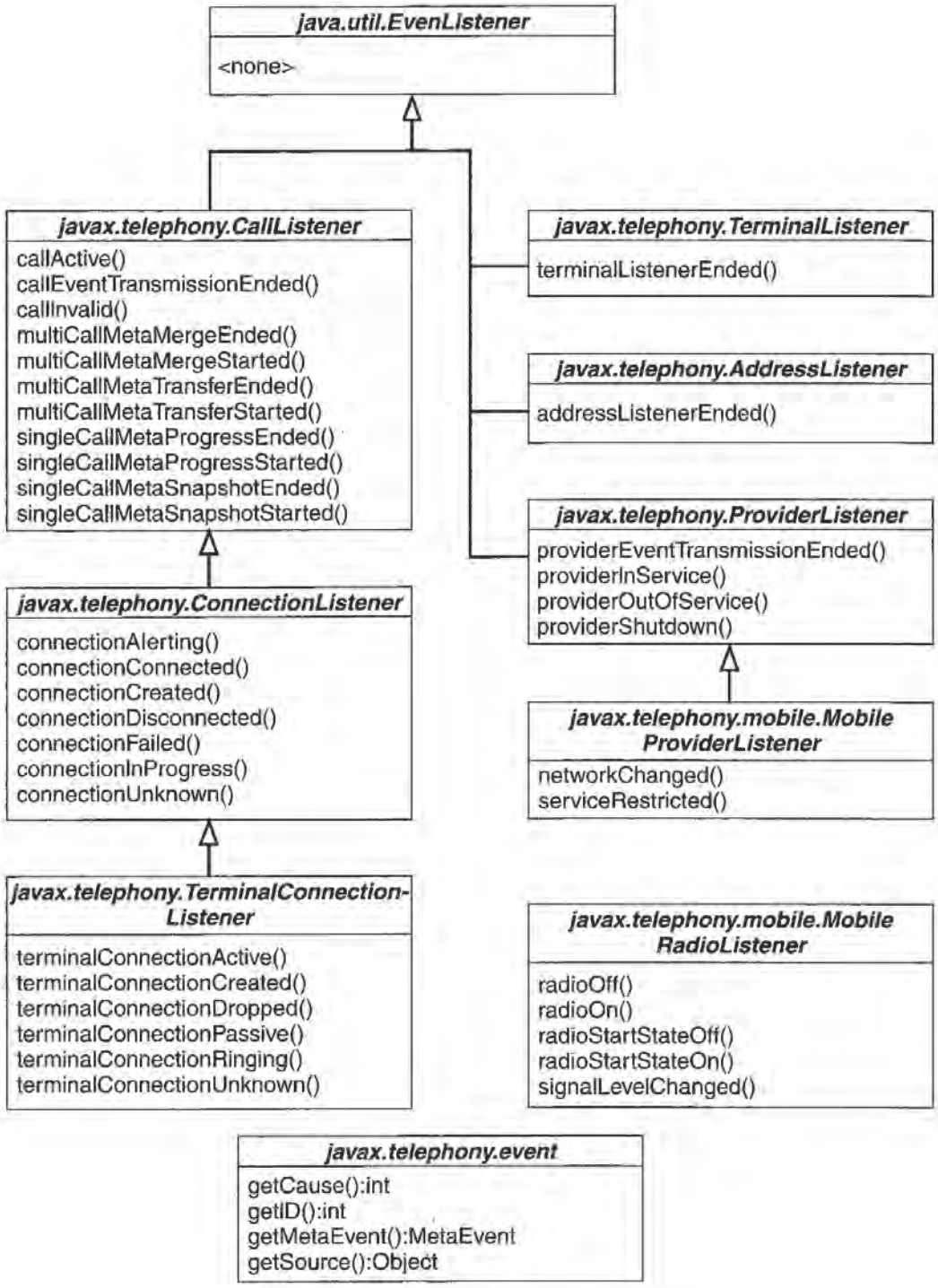


Figure 5.3 The JTAPI event listeners.

The JTAPI core package defines a hierarchy of event interfaces. The base of this hierarchy is the javax.telephony.Event interface. Figure 5.4 shows the complete structure.

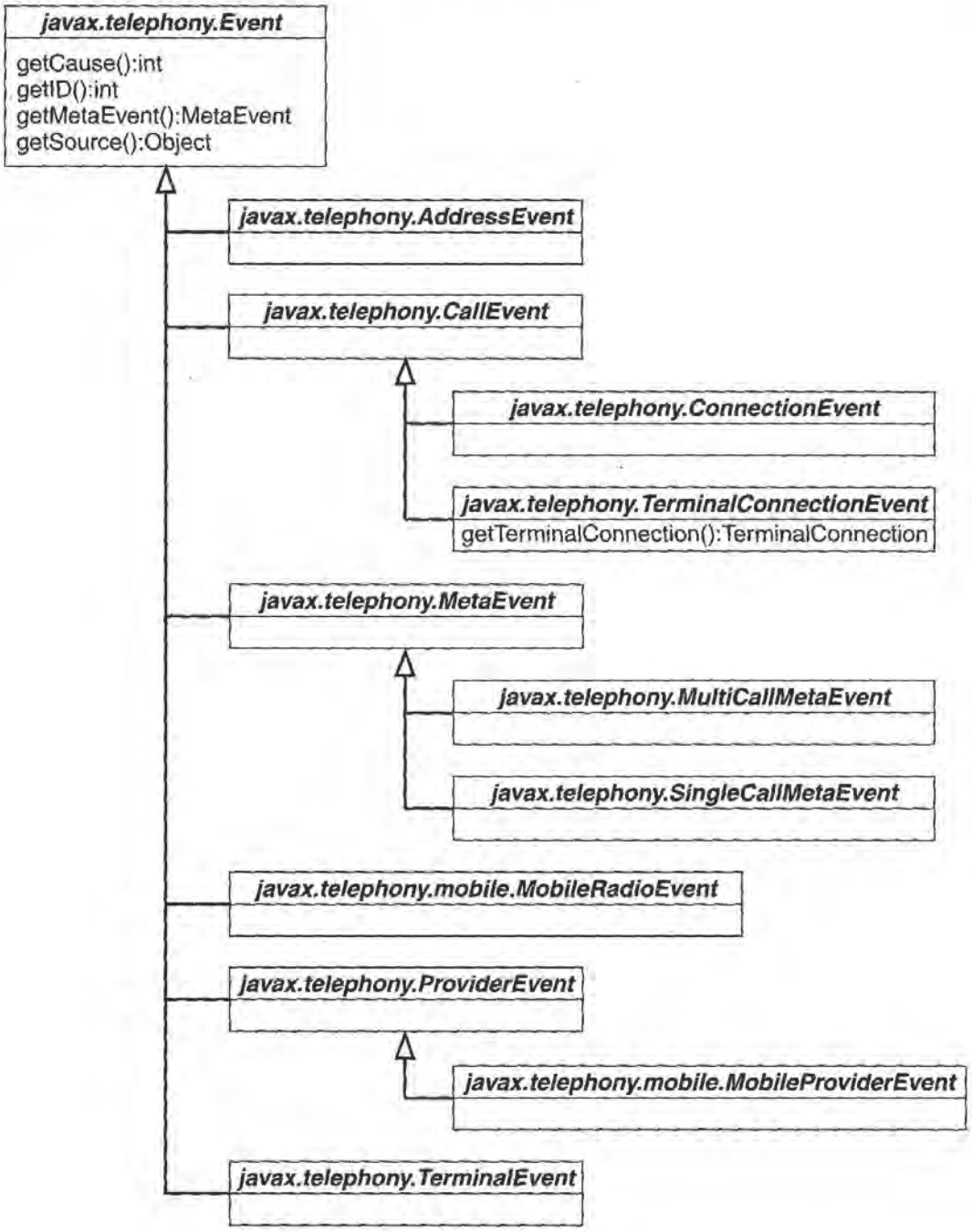


Figure 5.4 The JTAPI events.

5.4.6 The JTAPI Mobile Package

The mobile package, javax.telephony.mobile, covers features specific to mobile networks. Table 5.2 gives a summary of the principal classes and their functions.

Table 5.2 The principal classes and their functions

Interface	Description
MobileProvider extends Provider	The interface to the wireless implementation Provides methods to list the available MobileNetworks, select a MobileNetwork, get the current MobileNetwork, and get the type of network (e.g. GSM, CDMA, etc)
NetworkSelection extends MobileProvider	Provides methods to set up lists of preferred or forbidden networks and whether the network is selected automatically or manually
MobileNetwork	The MobileNetwork object contains all available information about the wireless network and provides methods to access this information
MobileRadio	Provides simple radio management: principally turning the radio on and off and getting the signal strength
MobileTerminal extends Terminal	Represents a physical endpoint of a connection. Provides methods to identify the terminal, such as the ID String and the manufacturer's name, together with methods to generate DTMF dial tones

The Symbian Provider implements both MobileProvider and MobileRadio interfaces. References to these are obtained by casting from a Provider, e.g.:

```
MobileProvider mobileProvider
    =(MobileProvider)JtapiPeer.getProvider(null);
```

or

```
MobileRadio mobileRadio
    = (MobileRadio)JtapiPeer.getProvider(null);
```

The JTAPI mobile package allows for very fine-grained control of options in the implementation. To find out if a MobileProvider supports a capability you would check for the existence of the capability's interface, and most methods can throw `javax.telephony.MethodNotSupportedException`. More use is made of return values, e.g. a get property call on an operation which is not supported will return a null String, or an operation will return a boolean to indicate success or failure.

5.4.7 Generating Dial Tones

We can use our originating Terminal obtained from a MobileProvider to generate DTMF dial tones. The general form is:

```
public boolean MobileProvider.generateDTMF(String digits)
```

The `digits` String consists of one or more of the following characters: the numbers zero through nine (0–9), the letters A through D, asterisk (*), or hash (#). Any other characters in the String are ignored.

The method returns true if the tone was successfully generated and throws MethodNotSupportedException if the Provider does not support DTMF dial tones.

The book's example code is available for download from the Developer Network website and it includes a Crystal version of PhoneCall which illustrates DTMF tone generation. However because of the limitations of the AT command set this won't generate tones under the emulator, though does work on a Nokia 9210.

5.4.8 Symbian's JTAPI Implementation

The current JTAPI implementation does not support caller identification.

Although the Symbian implementation provides the javax.telephony.events package it is not supported. The `add<>Observer()` methods will all throw `MethodNotSupportedExceptions`, and thus no Observer events are ever generated.

In more detail, Table 5.3 lists the methods not supported by Symbian's implementation.

Table 5.3 The methods not supported by Symbian's implementation

Class	Unsupported methods
Address	`addCallObserver(CallObserver obs)` `addObserver(AddressObserver addrobs)` `getAddressCapabilities(Terminal terminal)` `getCallObservers()` `getObservers()` `removeCallObserver(CallObserver callobs)` `removeObserver(AddressObserver obs)`
Call	`addObserver(CallObserver callObserver)` `getCallCapabilities(Terminal term, Address addr)` `getObservers()` `removeObserver(CallObserver callObserver)`
Connection	`getConnectionCapabilities(Terminal terminal, Address address)`
Provider	`addObserver(ProviderObserver observer)` `getObservers()` `getProviderCapabilities (Terminal terminal)` `getCallCapabilities(Terminal terminal, Address address)` `getConnectionCapabilities(Terminal terminal, Address address)` `getAddressCapabilities(Terminal terminal)` `getTerminalConnectionCapabilities(Terminal terminal)` `getTerminalCapabilities(Terminal terminal)` `removeObserver(ProviderObserver observer)`

(continued overleaf)

Table 5.3 *(continued)*

Class	Unsupported methods
Terminal	`addObserver(TerminalObserver terrobs)` `getObservers()` `removeObserver(TerminalObserver obs)` `addCallObserver(CallObserver callobs)` `getCallObservers()` `removeCallObserver(CallObserver obs)` `getTerminalCapabilities(Terminal ter, Address add)`
TerminalConnection	`getTerminalConnectionCapabilities(Terminal terminal, Address address)`
MobileProvider	`getHomeNetwork()` `MobileProvider.setNetwork()`
MobileRadio	`getRadioStartState()` `setRadioStartState()` `setRadioState()`
MobileTerminal	`getSoftwareVersion()`

5.4.9 JTAPI under the Emulator

JTAPI emulation under Symbian OS emulator relies on the AT command set to talk to a mobile phone. This allows the basic functionality to be implemented but does limit the scope of the emulation. In particular, the following features are not supported: call waiting, DTMF tone control, signal strength monitoring, obtaining the network time zone.

Disconnection events are not always reported by the Emulator. Using the Vodaphone provider we get the results shown in Table 5.4.

Table 5.4 The result of disconnection events

Disconnection reported?	Who initiates	Who disconnects	Comment
Yes		JTAPI application	
No	JTAPI application	Remote party	If you look at the SH888 at this point it still has a call ongoing. It's necessary to hang up the phone to terminate the call: this action does then deliver a disconnection event to JTAPI
No		Mobile phone	
Yes	Remote party	Remote party	

5.4.10 The Phonecall JTAPI Example Application

PhoneCall is a simple JTAPI example that uses the above ideas to set up a call, disconnect a call, and automatically pick up any incoming call. It is shown below running under Symbian's Quartz Emulator.

The status area reports the name of our Provider, "EpocGsmMobileProvider," and our handset's phone number. The first two events relate to making a call to the destination phone number and the next two events to disconnecting the call.

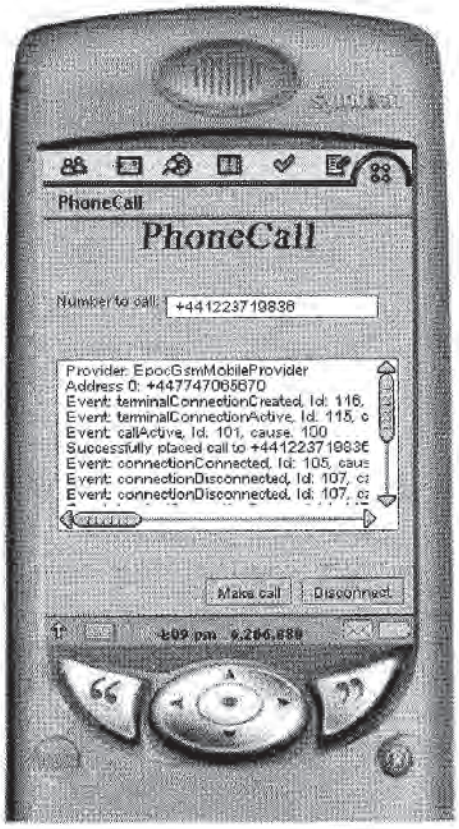


Figure 5.5 A simple JTAPI example running under the generic Symbian OS emulator.

The application's class diagram is shown in Figure 5.6.

The main class is PhoneCall. This extends QFrame which provides basic facilities for Quartz applications. QFrame itself extends Frame and is discussed in detail in Chapter 3. PhoneListener implements the TerminalConnectionListener interface and is used by PhoneCall to listen for telephony events in general and the TerminalConnectionRinging event in particular. PhoneCall creates an anonymous instance of PhoneListener; however, the latter maintains a reference back to PhoneCall so that it can use PhoneCall's `println()` method to report details of events.

Looking first at the PhoneCall class:

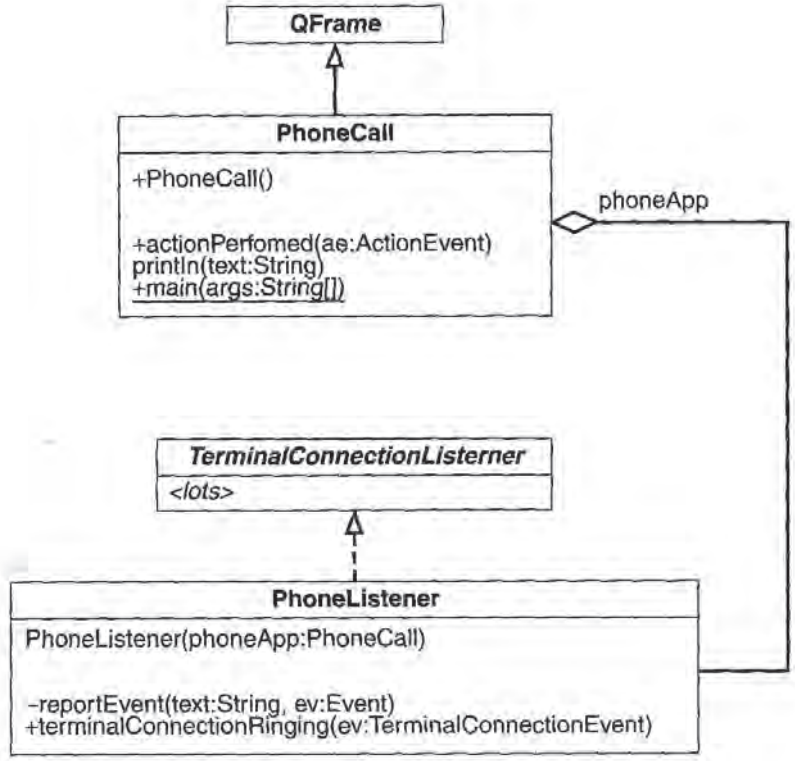


Figure 5.6 Structure of the PhoneCall application.

We create the application in the com.symbian.devnet.quartz.phonecall package and import the required telephony packages:

```
package com.symbian.devnet.quartz.phonecall;
import com.symbian.devnet.quartz.awt.*;
import javax.telephony.*;
import javax.telephony.events.*;
import java.awt.*;
import java.awt.event.*;

public class PhoneCall extends QFrame implements ActionListener
{
```

`provider` is effectively the class that provides our telephony implementation. `origAddress` is the address of my terminal. `origTerminal` is a reference to my terminal from which I will send calls and answer calls. `origConnection` is the connection between a call and my terminal:

```
private Provider provider;
private Address origAddress;
private Terminal origTerminal;
static Connection origConnection;

private Button makeCallB = new Button("Make call");
private Button dropCallB = new Button("Disconnect");
static TextArea receiveT = new TextArea(11, 30);
```

```
private TextField destNumberT = new TextField(20);
private String[] dialogText =
    { "--PhoneCall--",
    "Phone control by JavaPhone",
    "using javax.telephony",
    "Version 0.02, 14 October 2000",
    "Jonathan Allin"};
```

`main()` just creates an instance of our PhoneCall class.

```
public static void main(String[] args)
{
    new PhoneCall();
}
```

`println()` is used to report details of events and any errors:

```
private void println(String text)
{
    receiveT.append(text "\n");
}
```

The first half of the constructor creates the UI:

```
PhoneCall()
{
     super(1);
     createAboutDialog(dialogText);
     GridPanel mainPanel = new GridPanel();

     mainPanel.getConstraints().weighty=1.0;
     mainPanel.add(new Label("Number to call:"), 0, 1);
     mainPanel.add(destNumberT, 1, 1);
     mainPanel.getConstraints().gridwidth=4;

     Label titleL = new Label("PhoneCall");
     titleL.setFont(new Font("Serif", Font.BOLD + Font.ITALIC,20));
     mainPanel.add(titleL, 0, 0);
     mainPanel.add(receiveT, 0, 2);

     Panel buttonPanel = new Panel();

     ((FlowLayout)(buttonPanel.getLayout())).setAlignment
          (FlowLayout.RIGHT);
     buttonPanel.add(makeCallB);
     buttonPanel.add(dropCallB);

     makeCallB.addActionListener(this);
     dropCallB.addActionListener(this);
     destNumberT.addActionListener(this);

     Panel quartzPanel = getCardAt(0);
     quartzPanel.setLayout(new BorderLayout());
     quartzPanel.add(mainPanel, "Center");
     quartzPanel.add(buttonPanel, "South");
```

```
menuBar.remove(editMenu);
show();
```

The rest of the code in our constructor sets up the telephony environment. We first obtain a JTAPI implementation. We can either specify the name of the class, i.e. "com.symbian.javax.telephony.EpocJtapiPeer," or use null. In the latter case the default implementation will be used (as specified by an environment variable or similar).

We then obtain a Provider that our chosen Implementation supports. The Provider is the "window" through which JTAPI applications see the telephony system. If we use null we'll get the default Provider, or we can specify the Provider explicitly, e.g. "EpocGsmMobileProvider" (which in our case is also the default provider).

We need to get the Address and Terminal associated with the originating side of the telephone call. The Address is a logical endpoint, i.e. a "phone number," while the Terminal is a physical endpoint, i.e. a "phone set." We can have more than one Terminal per Address and vice versa.

We ask our Provider for a list of Terminals on it and arbitrarily choose one, then add our ConnectionListener to it (implemented by our PhoneListener class) so we can monitor our Terminal for incoming calls etc:

```
try
{
    JtapiPeer peer = JtapiPeerFactory.getJtapiPeer
        ("com.symbian.javax.telephony.EpocJtapiPeer");
    provider = peer.getProvider("EpocGsmMobileProvider");
    println("Provider: " + provider.getName());
    Address[] addresses = provider.getAddresses();
    for(int i = 0; i < addresses.length; i++)
    {
        println("Address "
            + i + ": "+ addresses[i].getName());
    }
    origAddress = addresses[0];
    Terminal[] terminals = provider.getTerminals();
    origTerminal = terminals[0];
    origTerminal.addCallListener(new PhoneListener(this));
}
```

The above code can throw MethodNotSupportedException, ResourceUnavailableException, and InvalidArgumentException. We'll keep life simple in this example:

```
catch(Exception e)
{
    println("Can't get a provider: "+ e.toString());
}
}
```

Most of the work is carried out by `actionPerformed()`. This is used to drop a call or make a call:

```
public void actionPerformed(ActionEvent ae)
{
```

Make a call. The destination number must be valid:

```
    if((ae.getSource() == destNumberT)
          || (ae.getSource() == makeCallB))
    {
        try
        {
```

Create the telephone call object:

```
                Call call = provider.createCall();
```

Place our telephone call. `Call.connect()` returns an array of connections. In a two-party call c[0] will be the originating terminal and c[1] the destination terminal. We keep a reference to our end of the connection so that we can drop the call:

```
                Connection c[] = call.connect(origTerminal,
                   origAddress,
                   destNumberT.getText());
                origConnection = c[0];
                println("Successfully placed call to "+
                   c[1].getAddress().getName());
        }
        catch(Exception e)
        {
                println("Can't connect: "+ e.toString());
        }
    }
```

Drop the call:

```
    else if(ae.getSource() == dropCallB)
    {
        try
        {
            if(origConnection != null)
            {
                origConnection.disconnect();
                origConnection = null;
            }
        }
```

The above code can throw PrivilegeViolationException, MethodNotSupportedException, InvalidStateException, and ResourceUnavailableException:

```
                catch(Exception e)
                {
                    println("Couldn't drop call: " + e);
                }
            }
        }
    }
```

The PhoneListener Class

The PhoneListener class provides our implementation of a ConnectionListener. Each method listens for a particular event, and in general we just report details of the event. The only method we are really interested in at the moment is `terminalConnectionRinging()`:

```
package com.symbian.devnet.phonecall;
import javax.telephony.*;
import javax.telephony.events.*;

public class PhoneListener implements TerminalConnectionListener
{
    static PhoneCall phoneApp;
```

The constructor provides us with a reference back to JavaCall so that we can use the latter's methods for printing:

```
PhoneListener(PhoneCall phoneApp)
{
    this.phoneApp = phoneApp;
}
```

`reportEvent()` displays information about the Event, including its ID and cause:

```
private void reportEvent(String text, Event ev)
{
    phoneApp.println("Event: "+ text + ", Id: "
     + ev.getID() + ", cause: " + ev.getCause());
}
```

The next eleven methods implement the CallListener interface. The principal events to listen for are CALL_ACTIVE and CALL_INVALID. Also the CALL_EVENT_TRANSMISSION_ENDED event ID is defined, which is sent

when the application can no longer listen for Call events. The other Events are MetaEvents, which are used to associate Events with underlying causes for both simple two-party calls and conference calls:

```
public void callActive(CallEvent ev)
     { reportEvent("callActive", ev); }
public void callInvalid(CallEvent ev)
     { reportEvent("callInvalid", ev); }
public void callEventTransmissionEnded(CallEvent ev)
     { reportEvent("callEventTransmissionEnded", ev); }
public void singleCallMetaProgressStarted(MetaEvent ev)
     { reportEvent("singleCallMetaProgressStarted", ev); }
public void singleCallMetaProgressEnded(MetaEvent ev)
     { reportEvent("singleCallMetaProgressEnded", ev); }
public void singleCallMetaSnapshotStarted(MetaEvent ev)
     { reportEvent("singleCallMetaSnapshotStarted", ev); }
public void singleCallMetaSnapshotEnded(MetaEvent ev)
     { reportEvent("singleCallMetaSnapshotEnded", ev); }
public void multiCallMetaMergeStarted(MetaEvent ev)
     { reportEvent("multiCallMetaMergeStarted", ev); }
public void multiCallMetaMergeEnded(MetaEvent ev)
     { reportEvent("multiCallMetaMergeEnded", ev); }
public void multiCallMetaTransferStarted(MetaEvent ev)
     { reportEvent("multiCallMetaTransferStarted", ev); }
public void multiCallMetaTransferEnded(MetaEvent ev)
     { reportEvent("multiCallMetaTransferEnded", ev); }
```

The next seven methods implement the ConnectionListener interface, allowing us to listen for state changes in a Connection. The corresponding events are: CONNECTION_IN_PROGRESS, CONNECTION_ALERTING, CONNECTION_CONNECTED, CONNECTION_DISCONNECTED, CONNECTION_FAILED, and CONNECTION_UNKNOWN. CONNECTION_CREATED is sent when a new Connection is created:

```
public void connectionAlerting(ConnectionEvent ev)
     { reportEvent("connectionAlerting", ev); }
public void connectionConnected(ConnectionEvent ev)
     { reportEvent("connectionConnected", ev); }
public void connectionCreated(ConnectionEvent ev)
     { reportEvent("connectionCreated", ev); }
public void connectionDisconnected(ConnectionEvent ev)
     { reportEvent("connectionDisconnected", ev); }
public void connectionFailed(ConnectionEvent ev)
     { reportEvent("connectionFailed", ev); }
public void connectionInProgress(ConnectionEvent ev)
     { reportEvent("connectionInProgress", ev); }
public void connectionUnknown(ConnectionEvent ev)
     { reportEvent("connectionUnknown", ev); }
```

The next six methods implement the TerminalConnectionListener interface, allowing us to listen for state changes in a TerminalConnection. The corresponding events are similar to those for a Connection: TERMINAL_CONNECTION_RINGING, TERMINAL_CONNECTION_ACTIVE, TERMINAL_CONNECTION_PASSIVE, TERMINAL_CONNECTION_DROPPED, and TERMINAL_CONNECTION_UNKNOWN. TERMINAL_CONNECTION_CREATED is sent when a new TerminalConnection is created.

Here are the first five:

```
public void terminalConnectionActive(TerminalConnectionEvent ev)
     { reportEvent("terminalConnectionActive", ev); }
public void terminalConnectionCreated(TerminalConnectionEvent ev)
     { reportEvent("terminalConnectionCreated", ev); }
public void terminalConnectionDropped(TerminalConnectionEvent ev)
     { reportEvent("terminalConnectionDropped", ev); }
public void terminalConnectionPassive(TerminalConnectionEvent ev)
     { reportEvent("terminalConnectionPassive", ev); }
public void terminalConnectionUnknown(TerminalConnectionEvent ev)
     { reportEvent("terminalConnectionUnknown", ev); }
```

And `terminalConnectionRinging()` allows us to answer the phone. We'll "auto answer":

```
public void terminalConnectionRinging(TerminalConnectionEvent ev)
{
     reportEvent("terminalConnectionRinging", ev);
     try
     {
          TerminalConnection tc = ev.getTerminalConnection();
```

Store a reference to the Connection so we can drop it if we want to, then answer it:

```
          PhoneCall.origConnection = tc.getConnection();
          tc.answer();
          phoneApp.println("Successfully answered incoming call");
     }
```

The code can throw ResourceUnavailableException, InvalidStateException, PrivilegeViolationException, MethodNotSupportedException:

```
     catch(Exception ex)
     {
          phoneApp.println("Problem answering incoming call: "
           + ex);
     }
   }
}
```

5.5 The Database Package

The database package (javax.pim.database) defines the objects used to access Address book and Calendar information. It uses a very simple flat, non-relational, model. The Database class contains methods to traverse, add, delete, and update the objects it contains.

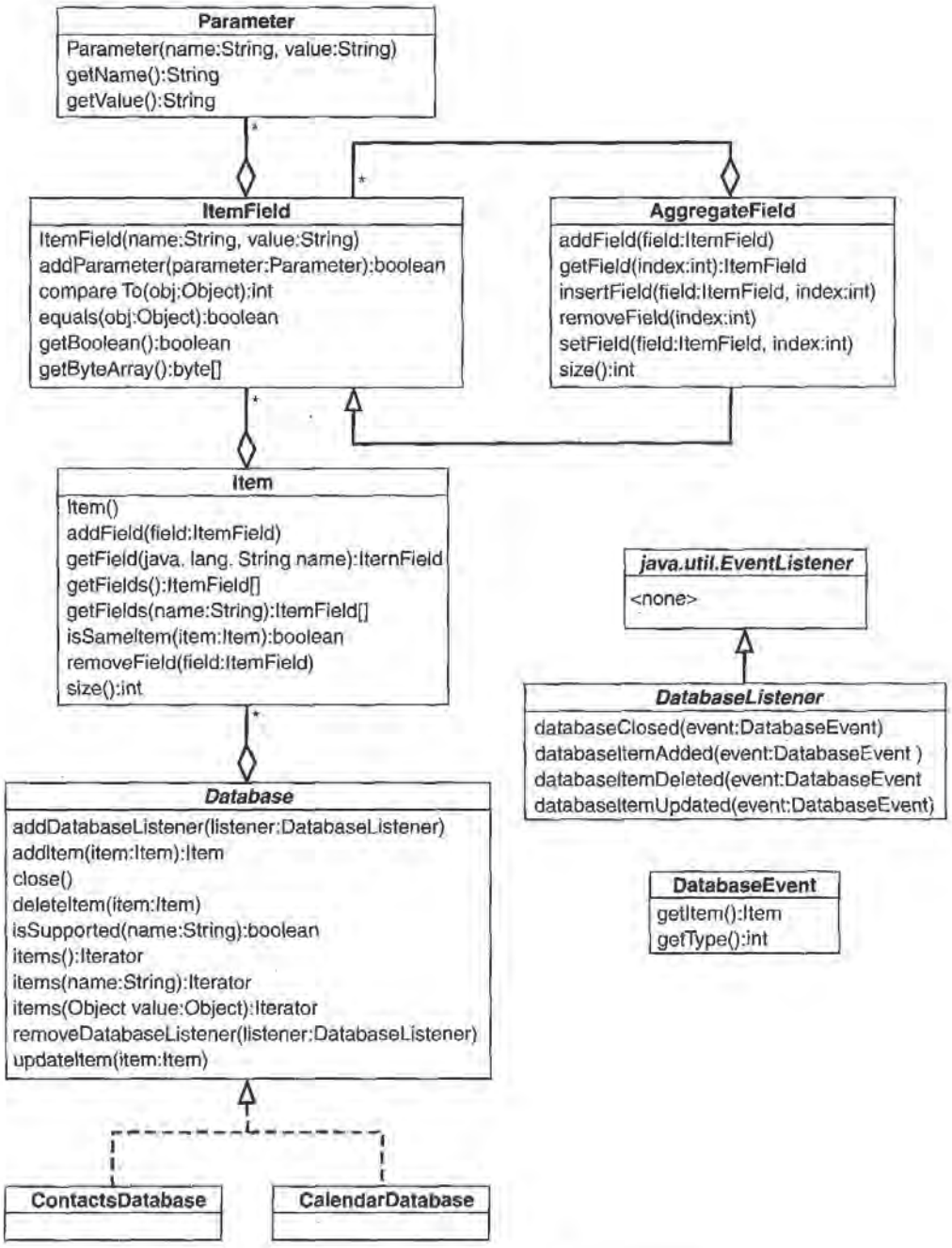


Figure 5.7 Classes of the Database package.

A Database contains multiple Items, where an Item is an entity such as a contact card that contains a person's details (name, address, etc.), or a calendar appointment that will include date information, repeat information, as well as details of the appointment.

The Item's details are encapsulated in ItemField objects: thus an Item contains multiple ItemFields. Each ItemField has a name and a value.

An ItemField can have a set of Parameter objects that contain information about the ItemField's name or value. They are used to specify such information as the location of an alternate text representation for a field value, the language of a text field value, and other attributes. Within a ContactCard Parameters are used to differentiate between the different phone numbers, e.g. voice, home, work.

Each Parameter object holds a name and optional value: the name cannot be null but its value can. Parameters are immutable and cannot be changed.

The AggregateField class extends the ItemField class. An AggregateField holds multiple ItemFields, and thus allows an ItemField to comprise a number of components. For instance the name ItemField used in a ContactCard is an AggregateField made up of five ItemFields. These refer to the contact's family name, given name, other names, honorific prefix, and honorific suffix.

The database package sends events to notify Listeners when changes are made in the database. Subclasses of databases such as AddressBook or Calendar can notify listeners of changes to the database. For example, registered listeners are notified if a date entry is modified.

Here are some guidelines for using the javax.pim.database package. They also generally apply to the addressbook and calendar packages:

- always use the defined strings in the API where possible for field and parameter names and values.
- when adding entries, consider reusing the ItemField and Parameter objects. Each addition is considered to be a totally new entry.
- as far as possible searching is carried out on the native side. Be as specific as possible when iterating through entries. Use the available iterator methods to find entries rather than searching the data on the Java side.
- use `Item.getField()` in preference to any of the `Item.getFields()` methods because it reduces the number of arrays created. The majority of ItemFields should only have a single instance: if there is a need to access the ItemFields that may have multiple instances, e.g. ATTENDEE, then use `Item.getFields()` with no argument rather than calling `Item.getFields(String name)` several times.
- Symbian's Address Book implementation only supports those fields supported by the native Contacts application.
- any repeat definition not supported by the native Agenda application will raise an exception.

5.6 The Address Book API

The Address Book package (javax.pim.addressbook) allows an application to locate and update contact information such as phone numbers, postal addresses, email addresses, and other information about individuals, groups, and organizations. The Address Book package also provides access to name and address information stored on a GSM's SIM card.

The class structure is shown in Figure 5.8.

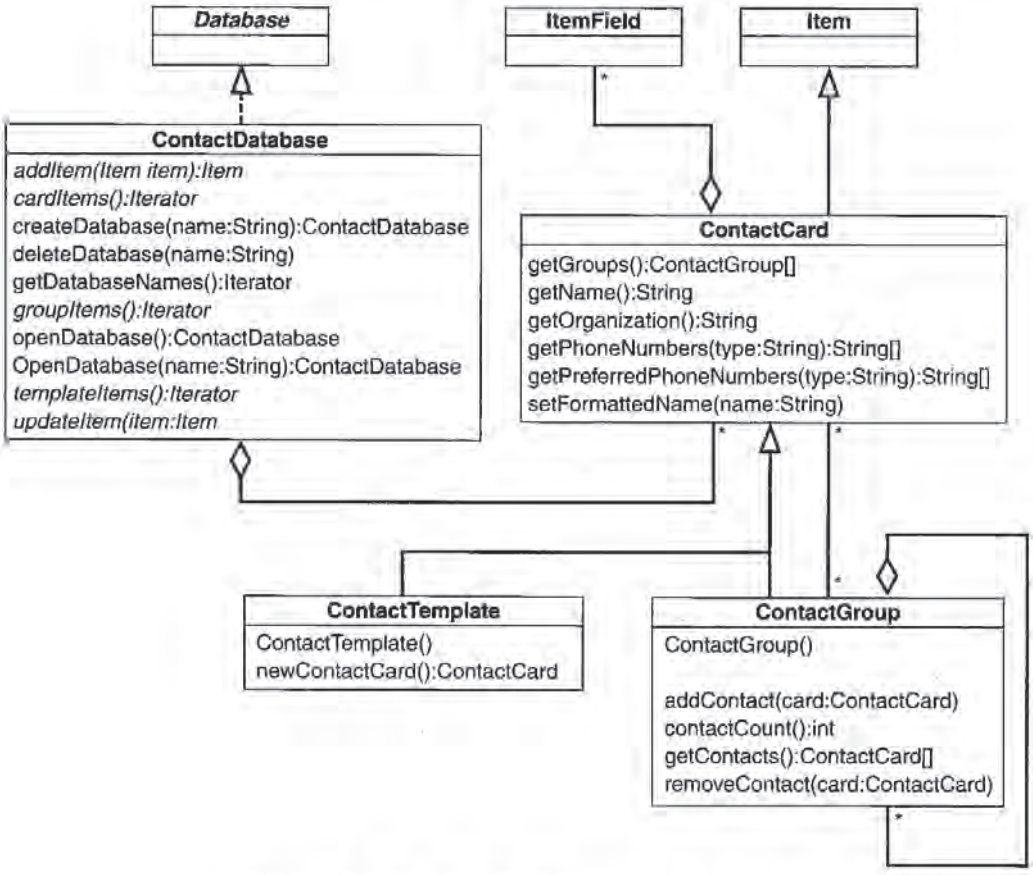


Figure 5.8 Classes of the Address Book package.

5.6.1 The ContactDatabase Class

The ContactDatabase class models a single database of contact information. It contains ContactCards, ContactTemplates, and ContactGroups. These objects in turn contain fields which contain the actual data. ContactDatabase extends Database while ContactCards, ContactTemplates, and ContactGroups, directly or indirectly, extend javax.pim.database.Item.

Field names, field values, parameter names, and parameter values used in the Address book package are conceptually the same as those defined in the vCard specification (see RFC 2426). The vCard property names are

used as naming conventions for field names and parameter strings. So, for example, the ContactDatabase class defines `public static final String N` which is equivalent to the vCard property name "N". The full set of vCard constants is defined as part of the ContactDatabase documentation in ***docs\javax\pim\addressbook\ContactDatabase.html***.

The vCard specifications allow parameter values for a property to be given as a parameter list. For example, in a textual vCard object the value for property parameter 'TYPE' can be defined as a list, e.g. TYPE = WORK, VOICE. This is not the case in the Address book package. If a ContactCard needs a parameter with multiple values then the parameter must be stored as several Parameter objects, i.e. we need two Parameters with the same name "TYPE" but with different values: "WORK" and "VOICE."

`openDatabase(ContactDatabase.DEFAULT_ADDRESSBOOK)` will retrieve the default address book, i.e. it is equivalent to `openDatabase()`.

The ContactDatabase's DatabaseFactory also allows access to the address book on the phone's SIM card. This can be obtained using `openDatabase(ContactDatabase.DEFAULT_SIM)`. ContactCards in this address book will contain ItemFields for `ContactDatabase.SIMSLOT`, `ContactDatabase.FN`, and `ContactDatabase.TEL`.

`getDatabaseNames()` will return an iterator of names suitable for `openDatabase(String name)`.

5.6.2 Contact Templates

The ContactTemplate class provides a template that can be used to create new ContactCards with default fields and possibly with default values. The ContactTemplate constructor is used to create an empty ContactTemplate to which we then add ItemFields as we would for any other ContactCard. We then use `newContactCard()` to provide us with a ContactCard containing these default fields and values.

A ContactCard retains no link to the ContactTemplate used to create it. Future changes to the ContactTemplate will not be reflected in any ContactCards created from it prior to the change.

`ContactDatabase.templateItems()` returns an Iterator of ContactTemplates. An application can then use one or more of these ContactTemplates to decide what the standard fields and values are for a default ContactCard in the given ContactDatabase.

Contact Groups

ContactGroups allow me to group my ContactCards into user-friendly categories. For instance, I can have a work contacts group, a home contacts group, or a chess club contacts group.

There is a many-to-many relationship between ContactGroups and ContactCards: a ContactGroup contains groups of ContactCards but can also contain

other ContactGroups. Further, a ContactCard can belong to many ContactGroups. The Address book implementation must retain the integrity of this many-to-many relationship. For example, when a ContactCard is removed from the database it must be removed from the ContactGroups to which it belongs.

ContactGroup's default constructor is used to create an empty ContactGroup. Other methods in the ContactGroup class allow me to add a ContactCard, count the number of ContactCards in the ContactGroup, delete a ContactCard, or return an array containing all the ContactCards in the ContactGroup.

Typically an application would use `setFormattedName()` to give the ContactGroup a user-friendly name and `getName()` to discover this name. It would then use `ContactDatabase.groupItems()` to get an iterator of all the ContactGroups in the ContactDatabase. `ContactCard.getGroups()` can be used to get an array of all the ContactGroups to which the ContactCard belongs.

5.6.3 The Little Black Book Address Book Example

Figure 5.9 shows a basic address book application that accesses the system's PDA address book or SIM address book. It allows a new contact to be added, a contact or contacts to be found, and a contact to be modified or deleted.

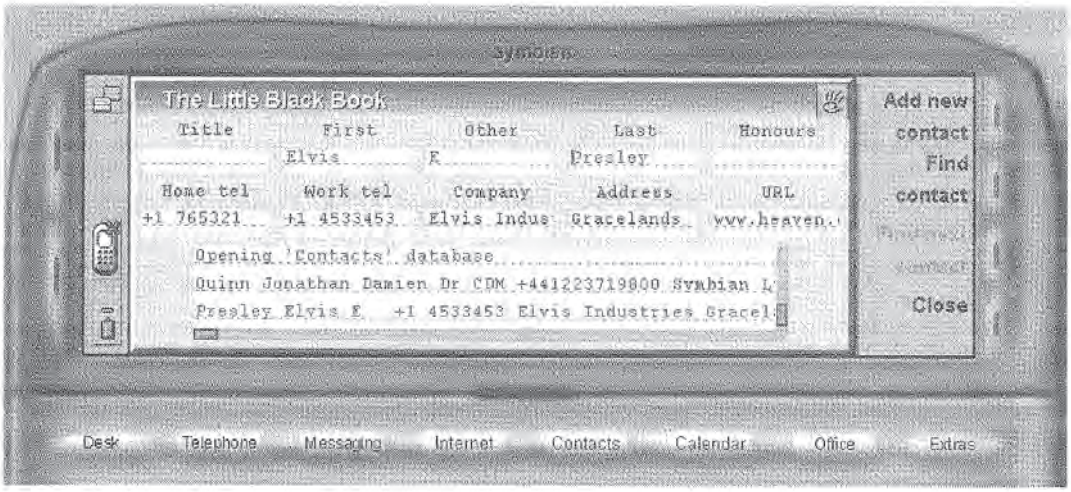


Figure 5.9 The little black book application running under Symbian's crystal emulator.

A contact is found by entering the name or part of the name in the "Last:" field and pressing the "Find" button. Further matching contacts can be found by pressing "Find next". The application will return those cards where the "Last:" field is a substring of the "N" ItemField, an aggregate field composed of the top five fields in the screen shot. If the "Last:" field is empty then all the contacts in the database will be found.

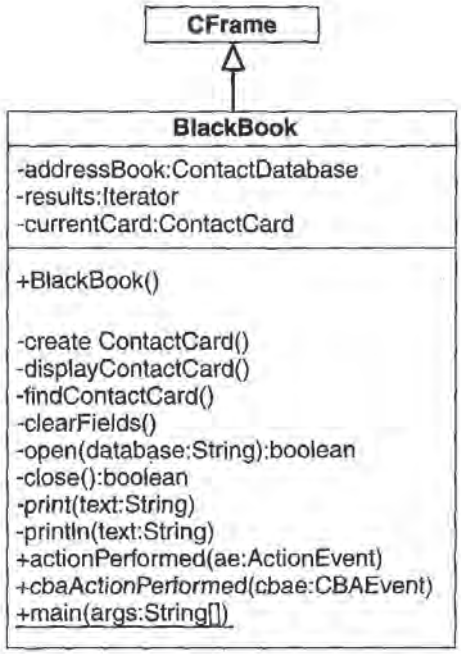


Figure 5.10 The Little Black Book's class diagram.

The application's class diagram is shown in Figure 5.10.

The application is contained in a single class, BlackBook. This extends CFrame which provides basic facilities for Crystal applications. CFrame itself extends Frame and is discussed in detail in Chapter 2.

`createContactCard()` takes the information entered by the user, creates a ContactCard, and adds it to the Contacts database. It sets `currentCard` to point to this ContactCard.

`findContactCard()` finds those ContactCards where the name entered in the Last name TextField, `lastT`, matches the `ContactDatabase.N` ItemField. It assigns results to the Iterator for the ContactCards found by the `findContactCard()` and then calls `displayContactCard()` to display the first ContactCard.

`displayContactCard()` takes the next ContactCard referenced by `results` Iterator, parses it into its components, and displays the results. It sets `currentCard` to point to the ContactCard being displayed.

`open(String database)` opens the specified contacts database and sets the `addressBook` variable to point to it. If database is null the default system contacts database is opened.

`close()` closes the database and sets `addressBook` to null.

`actionPerformed()` handles menu requests from the user (add contact, find and find next contact, update and delete contact). It makes use of `createContactCard()`, `findContactCard()`, and `displayContactCard()`.

`cbaActionPerformed()` handles the Crystal CBA button requests, i.e. add contact, find contact, find next contact, and close.

The next few sections describe each method in detail.

Setting up and the Constructor

The application-specific classes are in BlackBook.java. The application is created in the com.symbian.devnet.crystal.blackbook package and we import the javax.pim.database and javax.pim.addressbook packages:

```
package com.symbian.devnet.crystal.blackbook;
import javax.pim.addressbook.*;
import javax.pim.database.*;
:
:
```

Most of the class and instance variables are user interface components. We've listed the ten input fields which provide in order the contact's title (honorific prefix), first or given name, other names, last or family name, qualifications (honorific suffix), then the company's name, address, and URL.

The Address book specific variables are: `addressbook`, which is a reference to the system's contacts database; `results`, which is an iterator of ContactCards used to iterate through the results of a search; and `currentContact`, which is the currently active ContactCard:

```
public class BlackBook extends CFrame implements CBAListener
{
:
:
    private TextField titleT = new TextField(14);
    private TextField firstT = new TextField(14);
    private TextField otherT = new TextField(14);
    private TextField lastT = new TextField(14);
    private TextField qualificationsT = new TextField(14);
    private TextField homePhoneT = new TextField(14);
    private TextField workPhoneT = new TextField(14);
    private TextField companyT = new TextField(14);
    private TextField addressT = new TextField(14);
    private TextField urlT = new TextField(14);

    private ContactDatabase addressbook;
    private Iterator results;
    private ContactCard currentContact;
```

Most of the constructor's code is concerned with creating the user interface. Once this is accomplished we obtain the names of all the available contacts databases, create MenuItems for each of them, and add the MenuItem to the `fileMenu` Menu. We then call `open(null)` to open the default system contacts database:

```
BlackBook()
{
    :
    :
    try
```

```
        {
            Iterator databaseNames
                = ContactDatabase.getDatabase Names();
            while(databaseNames.hasNext())
            {
                MenuItem databaseItem = new
MenuItem((String)databaseNames.next());
                databaseItem.setActionCommand("Database");
                databaseItem.addActionListener(this);
                fileMenu.insert(databaseItem, 0);
            }
            open(null);
        }
        catch(DatabaseException dbe)
        {
            println("Couldn't get database names: "+ dbe);
        }
        show();
    }
```

print() and println()

`print()` displays text in the status area, `println()` calls `print()` but appends an end of line character to the text being displayed:

```
public final void print(String text)
{
    statusT.append(text);
}
```

The main() Method

`main()` just creates an instance of our BlackBook class:

```
public static void main(String[] args)
{
    new BlackBook();
}
```

actionPerformed()

`actionPerformed()` allows me to respond to menu selections, in particular to:

- create an entry
- search for matching entries
- update an entry
- delete an entry
- select a database

```
public void actionPerformed(ActionEvent ae)
{
```

Open a contacts database. Check if a MenuItem with the an ActionCommand equal to "Database" was selected; if so, open the relevant contacts database:

```
if(ae.getSource() instanceof MenuItem)
{
     MenuItem menuItem = (MenuItem)ae.getSource();
     if("Database".equals(menuItem.getActionCommand()))
     {
          close(); // Close the current database if it is open
          println("Opening " + menuItem.getLabel() + " database");
          open(menuItem.getLabel());
          return;
     }
}
if(ae.getSource() == createItem) createContactCard();
```

Get all the ContactCards that match the required name or name fragment (as entered by the user in the `lastT` TextField (i.e. the Last field in the user interface):

```
else if(ae.getSource() == findItem) findContactCard();
```

Having found, and displayed, the first ContactCard, we can now display any others that match the search criteria. The MenuItem, `findNextItem`, is only enabled if more ContactCards are available:

```
else if(ae.getSource() == findNextItem) displayContactCard();
```

Update the current ContactCard. We could use `Database.updateItem (Item)`, but it's easier, and probably quicker, to delete the Item (i.e. ContactCard) and replace it with a new one. The downside is that (a) any fields in the original ContactCard that are not handled by the application will be lost, and (b) the unique ID associated with the ContactCard will have changed:

```
else if(ae.getSource() == updateItem)
{
     try
     {
          if(currentContact != null)
          {
               println("Updating "+ currentContact.getName());
               addressbook.deleteItem(currentContact);
               createContactCard();
          }
     }
     catch(DatabaseException dbe)
     {
```

```
println("Problem updating Contact Card " +
    currentContact.getName());
}
}
```

Delete the current ContactCard (i.e. the one that is being displayed) as referenced by `currentContact`:

```
else if(ae.getSource() == deleteItem)
{
    try
    {
        // Delete the current ContactCard
        if(currentContact != null)
        {
            println("Deleting "+ currentContact.getName());
            addressbook.deleteItem(currentContact);
            results = addressbook.cardItems();
            //  Get an iterator for all the ContactCards in the
            //  database
            displayContactCard();
            //  Display the first ContactCard
        }
    }
    catch(DatabaseException dbe)
    {
        println("Problem deleting Contact Card "
            + currentContact.getName());
    }
}
```

Clear all the input fields (doing so makes it easier for the user to add a completely new entry). `clearFields()` is a utility that sets all the TextFields in the UI to an empty String:

```
else if(ae.getSource() == clearItem) clearFields();
```

Otherwise pass on the event to the super class:

```
    else super.actionPerformed(ae);
}
```

cbaActionPerformed()

`cbaActionPerformed()` handles the Crystal CBA button presses. The CBA buttons provide a subset of the functionality accessed from the menus; in particular, the four buttons are used to create a new contact, find a contact, find the next contact, or close the program:

```
public void cbaActionPerformed(CBAEvent cbae)
{
    switch (cbae.getID())
```

```
        {
            case ADD:
                createContactCard();
            break;

            case FIND:
                findContactCard();
            break;

            case FIND_NEXT:
                displayContactCard();
            break;

            case CLOSE:
                shutDown();
            break;

            default:
        }
}
```

createContactCard()

`createContactCard()` creates a ContactCard from the information entered by the user. It reads the ContactCard details from the various UI text fields and builds them into a ContactCard which it then adds to the Contacts database:

```
private void createContactCard()
{
```

The name AggregateField contains exactly five ItemFields in the order: family name, given name, other names, honorific prefix, honorific suffix:

```
AggregateField name = new AggregateField(ContactDatabase.N);
name.addField(new ItemField(ContactDatabase.N, lastT.getText()));
name.addField(new ItemField(ContactDatabase.N, firstT.getText()));
name.addField(new ItemField(ContactDatabase.N, otherT.getText()));
name.addField(new ItemField(ContactDatabase.N, titleT.getText()));
name.addField(new ItemField(ContactDatabase.N, qualificationsT.
    getText()));
```

Create ItemFields for the work and home phone numbers. The ItemFields are differentiated by their Parameters: the work phone number has a Parameter of name "TYPE" and value "WORK"; the home phone number has a Parameter of name "TYPE" with the value "HOME":

```
ItemField workTel = new ItemField(ContactDatabase.TEL,
    workPhoneT.getText());
workTel.addParameter(new Parameter(ContactDatabase.TYPE,
    ContactDatabase.WORK));
```

```
workTel.addParameter(new Parameter(ContactDatabase.TYPE,
    ContactDatabase.VOICE));

ItemField homeTel = new ItemField(ContactDatabase.TEL,
    homePhoneT.getText());
homeTel.addParameter(new Parameter(ContactDatabase.TYPE,
    ContactDatabase.HOME));
homeTel.addParameter(new Parameter(ContactDatabase.TYPE,
    ContactDatabase.VOICE));
```

Create an AggregateField for the organization information. The number of ItemFields in the ORG field is unspecified. We will create two:

```
AggregateField org = new AggregateField(ContactDatabase.ORG );
org.addField(new ItemField(ContactDatabase.ORG,
    companyT.getText()));
org.addField(new ItemField(ContactDatabase.ORG,
    addressT.getText()));
```

Create an ItemField for the URL:

```
ItemField url = new ItemField(ContactDatabase.URL, urlT.getText());
```

Create a ContactCard, add our ItemFields to it, and add the ContactCard to the database. Setting `currentContact` to our newly created ContactCard means that we can update it if we want to:

```
        try
        {
            ContactCard contact = new ContactCard();
            contact.addField(name);
            contact.addField(workTel);
            contact.addField(homeTel);
            contact.addField(org);
            contact.addField(url);
            addressbook.addItem(contact);
            currentContact = contact;
            println("Added contact ' " + contact.getName() + " ' ");
        }
        catch(DatabaseException dbe)
        {
            println("Unable to create a new contact: "+ dbe);
        }
    }
```

findContactCard()

`findContactCard()` finds all the ContactCards where the name entered in the Last name TextField, `lastT`, matches the ContactDatabase.N ItemField. It uses `displayContactCard()` to display the first ContactCard found:

```
private void findContactCard()
{
try
    {
        results = addressbook.items(ContactDatabase.N,
            lastT.getText());
        displayContactCard();
    }
    catch(DatabaseException dbe)
    {
        println("Problem during search: "+ dbe);
    }
}
```

displayContactCard()

`displayContactCard()` displays the next ContactCard referenced by the `results` Iterator. Parsing the ContactCard is fairly tedious:

```
private void displayContactCard()
{
    if(results == null)
    {
        println("No contact to display");
        return;
    }
```

Get the next ContactCard if one is available:

```
    if(results.hasNext())
    {
        ContactCard contact = (ContactCard)(results.next());
        currentContact = contact;
        println(contact.getName() + "" + contact.
            getOrganization());
```

Get the name field and split it apart. The name field is an AggregateField that always has five ItemFields:

```
        AggregateField name = (AggregateField)contact.
            getField("N");
        lastT.setText(name.getField(ContactDatabase.FAMILY_NAME).
            getString());
        firstT.setText(name.getField(ContactDatabase.GIVEN_
            NAME).getString());
        otherT.setText(name.getField(ContactDatabase.ADDITIONAL_
            NAMES).getString());
        titleT.setText(name.getField(ContactDatabase.PREFIXES).
            getString());
        qualificationsT.setText(name.getField(ContactDatabase.
            SUFFIXES).getString());
```

Get the home phone number and the work phone number. This is a slightly convoluted process:

- we get all this ContactCard's ItemFields that have a "TEL" field name.
- we then go through each ItemField and get a list of parameter values for that ItemField whose name is "TYPE."
- we then look for "HOME" and "WORK" values, setting the appropriate TextField to the String value of the matching phone number:

```
        ItemField[] phoneNumbers = contact.getFields("TEL");
        for(int i=0; i < phoneNumbers.length; i++)
        {
            String[] parameters = phoneNumbers[i].
                getParameterValues("TYPE");
            for(int j = 0; j < parameters.length; j++)
            {
                if("HOME".equals(parameters[j]))
homePhoneT.setText(phoneNumbers[i].getString());
                else if("WORK".equals(parameters[j]))
workPhoneT.setText(phoneNumbers[i].getString());
            }
        }
```

We then get the organization details. This is also an aggregate field:

```
        AggregateField org = (AggregateField)contact.getField
            ("ORG");
        int itemCount = org.size();
        if(itemCount > 0)
companyT.setText(org.getField(0).getString());
        if(itemCount > 1)
addressT.setText(org.getField(1).getString());

        ItemField url = contact.getField("URL");
        urlT.setText(url.getString());
    }
```

We then set the state of the "Find next contact" CBA button and MenuItem appropriately: enable them if the `results` Iterator references more ContactCards, disable them if we are displaying the last ContactCard:

```
    if(results.hasNext())
    {
        findNextItem.setEnabled(true);
        cba.setEnabled(FIND_NEXT, true);
    }
    else
    {
        results = null;
        findNextItem.setEnabled(false);
        cba.setEnabled(FIND_NEXT, false);
    }
}
```

open()

open() attempts to open the specified ContactsDatabase. If databaseName is null open() will attempt to open the system's default database. Having opened a database we call displayContactCard() to display the first ContactCard in the database:

```
private boolean open(String databaseName)
{
    try
    {
        if(databaseName == null) addressbook = ContactDatabase.
            openDatabase();
        else addressbook = ContactDatabase.openDatabase
            (databaseName);
        results = addressbook.cardItems();
        displayContactCard();
    }
    catch(DatabaseException dbe)
    {
```

If we can't open the database we'll try to create it:

```
        try
        {
            addressbook = ContactDatabase.createDatabase
                ("c:/system/data/Contacts.cdb");
        }
        catch(DatabaseException dbe2)
        {
            println("Unable to open or create database: "+ dbe2);
            return false;
        }
    }
    return true;
}
```

close()

Close the contact database and set addressBook to null:

```
private boolean close()
{
    try
    {
        if(addressbook != null)
        {
            addressbook.close();
            addressbook = null;
        }
    }
    catch(DatabaseException dbe)
```

```
        println("Unable to close the Contact database: " + dbe);
        return false;
    }
    return true;
}
```

shutDown()

`shutDown()` overrides `super.shutDown()` and ensures that the database is closed when the application terminates:

```
public void shutDown()
{
    close();
    super.shutDown();
}
```

5.7 The Calendar API

The Calendar package (javax.pim.calendar) provides applications with access to schedule information by defining the necessary objects to access, add, delete, and update calendar and to-do entries in a Calendar database.

CalendarDatabase implements javax.pim.database.Database which means that, for instance, `items()` can be used to return an iterator of all items in the database and `deleteItem(Item)` to remove the specified Item.

A CalendarDatabase contains CalendarEntry Items and CalendarToDo Items, both of which extend javax.pim.database.Item. CalendarEntry objects provide access to descriptions, dates, and repeat information. CalendarToDo objects provide access to notes and items that represent tasks that need to be accomplished by a certain date. CalendarEntry and CalendarToDo Items can be retrieved using `entryItems()` and `toDoItems()` respectively.

CalendarToDo is a relatively straightforward class. It provides methods to set and get the description, due date, and priority of an Item. These access the DESCRIPTION, DUE, and PRIORITY ItemFields respectively. The default constructor is used to create an empty CalendarToDo Item.

CalendarEntry provides methods for setting and getting the Item's description, start and end date, and repeat information. It provides a utility method for returning a list of attendees as an array of ItemFields: using ItemFields rather than attendee String names means that Parameters associated with the attendee are also available. Finally `getDates(Date beginDate, Date endDate)` returns an iterator of all those Dates on which the CalendarEntry falls between the specified end and start Dates. For a non-repeating CalendarEntry there will either be one Date or no Dates; for a repeating CalendarEntry there could be an unlimited number of Dates.

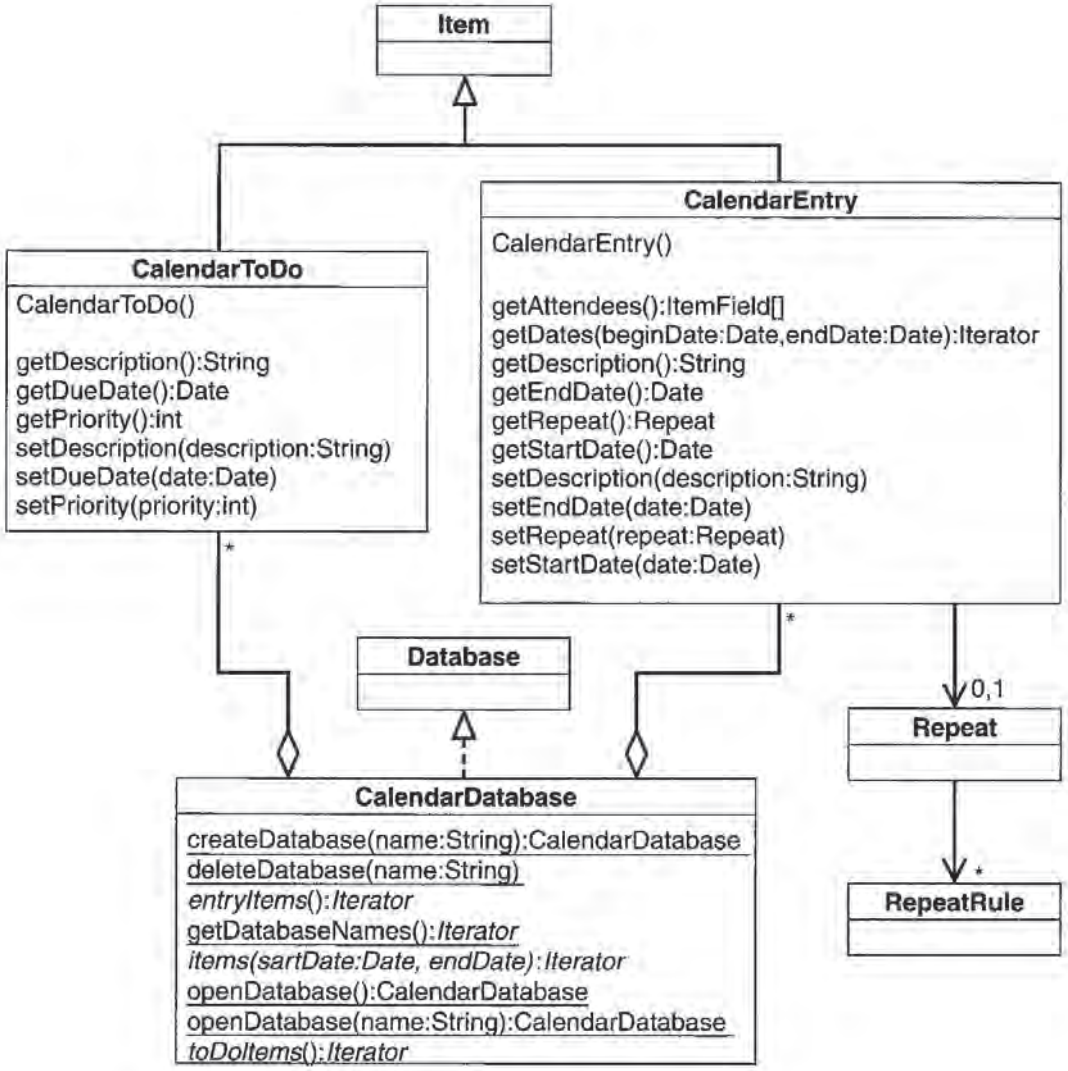


Figure 5.11 Classes of the Calendar package.

The Repeat class defines how a CalendarEntry repeats. It uses the RepeatRule class to define a sequence of Dates on which the CalendarEntry should fall. A RepeatRule can also be used to define exceptions, i.e. Dates when a CalendarEntry shouldn't fall. Individual Dates can also be added to a Repeat object to define when the CalendarEntry will and won't fall.

5.7.1 Symbian's Calendar Implementation

The Symbian implementation does not support all the repeats that can be described by a RepeatRule. In particular, the underlying Agenda model imposes a number of limitations:

- repeats, and exceptions to repeats, can be daily or coarser, but not finer than daily, i.e. hourly exceptions are not supported

- the model supports one repeat rule per entry and explicit exception dates. It does not support explicit repeat dates
- Calendar exception rules are therefore converted to explicit exception dates. This means in theory that an exception rule that repeats forever would generate an infinite list of exceptions. This is limited by the Agenda model which stops at the year 2100. Thus the largest number of exceptions will be, at one per day, about 100 × 365 or 36,500 entries. Still big enough to blow away a PDA or WID!

5.7.2 The QCalendar Example

QCalendar uses the Calendar API to view calendar entries in the PDA's calendar database. The application can be used to review and modify some of an entry's repeat details, including exception dates.

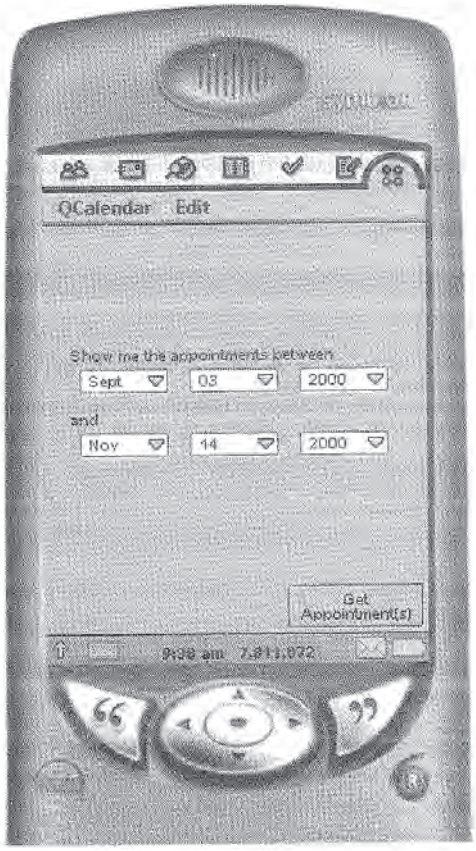


Figure 5.12 The QCalendar application running under Symbian's Quartz emulator: base view.

Tapping on Get Appointments brings up a list of appointments which fall between the selected dates (Figure 5.13).

Tapping on a single appointment brings up the details for that appointment (Figure 5.14).



Figure 5.13 QCalendar's list view.

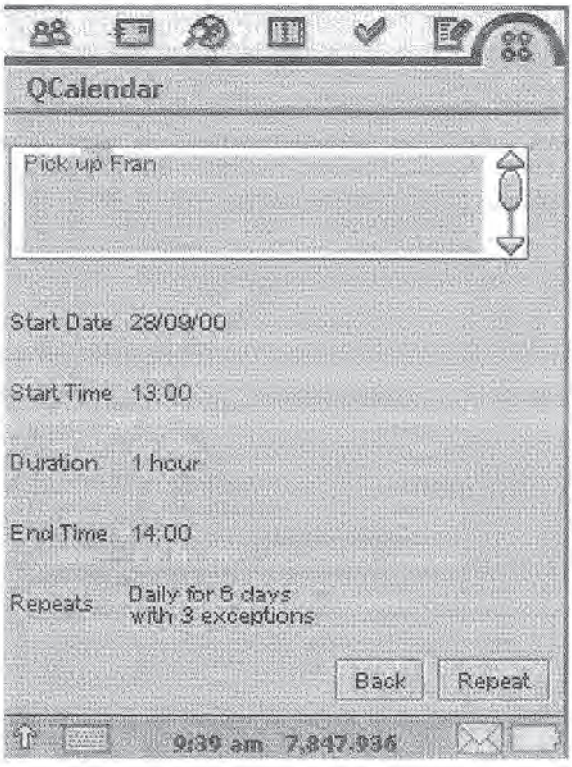


Figure 5.14 QCalendar's details view.

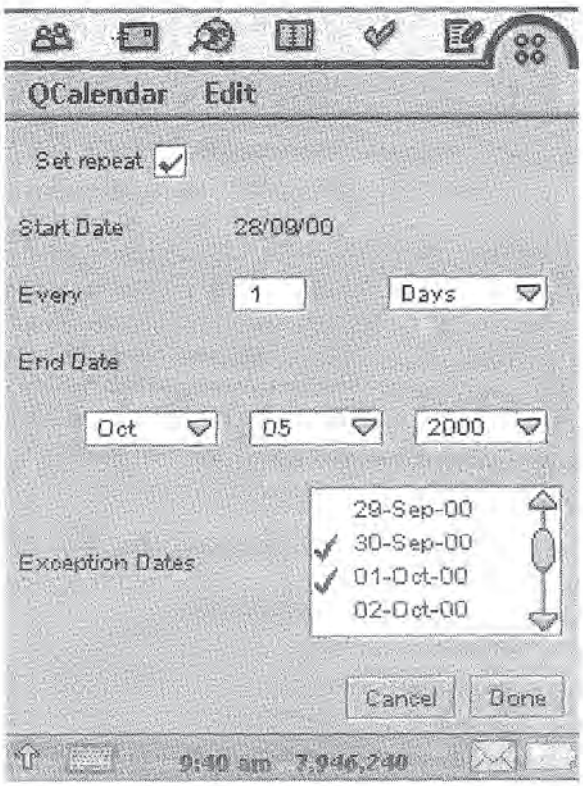


Figure 5.15 QCalendar's repeat view.

Tapping on Repeat brings up the appointment's repeat information (Figure 5.15).

The appointment's repeat dates are listed: tapping on a repeat date will make it an exception, as indicated by the tick. Tapping on the Done button will save any changes to the appointment's repeat details.

Of course, the Agenda application can be used to create appointments which can then be reviewed and modified by QCalendar.

5.7.3 QCalendar's Architecture

QCalendar is an extensive program, so rather than cover the whole application in detail we'll concentrate on the areas relevant to the Calendar package.

Table 5.5 summarizes the responsibilities of QCalendar's classes.

QCalendar contains methods to display the various views. Initially the Base view is displayed. `shutDown()` overrides `QFrame.shutDown()`: its main purpose is to call `CalendarModel.close()` which closes the Calendar database.

The Base view contains two DateControls: these are used to enter a start and end date. Tapping on Get Appointments invokes a call to `displayAppointment()`. This reads the dates from the DateControls, checks their validity, and

Table 5.5 The responsibilities of QCalendar's classes

Class	Responsibilities
QCalendar	Initialization, controlling the application, launching the required view
BaseView	Creates a UI to select a range of dates
ListView	Displays the selected entries in a list
DetailView	Displays the details of a single entry
RepeatView	Displays the repeat details of a single entry and allows these to be modified
DateControl	UI component used to enter a date
CalendarModel	Provides methods to access the Calendar database
CEntryDetail	A wrapper for a javax.pim.calendar.CalendarEntry object. It provides easier and more intuitive access to the underlying functionality of its associated CalendarEntry instance

calls `QCalendar.showListView(Date dtStart, Date dtEnd)` with these dates.

`QCalendar.showListView(Date dtStart, Date dtEnd)` retrieves appointment data from the CalendarModel using `CalendarModel.findEntry(Date startDate, Date endDate)`. This method returns an array of CEntryDetails. `showListView()` assigns the CEntryDetail array to the `details` instance variable (for use by other views) and then calls `List.display(CEntryDetail[])` to display the list of appointments.

`List.display()` uses the CEntryDetail methods `getEntryStartDate()` and `getDescription()` to obtain summary information about each appointment.

Tapping on one of the appointments in the List view invokes `QCalendar.showDetailView(int index)`, where index is the position of the appointment in the `CEntrydetail` array (as referenced by QCalendar.details). `showDetailView()` extracts the relevant CEntryDetail from the array and calls `DetailView.display(CEntryDetail)`, which displays the appointment's details.

`DetailView.display(CEntryDetail)` uses `getEntryStartDate()`, `getEntryEndDate()`, and `getHourDuration()` from CEntryDetail to display the appointment's text, dates, and duration in hours.

Other methods from CEntryDetail are used to examine the appointment's repeat details. `hasRepeat()` is used to check if there is any repeat information. If there is then `getRepeatType()`, `getRepeatDuration()`, and `getExceptionCount()` are used to determine the repeat details (e.g. daily, monthly, yearly; number of days, months, years; number of exceptions).

Tapping on the Repeat button brings up the Repeat view and the appointment's repeat details by invoking `QCalendar.showRepeatView(CEntryDetail)`. This in turn calls `RepeatView.display(CEntryDetail)`.

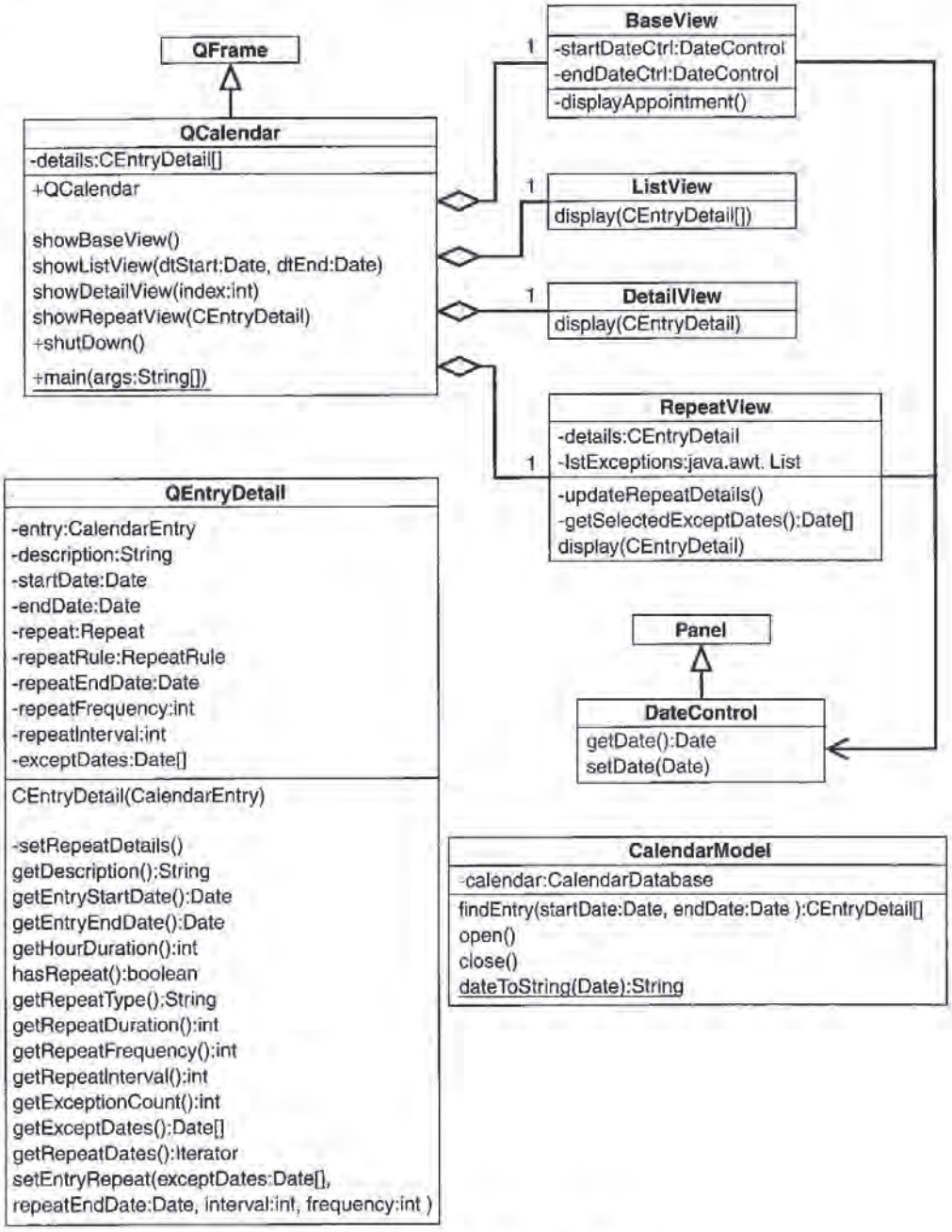


Figure 5.16 QCalendar class diagram.

5.7.4 Displaying the CalendarEntry's Repeat Details

Here is the code for `RepeatView.display(CEntryDetail details)`. It extracts the repeat entry information from the CEntryDetail, including exception dates, and displays the results in a List component. If a repeat date is an exception it is ticked.

`CEntryDetail.getEntryStartDate()` extracts the associated CalendarEntry's start Date, which is then converted to a String of the form dd/mm/yy

using the static method `CalendarModel.dateToString(Date)`. The start Date is displayed in the label `lbldtStart`. `CEntryDetail.hasRepeat()` checks if the associated CalendarEntry has a repeat; if so, it displays the repeat interval (i.e. repeat every repeat interval days/weeks/months, etc) using `CEntryDetail.getRepeatInterval()`, the repeat frequency (i.e. daily, weekly, monthly, etc) using `CEntryDetail.getRepeatFrequency()`, and the end Date using `CEntryDetail.getRepeatEndDate()`. Then we clear `lstExceptions`, the List component that displays the repeat dates:

```
void display(CEntryDetail details)
{
     this.details = details;
     String dtStart = CalendarModel.dateToString
         (details.getEntryStartDate());
     lbldtStart.setText(dtStart.substring(0, 8));
:
:
     boolean repeat = details.hasRepeat();
     txtInterval.setText((repeat) ?
String.valueOf(details.getRepeatInterval()) : "1");
     choiceFreq.select((repeat)? details.getRepeatFrequency()-1:1);
     endDateCtrl.setDate((repeat)? details.getRepeatEndDate():
details.getEntryStartDate());
     lstExceptions.removeAll();
```

If we have repeating entries we use `CEntryDetail.getRepeatDates()` to return an iterator of the repeats, use a DateFormatter `dfMed` to convert this to a String, and add these formatted dates to the component displaying the repeats, `lstExceptions`:

```
     if(repeat)
     {
          Iterator it = details.getRepeatDates();
          while(it.hasNext())
          {
               Date d = (Date)it.next();
               lstExceptions.add(dfMed.format(d));
          }
```

We get an array of Dates which are exceptions to those defined by the RepeatRule using `CEntryDetail.getExceptDates()`. Then we go through the entries displayed in `lstExceptions` and for each entry check if there is a corresponding entry in the array of exception Dates, `exceptDates`. To do the check we need to use the DateFormatter `dfMed` to obtain a suitable String representation of the exception Date:

```
          Date[] exceptDates = details.getExceptDates();
          if (exceptDates != null)
          {
```

```
                for(int dispIndex = 0; dispIndex <
                    lstExceptions.getItemCount(); dispIndex++)
                {
                    String repeatDateString = lstExceptions.
                        getItem(dispIndex);
                    for(int exceptIndex = 0; exceptIndex <
                        exceptDates.length; exceptIndex++)
                    {
                        String exceptDateString = dfMed.
                            format(exceptDates[exceptIndex]);
```

If the displayed repeat date matches an exception date we select it, i.e. tick, the entry in `lstExceptions`:

```
                        if(exceptDateString.equals(repeatDateString))
                            lstExceptions.select(dispIndex);
                    }
                }
            }
        }
```

5.7.5 Modifying the CalendarEntry's Repeat Details

We can change the exception dates by tapping on the corresponding entry in the displayed dates. Tapping will mark a repeat date as an exception, and an exception date as a repeat. The following code is used to obtain those entries we've selected as an array of Dates. This time we are using the DateFormatter `dfMed` to parse the String and return a Date:

```
private Date[] getSelectedExceptDates()
{
:
:
    String[] str = lstExceptions.getSelectedItems();
    int len = str.length;
    Date[] dates = new Date[len];
    try
    {
        for (int i = 0; i <len; i++)
        {
            dates[i]=dfMed.parse(str[i]);
        }
        return dates;
    }
    catch(ParseException pe)
    {
        return null;
    }
}
```

Finally we need to update our CEntryDetail instance, `details`, with the repeat information including the list of exception dates. This uses `CEntryDetail.setEntryRepeat(Date[] exceptDates, Date repeatEndDate, int interval, int frequency)`:

```
private boolean updateRepeatDetails()
{
:
:
    details.setEntryRepeat(getSelectedExceptDates(),
        endDateCtrl.getDate(), interval, frequency);
    return true;
}
```

5.7.6 The CEntryDetail Class

The CEntryDetailClass provides a wrapper around a CalendarEntry, and together with the CalendarModel class (discussed below) provides the interface between the QCalendar application and the Calendar database.

Methods and Variables

Tables 5.6 and 5.7 summarize the principal CEntryDetail instance variables and methods and how they interact with the JavaPhone Calendar package.

Table 5.6 The principal CEntryDetail instance variables

Instance variables (all private)	Description
CalendarEntry entry	The CalendarEntry associated with this CEntryDetail instance
String description	The CalendarEntry's descriptive text
Date startDate	The CalendarEntry's start Date
Date endDate	The CalendarEntry's end Date
Repeat repeat	Repeat object associated with the CalendarEntry
RepeatRule repeatRule	The RepeatRule associated with the CalendarEntry's Repeat Object
Date repeatEndDate	The end date for the RepeatRule
int repeatFrequency	The RepeatRule's frequency (daily, weekly, monthly, etc)
int repeatInterval	The RepeatRule's interval (every third day, every second week, etc)
Date[] exceptDates	Exception dates to the Repeat

Table 5.7 The principal CentryDetail methods

Package methods	Description
CEntryDetail(CalendarEntry)	Constructor. Parses a CalendarEntry object. Uses `CalendarEntry.getDescription()` to get the CalendarEntry's String description, and `CalendarEntry.getStartDate()` and `getEndDate()` to get its start and end Dates. Uses the private method `setRepeatDetails()` to parse the repeat information (described below)
getDescription():String	Returns the CalendarEntry's description, as referenced by `description`
getEntryStartDate():Date	Returns the CalendarEntry's start Date, as referenced by `start`
getEntryEndDate():Date	Returns the CalendarEntry's end Date, as referenced by `end`
getHourDuration():int	Calculates the duration of the CalendarEntry in hours by taking the difference between `startDate` and `endDate`, dividing by the number of ms in an hour, and returning the answer
hasRepeat():boolean	Returns true if this CalendarEntry is repeating
getRepeatType():String	Returns the type of repeat as a String. Looks up the description of the type in a String Array using the RepeatRule's frequency as an index
getRepeatDuration():int	Returns the duration of the Repeat associated with the CalendarEntry instance. Calculates the duration in ms by `startDate` from `repeatEndDate`, then uses the `repeatFrequency` to return the duration in the appropriate units (days, weeks, etc)
getRepeatFrequency():int	Returns the RepeatRule's frequency, as referenced by `repeatFrequency`
getRepeatInterval():int	Returns the RepeatRule's interval, as referenced by `repeatInterval`
getExceptionCount():int	Returns the number of exception dates. Uses the length of the `exceptDate` Array
getExceptDates():Date[]	Returns a copy of the `exceptDates` array
getRepeatDates():Iterator	Returns the iterator returned by RepeatRule.dates()
setRepeatDetails()	See description below
setEntryRepeat(exceptDates:Date[], repeatEndDate:Date, interval:int, frequency:int)	See description below

setRepeatDetails()

`setRepeatDetails()` is a private utility method. It uses the following methods from the Calendar package to initialize the instance variables associated with the Repeat instance:

- `CalendarEntry.getRepeat()` to return the Repeat object associated with the CalendarEntry and `Repeat.getRepeatRules()` to return an array of RepeatRules. Only the first RepeatRule is used.
- `RepeatRule.getFrequency()` to obtain an int representing the frequency (daily, weekly, monthly), `RepeatRule.getInterval()` to obtain the interval (every third day, every second week, etc), and `RepeatRule.getEndDate()` to get the end date.
- `Repeat.getExceptDates()` to return an Array of exception Dates.

setEntryRepeat()

`setEntryRepeat()` is used by the RepeatView to modify the repeat details for this CalendarEntry.

`setEntryRepeat()` first creates a new RepeatRule and uses the arguments to set its parameters. It then creates a Repeat and adds the RepeatRule to it. If there are exception dates these are added to the Repeat using `Repeat.addExceptDate(Date)`. The Repeat is added to the CalendarEntry using `CalendarEntry.setRepeat(Repeat)`. Finally `setRepeatDetails()` (described in the table above) is used to update the CEntryDetail's instance variables relevant to the repeat:

```
void setEntryRepeat(Date[] exceptDates, Date repeatEndDate, int
interval, int frequency)
{
     repeatRule = new RepeatRule();
     repeatRule.setInterval(interval);
     repeatRule.setFrequency(frequency);
     repeatRule.setEndDate(repeatEndDate);
     repeat = new Repeat();
     repeat.addRepeatRule(repeatRule);

     if (exceptDates != null)
     {
          int len = exceptDates.length;
          for(int i = 0; i < len; i++)
          {
               repeat.addExceptDate(exceptDates[i]);
          }
     }
     entry.setRepeat(repeat);
     setRepeatDetails();
}
```

5.7.7 The CalendarModel Class

CalendarModel provides methods to open, close, and access the Calendar database, and a utility method to convert a Date to a String.

open()

Uses the static method `CalendarDatabase.openDatabase()` to return a reference to the default Calendar database. Assigns the reference to `calendar`.

close()

Checks that calendar is not null then calls calendar.close(). `CalendarModel.close()` is called from `QCalendar.shutDown()`.

dateToString

`dateToString(Date)` is a static method. It takes a Date as an argument and returns a String representation. It uses a DateFormat object to return a short representation of the Date, e.g. "23/04/00."

findEntry(startDate:Date, endDate:Date):CEntryDetail[]

`findEntry()` returns a CEntryDetail Array containing all the entries in the CalendarDatabase that fall between the specified dates.

The method uses `CalendarDatabase.items(startDate, endDate)` to return an Iterator of CalendarItems. The Iterator is used to add the retrieved CalendarItems to a Vector, `detailsV`:

```
CEntryDetail[] findEntry(Date startDate, Date endDate)
{
     Vector detailsV = new Vector(10);
     try
     {
          results = calendar.items(startDate, endDate);
          while(results.hasNext())
          {
               detailsV.addElement(results.next());
          }
```

Create a CEntryDetail Array of the right size, go through the Vector using the `CEntryDetail(CalendarItem)` constructor to convert the CalendarItem to a CEntryDetail, and insert the CEntryDetail into the Array:

```
          int numberEntries = detailsV.size();
          CEntryDetail[] details = new CEntryDetail[numberEntries];
          for (int i=0; i < numberEntries; i++)
               details[i] = new
CEntryDetail((CalendarEntry)detailsV.elementAt(i));
          return details;
     }
     catch (DatabaseException ex)
     {
          return null;
     }
}
```

It's worth noting that the implementation of this method is not ideal:

- we are not limiting the size of the Vector used as a temporary store for the entries. It can thus grow without limit.
- we are copying the temporary Vector into an Array which will double our storage requirements.

5.8 The User Profile API

The User profile package is used to obtain information about the current owner or user of the telephony device and, subject to security restrictions, to set information about the current user. User information is returned or set in the same format as the AddressBook package, i.e. as a ContactCard. In a GSM phone, for example, the User profile API provides access to the SIM card information for the owner, which might include the user's name and other attributes that identify the user.

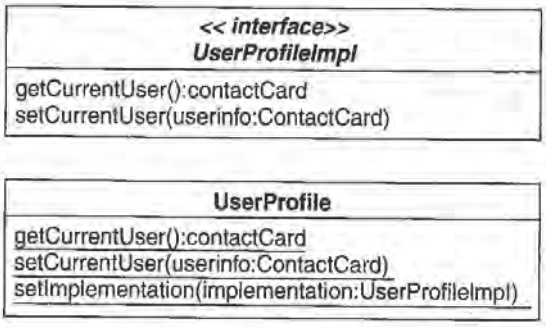


Figure 5.17 Classes of the User profile package.

UserProfile provides `getCurrentUser()` to get the current user's details. The telephony device uses the UserProfileImpl, which is the interface that implements the UserProfile class, to implement the UserProfile methods.

We can use the methods in the ContactCard class to obtain further information about the user, e.g. `ContactCard.getPreferredPhoneNumbers()` will return the user's preferred phone numbers.

`getCurrentUser()` will return null if there is no current user, and `setCurrentUser()` can be called with a null parameter to indicate that there is no current user.

The following example shows how to look up the preferred phone numbers for the current user:

```
import javax.pim.userprofile.*;
import javax.pim.addressbook.*;
import javax.pim.database.*;

public class ProfileSample1
{
```

Get the current user and print their phone numbers:

```
public static void main(String[] args)
{
    try
    {
        ContactCard user = UserProfile.getCurrentUser();
```

Get the array of preferred phone numbers and print them:

```
            String[] tel=user.getPreferredPhoneNumbers
                (ContactDatabase.VOICE);
            for (int i=0; i < tel.length; i++)
            {
                System.out.println("Name: " + user.getName()
                    + ": " + tel[i]);
            }
        }
        catch (DatabaseException ex)
        {
            System.out.println("Error getting user information:
                " + ex.getMessage());
        }
    }
}
```

5.9 The Datagram API

The Datagram API allows applications to send and receive datagrams that are independent of the underlying transport or bearer. More specifically, the API is designed to:

- encapsulate transport-dependent address information in a name service.
- allow applications to run on different telephony devices or networks without being rewritten.
- support non-IP transports such as SMS, GSM data, or Pagers.
- support client and server applications.
- support abstract and concrete addressing. In the former the Address is obtained indirectly through the DatagramNameService, typically from an entry in the Address book. In the latter addressing details are assumed to be known and so the Address can be created directly.

Symbian's implementation supports UDP over IP, SMS over GSM, and WDP over SMS.

Figure 5.18 shows the classes that make up the javax.net.datagram package.

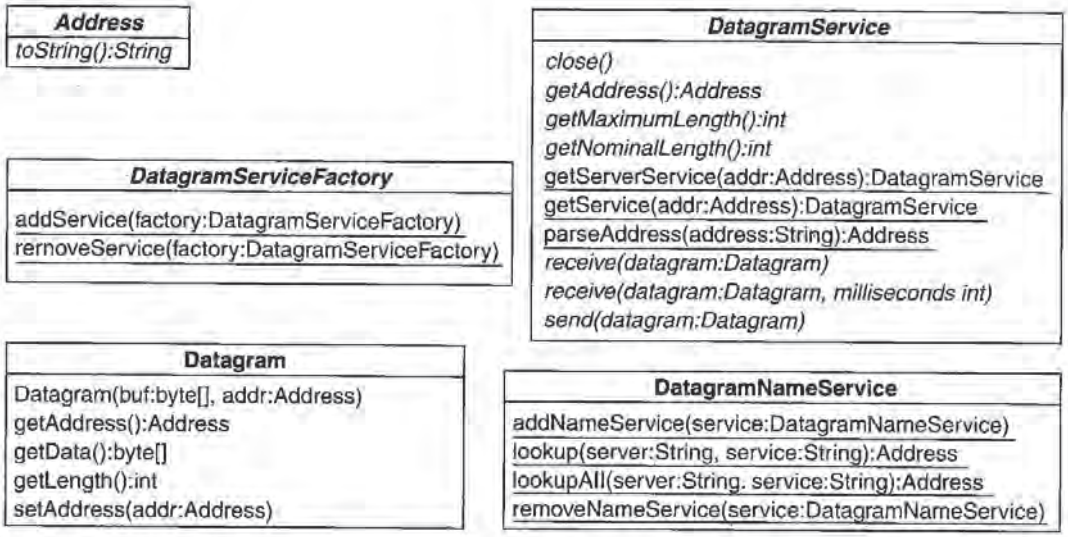


Figure 5.18 Classes of the Datagram package.

5.9.1 Addressing

An Address identifies a Datagram's destination or source together with the transport service used to carry the Datagram. You can create an address in two ways: concrete addressing and abstract addressing.

Concrete addressing requires a URL-like text description that identifies the destination and required transport. We then use `DatagramService.parseAddress(String textDescription)` to obtain an Address. For example:

- `DatagramService.parseAddress("sms://+441234567890")` will return an address for a text SMS datagram for the given phone number. You can use this to send an SMS message to a phone. The data you send will be interpreted as 7-bit text and sent as a vanilla text message. Note: the SMS bearer is send-only. You cannot use it to receive SMS messages.
- `DatagramService.parseAddress("wdpsms://+441234567890:5678")` will return an address for WDP over SMS datagram for the given phone number and port.
- `DatagramService.parseAddress("udp://foo.symbian.com:5678")` will return an address for a UDP datagram for the given host and port.

`Address.toString()` returns a String which has the same format as the text description used by `DatagramService.parseAddress()`.

Abstract addressing starts with the name of a service and of a server which can deliver the required service. `DatagramNameService.lookup(String server, String service)` is then used to return an Address. Our DatagramNameService implementation obtains the Address by querying our device's address book: the server name is used to identify the ContactCard for

the server and selected ItemFields are used to select the service and hence the transport and destination information.

The following code creates a suitable ContactCard. The first address supplied is marked as PREF to indicate it is the preferred address. You must prepend "X-DATAGRAM-" to the ItemField name to make it available to the lookup service:

```
String[] addresses
     = {"udp://chessmasters.org.uk:666",
          "wdpsms://+441234567890:666"};
static Parameter Kpref
     = new Parameter(ContactDatabase.TYPE, ContactDatabase.PREF);
ContactCard card = new ContactCard();
card.setFormattedName("Chess Club");

for (int i = 0; i < addresses.length; ++ i)
{
     ItemField field = new ItemField("X-DATAGRAM-" + "Fixtures",
         addresses[i]);
     if(i == 0) field.addParameter(KPref);
     card.addField(field);
}
contactDatabase.updateItem(card);
```

Later we can look up the concrete Addresses using the contact's full name and the service name. Assuming that "Chess club" is the formatted name (i.e. the FN field's value) of the ContactCard we can then use:

```
Address address = DatagramNameService.lookup("Chess club",
"Fixtures");
```

to get the preferred Address ("udp://chessmasters.org.uk:666" in our example).

The first parameter passed to `lookup()` is the formatted name (FN) of the relevant contact card. The second parameter is the name of the ItemField containing the address with the prefix "X-DATAGRAM-" removed.

To get all the addresses use:

```
Address[] addresses
    = DatagramNameService.lookupAll("Chess club", "Fixtures");
```

In Symbian's implementation, if no formatted name exists the formatted name will be constructed from the ContactCard's name components (as identified by its N field's value).

The Datagram API purposely omits classes implementing Address. This is because, in general, applications do not need to be aware of the addressing details. Receiving Datagrams should be as bearer agnostic as possible. For

instance, if the AddressBook contains multiple bearer/transport/protocol options for a particular server and service then the implementation can select one that is working.

To find what protocols are supported by a particular service (as listed in the address book) you can first use `DatagramNameService.lookupAll()` to return an Array of Addresses, `toString()` for each Address, and then compare the first characters up to `"://"` with the known `"sms://"`, `"wdpsms://"`, `"udp://"` Strings which are defined in the specification.

5.9.2 Sending a Datagram

To send or receive a datagram we first need to obtain a DatagramService, achieved using `DatagramService.getService(Address)`. Then, given an Address, we can send a Datagram:

```
DatagramService client = DatagramService.getService(address);
client.send(new Datagram("When do we play the Old
Masters?".getBytes(), address));
```

Or to create a Datagram server:

```
DatagramService server = DatagramService.getServerService(address);
```

To send a UDP datagram to yourself use `"udp://127.0.0.1:<portnumber>"`. If your machine is configured appropriately (e.g. if the local host's file contains a localhost mapping) you can also use `"udp://localhost:<portnumber>"`.

It's helpful to know that if you use `Datagram.getService()` a port number is automatically assigned. So another way of sending a UDP datagram to yourself is to open a DatagramService using a prototype Address of the correct class, then calling `getAddress()` on this DatagramService instance, e.g.:

```
Address localAddress =
DatagramService.parseAddress("udp://127.0.0.1");
DatagramService localService =
DatagramService.getService(localAddress);
localAddress = localService.getAddress();
localService.send(new Datagram("Hello to me".getBytes(),
localAddress));
```

You can choose your own fixed port on which to listen by using `DatagramService.getServerService()`. So to receive UDP datagrams on port 7654:

```
Address receiveAddress =
DatagramService.parseAddress("udp://127.0.0.1:7654");
DatagramService service =
```

```
DatagramService.getServerService(receiveAddress);
byte[] buffer = new byte[service.getMaximumLength()];
Datagram datagram = new Datagram(buffer);
while(true)
{
     datagram.setLength(buffer.length);
     service.receive(datagram);
```

Send a reply:

```
     String msg = "Received datagram";
     Datagram reply = new Datagram(msg.getBytes(),
         datagram.getAddress());
     service.send(reply);
}
```

5.9.3 The Telegraph Datagram Example

Telegraph uses the Datagram APIs to send an SMS text message and to send and receive WDP and UDP datagrams.

From the status area you can see that we first sent a UPD Datagram to ourselves. If a second copy of the example was running on the network we could have sent a message to it instead. We then sent a WDP message to +44 7747 065 691. Because this is also "our" phone number the message was first handled by the network operator, returned to our phone, and then picked up by our Telegraph application.

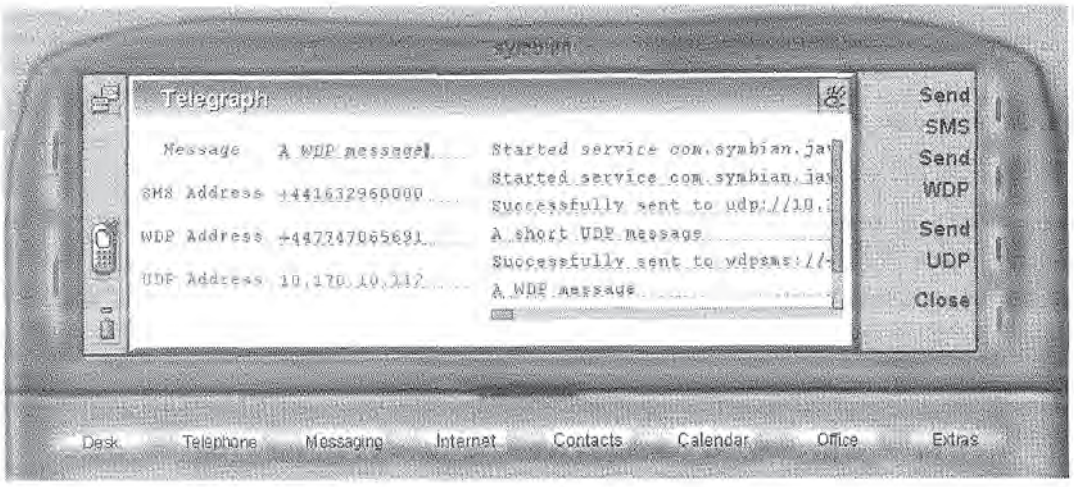


Figure 5.19 The Telegraph application running on the Crystal Emulator.

The application's class diagram is shown in Figure 5.20.

Telegraph is the main class.

DatagramListener provides a DatagramServerService that listens on the requested address. The address implies a service, which in our case is either

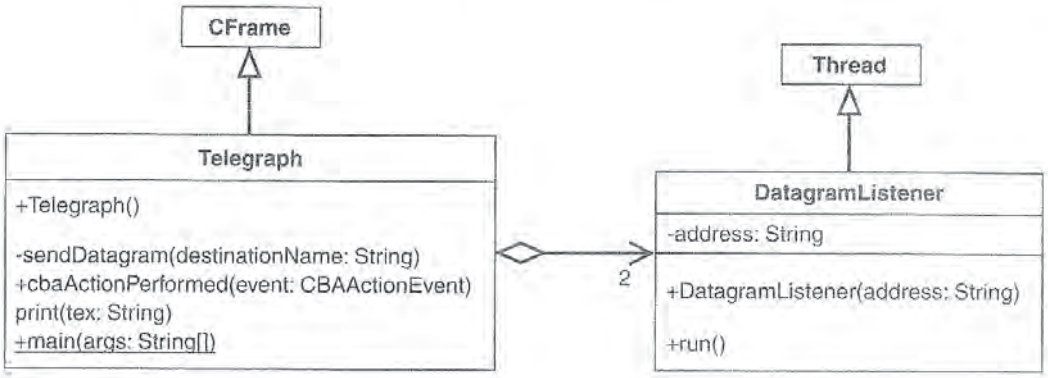


Figure 5.20 Telegraph's class diagram.

UDP or WDP. DatagramListener is an inner class to Telegraph, and as such has access to its methods and variables.

The application is created in the com.symbian.devnet.crystal.telegraph package and we import the javax.net.datagram JavaPhone package:

```
package com.symbian.devnet.crystal.telegraph;
import javax.net.datagram.*;
:

public class Telegraph extends CFrame
{
    :
    :
```

`main()` creates an instance of our class:

```
public static void main(String[] args)
{
    new Telegraph();
}
```

Most of the constructor is concerned with building the user interface. After this we find out our local IP address and display it in the TextField `udpDestination`. We then create and start two listeners for incoming Datagrams. The first is our UDP listener, the second our WDP listener:

```
Telegraph()
{
:
:
    try
    {
        String address = InetAddress.getLocalHost().toString();
        int slash = address.indexOf('/');
```

```
            String localAddress = (slash != -1)
                ? address.substring(slash + 1) : address;
            udpDestination.setText(localAddress);
        }
        catch (UnknownHostException uhe)
        {
            println("Cannot obtain a local IP address: " + uhe);
        }

        new DatagramListener("udp://"+udpDestination.getText()
            + ":7654").start();
        new DatagramListener("wdpsms://+441632960000:7655").start();
    }
```

`print()` is used to print useful information in a TextArea, `receiveT`:

```
    final void print(String text)
    {
        receiveT.append(text);
    }
```

`println()` adds an end of line character to the text to be displayed and then calls `print()`.

`sendDatagram()` sends a Datagram to the destination specified by `destinationName`. The message to send comes from the `sendMessageT` input field (a TextField).

`parseAddress()` creates an Address for the destination using `destinationName`. Given the destination Address, `destination`, we can create a DatagramService, `service`. Finally we use `service` to send our Datagram:

```
    private void sendDatagram(String destinationName)
    {
        try
        {
            Address destination = DatagramService.parseAddress
                (destinationName);
            DatagramService service = DatagramService.getService
                (destination);
            service.send(new Datagram(sendMessageT.getText().getBytes(),
                destination));
            println("Successfully sent message to " + destination);
        }
```

The above can throw AddressNotSupportedException, MalformedAddressException, IOException, IntermittentNetworkException, InterruptedIOException:

```
        catch(Exception e)
        {
            println("Failed to send message: " + e);
        }
    }
```

`cbaActionPerformed()` handles user input. It serves a similar purpose to `actionPerformed()` on Quartz.

In the Telegraph example `cbaActionPerformed()` sends a WDP, UDP, or SMS message in response to user input. It takes the address entered by the user, adds in the required address components, and calls `sendDatagram()` which does the real work. Note that we are arbitrarily sending and listening on port 7655 for WDP datagrams and on port 7654 for UDP datagrams:

```
public void cbaActionPerformed(CBAEvent e)
{
    switch (e.getID())
    {
        case SEND_SMS:
            sendDatagram("sms://" + smsDestination.getText());
            break;

        case SEND_WDP:
            sendDatagram("wdpsms://" + wdpDestination.getText() +
                ":7655");
            break;

        case SEND_UDP:
            sendDatagram("udp://" + udpDestination.getText() +
                ":7654");
            break;

        case CLOSE:
            shutDown();
            break;

        default:
    }
}
```

DatagramListener is an inner class to Telegraph. It extends Thread with the `run()` method used to listen for incoming Datagrams on the specified address:

```
class DatagramListener extends Thread
{
```

`address` holds the address which we will listen on. The constructor sets up this address.

```
String address;

DatagramListener(String address)
{
    this.address = address;
}
```

`run()` first creates an Address from our address String and then uses this to get a DatagramServerService, `service`:

```
public void run()
{
    try
    {
        Address receiveAddress
            = DatagramService.parseAddress(address);
        DatagramService service
            = DatagramService.getServerService(receiveAddress);
        println("Started service " + service + "on "
            + receiveAddress);
```

Having set up the Address and the DatagramService we loop forever waiting for incoming Datagrams. To hold the received message we create the largest buffer this service supports and then create a new Datagram, `datagram`, using our buffer. When a message arrives we receive it in `datagram` and print its contents in the status area:

```
        while( true )
        {
            byte[] buffer
                = new byte[service.getMaximumLength()];
            Datagram datagram = new Datagram(buffer);
            datagram.setLength(buffer.length);
            service.receive(datagram);
            println((new String(datagram.getData())).trim());
        }
    }
```

There are a couple of problems with this code. We should really check the maximum length before we create the buffer: an SMS message is limited to 160 characters, which is reasonable, however a WDP/SMS message is up to 16,384 characters, and a UDP message can be 49,152 characters! Also we create a new Datagram for each received message: it would be better to create a Datagram once for each service and reuse it.

The above can throw a variety of exceptions. For simplicity we'll just catch a general Exception:

```
    catch(Exception e)
    {
        println("Problem in UDP listener: " + e);
    }
}
```

5.10 The Power Monitoring API

The Power Monitoring API allows an application to monitor and respond to changes in the power level of the system and is implemented by the

javax.power.monitor package. An application is able to retrieve the current battery level, an estimate of remaining battery life, and whether an external power source is being used. It also enables an application to be notified when the battery power is running low and a service is about to be terminated as a result.

5.10.1 The Power Monitoring Classes

Power monitoring is probably the most straightforward of the JavaPhone APIs. The simplified class structure is shown in Figure 5.21.

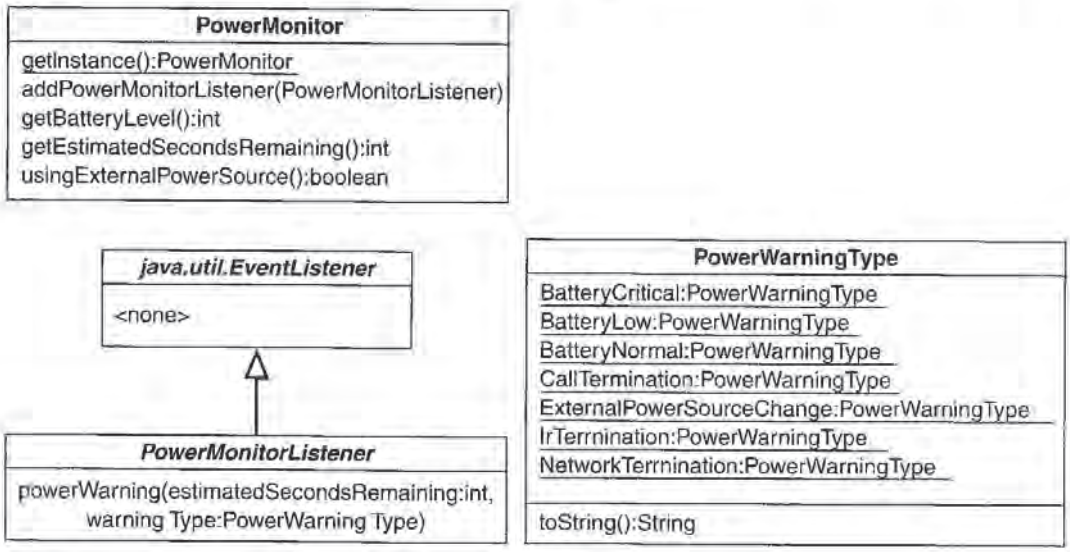


Figure 5.21 Power monitoring classes.

If an external power source is being used `getBatteryLevel()` and `getEstimatedSecondsRemaining()` will return Integer.MAX_VALUE. They will return −1 if a reading is not available. `getBatteryLevel()` will otherwise return an integer value in the range of 0 (no battery life remaining) to 100 (batteries are fully charged).

The PowerWarningType class indexes the types of power warnings an application can listen for. The intent is for this class to define the standard set of warnings across a range of devices, though hardware constraints may mean that it is not possible to distinguish between all of these events.

5.10.2 The Power Monitor Example

On start-up the Power Monitor example determines the time remaining on battery, the battery level, and whether we are on batteries or an external supply. These values can be rechecked by pressing the Check power status button.

The application will also pick up power change events, such as a change from battery to external power, and display the results in the status area.

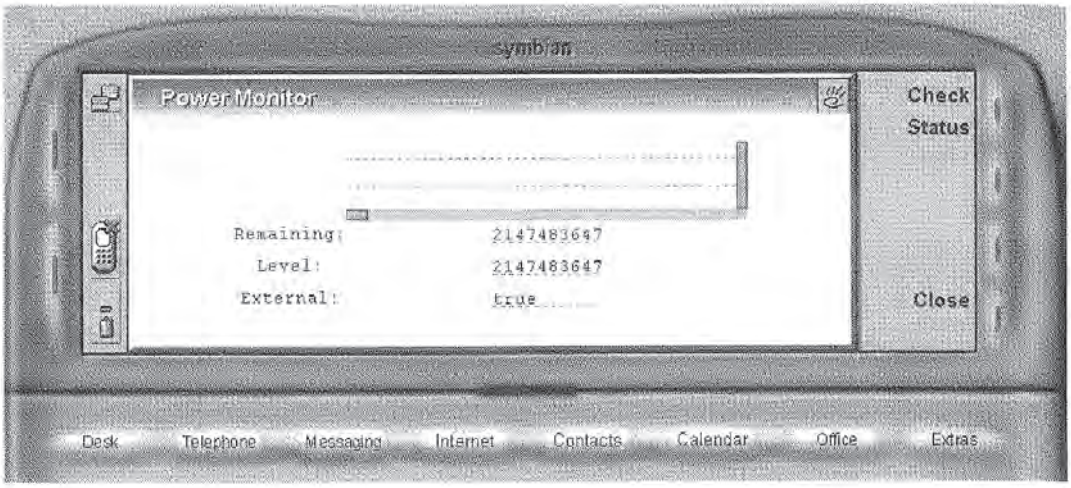


Figure 5.22 The Crystal version of the Power Monitor application.

The Remaining and Level values are at Integer.MAX_VALUE because the example is being run under an Emulator, and hence effectively on external power.

All the code for the application is listed to help illustrate how the other examples in this paper are written.

Each example is in its own package within com.symbian.devnet.crystal or com.symbian.devnet.quartz, in this case com.symbian.devnet.crystal.power. We need to import the javax.power.monitor package and also com.symbian.devnet.crystal.awt, which contains our CFrame utility class plus other useful classes.

```
package com.symbian.devnet.crystal.powermonitor;
import com.symbian.devnet.crystal.awt.*;
import com.symbian.epoc.awt.*;
import javax.power.monitor.*;
import java.awt.event.*;
import java.awt.*;
```

The Power Monitor example extends CFrame and implements the PowerMonitorListener and CBAListener interfaces:

```
public class PowerMonitor extends CFrame implements CBAListener,
    PowerMonitorListener
{
```

Set up the various UI components:

```
private static GridPanel panel = new GridPanel();
private static TextArea statusF = new TextArea(4, 40);
private static TextField remainingF = new TextField(10);
private static TextField levelF = new TextField(10);
```

```
private static TextField externalF = new TextField(10);
private static javax.power.monitor.PowerMonitor monitor;

private static CBAHandler cba;

final static int REPORT = EikCommandButtonGroup.BUTTON1;
final static int CLOSE = EikCommandButtonGroup.BUTTON4;
```

`main()` creates an instance of our example:

```
public static void main(String args[])
{
    new PowerMonitor();
}
```

The constructor builds the UI, including the Crystal Control Button Array (CBA):

```
PowerMonitor()
{
    setTitle(Version.TITLE);
    aboutDialog.setTitle("About " + Version.TITLE);
    aboutDialog.text = Version.DIALOG_TEXT;

    cba = new CBAHandler(this);
    cba.setText(REPORT, "Check\nStatus" );
    cba.setText(CLOSE, "Close" );
    cba.activate();

    panel.add(statusF, 1, 0);
    panel.add(new Label("Remaining:"), 0, 1);
    panel.add(remainingF, 1, 1);
    panel.add(new Label("Level:"), 0, 2);
    panel.add(levelF, 1, 2);
    panel.add(new Label("External:"), 0, 3);
    panel.add(externalF, 1, 3);
    add(panel);

    show();
```

Obtain a reference to the system's PowerMonitor and add a Listener (this instance of the PowerMonitor class) to it. `reportStatus()` checks and displays current power status:

```
    monitor = javax.power.monitor.PowerMonitor.getInstance();
    monitor.addPowerMonitorListener(this);
    reportStatus();
}
```

`reportStatus()` uses our instance of the javax.power.monitor.PowerMonitor to obtain power information and to display the results:

```
private static void reportStatus()
{
    remainingF.setText("Checking");
    levelF.setText("Checking");
    externalF.setText("Checking");
```

Get the current time and battery level:

```
    int remaining = monitor.getEstimatedSecondsRemaining();
    int level = monitor.getBatteryLevel();
    boolean external = monitor.usingExternalPowerSource();

    remainingF.setText("" + remaining);
    levelF.setText("" + level);
    externalF.setText("" + external);
}
```

`print()` is used to display output and status information. A console is by default not available to EPOC devices, so output from `System.out.println()` is lost:

```
private void print(String text)
{
    statusF.append(text);
}

private void println(String text)
{
    statusF.append(text + "\n");
}
```

`powerWarning()` implements our PowerMonitorListener interface. We just display information about the event, e.g. change from external to battery power:

```
public void powerWarning(int estimatedSecondsRemaining,
    PowerWarningType warning)
{
    println(estimatedSecondsRemaining + "s remaining: " + warning);
}
```

`cbaActionPerformed()` picks up user events, in this case a request to check on the power status:

```
public void cbaActionPerformed( CBAEvent e )
{
    switch (e.getID())
    {
        case REPORT:
            reportStatus();
            break;
```

```
                case CLOSE:
                    shutDown();
                    break;

                default:
            }
        }
    }
```

5.11 The Communications API

The Communications API is implemented by the javax.comm package. It provides access to a device's serial ports, including IR, and parallel ports.

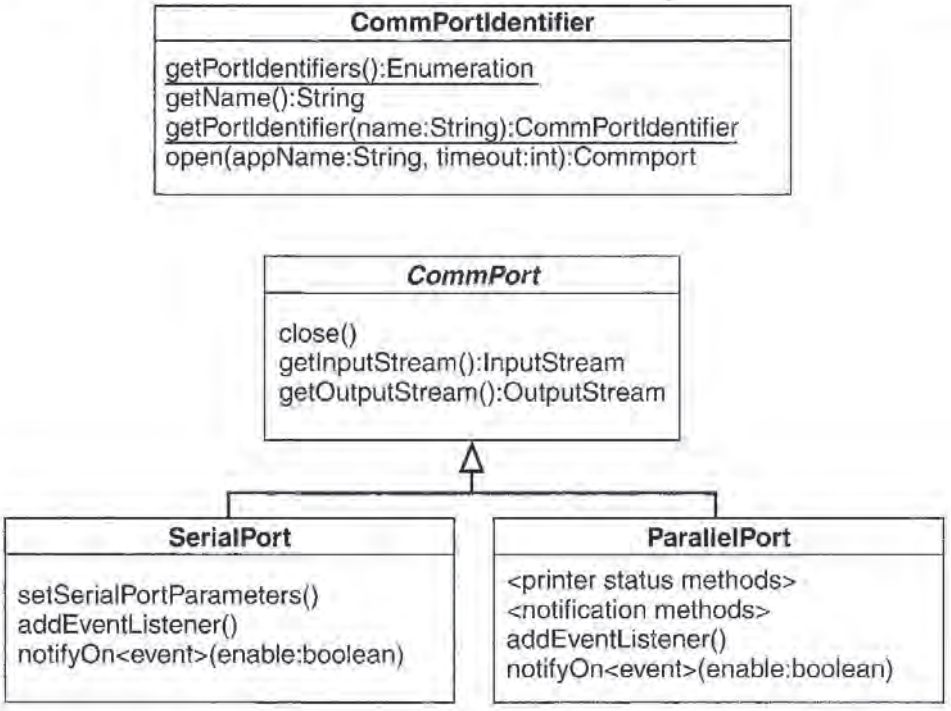


Figure 5.23 The communications package's main classes.

5.11.1 The Communications Classes

The CommPortIdentifier class is used to discover the device's available ports, to find out about a port (its name, its type), and to open a CommPort.

An open CommPort is obtained using `CommPortIndentifier.open(String appName, int timeout)`, where `appName` is the name representing the application (used to resolve port ownership conflicts) and `timeout` is the maximum time in ms that the call should block waiting for the CommPort to open.

Once we have a CommPort we can obtain an InputStream and an OutputStream (using `getInputStream()` and `getOutputStream()`), and we can close it.

We can add an EventListener to a port which will allow us to listen for events generated when the port changes state. To enable an event so that it will be caught by our EventListener use `notifyOn<event>()`, e.g. `SerialPort.notifyOnOutputEmpty(true)` will generate an event when all the data in the port's output buffer has been sent. When we receive an Event we can use `Event.getNewValue()` and `Event.getOldValue()` to determine the relevant status of the port (which will be either true or false) after and before the event.

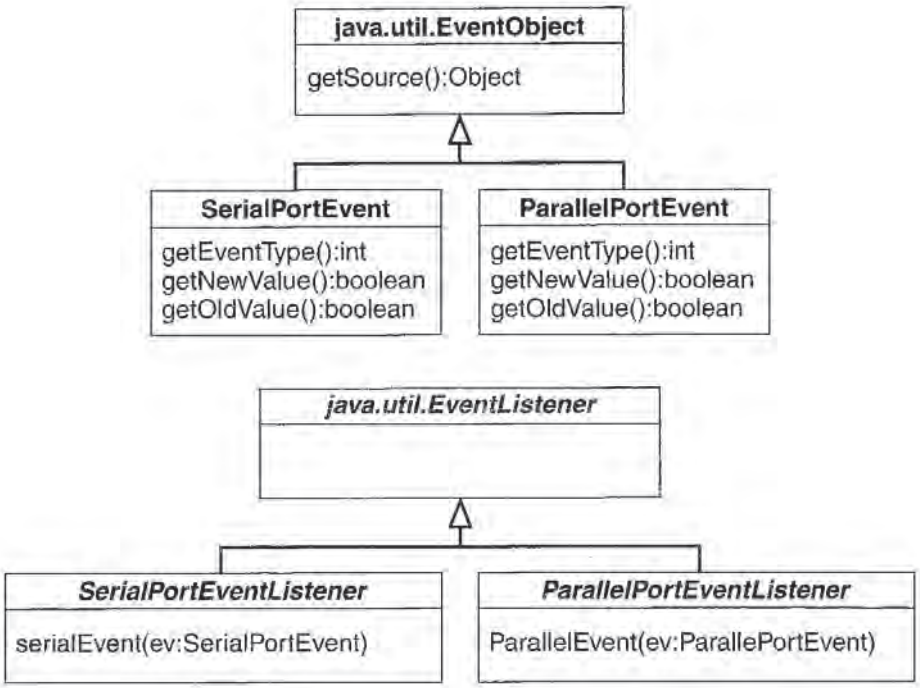


Figure 5.24 The communications package's events and event listeners.

5.11.2 The Serial Chat Example Application

The Serial Chat example provides a chat facility between any two machines running a JVM (in fact, provided a correct javax.comm is on the class path this application will run under Solaris, Windows, etc.). Either run the example on both machines, or run a general comms program on one of them. The code arbitrarily sets the baud rate to 2400, with 8 data bits, 1 stop bit, and no parity, though this can be easily changed.

The program displays a list of the ports on the device when it starts up. Press one of the initialize buttons, type some text into the top window, then either press SEND or hit the ENTER key. Received data is displayed in the bottom window along with any status information.



Figure 5.25 The Quartz version of the Serial Chat application.

The Serial Chat's class structure is shown in Figure 5.26.

The application uses the javax.comm package. This is the Quartz version and so the main class, SerialChat, extends QFrame. It implements the ActionListener interface to handle user input and the SerialPortEventListener to handle serial port events. `port` is a reference to our open serial port, itself a specialization of a CommPort. `outputStream` is an OutputStreamWriter so that we can write a line of text at a time to the port:

```
import javax.comm.*;
:
public class SerialChat extends QFrame implements ActionListener,
SerialPortEventListener
{
    private SerialPort port;
    private InputStream inputStream;
    private OutputStreamWriter outputStream;
```

Let's now go through each method in turn:

Most of the constructor is involved with creating the UI. After creating the UI we go through the available ports and print out their names:

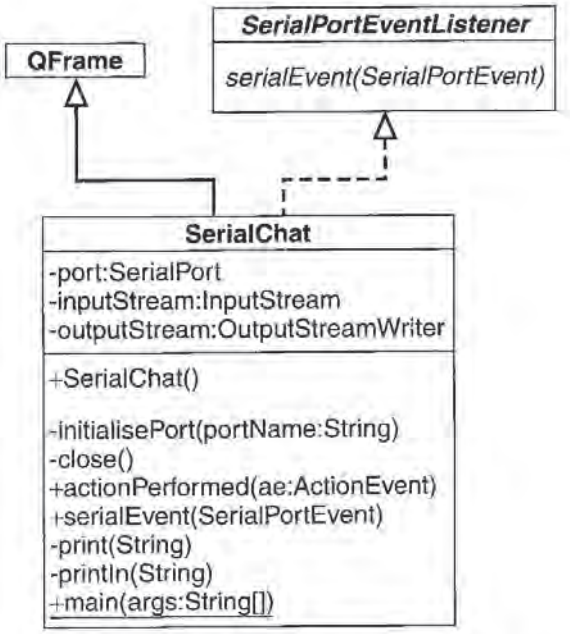


Figure 5.26 Structure of the Serial Chat program.

```
public SerialChat()
{
    :
    :
    Enumeration ports = CommPortIdentifier.getPortIdentifiers();
    while(ports.hasMoreElements())
    {
        println(
            ((CommPortIdentifier)(ports.nextElement())).
                getName());
    }
}
```

`main()` just creates an instance of our SerialChat class.

`actionedPerformed()` picks up user events, and in response will do one of:

- initialize the IR or RS232 port
- send text
- close the port

```
public void actionPerformed(ActionEvent ae)
{
```

Initialize the relevant port:

```
    if(ae.getSource() == initRS232B) initializePort("COM1");
    else if(ae.getSource() == initIRB) initializePort("IrCOMM");
```

If the Send button was pressed get the text from the `sendT` text field and write it to our port's output stream. After writing we need to flush the output stream:

```
        else if((ae.getSource() == sendT) || (ae.getSource() == sendB))
        {
            if(outputStream == null)
                println("-- Port is not yet initialized");
            else
            {
                println("-- Sending");
                try
                {
                    outputStream.write(sendT.getText() + "\n");
                    outputStream.flush();
                }
                catch(IOException ioe)
                {
                    println("-- SerialChat.actionPerformed()
                        exception:"+ ioe);
                }
                sendT.selectAll();
            }
        }
```

Close the port if the Close button was pressed:

```
        else if(ae.getSource() == closeB) close();
    }
```

`initializePort(String portName)` initializes and opens the required port:

```
    private void initializePort(String portName)
    {
        println("-- Initializing");
        try
        {
```

Get a CommPortIdentifier for the specified port, `portName`. Cast it to a serial port and configure it:

```
            CommPortIdentifier portId
                = CommPortIdentifier.getPortIdentifier(portName);
            port = (SerialPort)portId.open("Serial Chat", 5000);
            port.setSerialPortParams(2400,
                SerialPort.DATABITS_8,
                SerialPort.STOPBITS_1,
                SerialPort.PARITY_NONE);
```

Add the event listener (which is this class instance) to the port. Note that only one listener can be added to a serial port. Request that we receive notification when the write buffer is empty and when we have received data:

```
            port.addEventListener(this);
            port.notifyOnDataAvailable(true);
            port.notifyOnOutputEmpty(true);
```

Get an OutputStreamWriter from our port so that we can write Strings and also get an InputStream:

```
            outputStream = new
OutputStreamWriter(port.getOutputStream());
            inputStream = port.getInputStream();
            initRS232B.setEnabled(false);
            initIRB.setEnabled(false);
            println("-- Ready to send and receive");
    }
```

The above code can throw: UnsupportedCommOperationException, NoSuchPortException, PortInUseException, or TooManyListenersException. So we'll keep things simple and just have one catch clause:

```
        catch(Exception e)
        {
            println("-- Failed to open port '"
                + portName + "'. Exception: " + e);
            close();
        }
}
```

The serial port events are handled by `serialEvent()`. First print out details of any event received. If the event is `OUTPUT_BUFFER_EMPTY` just write a message saying we've sent our text:

```
public void serialEvent(SerialPortEvent spe)
{
    println("Event: " + spe.getEventType() + "; " + spe);

    switch(spe.getEventType())
    {
        case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
            println("-- Sent --");
        break;
```

This is where we handle received data in response to a `DATA_AVAILABLE` event. We see how much data has arrived, create the correct sized buffer, and read the data into the buffer. This is not the most efficient code: we are creating

a byte array and a String for each character or group of characters received, which is excessive:

```
        case SerialPortEvent.DATA_AVAILABLE:
             try
             {
                  int bytesAvailable = inputStream.available();
                  byte[] readBuffer = new byte[bytesAvailable];
                  int numBytes = inputStream.read(readBuffer);
                  System.out.print(numBytes);
                  print(new String(readBuffer));
             }
             catch(IOException ioe)
             {
                  println("-- serialEvent() exception: " + ioe);
             }
        break;
    }
}
```

And finally we have the `close()` method. All this does is check if the port is valid, and if it is closes the output stream, the input stream, and the port:

```
private void close()
{
    try
    {
         if(port != null)
         {
              outputStream.close();
              inputStream.close();
              port.close();
              port = null;
              initRS232B.setEnabled(true);
              initIRB.setEnabled(true);
              sendB.setEnabled(false);
              closeB.setEnabled(false);
              println("-- Closed --");
         }
    }
    catch(IOException ioe)
    {
         println("-- SerialChat.close() exception: " + ioe);
    }
}
```

5.12 Further Reading

The JavaPhone 1.0 API specification can be downloaded from ***http://java.sun.com/products/javaphone/***. This also provides a general starting point for JavaPhone, including some JavaPhone use cases.

A good description of the JTAPI telephony model can be found within the API specification in ***doc/javax/telephony/package-summary.html***.

The javax.comm specification is located at ***http://java.sun.com/products/javacomm/***.

The javax.net.ssl specification is located ***http://java.sun.com/products/jsse/***.

6

Wireless Java Games

Lucy Sweet

In this chapter, we demonstrate how to develop a strategy game for Symbian devices and look at the special considerations that action games require on constrained devices.

6.1 Developing a Strategy Game

6.1.1 The Multiplayer Strategy Game

Although there are many ways of categorizing games, for example, single-player and multi-player, we will initially split them into strategy and action games, before looking in detail at an example of a strategy game. Strategy games differ from action games in that they are generally turn-based and require a significant amount of thought before each turn. As a result, there are periods of inactivity as other players make their moves. In contrast, action games require constant attention from the user. Both strategy and action games may be either single player, like Patience, or multi-player, like Whist or the ever-popular Quake. An additional consideration for developing both single-player and multi-player games is games and game-modes in which the computer plays the part of one or more opponents.

There are many permutations that can be achieved with this form of categorization and we will also look in detail at two other categories, *server-less* and *server-enabled* games. In the wireless market, however, we are less interested in single-player games. One reason is that many current mobile devices already provide a variety of basic single-player games, some of which can be linked by infrared. But the compelling reason for developing multi-player games for advanced clients is that we can use the latest communication technology to earn revenue through careful use of network airtime.

6.1.2 Wireless Whist

The game of Whist is a simple turn-based card game, but it is ideal for demonstrating a multi-player strategy game that will run over several WIDs.

This particular example doesn't make use of remote services but uses peer-to-peer protocols. It is also a good demonstration of an application that runs over a mobile telephone network without requiring a continuous connection throughout the duration of play.



Figure 6.1 Introducing Whist.

This game of Whist requires four players, split into two partnerships. These partnerships are decided by the order the players are selected at initiation and, once decided, cannot be changed. The application mimics shuffling and dealing a pack of cards and distributes the correct number to each player. The first player selects the card in their hand that they wish to play. This is sent to the next player where it is shown to them. The second player completes their turn and the cards played are sent to the next player, and so on. When all four players have taken a turn the application establishes the winner of the trick. The winner then starts the next trick. The winner is the partnership that is the first to reach five points once the first six tricks have been deducted.

All communication for the game occurs via the Short Messaging Service (SMS). In this example, the emphasis is placed on minimizing the cost to the user, so only the player whose turn it is next is sent an SMS; the other players must wait their turn. The disadvantage to this is that the other players will always be slightly behind the game.

One of the concepts behind this implementation of Whist was to develop a generic model for transferring data between WIDs that could be easily reused by other turn-based games. The result of this was an application which had two engines, one used for controlling the connectivity and data transfer details and the other used for processing the data belonging to the game once it has

been extracted from the SMS. On top of this sits the *User Interface Controller*, which looks after the graphics and allows the users to enter data.

WAP-based SMS (WDP/SMS) is used as it allows the Whist application to receive messages of this type sent to a specific port. (With traditional SMS, a message goes straight to the inbox on the mobile phone and there is no mechanism for a Java application to intercept the message.)

The design principles used in the development of this game can be applied to many other games and applications and the underlying code can, with the addition of some extra features to increase robustness and increase flexibility, be used for most turn-based games, regardless of the number of players involved.

6.1.3 Packages

The application is split into three separate packages: the *transport package*, "TokenRing"; the *game processing package*, "Game"; and the *user interface package*, "awt." The packages are designed so that each has a distinct function and there is limited (and well-defined) data exchange between them. This allows us to replace packages, provided that they supply the basic functionality required by the remaining classes. Figure 6.2 demonstrates the relationship between the three engines.

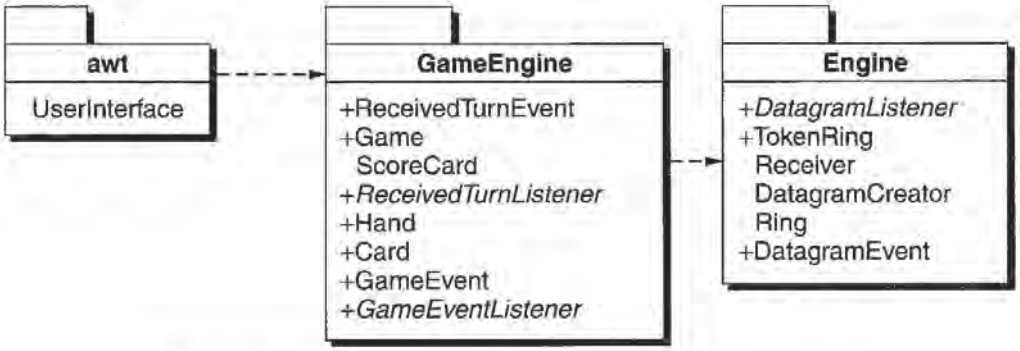


Figure 6.2 The relationship between the game's three engines.

This application uses both the Datagram and Contacts APIs from JavaPhone. The underlying TokenRing, which is responsible for sending, receiving, and performing preliminary data processing on the Datagrams, uses the Datagram API.

6.1.4 TokenRing Engine

If you imagine watching a group of people playing Whist, you would likely envisage them sitting in a circle, with the cards being played in order, clockwise or anti-clockwise, one player after the other. This ordered arrangement is similar to the token ring arrangement used, for example, in computer networks where each machine takes its turn to transmit data. It was because of this similarity that

a token ring-style engine was developed to send the data to the players. Closer inspection of the package does, however, highlight some differences — we are able to send a message to more than one player from any machine. We also have the ability to send a message to any one of the players, not just the person whose turn follows. This added functionality and flexibility allow more games to utilize this engine, while ensuring that the mechanism is simple, effective, and reliable.

We can look at the TokenRing engine by following a Datagram from its creation and sending, through to its reception at its destination. The Datagram itself is created in the `DatagramCreator` class, which is constructed using `DatagramCreator(DatagramService service, Ring ring)`. This method takes a reference to the DatagramService we create in the TokenRing and also a reference to the Ring class, where the addresses of the other players are stored.

```
class DatagramCreator
{
    private DatagramService dgService;
    private Ring ring;

    DatagramCreator(DatagramService service, Ring ring)
    {
        dgService = service;
        this.ring = ring;
    }
```

There are two methods for sending Datagrams, both taking a String of text to send, and a unique identification number. The difference between the two methods is the destination argument. In the method shown below, we pass an array of destinations. These correspond with the addresses in the array above; for example, if we stored addresses for Sue, Bob, Jane, and John and want to send to just Bob and Jane, we would pass {1, 2} for the destination. This method then makes two successive individual calls to `public void createAndSend(String str, int destination, int idNo)`. This method is rarely used, as the game was designed to ensure that each player sends only one message per turn. However, its inclusion demonstrates how to send Datagrams to more than one WID and can be used for a vital real-world function if one player were to prematurely finish a game, allowing them to send a message to each of the other players informing them of the game end. If the Datagram cannot be sent, for example if there is no network coverage, a dialog will appear on the screen, prompting the user to either resend the message or exit the game. This class uses the `TaskSwitch.notify()` method discussed in Chapter 13 and the same notification process is used elsewhere in the TokenRing package to inform the user that sending has failed. Note

that we use "Information" as a title to the error message to avoid panicking the user.

```
public void createAndSend(String str, int[] destination, int idNo)
{
    boolean sentSuccessfully = true;
    int length = destinations.length;
    for (int i = 0; i < length; i++)
    {
        sentSuccessfully = createAndSend(str, destinations[i],
          idNo);
        while (!sentSuccessfully)
        {
            int ret = TaskSwitch.notify("Information", "Failed to
                send SMS", "Retry", "Exit");
            if (ret == 0)
            {
                sentSuccessfully = createAndSend(str,
                   destinations[i], idNo);
            }

            else
                System.exit(0);
        }
    }
}
```

This method sends a Datagram to the address that corresponds to the destination passed in; for example, using Sue, Bob, Jane, and John, as above, we would send to John by setting the destination to 3. Note that if messages are to be sent to groups of people on a regular basis, it would be more efficient to have this method calling the one above, with an array of just one int. We return false if we are unable to send a Datagram. This allows the calling code to take appropriate action, one course of which is discussed in the method above. Another solution would be to wait for a short time and then to try again without informing the user.

```
public boolean createAndSend(String str, int destination,
    int id)
{
     int id = idNo;
     String st = (String.valueOf(id) + "?" + str).trim();

     Datagram dg = new Datagram(st.getBytes(), ring.
         getAddress(destination));
     try
     {
          senderService.send(dg);
     }
     catch (Exception e)
```

```
            {
                return false;
            }
            return true;
        }
    }
```

Initially, only the device that starts the game knows the addresses of all the participants, so the DatagramCreator class is not constructed in the other devices until the first WDP/SMS containing the player information arrives. We are also unable to construct the Ring class for the same reason.



Figure 6.3 Initially, only the device that starts the game knows the addresses of all the participants.

We have seen how the Datagrams are created and sent, but how are they received by the application? The answer is simple: we have a Receiver class (which is shown below) that listens on a port for WDP/SMS messages to arrive.

The maximum length of a WDP/SMS is defined by the operator, so we call `DatagramService.getmaximumLength()` in the TokenRing class to determine the maximum size and then create a buffer of that size. We must ensure that we have enough buffer space for all the data coming in as we do not know the length of the Datagram in advance. We are able to use the same DatagramService as the DatagramCreator uses for sending the Datagrams.

```
class Receiver
{
    private static int maxSize;
    private DatagramService dgService;
    private Vector listeners = new Vector();

    private Thread receiveThread;
    private byte[] buffer;

    Receiver(DatagramService dgService, int maxSize)
    {
        this.dgService = dgService;
        this.maxSize = maxSize;
    }
```

This method requires a new thread to receive Datagrams to ensure that no other threads are blocked while it is running. We create an array of bytes of the maximum size of a Datagram to buffer the data once it has arrived. This array is then used to construct a new Datagram. We then call `dgService.receive(dg)` to receive the incoming Datagram.

Once a Datagram has been successfully received, we create a DatagramEvent. Finally, we start the receiveThread and let it wait for messages.

```
public void listen()
{
    receiveThread = new Thread()
    {
        public void run()
        {
            while(true)
            {
                try
                {
                    buffer = new byte[maxSize];
                    char zero = ' ';
                    Datagram dg = new Datagram(buffer);
                    int sizeOfDatagram = dg.getLength();
                    for (int i = 0; i < sizeOfDatagram; i++)
                        buffer[i] = (byte)zero;
                    dgService.receive(dg);
                }
                catch(Exception ex)
                {
                    System.exit(0);
                }
                DatagramEvent dge = new DatagramEvent(this,
                    buffer);
                processDatagram(dge);
```

```
                }
            }
        };
        receiveThread.start();
    }
```

Once a Datagram has been successfully received and passed into the TokenRing class, it is processed in `processMessage()`, which extracts the identification number from the data received. It then checks the local identification number. If this is zero we know that no previous messages have been received and that the message which has just arrived contains the information to set up the token ring. If this is the case, the data is extracted into a String that is passed to the constructor for the Ring class. Once the Ring has been created, we can construct the DatagramCreator class. The player number is calculated by adding one to the identification number from the Datagram. This is also used to determine if the data needs to be sent to any other players. If this is the case, the data is passed on unaltered to the next player in the list, otherwise the player receiving the message is the last one in the list of selected players and passes an "X" back to the initiator to indicate that the ring is complete.

```
public class TokenRing implements DatagramListener
{
:
    private void processMessage()
    {
        int id = getidNo();
        boolean sentSuccessfully = true;
        if (idNo == 0)
        {
            idNo = id;
            playerNo = idNo + 1;
            String string = getData(data);
            ring = new Ring(string);
            dgCreator = new DatagramCreator(dgService, ring);
                        setNextAddress();
            if (playerNo < ring.noOfPlayers)
            {
                sentSuccessfully = dgCreator.
                    createAndSend(string, nextAddress, ++idNo);
                while (!sentSuccessfully)
                {
                    int ret = TaskSwitch.notify("Information",
                        "Failed to send SMS", "Retry", "Exit");
                    if (ret == 0)
                    {
                        sentSuccessfully = dgCreator
                            .createAndSend(string, nextAddress,
                                idNo);
                    }
```

```
                else System.exit(0);
            }
            else
            {
                ReceivedTurnEvent ge = new ReceivedTurnEvent
                    (this, c);
                sentSuccessfully = dgCreator.createAndSend("X",
                    nextAddress, ++idNo);
                while (!sentSuccessfully)
                {
                    int ret = TaskSwitch.notify("Information",
                        "Failed to send SMS", "Retry", "Exit");
                    if (ret == 0)
                    {
                        sentSuccessfully = dgCreator.
                            createAndSend(string, nextAddress,
                              idNo);
                    }

                    else
                        System.exit(0);
                }
            }
        }
```

If the current identification number is not zero and is greater than the identification number attached to the message, we know that this is an old or duplicate message that can be discarded. However, it is conceivable that some games will want to use old information, for example to build player profiles or to ensure data consistency throughout play. It may also be useful to check for old and duplicate messages during testing phases.

```
else if (id <= idNo)
{
}
```

If the current identification number is not zero and is less than the identification number attached to the message, we know that this is a new, valid message and the data is to be extracted and passed to the Game class for processing.

```
else
{
    idNo = id;
    String string = getData(data);
    ReceivedTurnEvent ge = new ReceivedTurnEvent(this,
       string);
    processReceivedTurn(ge);
}
}
```

The Ring class contains all the information required for the token ring to run, which is essentially the names and addresses of all those involved. There are two constructors, which are shown below. The first is used by the device which initiates a game and takes two string arrays containing the names and addresses of the participants, for example {Jane, Bob, Sue, James} and {+44123456789, +44234567890, +44345678901, +44456789012}. In this constructor, we parse the String addresses, returning an Address which can be used to send Datagrams. This constructor assumes that the same number of addresses as names is passed in. This checking is done in the TokenRing class before we initiate the Ring. We also assume that we are able to parse all of the addresses we pass in. In reality this might not be possible so we should provide some mechanism to prompt the user to check that the number entered is correct.

```
class Ring
{
        private static final char QUESTION_MARK = '?';
        public static int noOfPlayers;
        private String[] names;
        private Address[] addresses;
        Ring(String[] names, String[] addresses)
        {
                for (int i = 0; i < noOfPlayers; i++)
                {
                        this.names[i] = names[i];
                        try
                        {
                             this.addresses[i] =
                                 DatagramService.
                                  parseAddress(addresses[i]);
                        }
                        catch(Exception e)
                        {
                        }
                }
        }
;
}
```

The second constructor takes a String containing the data which is received by the TokenRing from another player. An example of this data would be "+44123456789? + 44234567890? + 44345678901? + 44456789012?Jane?Bob?Sue?James?". This constructor cycles through the data, counting question marks. It then divides this number by 2 to determine the number of players in the game. Once this is done, it extracts the relevant number of addresses (as Strings) and sends this temporary array of addresses to setUpAddresses which parses them, filling the array Address[] `addresses`. We extract the names of the players in a similar manner; however, no further

processing is required. We can assume that the number of addresses and names are identical as this was checked by the machine that started the game.

```
Ring(String details)
{
      // Count the number of question marks to determine
      // the number of players.
      int length = details.length();
      int count = 1;
      int index = details.indexOf(QUESTION_MARK);
      while ((index < length) && (index != -1))
      {
            index = details.indexOf(QUESTION_MARK,
               ++index);
            count++;
      }
      noOfPlayers = count / 2;
      names = new String[noOfPlayers];
      String[] tempAddresses = new String[noOfPlayers];
      count = 0;
      int start = 0;
      int end = 0;
      for (int i = 0; i < noOfPlayers; i++)
      {
            end = details.indexOf(QUESTION_MARK,
                 start);
            tempAddresses[i] = details.substring
                 (start, end);
            start = ++end;
       }
       for (int i = 0; i < noOfPlayers; i++)
       {
            end = details.indexOf(QUESTION_MARK,
                start);
            names[0] = details.substring(start, end);
            start = ++end;
       }
       setUpAddresses(tempAddresses);
}
```

Again we assume that the addresses can be successfully parsed; however, in reality we should assume that this may fail and inform the user of the problem. We could prompt the user to check the address that has failed and allow them to change it. Another course of action would be to request that the sender resends the message. This could be performed without notifying either user.

```
private void setUpAddresses(String[] tempAddresses)
{
      addresses = new Address[noOfPlayers];
      for (int i = 0; i < noOfPlayers; i++)
```

```
        {
                try
                {
                        addresses[i] = DatagramService.
                            parseAddress(tempAddresses[i]);
                }
                catch(Exception e)
                {
                }
        }
}
```

The Ring also provides accessor methods to allow other classes to retrieve the names and addresses.

The class that ties the TokenRing engine together is the TokenRing class itself. We have already looked at the `processMessage` method of this class, and we will now take a closer look at its construction and some of the other key functions it provides.

The TokenRing class inherits from DatagramListener, an interface that is discussed in section 6.2.5 on communication between packages.

```
public class TokenRing implements DatagramListener
{
        private static final int UDP = 0;
        private static final int WDPSMS = 1;
        private static final int INITIAL_PORT = 1010;
        private static final char QUESTION_MARK = '?';
        private static final String DUMMY_ADDRESS_UDP =
            "udp://+1234567890:9999";
        private static final String DUMMY_ADDRESS_WDPSMS =
"wdpsms://+123456789:9999";
        private int maxDatagramLength;
private String commsAddress;
        private String port;
        private Ring ring;
        private Receiver receiver;
        private DatagramService dgService;
        private DatagramCreator dgCreator;
        private Address myAddress;
:
```

The constructor itself is quite simple, as it just allocates the port number for the DatagramService to use and starts the Receiver. The constructor allocates the port number by trying each one in an array until it finds one that is free. When this application is using WDP/SMS this procedure is unnecessary, as we do not have port conflicts on the devices. However, when we run more than one emulator on the same machine, we will have a conflict as two emulators are not able to listen to the same port. We initially set the protocol to UDP as this will be the one most commonly used for testing until real devices are available.

```
public TokenRing()
{
        protocol = UDP;
        port = String.valueOf(INITIAL_PORT);
        int count = 0;
        boolean b = setupEngine(port);
        while (!b)
        {
                port = String.valueOf(INITIAL_PORT + (++count));
                b = setupEngine(port);
        }
}
```

In the above code, we can see that the constructor makes a call to `setupEngine()`, passing it the port number obtained from the array. This method creates an address in the form of a String of the type indicated by the protocol variable and parses it to return an Address. We then create a DatagramService using this Address and create a Receiver using this service, having determined the maximum length of a Datagram allowed by the DatagramService being used.

We register interest in listening for incoming Datagrams and finally start the Receiver listening for Datagrams. If this is successful, we return "true"; however, if it fails we return "false." We expect this method to fail when the port we are trying to use has been taken up by another emulator, and this allows us to return to the constructor and select a different port number from the array.

```
private boolean setupEngine(String port)
{
      if (protocol == UDP)
              commsAddress = "udp://" + getNodeIPNo()
                 + ":" + port;
      else
              commsAddress = "wdpsms://" + getNodeIPNo()
                 + ":" + port;
      try
      {
              myAddress = DatagramService
                  .parseAddress(commsAddress);
      }
      catch(Exception ex)
      {
              return false;
      }


      try
      {
```

```
                dgService = DatagramService.
                    getServerService(myAddress);
                maxDatagramLength = dgService.getMaximumLength();
                receiver = new Receiver(dgService,
                    maxDatagramLength);
                receiver.addDatagramListener(this);
                receiver.listen();
        }
        catch(Exception e)
        {
              return false;
        }

        return true;
}
```

The following method to allows us to change from using UDP as the transport protocol to using WDP/SMS. This simply closes the `Receiver` which is using UDP, changes the protocol variable to WDP/SMS, and calls `setUpEngine(...)` with the default port number 1010. The current implementation of the `TokenRing` requires all emulators to use the same protocol. We will leave it as an exercise for the reader to ponder the value of allowing the `TokenRing` to use both protocols at the same time and how this may be implemented.

```
public void initiateSMS()
{
        receiver.close();
        protocol = WDPSMS;
        boolean b = setupEngine(String.valueOf(INITIAL_PORT) );
}
```

There are three methods for sending data to one or more other players. All three take a String as their first argument and increment the unique identification number.

The first method has no further arguments and passes the data on to the device which has been set as the next device in the ring. Note that we check that the message has been sent successfully and offer the user the option to resend or exit the game if sending fails.

```
public void sendMove(String s)
{
      boolean sentSuccessfully = true;
      sentSuccessfully = dgCreator.createAndSend
           (s,nextAddress, ++idNo);
      while (!sentSuccessfully)
      {
            int ret = TaskSwitch.notify("Information",
                "Failed to send SMS", "Retry", "Exit");
```

```
                if (ret == 0)
                {
                        sentSuccessfully = dgCreator.
                            createAndSend(s, nextAddress,
                            idNo);
                }
                else
                        System.exit(0);
        }
}
```

The second method allows specification of which device the data should be sent to, allowing us to free ourselves from the restrictions of a true ring architecture.

```
public void sendMove(String s, int next)
{
        boolean sentSuccessfully = true;
        sentSuccessfully = dgCreator.createAndSend(s, next,
            ++idNo);
        while (!sentSuccessfully)
        {
                int ret = TaskSwitch.notify("Information", "Failed
                    to send SMS", "Retry", "Exit");
                if (ret == 0)
                {
                        sentSuccessfully = dgCreator.
                            createAndSend(s, nextAddress, idNo);

                }

                else System.exit(0);
        }
}
```

The third method allows us to send the required data to all the other players. Note that this method does not require us to check that the message has been sent successfully. This checking is performed in the DatagramCreator class for each Datagram sent which is to inform the user which message(s) failed and prompt them for action on each individual failure. This course of action was deemed to be more flexible and informative to the user than performing the check in the TokenRing where we could only notify them that one or more messages could not be sent without giving them the opportunity to selectively resend messages.

In the Whist game this method is only used when a player decides to terminate a game before its natural end. For this reason, we take the identification number and add the number of players multiplied by two to ensure that this message has an identification number which is ahead of those on

other machines. It is important to note that the player terminating the game is not necessarily the one currently taking a turn and may have an identification number which is "out of date." If this method is required frequently, the current use of identification numbers may need to be changed.

```
public void sendMoveToAll(String s)
{
        int[] addresses = new int[ring.noOfPlayers - 1];
        int count = 0;
        for (int i = 0; i < ring.noOfPlayers; i++)
        {
                if (playerNo-1 != i)
                        addresses[count++] = i;
        }
        dgCreator.createAndSend(s, addresses, (ring.noOfPlayers
            * 2 ));
}
```

Although the functionality has not been implemented here, it is possible to specify methods that would allow us to send data to any combination of players. However, if we required this type of functionality we may decide that a different engine model may be more suitable to our game than the token ring model implemented here.

The penultimate method we will look at in this class is `start()` detailed below. This is called by the device that initializes the game, and is therefore the one that contains details of all the players who are to be involved, as well as their addresses. We first perform a check to ensure that we have received the same number of names as addresses!

```
public void start(String[] names, String[] addresses)
{
        if (names.length == addresses.length)
        {
                ring = new Ring(names, addresses);
                dgCreator = new DatagramCreator(dgService,
                    ring);

                StringBuffer strBuf1 = new StringBuffer();
                StringBuffer strBuf2 = new StringBuffer();
                for (int i = 0; i < ring.noOfPlayers; i++)
                {
                        strBuf1.append(addresses[i]);
                        strBuf1.append(QUESTION_MARK);
                        strBuf2.append(names[i]);
                        strBuf2.append(QUESTION_MARK);
                }
                strBuf1.append(strBuf2);
                String string = new String(strBuf1);
                idNo = 1;
```

```
                dgCreator.createAndSend(string, 1, idNo);
                playerNo = 1;
        }
        else
        {
                throw new IllegalArgumentException("The two
                   arrays passed should be the same length");
        }
}
```

The final method we will look at is `close()`. This allows us to close the Datagram service, freeing up system resources and the port number for future games. Currently we assume that this method is only called when the game is being closed down or we are changing protocol. In the former case it is not a problem if this fails; however, we may wish to take further action if this method fails in the latter case.

```
public void close()
{
        try
        {
                dgService.close();
                dgService = null;
        }
        catch(IOException ioe)
        {
                System.exit(0);
        }
}
```

The remaining methods in the `TokenRing` class are simple getter methods and data manipulation methods.

```
        ...
}
```

The `TokenRing` engine implementation discussed above is just one way of implementing this type of transport mechanism and many others exist, each with its own advantages and disadvantages. The discerning reader will have noticed that we have added functionality to our token ring which allows it to act in a far more flexible manner than a traditional token ring; however, we kept the name to demonstrate that the core functionality of this engine is to pass turns (tokens) around a circle of players. The implementation we have just described is still, in many ways, just a skeleton implementation, to which much functionality can be added. We will leave that to the discretion of the reader!

We have discussed the TokenRing package in detail and seen how the data is sent between devices, extracted, and processed. We now need to look at the way in which this data is passed between the packages and also how it

is transferred from the `Receiver`, which is running in its own thread, to the `TokenRing`.

6.1.5 Communication Between Packages

A design aim of the Whist application is to demonstrate good object-oriented design with data abstraction and minimal contact between packages. To achieve this we implemented a series of EventListener interfaces that listened for specific EventObjects. These allow data to be passed up the chain of packages, but without the need for a reference to that package. An example of this interfacing is between the Receiver Thread and the TokenRing package, which we will look at in more detail. The principle, however, is exactly the same for communicating between packages.

In the example, the Receiver class runs in its own thread to ensure that no other threads are blocked. Each class that will be generating events has a Vector for listeners:

```
private Vector listeners = new Vector();
```

and on examining the Receiver class you will notice that there is an `addDatagramListener(DatagramListener listener)` method, allowing other classes to register interest in incoming Datagrams. This takes as its argument the implementation of the DatagramListener interface which is to be called when a new Datagram is received, with its sole function being to add the DatagramListener interface calling the method to the Vector of listeners.

```
public void addDatagramListener(DatagramListener listener)
{
      listeners.addElement(listener);
}
```

This method is called from the `TokenRing` class, and is described above. It is conceivable that more than one class may be interested in listening for Datagrams, hence the allowance for more than one Listener. This additional functionality is not used in the Whist game and is just shown for the reference of the reader. It should be noted, however, that in the interests of performance a Vector may not be the most efficient method of storing the Listeners. The reader may wish to refer to Chapter 8 on optimization.

We also have a corresponding method to remove DatagramListeners from the Vector, shown below:

```
public void removeDatagramListener(DatagramListener listener)
{
      listeners.removeElement(listener);
}
```

Once a Datagram has been received, the data contained in it should be passed to the class or classes that have registered interest in incoming Datagrams. In this case, the only class to have done this is the TokenRing class. The `receiveThread` creates a DatagramEvent, which is passed to the `processDatagram()` method.

```
DatagramEvent dge = new DatagramEvent(this, buffer);
processDatagram(dge);
```

The `processDatagram(...)` method enumerates through the Vector of listeners, passing the `DatagramEvent` to each of the listeners in turn.

```
private void processDatagram(DatagramEvent notification)
{
      for (Enumeration en = listeners.elements();
          en.hasMoreElements();)
      {
            DatagramListener listener
                =(DatagramListener)en.nextElement();
            listener.DatagramReceived(notification);
      }
}
```

On receipt of a DatagramEvent from the Receiver class, the TokenRing class makes a copy of the data from the DatagramEvent and notifies the wait which is waiting on the same synchronized lock. This method is the implementation of the DatagramListener interface that the TokenRing class extends.

```
public class TokenRing implements DatagramListener
{
:
:
      public void DatagramReceived(DatagramEvent de)
      {
            dataReceived = de.getData();
            synchronized(this)
            {
                  notify();
            }
      }
```

Once `wait()` has been interrupted, we can process the data collected from the Datagram in any way required. This processing is done in the `processData()` method discussed in the section on the TokenRing class. Note that we do not anticipate any interruptions.

```
public void eventLoop()
{
```

```
        while (true)
        {
                try
                {
                        synchronized(this)
                        {
                                wait();
                        }
                }
                catch(InterruptedException ie)
                {
                }
                data = dataReceived;
                processMessage();
        }
}
```

Let's take a closer look at the DatagramEvent class, which extends java.util.EventObject. This class keeps a reference to the content of the Datagram and provides an accessor method to allow us to make a copy for use elsewhere.

```
public class DatagramEvent extends java.util.EventObject
{
        private byte[] data;
```

A `DatagramEvent` is constructed by passing the Receive object that produced the event and a reference to the raw data contained within the Datagram. The event source is passed to the superclass, and the reference to the data is stored locally to prevent new Datagrams from arriving and overwriting the buffer.

```
public DatagramEvent(Object source, byte[] data)
{
        super(source);
        this.data = data;
}
```

Finally, we have the method by which we return the data contained in the Datagram to the listeners or listeners interested as a String.

```
public String getData()
{
        String newdata = new String(data);
        newdata.trim();
        return newdata;
}
```

The last part of the chain is the DatagramListener interface itself. It extends java.util.EventListener which is purely a tagging interface and provides no functionality.

```
Interface DatagramListener extends java.util.EventListener
{
        public void DatagramReceived(DatagramEvent event);
}
```

This mechanism of events and listeners is implemented twice more in the Whist application. The first instance of this is between the TokenRing engine and the Game engine, and the second between the Game engine and the UserInterface engine. The implementation is virtually identical to that described above—the main differences are in the names of the classes and methods and also in the type of data passed on in the events generated.

What happens if we receive two Datagrams, one immediately after the other? Do we lose the data contained in the first Datagram by it being overwritten by the data from the second Datagram or is the second Datagram ignored? If we do get loss of data, how can we avoid it? One suggestion includes queuing the events, using a similar principle to that in java.awt.EventQueue. Another consideration to take into account is what happens if the TokenRing is still processing a previous Datagram when a new one comes in. We can guard against problems caused by a second Datagram arriving before the TokenRing has finished processing the first by calling `processMessage()` in a new Thread for each arriving message. This should not be undertaken lightly as the creation of each new Thread uses valuable resources. For more information on this, the reader may wish to consult the chapter on optimization (Chapter 8).

We have now studied the main functionality of the TokenRing class and the process by which we transfer data from one part of the program to another. We will now move on to the Game engine, where most of the data received in the TokenRing engine is processed.

6.1.6 Game Engine

The game engine is where the processing of all turns occurs and where information on scores and the cards in each hand is stored. The main class in this package is `Game`. This class implements both ReceivedTurnListener and Runnable. By implementing ReceivedTurnListener we are able to pick up ReceivedTurnEvents created in the TokenRing engine. Implementing the Runnable interface allows us to enable this class to run in its own thread, thus preventing some threading issues which developed when we moved from using a text user interface to a graphical one. The reader might like to remove the implementation of Runnable and observe the consequences of their action!

```
public class Game implements ReceivedTurnListener, Runnable
{
        :
        :
```

The construction is simple; we create instances of a ScoreCard, a Deck and a TokenRing and register interest in ReceivedTurnEvents created by the TokenRing engine.

```
public Game()
        {
            deck = new Deck();
            sc = new ScoreCard();
            tokenRing = new TokenRing();
            tokenRing.addReceivedTurnListener(this);
        }
```

As with the TokenRing class, we have a method for processing the data from the events posted to the listener. This is `processData()`. This creates a String array containing one or more of the following letters depending on the action or actions it requires the UserInterface to take.

Letter	Action
C	The game is in the middle of a trick and the cards that have already been played should be shown to the user
E	A game has been completed and/or the decision has been taken to end it. The final score should be displayed to the user
F	The game has come to an end and the score is to be displayed to the user. The user is also given the option of playing another game or ending the current session
H	The hand of cards that the user is playing with should be displayed
O	The cards played in the previous trick should be shown to the user
Q	One of the other players has ended the game prematurely and the current score is displayed

This method checks that the first character of the data is neither an "X" nor a "Q". The "X" indicates that this is the start of a new game and that the deck should be shuffled and the hands dealt. The "Q" indicates that the game has finished prematurely. Assuming that we do not have an "X" or a "Q", we step through the array of characters until we reach a "/" and then look at the

character directly following the slash. If this is an "F", we know that this data is the continuation of a game and not the start of a new game.

```
private void processData()
{
        if (dataReceived[0] != 'X' & & dataReceived[0]
           !='Q')
        {
                int count1 = 0;
                while (dataReceived[count1] != '/')
                      count1++;
                count1++;a
                if (dataReceived[count1] == 'F')
                        isNewGame = false;
                else
                        isNewGame = true;
        }
```

If we receive an "X" as the first character in the data we know that this is the start of a new game. If no other games have been played, we instantiate a Hand and deal the cards. The only data we have to pass on to the next device is the other three hands of cards, which are to be dealt to the other players. We get the number of players in the game from the TokenRing package using a simple getter method.

```
if (dataReceived[0] == 'X')
{
        isNewGame = true;
        :
        if (firstGame == true)
        {
                Card[][] cards= deck.deal(noOfPlayers);
                myHand = new Hand(cards);
                moved = deal(cards);
                st = moved.trim() + "/";
        }
```

If previous games have been played, we extract the cards played in the last trick and update the score accordingly. The score has a unique identification number, similar in purpose to that in the TokenRing engine, and allows us to check that we have not updated the score for this trick already. Again, we instantiate the Hand and deal the cards. This time we need to pass on the four cards played in the previous trick to the next device, along with the remaining three hands we have dealt, the first card played in that trick (to allow us to determine which player went first) and the score identification number. If this is not a new game, we append an "F" to the end of the String to be sent.

```
        else
        {
                Card[] cs = getLastRound();
                if (getScoreId() != scoreId)
                {
                        scoreId = getScoreId();
                }
                sc.hasGameEnded();
                myHand = new Hand();
                moved = deal(myHand.getHands());
                String s = "";
                for (int i = 0; i < 4; i++)
                     s += String.valueOf(cs[i].getCard()
                          + "@");
                st = (moved.trim() + "@" + s + "~" + String.
                     valueOf(getFirstCard().getCard()) + "~;"
                     + scoreId + ";/").trim();
                if (!isNewGame) st += "F";
        }
```

Unless this game has been stopped, we send the move made by this player on to the next player.

```
        if (!gameStopped) tokenRing.sendMove(st);
```

If the game has been stopped, we reset the `gameStopped` variable in preparation for a new game and notify anything waiting on the lock object.

```
        else
        {
                gameStopped = false;
                synchronized(lock)
                {
                        lock.notify();
                }
        }
}
```

We check that the game has not been stopped before sending any move, and it can be seen that the above check is present before each call to `sendMove(String st)` in the source; however, it will not be shown here again.

If we receive a "Y", the player receiving this should take their turn. We check if this is the first game to determine if we need to display a previous trick. The main difference is that we create a new `GameEvent`, passing a reference to ourselves and an "H" as arguments. We then call the `processGameEvent` method and wait on a synchronized lock until we are notified.

```
else if (dataReceived[0] == 'Y')
{
        if (firstGame == true)
        {
                :
                gameEventData = "H";
                GameEvent ge = new GameEvent(this,
                    gameEventData);

                processGameEvent(ge);
                synchronized(lock)
                {
                     try
                     {
                             lock.wait();
                     }
                     catch (InterruptedException ie)
                     {}
                }
```

If the game has not been stopped, the remaining hands from the `dataReceived` are stored in a String, along with the value of the card the player has just selected (taken) and the score identification number.

```
        if (!gameStopped)
        {
                :
                moved = getRestOfCards() + "% " + String.
                    valueOf(taken) + "% *;" + scoreId + ";/";
        }
        :
}
```

The `processData()` method continues in a similar vein throughout, which can be seen by consulting the code on the CDROM with this book. Many of the methods provided in the `Game` class either perform data manipulation or are getter methods to return private data for other classes or to extract parts of the `dataReceived` array. The essence of the data encoding is the use of various separation characters, for example the score identification number is placed between semi-colons. This allows the data manipulation methods to search the array for the required character and extract the following data, and is a simple way of stringing different data items together for sending in Datagrams.

A modification to the Game engine that would reduce the complexity of the code would be to use java.lang.String instead of char arrays. We could also improve the performance of this engine by using StringBuffers instead of Strings. For a discussion of why this is better, the reader should consult the chapter on optimization (Chapter 8).

It can be seen that we have methods for processing the events which occur at the end of a trick and at the start of a new game; however, these again are mainly data manipulation methods and rely on the principles explained above.

As discussed above, we create both a deck and a hand of cards, similar to that in a real card game, but containing virtual cards. We will now take a brief look at the Deck, Hand and Card classes.

`Deck` is a class that allows creation of a deck of 52 cards, and shuffles and deals them.

```
public class Deck
{
      private static final int NO_OF_CARDS_IN_DECK = 52;
      private static Card[] deck;
      private static int noOfPlayers;
      private static int maxNoOfCardsInAHand;
      private Card[][] hands;
      private int[] indexToPlayers;
```

The constructor just creates the array of Cards and fills it.

```
public Deck()
{
     deck = new Card[NO_OF_CARDS_IN_DECK];
     for (int i = 0; i < NO_OF_CARDS_IN_DECK; i++)
           deck[i] = new Card(i);
}
```

The `deal()` method has two functions, which it performs at the same time. Those two functions are shuffling and dealing. It does this by cycling through the deck of cards, one by one, and selecting the player to receive the card at random. Once a player has a full hand, they are essentially removed from the array of players.

```
public Card[][] deal(int noOfPlayers)
{
      this.noOfPlayers = noOfPlayers;
      indexToPlayers = new int[noOfPlayers];
      maxNoOfCardsInAHand = NO_OF_CARDS_IN_DECK / noOfPlayers;
      hands = new Card[noOfPlayers][maxNoOfCardsInAHand];
      for (int i = 0; i < noOfPlayers; i++)
            indexToPlayers[i] = i;
      Random random = new Random();
      int generatedNumber;
      int index;
      int[] noOfCardsDealtToEachPlayer = new int[noOfPlayers];
      for (int i = 0; i < noOfPlayers; i++)
            noOfCardsDealtToEachPlayer[i] = 0;
      int noOfRemainingPlayers = noOfPlayers;

      int playerReceivingCard;
      for (int i = 0; i < (NO_OF_CARDS_IN_DECK); i++)
      {
            generatedNumber = random.nextInt();
            if (generatedNumber < 0)
```

```
            generatedNumber = generatedNumber * -1;
            index = generatedNumber % noOfRemainingPlayers;
            playerReceivingCard = indexToPlayers[index];

            hands[playerReceivingCard][(noOfCardsDealtToEachPlayer
                [playerReceivingCard])] = deck[i];
            noOfCardsDealtToEachPlayer[playerReceivingCard]++;
            if (noOfCardsDealtToEachPlayer[playerReceivingCard] >=
                maxNoOfCardsInAHand)
                indexToPlayers[playerReceivingCard] =
                    indexToPlayers[--noOfRemainingPlayers];
        }
        return hands;
    }
```

The Hand class allows us to store the hand currently in play and the trump suit.

```
public class Hand
{
        private static final char QUESTION_MARK = '?';
        private static final char HASH = '# ';
        private int noOfCardsInAHand;
        private Card[] cardsInHand;
        private int trump;
        private int cardsRemainingInHand;
```

We have two constructors; the first takes the double array of Cards created by deck.deal(...) and extract the cards for that hand and also the trump suit.

```
Hand(Card[][] cards)
{
        noOfCardsInAHand = cards.length;
        cardsRemainingInHand = noOfCardsInAHand;
        for (int i = 0; i < noOfCardsInAHand; i++)
                cardsInHand[i] = cards[0][i];
        trump = Card.getIntSuit(cards[0][noOfCardsInAHand-1]);
}
```

The second takes the data received by the Game and extracts the cards for that hand. These cards will have been dealt by another device.

```
Hand(String dataReceived)
{
        int length = dataReceived.length();
        int count = 1 ;
        int index = dataReceived.indexOf(QUESTION_MARK);
        while ((index < length) && (index != -1))
        {
                index = dataReceived.indexOf(QUESTION_MARK,
                    ++index);
                count++;
        }
```

```
        noOfCardsInAHand = count;
        cardsRemainingInHand = noOfCardsInAHand;
        cardsInHand = new Card[noOfCardsInAHand];
        createHand(dataReceived);
}
```

`CreateHand()` extracts the cards from the data received and fills the array with them. It is also responsible for extracting the trump suit and storing it appropriately.

```
private void createHand(String dataReceived)
{
     int count = 0;
     int start = 0;
     int end = 0;
     byte[] bytes;
     for (int i = 0; i < noOfCardsInAHand; i++)
     {
          end = dataReceived.indexOf(QUESTION_MARK, start);
          bytes = dataReceived.substring
               (start, end).getBytes();
          cardsInHand[i] = CardConverter.getCard(bytes[0]);
          start = ++end;
     }
     int trump;
     start = dataReceived.indexOf(HASH);
     end = dataReceived.indexOf(HASH, start);
     bytes = dataReceived.substring(start, end).getBytes();
     trump = Card.getIntSuit
         (CardConverter.getCard(bytes[0]));
}
```

We also provide methods for accepting a new hand from an incoming Datagram; however, this method is very similar to the above methods and differs only in that there is already an array in which we can hold the cards.

The following method ensures that the cards selected by the player during their turn is a valid move and returns true if this is the case. We use the first card taken in the current trick to determine the suit for that trick and then determine if the card taken by the player is of either that suit or the trump suit. The only exception to this rule is when there are no cards of the suit of the first card, in which case the player can select any card.

```
public boolean validateMove(int t, Card firstCard)
{
       Card taken = Deck.getCard(t);
       boolean move = false;
       if ((Card.getIntSuit(taken) == Card.getIntSuit(firstCard))||
            (Card.getIntSuit(taken) == trump))
       {
              move = true;
       }
```

```
        else
        {
                boolean canGo = false;
                for (int i = 0; i < cardsRemainingInHand; i++)
                        if (Card.getIntSuit(cardsInHand[i])
                            == Card.getIntSuit(firstCard))
                                canGo = true;
                move = !canGo;
        }
        return move;
}
```

If the move is valid, we remove that card from the array to ensure it cannot be played again in this game.

This class makes use of the `Card` class, which allows us to map cards (rank and suit) to a number between 0 and 51.

```
Card(int num)
{
        createCard(num);
}
```

The `createCard` method performs some simple arithmetic operations to map a number between 0 and 51 to a rank and suit, as shown below.

```
private void createCard(int num)
{
        int t = num % NO_OF_SUITS;
        switch (t)
        {
                case 0: suit = HEART; break;
                case 1: suit = CLUB; break;
                case 2: suit = DIAMOND; break;
                default: suit = SPADE; break;
        }
        t = num / NO_OF_SUITS;
        switch (t)
        {
                case 0: rank = 2; break;
                :
                :
                case 8: rank = 10; break;
                case 9: rank = JACK; break;
                case 10: rank = QUEEN; break;
                case 11: rank = KING; break;
                default: rank = ACE; break;
        }
}
```

There are also utility methods which allow other classes to access the values and suits of cards and convert suits and ranks into the corresponding unique identification number, and vice versa.

The final class in this package we will look at is the ScoreCard class. This class simply increments the score for the player who won the previous trick and checks that the number of tricks won by that player is not enough to win a game. We allow other classes to interrogate the ScoreCard to determine if a player has won enough tricks to win the game by the `hasGameEnded()` method. This returns true if the game has been won by someone. Note that, for testing, it is often convenient to reduce the number of points required to win a game from 5 to 1 to save time.

This is the essence of the Game package. True to the design principles, this package has no knowledge of the underlying transport mechanism and it would be very simple to change the TokenRing package for another one that provided the same services.

These two packages are of no obvious use to the user, however, as they are unable to display anything on the screens of the devices. We have a separate package for this, the User Interface package.

6.1.7 User Interface Package

The User Interface package started life as a simple class that only displayed information in a command prompt. This was used to develop and debug the TokenRing and Game engines, and the graphical user interface (GUI) displayed on the Quartz emulator was only added at the end of development once the majority of problems had been ironed out. This demonstrated how the packages had been developed to allow them to be interchanged as required.

The main class in this package is Whist, which extends Qframe. (QFrame is discussed in Chapter 3). We make a call to super, passing the argument "4" to create four Cards. We then create the four main views and assign one Card to each.

```
public Whist()
{
    super(4);
    :
    smsView = new SMSBaseView(getCardAt(0), this);
    demoView = new DemoBaseView(getCardAt(1), this);
    wait = new WaitingView(getCardAt(3));
    gui = new GUInterface(getCardAt(2), this);
    :
}
```

We have methods allowing us to show different views and which set the menu options as required; for example, the code below displays the DemoBaseView.

```
public void showDemoBaseView()
{
      displayCardAt(1);
      editMenu.removeAll();
      menuBar.add(editMenu);
      editMenu.add(mWDPSMS);
      editMenu.add(mDemo);
}
```

The DemoBaseView allows us to run up to four emulators playing the game on one machine. The remaining emulator will be on one or more other machines. The required setup for this mode is determined in the launch script by the number of addresses passed in. If the four emulators are running on one machine, the user is unable to start the game until the fourth emulator has successfully loaded. Note that for the case where one or more emulators are on a different machine, there is no such checking mechanism.



Figure 6.4 Up to four emulators can run on one machine. The highlighted areas mark the trump suit and cards played.

The other option we have is to start a game playing via SMS using the SMSBaseView class. This class makes use of the Contacts database, which is discussed in Chapter 5 to extract the names and cell phone numbers from the native Contacts database and use them to create WDP/SMS addresses for use by the TokenRing engine. The user is able to choose which players they would like to join the game by selecting the names from Choice boxes. The SMSBaseView class checks the user's selection to ensure that they have not selected the same person more than once. This view assumes that there are three or more contacts with cell phone numbers in the database.

Once the game has been started using either WDP/SMS or UDP, a `WaitingView` is displayed. This simply informs the user that the game is starting and that it might take a while for the Datagrams to pass round all of the players.

Once we have all four TokenRing engines configured correctly, we display the remaining Card, which is called GUInterface. This is the class that displays the cards in the current hand. It has a `processData()` method similar to the TokenRing and Game engines allowing it to process the data it receives from Events posted to it. This is a large, yet simple class and readers are advised to look at it in detail only if they have a special interest in User Interfaces.

The final two classes we will discuss here are CardImage and PanelWithStruts. The CardImage class simply associates an image of a card with the suit and rank that correspond to the image. The PanelWithStruts class allows us to create a Panel that is sized suitably to fit a row of card images onto it with small struts to separate each suit from the neighboring one.

6.1.8 Design Issues

We have taken a rather whirlwind approach to the three packages supplied with the Whist application, but fully documented source code is available for you to experiment with and improve on. We will now take a look at some if the issues we encountered when developing this game.

Cost

Many Internet gamers today are young children and it is anticipated that wireless games will also appeal to the younger generation. Because of the usually limited finances of children, it was decided to limit the overall cost for the user to the minimum as far as practical. In practice, it will not only be children who will thank you for minimizing the cost of playing games. Conversely, network operators may take a different view!

Costs were minimized by not sending each move to each player as it was taken, but waiting for it to propagate round the ring. The former method requires three messages to be sent per turn, while the latter only requires one per turn. This means that the devices are not completely synchronized with one other, but that they would never be more than one trick out of date. This was not deemed to be a significant problem, and the saving to the user outweighs the disadvantage. All the data required for one turn could fit into one message (160 characters) by sending the cards as ints (numerical representations of their rank and suit). Games requiring more than four players may need more than one SMS message to be sent to carry all the addresses of the players.

In this version there is no checking to ensure that the message reaches its destination, or even if it has been understood when it arrives. A solution based on the three-way handshake would be possible, but it would require many more Datagrams to be sent, which is an undesirable side effect in view of the cost constraints.

Possibility of Messages Being Lost

How likely is it that we will lose one or more messages throughout the duration of the game? This may be due to a delivery failure (for which you may or may not get a notification from your provider) or to the data becoming corrupted during transit. Possible solutions to this are the three-way handshake to ensure that delivery is successful; and a resend option, which would require the device receiving the corrupt message contacting the sender and asking them to resend.

6.1.9 Future Improvements

There are several enhancements this application might require to develop it into a full application that would be suitable for sale. Possible improvements include:

- adding sound which can be switched on or off as required by the user
- the application could also be modified to detect if it is in the background when a new message comes in. If it is in the background, it could display a system modal dialog informing the user that it is now their turn with options to play now or wait. If the user decides to play, the application will be brought into the foreground.
- intelligent computer opponents could be created to take the place of one or more real players
- a resend option could be included to allow the previous move to be re-sent if lost or corrupt
- a server could be added to allow the players access to more opponents. (This might also provide a further revenue source for the network operator)
- currently it is assumed that all people included in the game want to play. This may not be the case so it is advisable to invite the players to join the game before it is set up. If a player declines, the person who started the game should be given the option of choosing a replacement
- there is the possibility that the application may be closed down by the system, so there should be a solution to save the current state of the game so that the player can restart it
- there is no guarantee that the player will take their turn when the Datagram arrives, so timed notifications could be used to remind the player to take his or her turn
- this version of the game is only for Quartz and it could be ported to other form factors that support the JavaPhone APIs, like Crystal.

6.1.10 Platform Constraints

There is limited text entry due to the lack of a keyboard, but methods for entering text exist, using either a virtual keyboard or handwriting recognition.

The screen will be small and the colors available limited. Relative to a desktop PC, the platform has only a small amount of memory available to it and we should ensure that we use the minimum amount possible. Processing power is also constrained — the processors available on WIDs will be far less powerful and there will be no access to the kind of hardware acceleration we are all used to on the desktop.

6.1.11 Where to go from here

We've given you some ideas as to how to go about writing wireless Java games; however, we have only been able to scratch the surface of the issues involved here. Here are a few more ideas for you to think about:

- the application does not provide computer opponents
- the application does not use the address book application for the names and addresses of the players
- the application does not allow the user to select players and does not ensure that the correct number of addresses have been entered
- the application does not allow the player initiating the game to invite other players to join the game
- the application does not check that the invited players have the game on their machine
- the application does not allow messages to be re-sent if they are not delivered
- the players cannot choose their partners.



Figure 6.5 Game over.

6.2 Developing Action Games

In this chapter we've already discussed a number of the issues around developing games for WIDs. Here's another question to ask yourself: what is unusual about wireless games and what is unusual about action games on wireless devices? Let's look at some of the restrictions and the opportunities in designing action games for constrained environments.

6.2.1 Constraints

WIDs are constrained by the following:

- screens are small, with low-resolution graphics, more often than not in monochrome. You must also cater for different sized screens on different devices with a range of input methods. Remember that not all devices will have a touch-sensitive screen and many will not have a keyboard. To achieve this flexibility you may need to produce several different user interfaces to allow your game to reach the biggest possible audience. By using the model described above, it is a relatively simple process to port your game to different devices. Processing performance and graphical rendering ability are slow compared to other platforms like PCs and consoles so you can't port games like Quack and expect them to have the same visual impact as the PC version
- controls are awkward and tend to be designed primarily for data interaction. Different devices will have different controls and there is no guarantee that devices within the same family will have the same controls
- WIDs are more constrained than PCs and consoles in their ability to deliver games. Although processing power continues to grow, for the foreseeable future, the disparity will remain.

There are also disparities between devices, with some offering PersonalJava but others offering the more limiting MIDP.

These constraints may seem very limiting; however, with a bit of imagination and patience it is possible to use techniques such as clipping and double buffering to increase the apparent quality of the graphics (see Chapter 8). We cannot expect to emulate processor-intensive applications which require fast Internet connections and dedicated graphics hardware, for example Quake, for the time being; however, we can still create some great games with a little skill and practice.

6.2.2 Advantages

All platforms are constrained in some way: we could easily draw up a table of constraints for PCs and consoles. For instance, consoles are constrained by analogue TV, and cannot be upgraded. The complexity of PCs that they

can suffer from compatibility problems, difficult configuration and so on. The constraints of developing applications for WIDs also show us that a new kind of games idiom will emerge. What are the advantages of WIDs?

- they're wireless: You can play multi-player games anywhere and on multiple bearers like GPRS, 3G, Bluetooth, IrDA etc
- they're advanced clients. They have processing power on board, which means they support the online games client–server model
- they're communications-centric. Games can be integrated into telephony, to take advantage of a phone's settings, profiles, and features. With Java you have the unique advantage of being able to access new and exciting features like WDP/SMS, which is not available to those programming natively for security reasons.

6.2.3 Bearer Technology

Key wireless technologies include:

- GPRS (2.5G) and W-CDMA/cdma2000 (3G): GPRS already exists in a commercial form and is being rolled out to consumers in Europe. The advantage of GPRS is that it is being deployed globally, which means that games developed for one region will have similar network environments in the US, Europe and Asia. What bandwidth can we expect from GPRS? Probably 20–50 kbps over the air when not in a laboratory environment
- Bluetooth technology implementations will allow localized games (say in a radius of 10 meters) which is a money-saving way for players to play "pico-networked" games without the need to pay for airtime.

6.2.4 Latency

Over the wired Internet there is typically a delay or latency caused by the processing of network traffic between two machines, often related to the number of Internet routers on the round trip between players. This delay is typically anything from 0.05 to 0.5 seconds, which can make it difficult for a player to react to another's actions.

One solution is turn-based games, where the processing takes place on board the WID, with scores at the end of each round being forwarded to each player, for instance. Another approach might be to use prediction, extrapolating the movements of other players until confirmation of their positions is received.

6.2.5 Rethinking Action

Rethink the paradigms of action games.

Is constant action necessary or could or should it be interspersed with pauses, text, or speech? What are the new paradigms of action games on a WID?

What happens if the player receives a phone call mid-way through a game? Does the call go to voice mail and if so, how can the game be integrated with the JavaPhone APIs on a WID?

How can buttons and keys be used? Communicator devices can come with and without keyboards and pointing-devices, even touch-screens. How can these be used to enhance the games experience?

Smartphones are expected to dominate WID sales as they replace today's "dumb" terminals. A dumb terminal is a phone with no on-board processing power and is unable to run applications, although they may have WAP browsing facilities. Unlike Symbian's Communicator reference design, Symbian Smartphones are intended for one-handed operation. How does this fit in with action games?

Is network latency an important issue for your action game?

6.2.6 Optimizing for Mobile Environments

Many games are now being designed as "wireless friendly" applications and there is help for developers looking to learn more in this area.

Symbian was a founder member of the GPRS Application Alliance (GAA) at ITU Telecom '99. Its mission is to "enhance the quality of your mobile and business life, anytime, anywhere."

The initiative supports application developers creating robust, simple, and personalized applications for the wireless marketplace. As a result of the GAA, Symbian shareholder Ericsson launched the Mobile Applications Initiative (MAI) in June 1999 for applications developers, platform designers, system integrators, service and content providers, and handset manufacturers, to test and take applications to market. The initiative is open to the entire wireless industry.

The MAI supplies three services:

- application optimization. Developers can perfect applications in two different testing environments. The Global Application Test Environment (GATE) gives developers an indication of how applications might behave on a GPRS network. A second environment, the GATE Application Performance Evaluation, is targeted toward application/software developers with applications capable of performing in the GPRS environment. It supports IP over GPRS/EDGE with UMTS (WCDMA), CDPD, and cdma2000 in the near future
- knowledge sharing. A series of training courses that range from the most basic knowledge of wireless to advanced technical training. Courses include GSM, WAP and GPRS and other mobile Internet-related topics. Knowledge sharing also includes an initial consultation with MAI technical personnel

- marketplace networking. MAI provides a global networking opportunity for member companies through online interaction on its website, ***http://www.MobileApplicationsInitiative.com***. Members can also meet and work with other application developers at Ericsson-sponsored and industry-related events.

  The website also has a list of MAI testing centers around the world — including core centers in Berkeley, Calif., and Stockholm, Sweden.

7

Building User Interfaces for WIDs

Jonathan Allin

This chapter looks at guidelines and general rules for creating user interfaces for constrained devices, with examples focused mainly on EPOC Release 5 devices.

The chapter's objective is twofold: to encourage you to think about opportunities for Java on constrained devices, and to emphasize that developing for such devices requires more care than developing for desktop machines.

We'll start with a look at some current Symbian devices and examples of applications running on these devices, look at a range of large and small Java applications and consider their suitability for WIDs and PDAs, then look at guidelines for creating UIs for constrained devices. The final section discusses some UI aspects of Symbian's reference designs, including Quartz and Crystal, which were first discussed in chapters 2 and 3.

7.1 Symbian Devices

The range of Symbian devices is continuously expanding. Table 7.1 summarizes the principal characteristics of Symbian devices based on EPOC release 5. A comprehensive list of the features of these and Symbian OS v6.0 devices appears in the appendices.

Table 7.1 shows how these devices vary in screen size, user input method, and processing power; ranging from the Revo Plus, a shirt pocket device with a 480 × 160 pixel monochrome screen, to the netBook, a sub-notebook with a full VGA color screen.

The next two devices are wireless information devices in the true sense, combining both telephony and PDA functionality.

Ericsson's communicator concept is based on the Quartz reference design. It uses a quarter VGA color screen (240 × 320 pixels) in portrait orientation. It doesn't use a keyboard or keypad but relies on a pen for input, backed up by handwriting recognition and a virtual keyboard.

Table 7.1 Principal characteristics of Symbian EPOC Release 5 devices

	Revo Plus	5MX	MC218	netBook	netPad
Manufacturer	Psion	Psion	Ericsson	Psion	Psion
Format	Shirt pocket	Pocket, brief case	Pocket, brief case	Small laptop	Tablet
Screen	480 × 160, mono	640 × 240, mono	640 × 240, mono	640 × 480, color	640 × 240, color
Memory	16 MB	16 MB	16 MB	48 MB	–
Processor	36 MHz ARM 710T	36 MHz ARM 710T	36 MHz ARM 710T	190 MHz StrongArm	190 MHz StrongArm
Java	JDK1.1	JDK1.1	JDK1.1	JDK1.1	PersonalJava
Input	Keyboard, pen	Keyboard, pen	Keyboard, pen	Keyboard, pen	Pen



Figure 7.1 Ericsson Communicator concept.



Figure 7.2 Nokia 9210 Communicator.

The Nokia 9210 is based on the Crystal reference design. It uses a 640 × 200 pixel color display. It doesn't have a touch screen in order to achieve pin-sharp resolution even with a small pixel size. Applications are controlled from the keyboard, the four buttons in the Command Button Area to the right of the screen, and by menus.

The significance from our point of view is that the 9210 is the first Symbian licensee device to support PersonalJava and the JavaPhone APIs (discussed in detail in Chapters 4 and 5 respectively).

7.2 Running Java Application Written for Desktop Computers

7.2.1 Write Once, Run Anywhere?

The J2ME's concepts of Profiles and Configurations (see Chapter 14) is a recognition that Java's promise of "write once, run anywhere" was probably a bit optimistic. Expecting to be able to run a large application written for desktop PCs on a PDA without changing anything is unrealistic, no matter what language it was written in.

However, Java is designed for portability, which it achieves more convincingly and on a wider range of platforms than probably any other language. This is due to tight standardization of the built-in types, binary formats, and VM specifications, plus a full library that includes a GUI. As a consequence, simple Java applications and Applets can often be run without any changes (and hence without having to recompile the class files) on most devices, including Symbian's, while porting more complex applications and Applets is generally straightforward.

We'll look at running and installing a number of Java applications, written for desktop machines, on the Symbian OS. For the moment we'll stay with EPOC Release 5.

The issues we encounter will lead us into the next Section 7.3, **Creating A User Interface That Works**.

7.2.2 Simpler Applications and Applets

The Symbian web browser supports Applets, so running an Applet is just a question of pointing the browser at the correct URL. If the Applet is local it can also be run by double-tapping on the relevant HTML file.

Figure 7.3 shows the ArcTest example from Sun's JDK 1.1.8 (see **demo\ArcTest** in your JDK installation), running under the emulator with a 640 × 480 screen. It can also be copied to a Psion netBook as **C:\Demos\ArcTest**, and then run by double-tapping on the file **example1.html**.

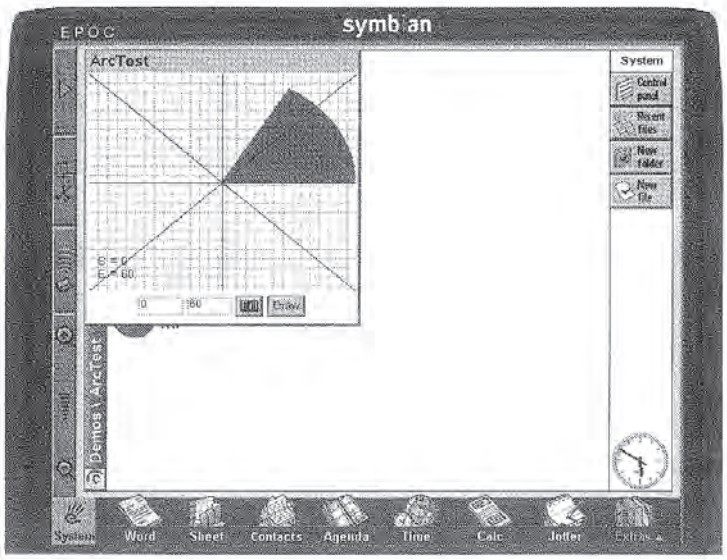


Figure 7.3 ArcTest run as an Applet in the web browser under the emulator.

The ArcTest example can also be run as an application. Instead of double-tapping on the HTML file, double tap on the class file, **ArcTest.class**.

Alas, not everything in the garden is rosy. Figures 7.4 and 7.5 are taken from the DrawTest example (also from JDK 1.1.8). A Symbian OS check box is wider than a Windows check box to make it more visible on low-contrast displays and easier to use with a pen. The consequence is that there's insufficient space for all the UI controls. Arguably the error is with the application because it has made non-portable assumptions about the size of the controls.

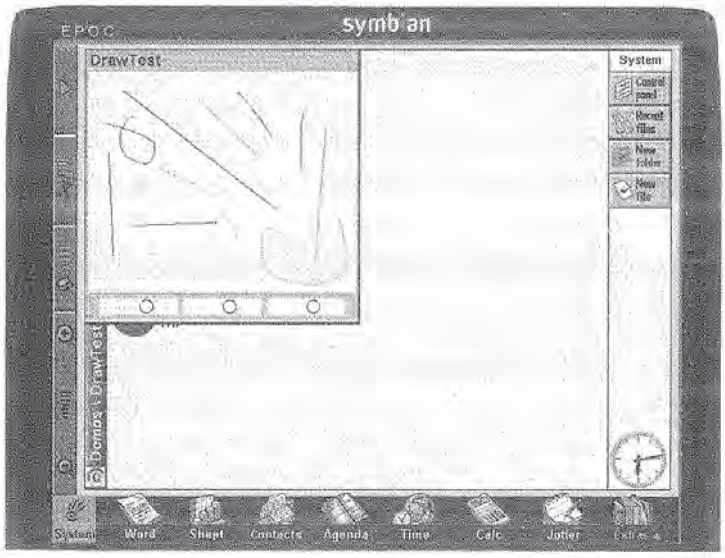


Figure 7.4 DrawTest running as an application under the Symbian OS emulator.

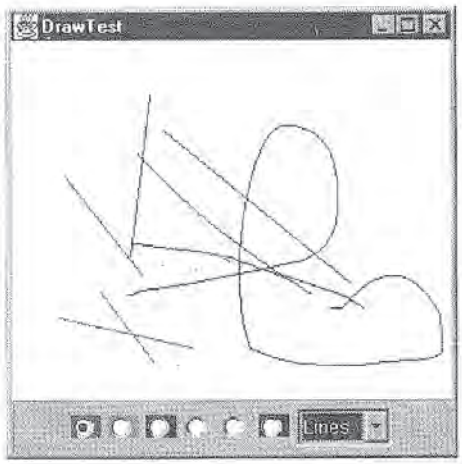


Figure 7.5 DrawTest run as an application on Windows.

You might also have noticed that under Symbian OS the check boxes don't have a colored background: this is a feature which is not supported in EPOC Release 5.

7.2.3 Larger Java Applications

In the next two sections we'll discuss two applications: JForge and Adobe Acrobat Viewer.

The first thing to say is that it is a testament to Symbian's Java implementation that such applications can be run. The main reasons that a **pure** Java application won't run are through lack of memory or disk space, or because it requires Java 2. An application will be unusable if it has windows, i.e. Frames, that need to be resized (those with multiple top-level windows fall into this category), or if it requires lots of processing power.

There is a bug in the Series 5 emulator which means that larger Java applications, such as these, can only be launched from the command line, not from the emulator's Extras bar. This is not a problem on Symbian hardware, nor with Symbian OS v6.0 emulators.

7.2.4 Installing and Running JForge

JForge is a GUI builder from Tek-Tools. It allows drag and drop editing of GUI components to create a Java source file which can then be compiled. JForge can be downloaded from ***http://www.tek-tools.com/jforge***. The latest version of JForge requires J2SE support and so won't run on the Symbian OS: the version illustrated in Figure 7.6 is an earlier one that was written to JDK1.1.

JForge has lots of useful configuration options, including the ability to set the initial window (i.e. Frame) size and position. It's therefore possible to ensure

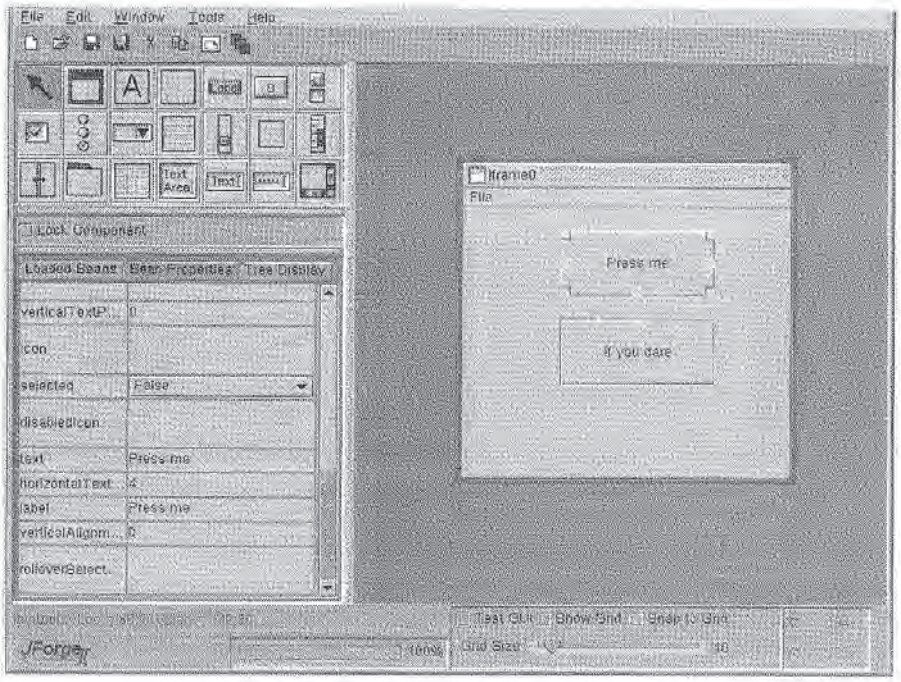


Figure 7.6 Screen shot of JForge, taken from the Symbian OS emulator set to an 800 × 600 pixel display.

that the application fills the screen, and the title bar can be pushed off the top of the screen by giving the y offset a negative value of about −20.

Even so, the user interface is such that it cannot sensibly be run on anything smaller than an 800 × 600 display. This is partly due to a very busy layout, and partly to the large amount of gray, unused screen real estate. Further, screen space is wasted because JForge uses Swing components, so the Swing menu bar is used instead of the Symbian OS's menu bar.

Installing JForge is a little more intricate than Acrobat Viewer. In order to run the configuration program it is first necessary to install JForge on Windows. This creates a series of files in the root of the C drive in a directory **.jforge2**. This directory (and its contents) then have to be copied to the root of the C drive on the emulator (i.e. **\Epoc32\wins\c**).

The following steps are needed to complete the installation:

1. Create the application directory, **\epoc\wins\c\system\apps\JForge**.
2. Copy into it **jforge.jar**, **public.jar**, and **swingall.jar**.
3. Create a **.app** file, **JForge.app**, so that an icon for the application will appear on the Extras bar. This can be achieved by entering `Appbuild JForge 100058A7` at a command prompt.
4. Create a file **JForge.txt** in the same directory. The contents of this should be

```
-DJFORGE_HOME=c:\system\apps\jforge -cp
jforge.jar;public.jar; swingall.jar uk.co.demon.sspl.jforge.Builder
```

the `-cp` parameter adds the three JAR files to the existing class path. `uk.co.demon.sspl.jforge.Builder` is the fully qualified name of the application's main class.

5. The application can be launched from the emulator by entering

```
\epoc32\release\wins\rel\java.exe -Mlarge-- -cd
c:\system\apps\JForge-cp jforge.jar;public.jar;swingall.jar
uk.co.demon.sspl.jforge. Builder
```

at a command prompt. `-Mlarge--` will start the emulator with an 800x600 screen size (note the bug that prevents the application from being launched from the emulator's Extras bar).

To run on a Symbian device:

1. Copy the **JForge** directory and its contents into **\System\Apps** on any drive.
2. Copy the **.jforge2** directory from **\Epoc32\wins\c** to the root of the C drive on the Symbian device.
3. The application can be run from the Extras bar (its icon will be a question mark because we have not created a special icon for it).

JForge includes an option to compile the Java code it creates into a class file. This option will cause the application to crash with an EPOC KERN-EXEC, because the Sun tools Java compiler is not part of the Symbian OS license. (There are several websites which describe how to install Sun's Java compiler onto a Symbian licensee device.)

7.2.5 Installing and Running Adobe Acrobat Viewer

Adobe Acrobat Viewer is used to view Adobe PDF-format files. It is a Java version of Acrobat Reader for use on platforms on which Acrobat Reader is not supported. It can be downloaded from ***http://www.adobe.com/products/acrviewer/main.html***.

The application will run reasonably well on a Psion Series 7 or Psion netBook, though it's probably too slow to be usable on a Psion Series 5MX.

Acrobat Viewer uses the AWT windows tool kit and makes extensive use of lightweight components to achieve a pleasant appearance. It doesn't use Swing, which means that (a) the application will be smaller, and (b) the MenuBar will be implemented by the Symbian OS's native menu bar. This is only displayed when needed, saving screen real estate.

When Acrobat Viewer is first run it creates a file **acrobat.properties** in the root of the drive. This contains a number of setup parameters, including the size and position of the application's main Frame. The size parameters can be modified to match the netBook's VGA screen or the Psion Series 5mx's half VGA

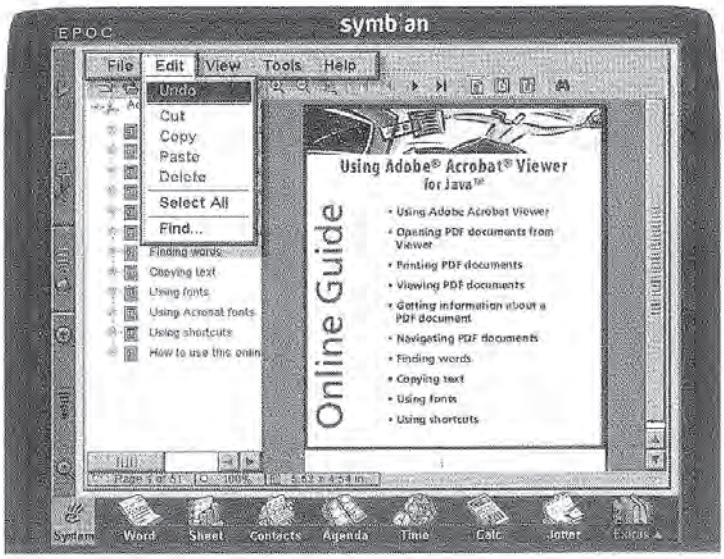


Figure 7.7 Adobe Acrobat Viewer screen shot running on the EPOC Release 5 VGA emulator.

screen. Unfortunately, the position parameters have to be positive. It is therefore not possible to hide the title bar by giving the Frame's origin a negative y value.

To run under the EPOC Release 5 emulator:

1. Copy the JAR file **acrobat.jar** into **\EPOC32\wins\c\System\apps\Acrobat** on the emulator.
2. The application can be launched from the emulator by entering

```
\epoc32\release\wins\rel\java.exe -Mvga-- -cd c:\system\apps\Acrobat
   -cp acrobat.jar com.adobe.acrobat.Viewer
```

at a command prompt. The `-Mvga--` tag will start the Emulator with a VGA screen size.

To run on a Symbian device:

1. Create a **.app** file, **Acrobat.app**, so that an icon for the application will appear on the Extras bar. This can be achieved by entering `Appbuild Acrobat 100058A6` at a command prompt (0x100058A6 is the UID assigned to Acrobat Viewer).
2. Create a file **Acrobat.txt** in the same directory. The contents of this should be

```
-cp acrobat.jar com.adobe.acrobat.Viewer -d -console -v
```

The `-cp` parameter adds **acrobat.jar** to the existing class path; `com.adobe.acrobat.Viewer` is the fully qualified name of the application's main class.

3. Copy the **Acrobat** directory and its contents into **\System\Apps** on any drive.
4. The application can be run from the Extras bar (its icon will be a question mark, because we haven't created a special icon for it).

Although a Java application can be launched by double-tapping on its class file, it's not possible to double-tap on a JAR file to launch the Java application within it.

7.3 Creating a User Interface that Works

The previous section highlighted the need to think carefully about screen real estate. JForge could be redesigned to fit within a VGA screen, and Acrobat Viewer would run nicely on a half VGA screen if the button bar and status area were optional.

Symbian and its partners invest considerable resource in developing a look and feel that is appropriate for constrained devices, ensuring that applications, and the devices themselves, have user interfaces that work well for the particular class of device and at the same time are attractive to use. It therefore makes sense for applications to adopt this look and feel, and the platform's guidelines, whenever possible.

What does look and feel cover? There are probably three requirements to think about:

1. Identical appearance and behavior of controls.
2. Identical appearance and behavior of application, e.g. applications should fill the screen, use similar menus, be launched in the same way.
3. Java applications have access to the same resources as C++ applications, such as UI controls and frameworks, native data types, sound management, IO ports.

The first requirement is achieved because Java UI controls use native peers (i.e. they use existing UIKON controls). The second can generally be achieved by following the platform's guidelines and making use of, and building on, application frameworks supplied by the Symbian Developer Network. Frameworks include CFrame for Crystal and QFrame for Quartz, and EFrame for the older EPOC Release 5.

Gaining access to native resources is perhaps the hardest to achieve and frequently requires the use of JNI programming (see Chapter 13).

7.3.1 AgendaWriter

AgendaWriter illustrates how a variety of techniques can be used to achieve a native look and feel and to access native resources.

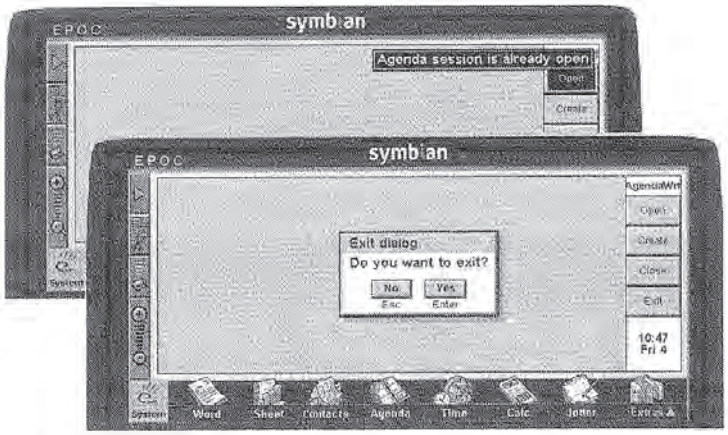


Figure 7.8 AgendaWriter: achieving a native look and feel.

The application can open and close an Agenda session and create a new appointment, achieved using JNI to access the Agenda engine.

JNI is used to display the message "Agenda session is already open" in a PrintNotify component and to display the Notify dialog. The latter looks like a standard Java dialog; however, it has the advantage that it will always be displayed on top of any other applications, so it will be visible even if your application is in the background.

The toolbar down the right-hand side of the application looks like a native toolbar. In fact it's a Java look-alike, created using Buttons and a Panel. The behavior is also broadly correct, for instance clicking in the title area above the Buttons brings up the open files dialog.

7.3.2 RemoteFiler

The RemoteFiler application is used to manage files remotely between any two machines running a JVM and supporting a TCP/IP connection. One of the machines runs the client (shown in Figure 7.13) and the second runs a server. The application allows files to be transferred, files to be deleted both remotely and locally, and directories to be created.

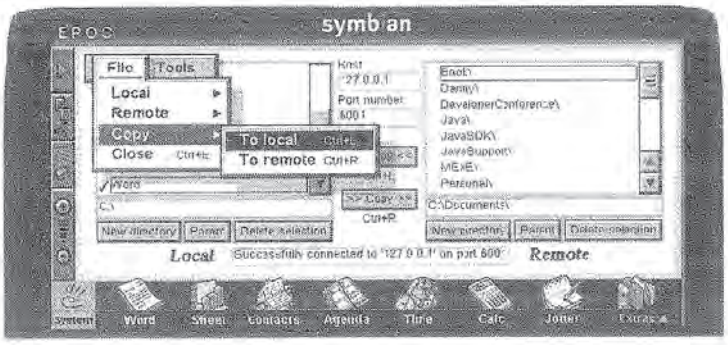


Figure 7.9 The RemoteFiler client.

The application has been written to have a Symbian OS look and feel, and, perhaps more importantly, to be usable on a small device. In particular:

- the application fills the screen, with no title bar
- menus are typical Psion Series 5 application menus. For instance, the File menu has a Close entry with a Ctrl+E shortcut
- the buttons have keyboard shortcuts to make it easier to drive the application from the keyboard, and buttons are matched with menu entries. It's therefore possible to drive the application from either the buttons, the keyboard, or the menu
- no dragging is required: files are transferred by selecting and copying
- the user interface is kept as simple as possible and doesn't waste screen real estate. The corollary is that you might feel the screen is too cluttered: arguably the TextFields used to specify the IP address and port number should be in a separate Dialog
- perhaps most importantly, neither the application nor user interface makes unrealistic demands on memory or CPU performance.

7.3.3 Personal Finance Manager

Personal Finance Manager was written by IBM for the Psion netBook. It is an easy-to-use application for managing cash, bank, and credit card accounts. Transactions associated with these accounts are displayed by the transaction log and can be categorized in order to track personal spending habits. The bill calendar tracks all bills, pending and upcoming, and provides a reminder of pending bills.

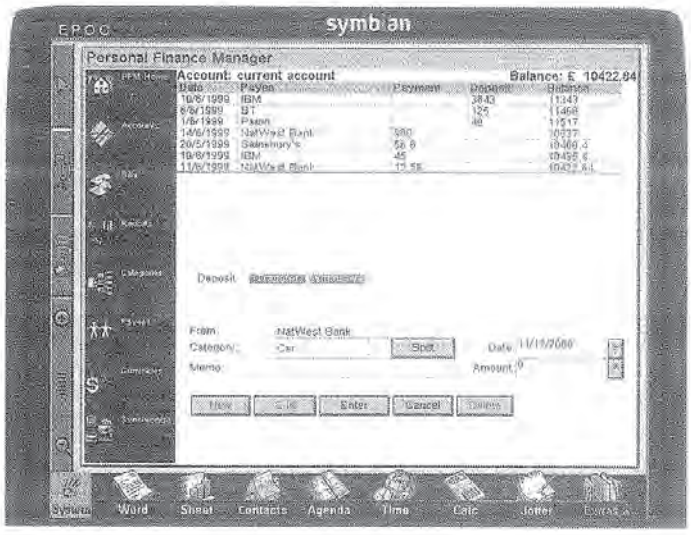


Figure 7.10 Personal Finance Manager from IBM.

The UI is straightforward and the application works smoothly. Arguably the Frame's title bar wastes space, especially important on a half VGA screen. More use could be made of menus and keyboard shortcuts to control the application, which would also mean that the left-hand tool bar could be optional, again saving screen space.

7.3.4 KahJongg

KahJongg is a variation on the classic Mah-jongg patience game. The idea is to remove all the tiles by removing matching pairs (though there are one or two wrinkles, as one might expect with a game that originated in China!).

The game was written by Andreas Kindlbacher and was the winner for the best Java game in the 2000 Symbian Developer Contest.

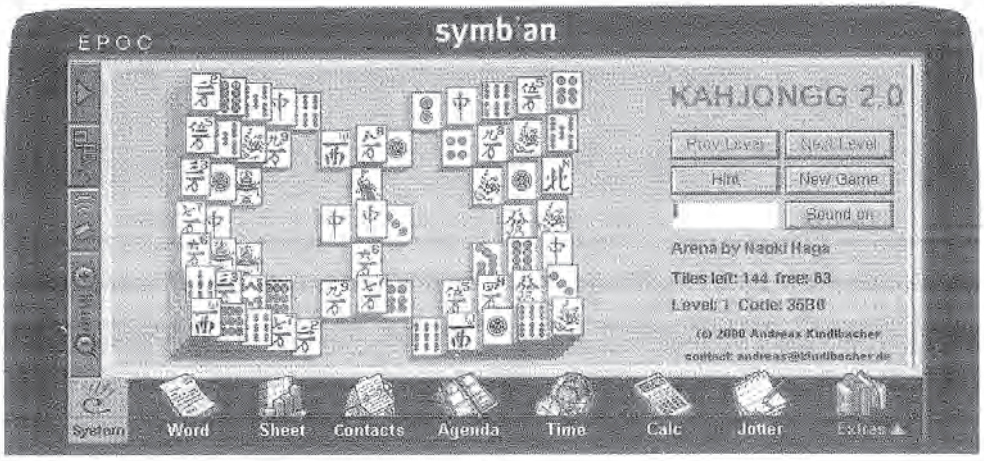


Figure 7.11 KahJongg by Andreas Kindlbacher.

KahJongg was written specifically for the Psion Series 5mx device. It works well on a grey scale screen despite using highly colourful and highly detailed images. The game play and the user interface is straightforward, and it looks like a native application. The game doesn't make unrealistic demands on the machine's performance: it is responsive when tiles are removed and the redraws are flicker free.

7.4 Symbian Reference Designs

Symbian's reference designs were introduced because of the wide variation in WIDs: screen sizes can vary from 100 × 100 to nearly half VGA, screens may be color or monochrome, input may be by touch screen and pen, keyboard, or keypad.

Devices within a reference design will have similar resources, binary compatibility so that an application written for one member will run on another member

of the family, and provide a shared user experience. This means that having learned to use one device in a family it should be straightforward to use another.

Even though the Ericsson R380 Smartphone and the Nokia 9210 Communicator have very different user interfaces, they have in common most of their code, and in fact over 70% of the code is common across all Symbian reference designs. All support WAP, Bluetooth and are Unicode based. And of course they will all support Java. The Quartz and Crystal reference designs are discussed in detail in chapters 2 and 3.

7.4.1 The Crystal Reference Design

Code-named Crystal, these devices are single-box solutions, which means PDA and phone functionality are combined into a single unit.

A typical Crystal device will have a half VGA (640 x 240 pixel) or near half VGA screen, almost certainly color, and a QWERTY keyboard to make entering information relatively straightforward. Such devices are driven by their keyboard, not by a pointer (though later devices may also support a pointer). The Nokia 9210 is an example of a Crystal reference design.

Crystal devices include both WAP and Web browsers, with the first release delivering PersonalJava and JavaPhone.



Figure 7.12 The Crystal reference design concept.

7.4.2 The Quartz Reference Design

Code-named Quartz, these devices are either one- or two-box solutions: in the latter case telephony functionality will be provided by a separate phone linked by Bluetooth, IR, or perhaps USB.

A typical device will have a quarter VGA screen, (240 × 320 pixel), in portrait orientation, and will typically be driven by a pointer. The user interface paradigm is therefore oriented toward information retrieval rather than information entry. However, handwriting recognition and a virtual keyboard is provided for text input—the Ericsson Communicator concept is an example.

Quartz devices ship with both Web and WAP browsers, and like Crystal devices will include Personal Java and JavaPhone.



Figure 7.13 The Quartz reference design concept.

7.4.3 Symbian's Smartphone Platform

Symbian smartphones will have a constrained user interface, with a small screen (Minimum screen size is 176 × 208 pixels, which gives QCIF compatibility) and a keypad. Devices will typically be thumb operated. The keypad will use T9 use predictive text entry and typically have just the 0–9 keys, "#", and "*". In addition, seven control buttons are mandated: End, Send, Application switcher, Up, Down, Do it, and Menu. Some devices may provide a pointer as an assistive input device, but primary input will always be through the keypad.

However, even though the UI is constrained, these are capable devices with reasonable processing capabilities and memory. They will use a WAP rather

than a Web browser, provide PIM functionality, and will support games and simple commerce. They are therefore far more than feature phones.

Because of the wide range of requirements for smartphones, rather than providing a single smartphone platform, Symbian provides a suite of applications, engines, and generic technology necessary to power the device. This enables the licensee to select applications and design a user interface that meets their specific requirements.

The level of Java support is likely to vary between products, though as a minimum will include MID Profile compatibility (the MID Profile is discussed in detail in Chapter 14).



Figure 7.14 The smartphone reference design concept.

7.5 WIDs are not PCs

Moving from desktop Java applications to writing Java applications for small devices is relatively straightforward (far simpler than if you were writing in C++). However, a paradigm shift is still required and we need to be aware of the differences between desktop computers and WIDs (and indeed other constrained devices).

7.5.1 Don't Make Assumptions

The following bit of code was written to check out an SDK installation. It was expected to do three things:

1. Print out the message "Hello there."
2. Make a beep.
3. Exit.

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("Hello there");
        java.awt.Toolkit.getDefaultToolkit().beep();
    }
}
```

In fact it only did one of these: it made a beep. The engineer concerned assumed that the installation was faulty; however, perhaps surprisingly, the observed behavior is the correct behavior. So what's happening?

Symbian devices do not generally present the user with a console: it's not an appropriate user interface for a WID. So, of course, anything sent to the console is simply swallowed, hence the "Hello there" message isn't displayed.

The reason the application didn't exit is a bit more subtle. The beep sound is created in a separate thread which doesn't terminate. The application therefore stays alive and the only way to exit at this point is to call `System.exit()`. The code also behaves in the same way under Windows, where we would perhaps use Ctrl-C to terminate the application without thinking too much about it. However, the Ctrl-C option is not a suitable approach on small devices.

7.5.2 Missing Features

The nature of constrained devices means that features we might take for granted on a desktop computer are not available to us.

We can't depend on:

- right-mouse clicks: in general we only have a pointer
- mouse motion: this means there are quite a few UI features that won't work, for instance: tool tips, rollover icons (icons which change shape when the mouse pointer is moved over them), mouse pointers that change depending on the underlying component (imagine trying to resize panels in a desktop window without the mouse pointer telling you when you are precisely over the narrow bar that separates them)
- a wait icon: a wait icon is displayed instead of the mouse pointer; however, a mouse pointer is generally not visible with a pen-based system. Symbian OS will only display the wait pointer (a small clock face) if a component has the focus and while the pen is in contact with that component

- the ability to move or resize a window. Frames in EPOC Release 5 can be moved but not resized, and neither resized nor moved in Quartz.

7.5.3 Some General Guidelines for User Interfaces

The following covers general ideas for creating UIs on constrained devices. Each reference design has its own specific guidelines, and these are discussed in the relevant chapters (see Chapters 2 and 3).

We shouldn't use double-click events. A double-tap generates a double-click event; however, this should be avoided as double-tapping is not easy for a pen-based user. EPOC Release 5 devices generally use single-tapping, i.e. tap once to select an item, tap again to activate it. Quartz uses a single tap to activate an object.

Mouse dragging should be avoided (other than for, say, scroll bars). There are two reasons:

1. It's too easy to accidentally drop what you are dragging (e.g. a document selection) in the wrong place.

2. `MOUSE_EXITED` and `MOUSE_ENTERED` events are not generated during a drag. Since we operate in pen environments, these events are generated by seeing which component has just produced a pointer event and, if it's not the same as the one that generated the last event, a `MOUSE_EXITED` event is posted for the old one and a `MOUSE_ENTERED` event for the new one. This goes wrong for dragging, where the event source doesn't change even though you may have dragged from one component to another, hence there are no enter/exit events until you release your drag and then tap on the component again.

Don't use desktop paradigms. You may be writing for people who have not been indoctrinated with the standard windows paradigms (as an example, horizontal scroll bars can cause problems, and so should be avoided). Your audience may be non-technical or not fascinated with technology, or they may be PC literate but operating in a different environment, perhaps out walking, waiting at a station, or on a plane.

Keep it simple! Your application should just work, like a domestic appliance. Keeping the interface simple means that there's less to learn and it will be more intuitive. This will also improve responsiveness and give the user more feedback.

The Quartz and Crystal reference designs support only a single input mode, pen for the former and keyboard for the latter. Other devices (such as the existing EPOC Release 5 PDAs) support both. If your target platform does support both pen and keyboard, don't force the user to switch input mode. If I've started a task using my pen, I should be able to complete it using the pen. Similarly, if I've started with a keyboard I should be able to complete it with the keyboard.

7.5.4 And Finally, Target the Device

To achieve the best results when writing (or porting) Java applications to WIDs we need to keep in mind the characteristics of the target device:

- smaller screen, which may be monochrome or color
- may use a keypad, keyboard, or pointing device
- less memory and processing power. WIDs and PDAs will always be constrained. Even though processing performance and available memory will increase on these devices, the gap will remain as desktop performance also improves.

Symbian is committed to Java because it provides a better programming environment: the device-independent characteristics of Java are less important. A Java application or service should therefore be written to a particular reference design — it is not expected that one application should work across all reference designs. In practice this means that a carefully architected wireless service will use a common engine, but a UI that is specific to the target device.